.NET

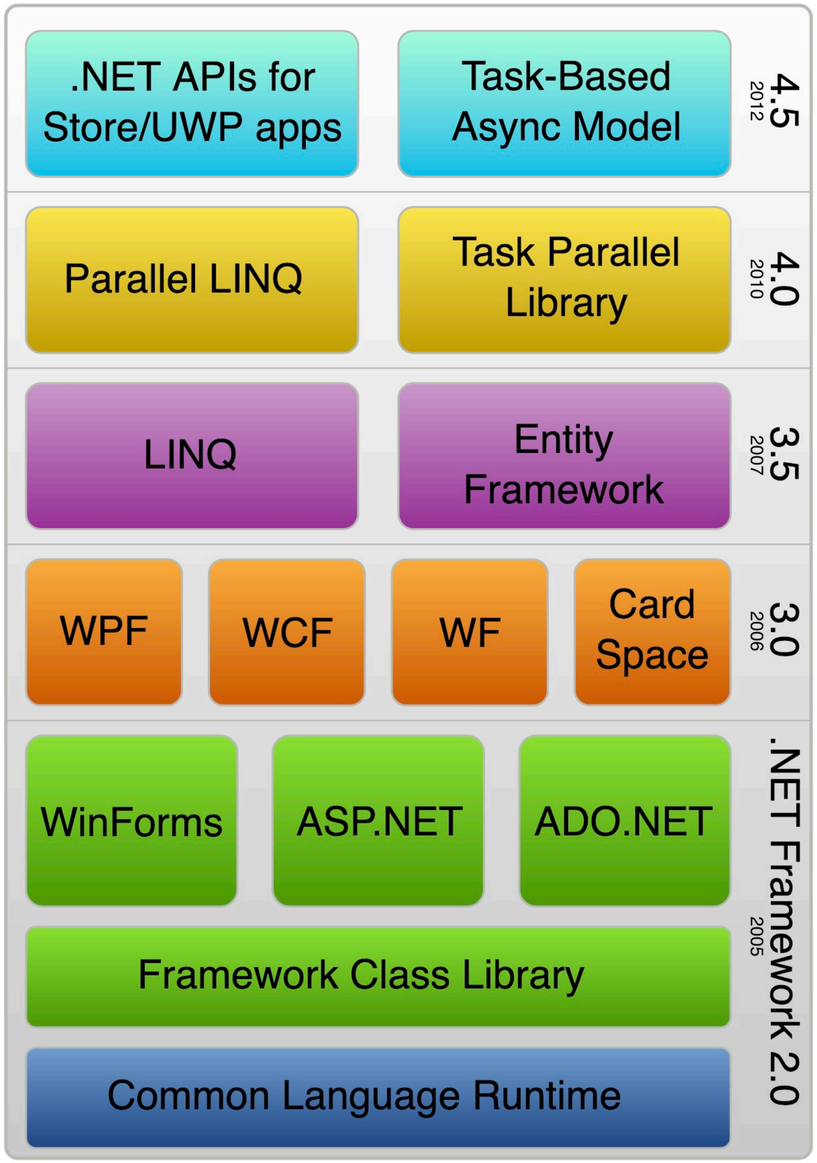
2160711

Unit 1

Introduction to .NET Framework

GTU Questions

1	Explain .Net framework architecture with diagram and also discuss IL,CLR, CTS and CLS	7
2	Differentiate between Managed and Unmanaged Code in .Net framework	4
3	What is Namespace ? Write a short note on Namespaces.	7
4	Explain practical importance of Window Application and Web Application	4
5	Explain unsafe code in C#.NET	3
6	What is dll hell problem? Discuss the solution of dll hell problem in .NET.	3
7	Explain Garbage Collection in .NET	3
8	Discuss .NET Assemblies and Assembly Contents in Detail	7
9	Explain .NET compilation process using Diagrammatic Flow	3



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

Microsoft Intermediate Language / Common Intermediate Language

MSIL

CLR

CLS

CTS

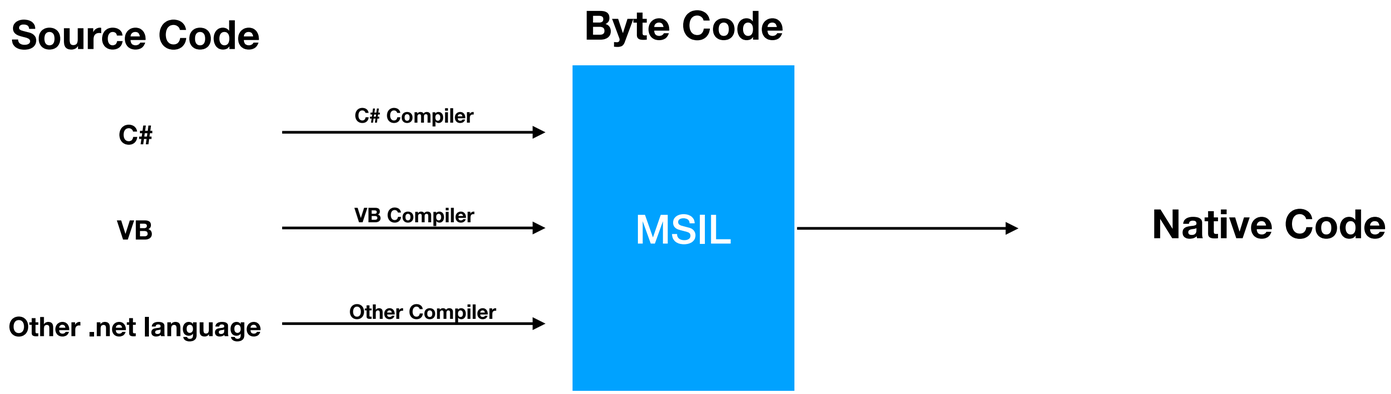
All .Net codes are first compiled to Intermediate Language.

The lowest-level human-readable programming language

Stack based Object-Oriented assembly language

Its byte-code is translated into native code

MSIL provides instructions for calling methods, initializing and storing values, operations such as memory handling, exception handling and so on.



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

Common Language Runtime

Manages the execution of .NET programs

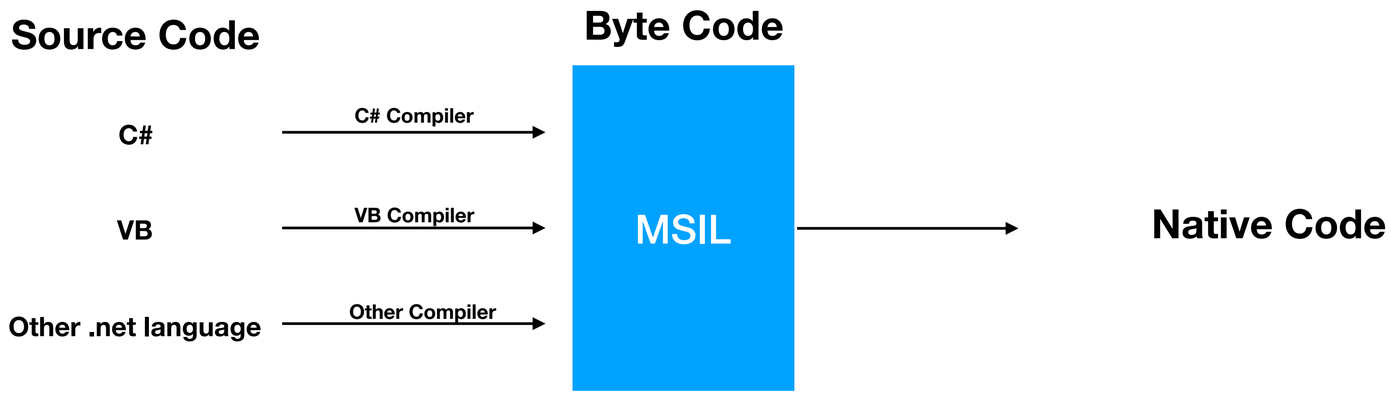
A process known as **just-in-time compilation,** converts compiled code into machine instructions which the computer's CPU then executes

MSIL

CLR

CLS

CTS



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

Common Language Specification

MSIL

CLR

CLS

CTS

CLS specifies a set of rules that needs to be satisfied by all language compilers.

The languages which follows these set of rules are said to be CLS Compliant.

CLS represents the guidelines to the compiler of a language, which targets the .NET Framework. CLS-compliant code is the code exposed and expressed in CLS form.

Even though various .NET languages differ in their syntactic rules, their compilers generate the Common Intermediate Language instructions, which are executed by CLR.

Thus, CLS acts as a tool for integrating different languages into one umbrella in a seamless manner.

Common Type System

MSIL

CLR

CLS

CTS

CTS defines the CLR supported types of data and the operations over them

Ensures data level compatibility between different .NET languages

E.g. string in C# is the same like String in VB.NET and in J#

Value types and Reference types

The CTS also defines the rules that ensures that the data types of objects written in various languages are able to interact with each other.

The CTS also specifies the rules for type visibility and access to the members of a type, i.e. the CTS establishes the rules by which assemblies form scope for a type, and the Common Language Runtime enforces the visibility rules.

The CTS defines the rules governing type inheritance, virtual methods and object lifetime.

All types derive from System.Object

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

Namespace

In Object Oriented world, many times it is possible that programmers will use the same class name, Qualifying NameSpace with class name can avoid this collision.

NameSpace is the Logical group of types or we can say namespace is a container.

Example:

Namespace	Description
System	Provides the Classes for commonly used data types, events and exception
System.Collections	List, Queue, Hash Table, Arrays
System.Data	For different data sources
System.IO	Data access with File

Assemblies

Compiled Library

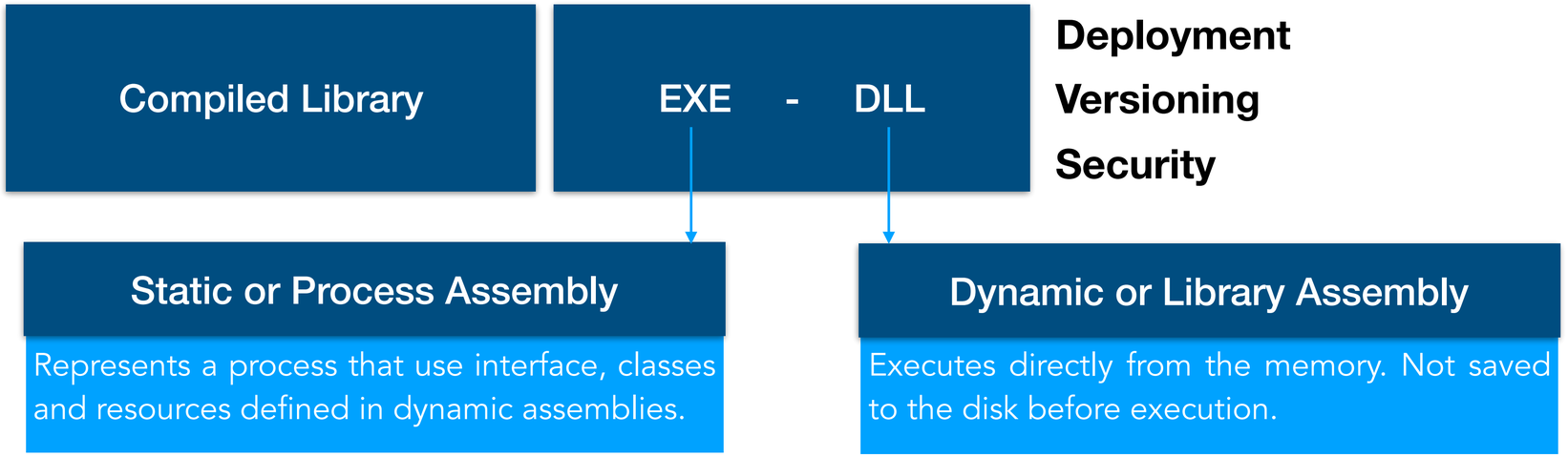
EXE - DLL

Deployment

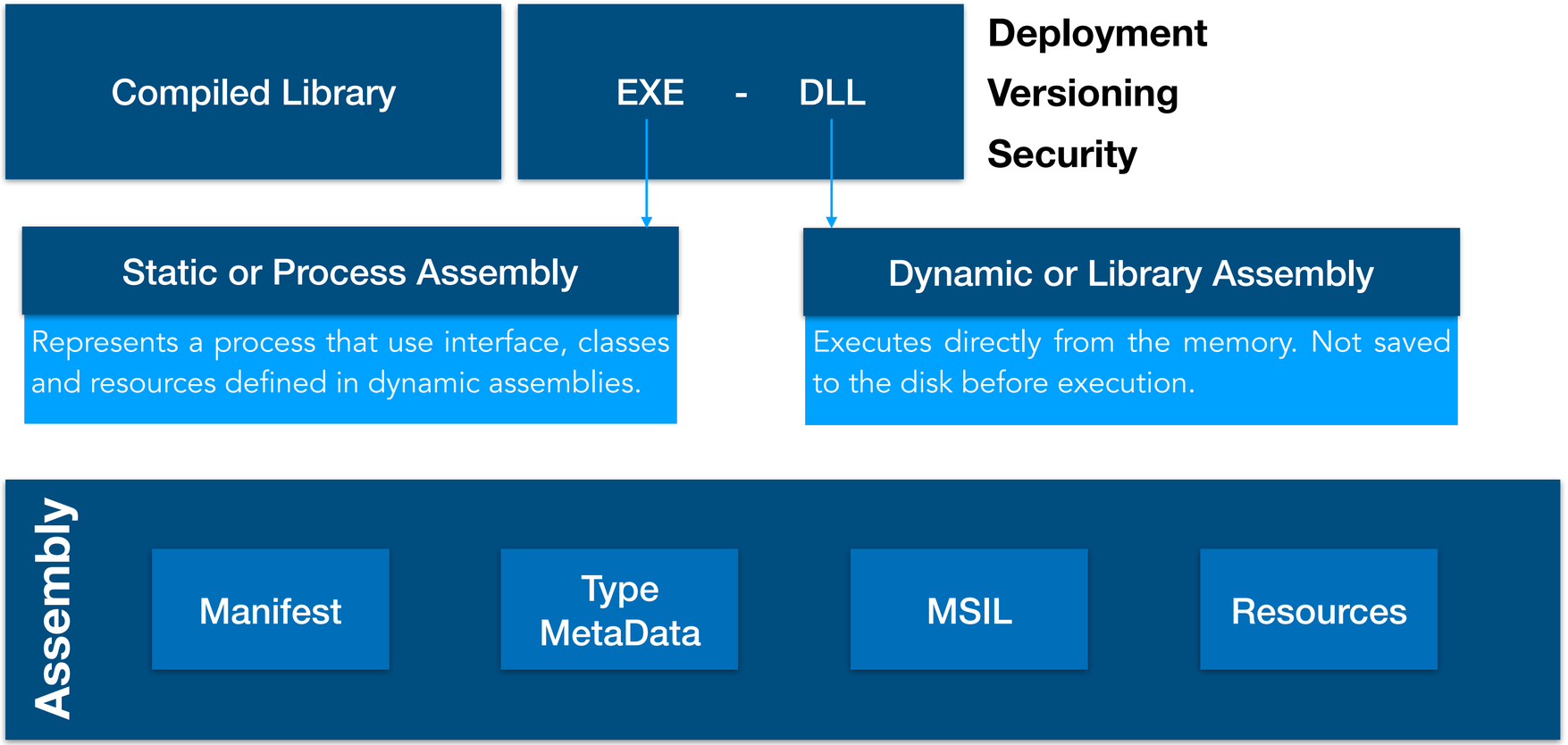
Versioning

Security

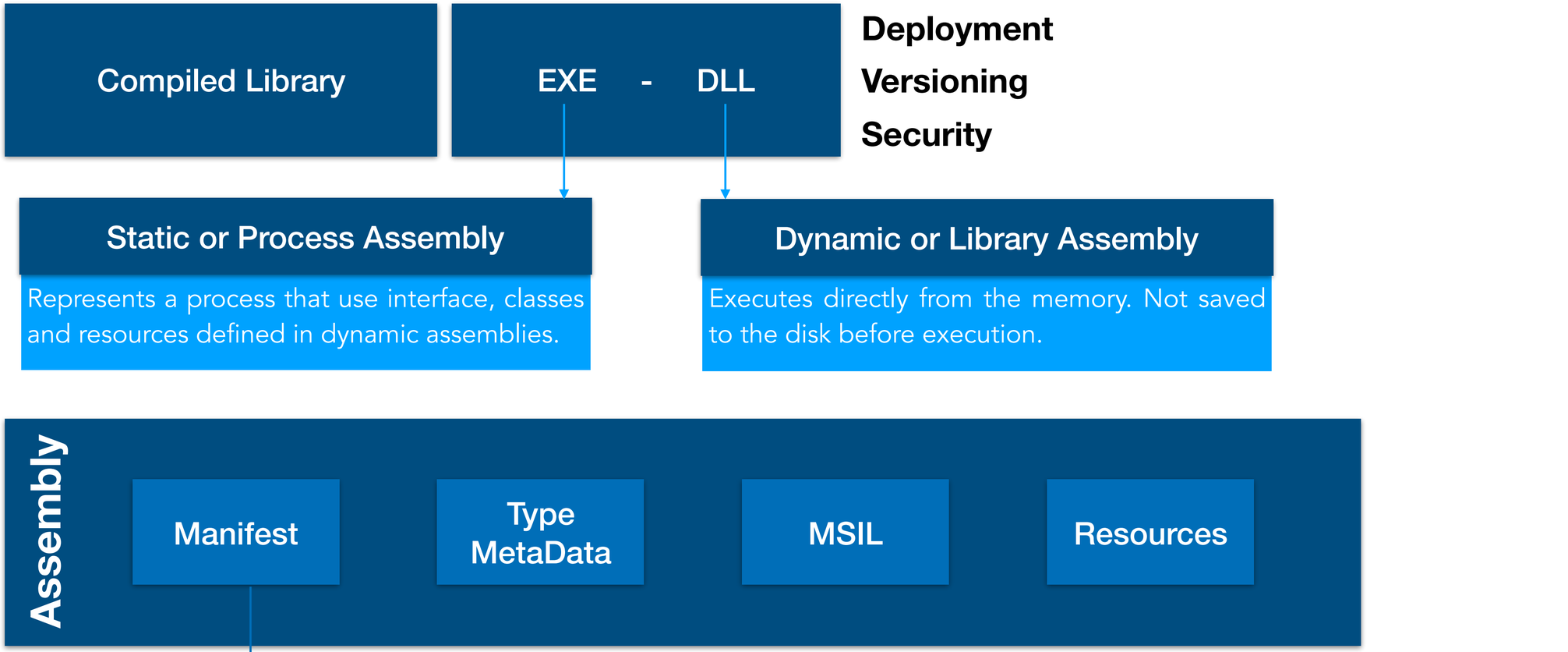
Assemblies


Compiled Library
EXE - DLL
Deployment
Versioning
Security
Static or Process Assembly
Represents a process that use interface, classes and resources defined in dynamic assemblies.
Dynamic or Library Assembly
Executes directly from the memory. Not saved to the disk before execution.

Assemblies


Compiled Library
EXE - DLL
Deployment
Versioning
Security
Static or Process Assembly
Represents a process that use interface, classes and resources defined in dynamic assemblies.
Dynamic or Library Assembly
Executes directly from the memory. Not saved to the disk before execution.
Assembly
Manifest
Type MetaData
MSIL
Resources

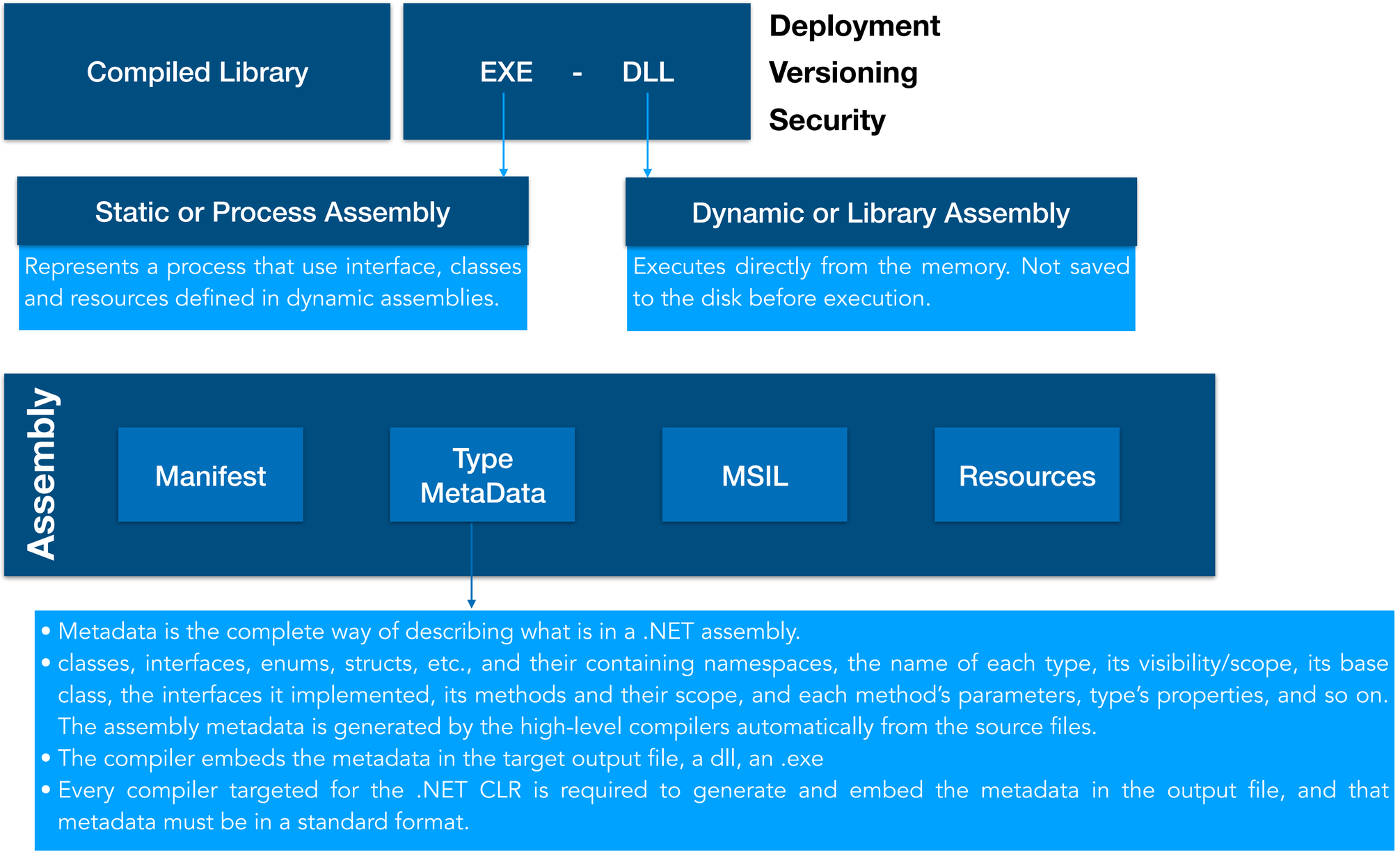
Assemblies



- Manifest maintains the information about the assemblies like assembly name, version number, list of files in the assembly
- This manifest information is used by the CLR.
- The manifest also contains the security demands to verify this assembly. It also contains the names and hashes of all the files that make up the assembly. The .NET assembly manifest contains a cryptographic hash of different modules in the assembly. And when the assembly is loaded, the CLR recalculates the hash of the modules at hand, and compares it with the embedded hash. If the hash generated at runtime is different from that found in the manifest, .NET refuses to load the assembly and throws an exception.

https://stackoverflow.com/questions/9407346/difference-between-metadata-and-manifest

Assemblies



https://stackoverflow.com/questions/9407346/difference-between-metadata-and-manifest

Assemblies

Compiled Library

EXE - DLL

Deployment

Versioning

Security

Static or Process Assembly

Represents a process that use interface, classes and resources defined in dynamic assemblies.

Dynamic or Library Assembly

Executes directly from the memory. Not saved to the disk before execution.

Assembly

- Manifest
- Type MetaData
- MSIL
- Resources

```
.method public hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       44 (0x2c)
  .maxstack  2
  .locals init (int16 V_0)
  IL_0000:  ldc.i4     0x7fff
  IL_0005:  stloc.0
  IL_0006:  ldstr      "sum = {0}"
  IL_000b:  ldloc.0
  IL_000c:  box        [mscorlib]System.Int16
  IL_0011:  call       void [mscorlib]System.Console::WriteLine(string,
                                                                object)
  IL_0016:  ldloc.0
  IL_0017:  ldc.i4.1
  IL_0018:  add
  IL_0019:  conv.i2
  IL_001a:  stloc.0
  IL_001b:  ldstr      "sum = {0}"
  IL_0020:  ldloc.0
  IL_0021:  box        [mscorlib]System.Int16
  IL_0026:  call       void [mscorlib]System.Console::WriteLine(string,
                                                                object)
  IL_002b:  ret
} // end of method Conversion::Main
```

Bitmap
Jpeg
...

Assemblies

Private Assembly	Public(shared) Assembly
Private assembly can be used by only one application.	Public assembly can be used by multiple applications.
Private assembly will be stored in the specific application's directory or sub-directory.	Public assembly is stored in GAC (Global Assembly Cache).
Strong name is not required for private assembly.	Strong name has to be created for public assembly.
Private assembly doesn't have any version constraint.	Public assembly should strictly enforce version constraint.

Garbage Collection

In the common language runtime (CLR), the garbage collector serves as an automatic memory manager.

It provides the following benefits:

- Enables you to develop your application without having to free memory.
- Allocates objects on the managed heap efficiently.
- Reclaims objects that are no longer being used, clears their memory, and keeps the memory available for future allocations. Managed objects automatically get clean content to start with, so their constructors do not have to initialize every data field.
- Provides memory safety by making sure that an object cannot use the content of another object.

https://deepak4blogger.blogspot.com/2017/09/generations-improving-performance.html

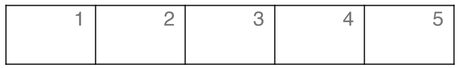
Garbage Collection

The CLR’s Garbage collector (GC) is a generational garbage collector

It has three generations: 0, 1 and 2

Generation 0 :

It contains all newly constructed object which are never examined by GC.



https://deepak4blogger.blogspot.com/2017/09/generations-improving-performance.html

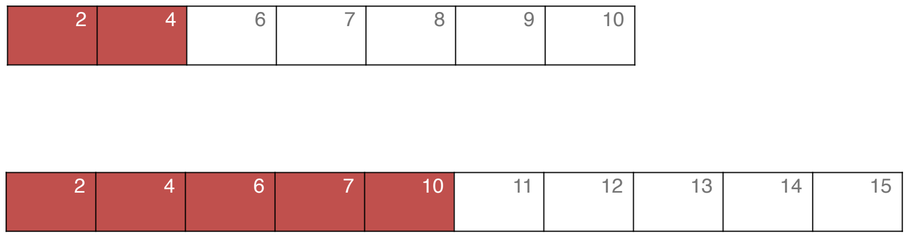
Garbage Collection

The CLR’s Garbage collector (GC) is a generational garbage collector

It has three generations: 0, 1 and 2

Generation 1 :

After creation of object 6, garbage allocation gets started which deallocates the garbage objects 1, 3 and 5 and moves 2 and 4 adjacent to each other in Generation 1.
The budget size of generation 1 is also selected by CLR upon initialization. Creation of object 11 causes the GC to start again which may move some more objects to generation 1.
Generation 1 is ignored for Garbage Collection until it reaches it’s budget size for Garbage collection



https://deepak4blogger.blogspot.com/2017/09/generations-improving-performance.html

Garbage Collection

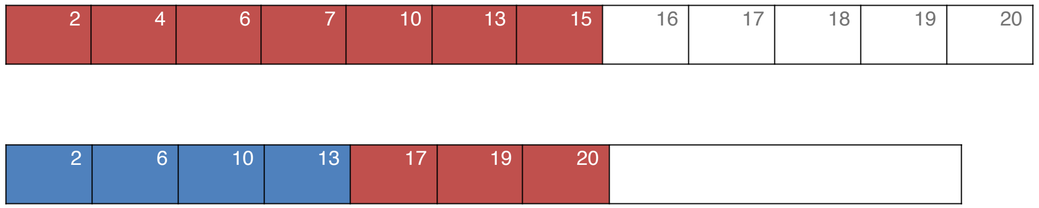
The CLR's Garbage collector (GC) is a generational garbage collector

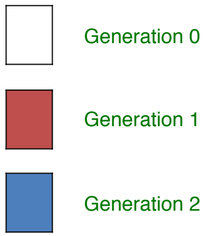
It has three generations: 0, 1 and 2

Generation 2 :

Over the several generation 0 collection, generation 1 may surpass it's budget limit which cause GC to collect the Garbage from both generations. In this case, generation 1 survivors are promoted to generation 2, generation 0 survivors are promoted to generation 1, and generation 0 is empty. Let's say allocation object 21 cause Garbage collection and generation 1 budget have been reached.

So heap would look like below with the object that survived in Generation 1 promoted to generation 2.



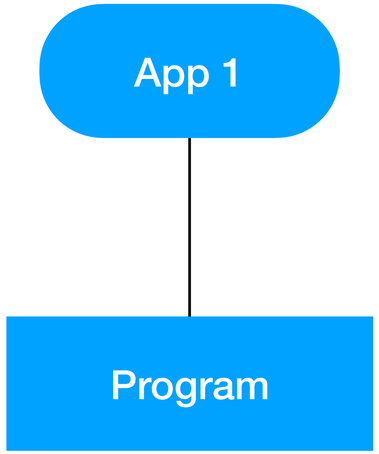


https://deepak4blogger.blogspot.com/2017/09/generations-improving-performance.html

End to DLL hell

Problem:

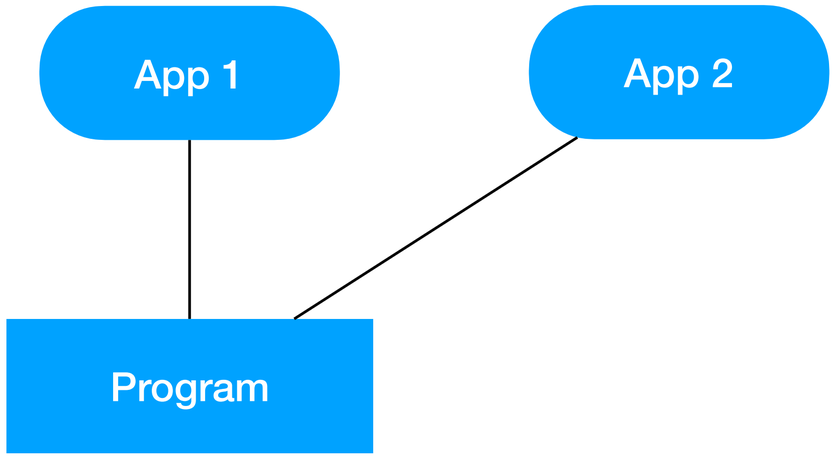
Dll Hell refers to a set of problems caused when multiple applications attempt to share a common component like a dynamic link library (DLL).



End to DLL hell

Problem:

Dll Hell refers to a set of problems caused when multiple applications attempt to share a common component like a dynamic link library (DLL).



End to DLL hell

Problem:

Dll Hell refers to a set of problems caused when multiple applications attempt to share a common component like a dynamic link library (DLL).

Solution:

Versioning

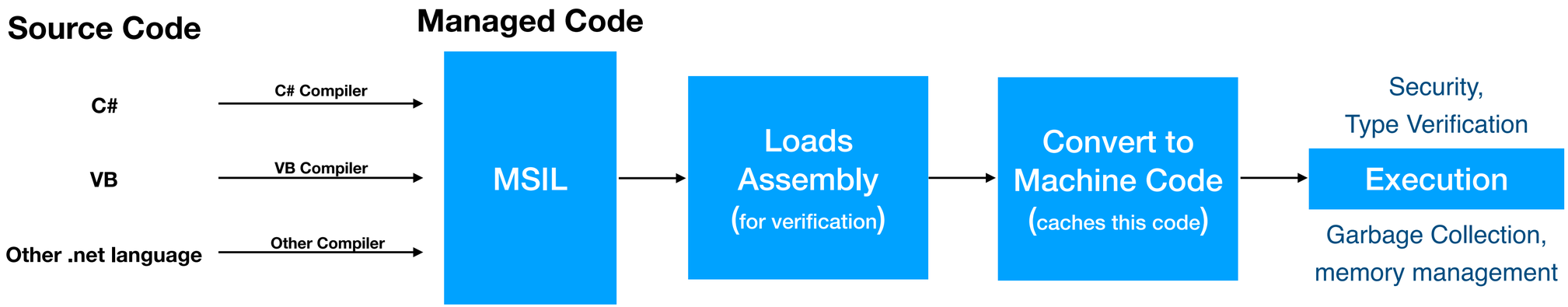
Managed Code

Managed Code is what Visual Basic .NET and C# compilers create.

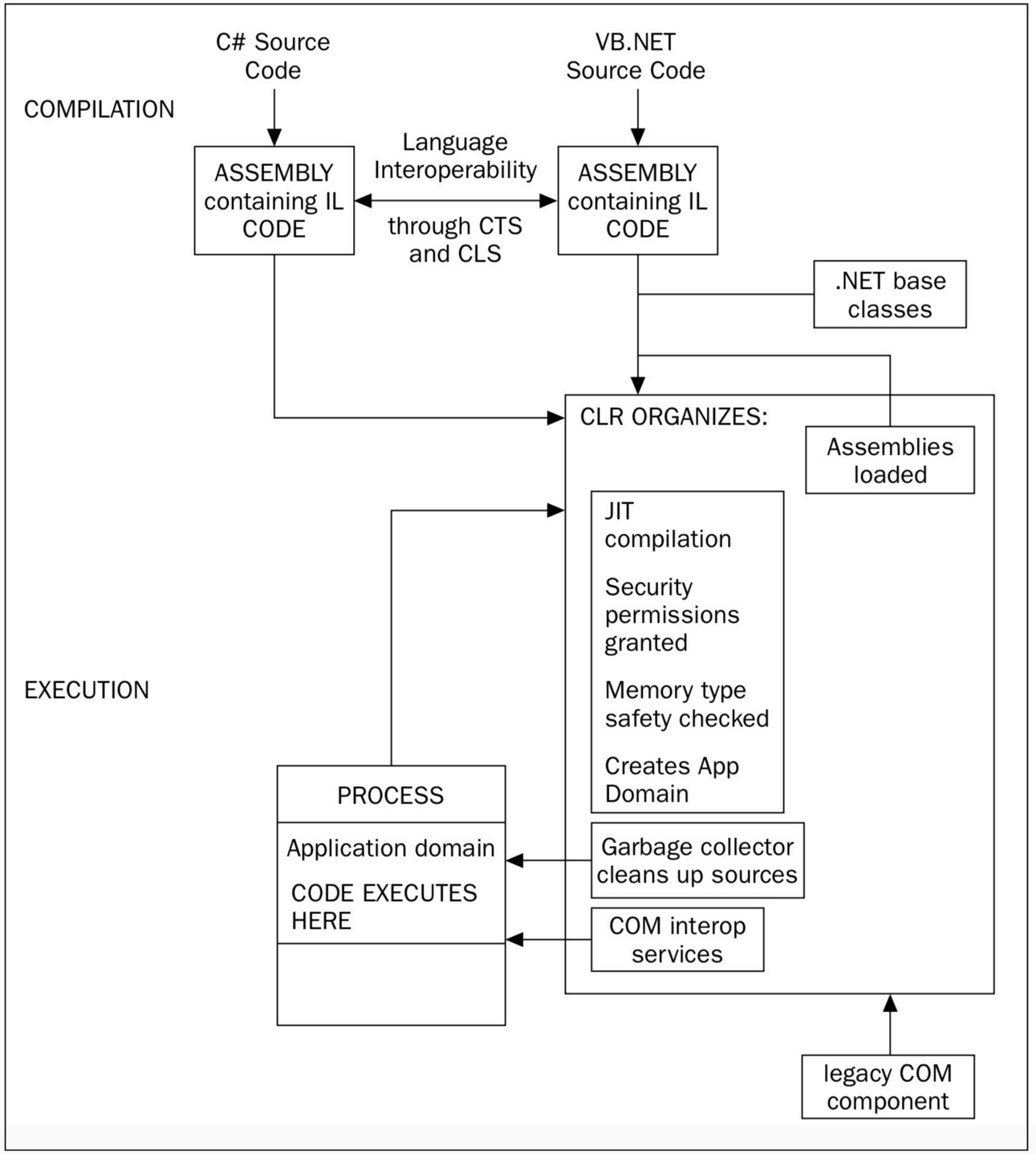
It compiles to Intermediate Language (IL).

The IL is kept in a file called an assembly, along with metadata that describes the classes, methods, and attributes (such as security requirements) of the code you've created.

Managed code runs in the Common Language Runtime. The runtime offers a wide variety of services to your running code.



Managed Code



Unmanaged Code

Unmanaged code is what you use to make before Visual Studio .NET 2002 was released.

It compiled directly to machine code that ran on the machine where you compiled it—and on other machines as long as they had the same chip, or nearly the same.

It didn't get services such as security or memory management from an invisible runtime; it got them from the operating system.

And importantly, it got them from the operating system explicitly, by asking for them, usually by calling an API provided in the Windows SDK. More recent unmanaged applications got operating system services through COM calls.

In unmanaged code a programmer is responsible for:

- Calling the memory allocation function
- Making sure that the casting is done right
- Making sure that the memory is released when the work is done

Unsafe Code

The core C# language differs notably from C and C++ in its omission of pointers as a data type.

Instead, C# provides references and the ability to create objects that are managed by a garbage collector.

This design, coupled with other features, makes C# a much safer language than C or C++.

In the core C# language it is simply not possible to have an uninitialized variable, a "dangling" pointer, or an expression that indexes an array beyond its bounds.

Whole categories of bugs that routinely plague C and C++ programs are thus eliminated.

In unsafe code it is possible to declare and operate on pointers, to perform conversions between pointers and integral types, to take the address of variables, and so forth. In a sense, writing unsafe code is much like writing C code within a C# program.

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/unsafe-code