NET Introduction

Modern .NET/C# Course

.NET Academy

Content

- .NET Framework Overview.
- .NET Standart/.NET Framework/.NET Core/.NET
- Applications that can be developed using .NET
- MSIL.
- The Common Language Runtime (CLR).
- Managed and Unmanaged Code.
- Common Type System (CTS).
- Common Language System (CLS).
- Compilation.
  - JIT.
  - OSR (On Stack Replacement).
  - AOT.
  - PGO.
  - SIMD.

.NET Framework Overview.

.NET Framework is a platform developed by Microsoft for building Windows application. .NET not a language, it is framework, that can be used in languages like C#, VB, C++, F#. This platform was developed in 2002. And since the development of this platform, many flavors of .NET have been developed. And now this platform is cross-platform, which allows it to run on all platforms, Linux, Windows, macOS etc.

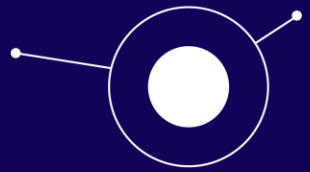
.NET Academy

What is .NET Standard?

1. .NET Standard – .NET Standard is a formal specification of .NET APIs that are available on multiple .NET implementations.
2. .NET Framework – platform for developing Windows applications.
3. .NET Core – Cross-platform framework and a new version of .NET Framework.
4. .NET – Still cross-platform framework, but newer than .NET Core.

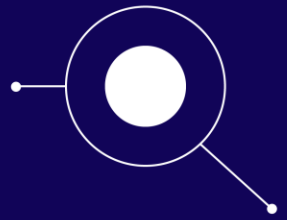
Applications that can be developed using .NET

.NET is huge and powerful, we can develop console applications (tools), desktop applications, mobile applications using Xamarin, powerful web-services and games.
So, that is a time to talk about how does .NET works, and how applications execute. The first thing that we need to discover is MSIL.



.NET Academy

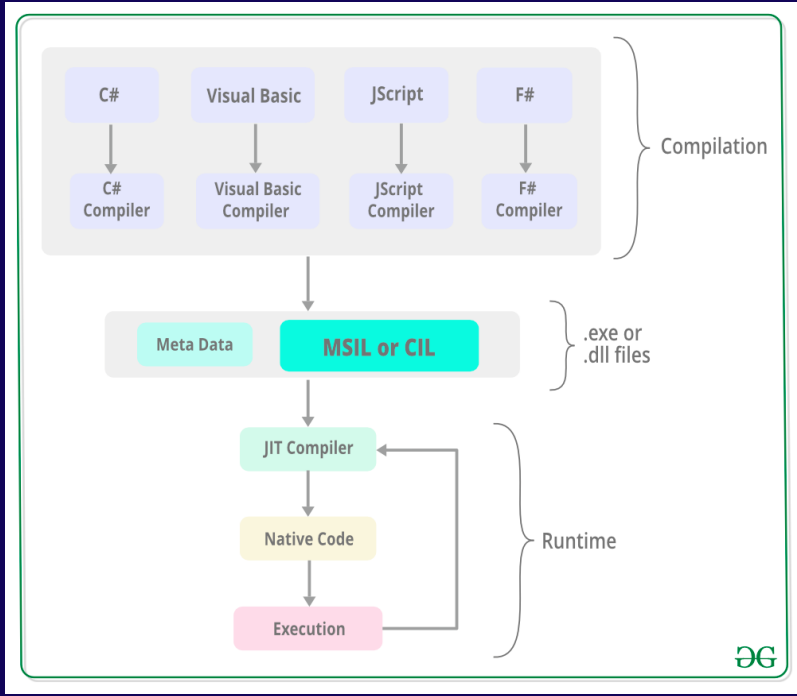
MSIL



MSIL (The Microsoft Intermediate Language) or CIL (Common Intermediate Language) are the same thing, is a set of instructions that are platform independent and are generated by the language-specific compiler from the source code. The MSIL is platform independent and consequently, it can be executed on any of the Common Language Infrastructure supported environments such as the Windows .NET runtime. The MSIL is converted into a particular computer environment specific machine code by the JIT compiler. This is done before the MSIL can be executed. Also, the MSIL is converted into the machine code on a requirement basis, i.e., the JIT compiler compiles the MSIL as required rather than the whole of it.

MSIL

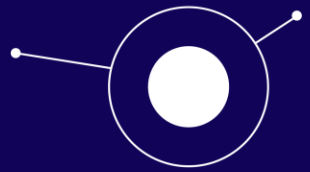
We talked about MSIL and its role in execution .NET applications, but what actually executes .NET applications? The answer is CLR (Common Language Runtime).



.NET Academy

The Common Language Runtime (CLR).

CLR is the main and Virtual Machine component of the .NET. It is the run-time environment for the .NET that runs the codes. Basically, it is responsible for managing the execution of .NET programs regardless of any .NET programming language. Internally, CLR implements the VES (Virtual Execution System) which is defined in the Microsoft's implementation of the CLI (Common Language Infrastructure).



.NET Academy

Managed and Unmanaged Code.

The code that runs under the Common Language Runtime is termed as the Managed Code. In other words, you can say that CLR provides a managed execution environment for the .NET programs by improving the security, including the cross language integration and a rich set of class libraries, etc. CLR is present in every .NET framework version. Below table illustrate the CLR version in .NET framework.

Managed and Unmanaged Code.

As we say before, managed code is MSIL code managed by CLR, and we also have Unmanaged Code. Unmanaged code is a COM Components & Win32 API do not generate the MSIL code, it means that is not managed by CLR, it managed by Operating System.

And talking about main aspects of .NET, we also need to talk about CLS (Common Language Specification) and CTS (Common Type Systems).

.NET Academy

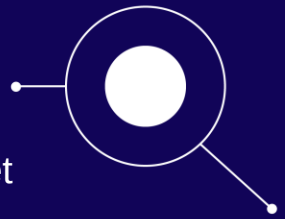
Common Language System (CLS).

Every programming language has its own data type system, so CTS is responsible for understanding all the data type systems of .NET programming languages and converting them into MSIL, and CLR understandable format which will be a common format. There are 2 Types of CTS that every .NET programming language have:

Common Language System (CLS).

1. Value Types will store the value directly into the memory location. These types work with stack mechanisms only. CLR allows memory for these at Compile Time.
2. Reference Types will contain a memory address of the value because the reference types won’t store the variable value directly in memory. These types work with Heap mechanism. CLR allot memory for these at Runtime.

Common Type System (CTS).



What about CTS? The Common Language Specification (CLS) is a fundamental set of language features supported by the Common Language Runtime (CLR) of the .NET Framework. CLS is a part of the specifications of the .NET Framework. CLS 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. It includes features common to many object-oriented programming languages. It forms a subset of the functionality of common type system (CTS) and has more rules than defined in CTS. CLS defines the base rules necessary for any language targeting common language infrastructure to interoperate with other CLS-compliant languages.

Just-In-Time (JIT) Compilation:

Code generation permeates every single line of code we write, and it's critical to the end-to-end performance of applications that the compiler doing that code generation achieves high code quality. In .NET, that's the job of the Just-In-Time (JIT) compiler, which is used both "just in time" as an application executes as well as in Ahead-Of-Time (AOT) scenarios as the workhorse to perform the code-gen at build-time. Every release of .NET has seen significant improvements in the JIT, and .NET 8 is no exception. In fact, I dare say the improvements in .NET 8 in the JIT are an incredible leap beyond what was achieved in the past, in large part due to dynamic PGO…

Profile guided optimization (PGO).

Generally, compilers have to assume that all possible behaviors that could happen at runtime will happen at runtime. But most of the time, the runtime behavior of programs only covers a small fraction of what could happen. So, PGO helps the compiler prioritize optimizations based on what is likely to happen.

PGO works by analyzing the program's behavior and optimizing it based on that information to prioritize behavior found in the past. This can make programs start up faster, run quicker, and have more predictable latency. The PGO data is used by several components of the system to achieve these goals.

Static PGO systems are particularly strong for use in situations where the behavior of an application is both testable outside of production, and in scenarios where startup performance is an important concern. Dynamic PGO systems have the potential for the best throughput in performance, but tend to have performance problems and unpredictable behavior during the startup phases of applications. It is possible to combine the two approaches, and that is what we seek to do in .NET 6. We will provide a static PGO system which can reduce the startup time spent jitting, and draw out some of the throughput benefits of PGO, as well as building a dynamic PGO system that can optionally be enabled for developers seeking the best throughput performance.

.NET Academy

Profile guided optimization (PGO).

PGO relies on a data pipeline of interesting data about application execution, as well as algorithms which take advantage of that data. Hard problems include:

- Building a data pipeline that moves data from one execution to the next. This may be in-process, in the case of dynamic PGO, and across multiple processes in the case of static PGO. The data collected comes in many subtly different forms, and is used for several purposes, which makes this a need for a general purpose data format. To provide a best-in-class solution, this pipeline needs to be generally useable for both models, and even for both models at once, and being resilient to changes in the application from build to build.

- In addition, there are many complex algorithms that need to exist to utilize this data effectively, as just having data is not useful. These algorithms include everything from effects on the register allocator and basic block layout in the JIT, to evaluation of which methods should be ahead-of-time compiled, to algorithms which adjust where in memory code is placed. These complex algorithms are a work in progress in .NET 6 and will continue to evolve for many years.

Ahead-of-Time (AOT) Compilation:

Publishing your app as Native AOT produces an app that's self-contained and that has been ahead-of-time (AOT) compiled to native code. Native AOT apps have faster startup time and smaller memory footprints. These apps can run on machines that don't have the .NET runtime installed.

The benefit of Native AOT is most significant for workloads with a high number of deployed instances, such as cloud infrastructure and hyper-scale services. Native AOT deployment is currently in preview for ASP.NET Core 8.0.

Ahead-of-Time (AOT) Compilation:

The Native AOT deployment model uses an ahead-of-time compiler to compile IL to native code at the time of publish. Native AOT apps don't use a just-in-time (JIT) compiler when the application runs. Native AOT apps can run in restricted environments where a JIT isn't allowed. Native AOT applications target a specific runtime environment, such as Linux x64 or Windows x64, just like publishing a self-contained app.

Native AOT shipped in .NET 7. It enables .NET programs to be compiled at build time into a self-contained executable or library composed entirely of native code: no JIT is required at execution time to compile anything, and in fact there’s no JIT included with the compiled program. The result is an application that can have a very small on-disk footprint, a small memory footprint, and very fast startup time. In .NET 7, the primary supported workloads were console applications. Now in .NET 8, a lot of work has gone into making ASP.NET applications shine when compiled with Native AOT, as well as driving down overall costs, regardless of app model.

.NET Academy

On Stack Replacement.

On Stack Replacement allows the code executed by currently running methods to be changed in the middle of method execution, while those methods are active "on stack."

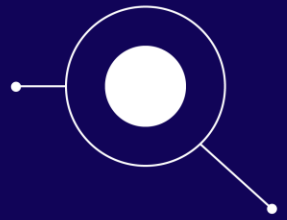
On Stack Replacement (hereafter OSR) refers to a set of techniques for migrating active stack frames from one version of code to another.

The two versions of the code involved in OSR may arise from different program sources (as in Edit and Continue) or different approaches to compiling or executing a single program (say, unoptimized code versus optimized code). The goal of OSR is to transparently redirect execution from an old version of code into a new version, even when in the middle of executing the old version.

Initial work on OSR was pioneered in Self as an approach for debugging optimized code. But in the years since, OSR has mainly seen adoption on platforms like Java and JavaScript that rely heavily on adaptive recompilation of code.

.NET Academy

On Stack Replacement.



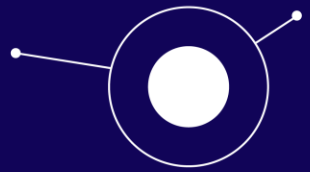
The ability to adaptively recompile and switch code versions while methods are running provides some key advantages:

•Platforms can offer both quick start up and excellent steady-state performance, interpreting or quickly jitting to enable initial method execution, and using OSR to update the methods with better performing or more completely compiled versions as needed.

•Platforms can take advantage of transient program facts and recover when those facts no longer become true. For example, a platform may compile virtual or interface calls as direct calls initially and use OSR to update to more general versions of code when overriding methods or other interface implementations arrive at the scene.

On Stack Replacement.

The CLR already supports various mechanisms for changing the code for a method in a runtime instance. Edit and Continue implements true OSR but is supported only on some architectures, works only when code is running under a managed debugger, and is supported only for unoptimized to unoptimized code. Profiler rejit and tiered compilation can update code used in future invocations of methods, but not code running in currently active methods.



.NET Academy

On Stack Replacement.

OSR is a technology that will allow us to enable tiered compilation by default for almost all methods: performance-critical applications will no longer risk seeing key methods trapped in unoptimized tier0 code, and straightforwardly written microbenchmarks (e.g. all code in main) will perform as expected, as no matter how they are coded, they will be able to transition to optimized code.

On Stack Replacement.

OSR also provides key building blocks for an eventual implementation of deopt and the ability of our platforms to make strong speculative bets in code generation.

In addition, OSR will also allow us to experiment with so-called deferred compilation, where the jit initially only compiles parts of methods that it believes likely to execute (say, based on heuristics or prior runs). If an uncompiled part of a method is reached at runtime, OSR can trigger recompilation of the missing part or recompilation of the entire method.

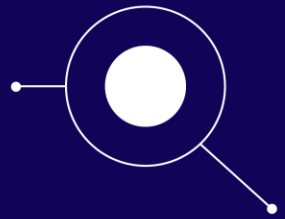
.NET Academy

Single Instruction, Multiple Data (SIMD):

SIMD (Single instruction, multiple data) provides hardware support for performing an operation on multiple pieces of data, in parallel, using a single instruction. In .NET, there's a set of SIMD-accelerated types under the System.Numerics namespace. SIMD operations can be parallelized at the hardware level. That increases the throughput of the vectorized computations, which are common in mathematical, scientific, and graphics apps.

In SIMD processing, data is divided into smaller elements, often called vectors or lanes. These vectors contain multiple data items that can be processed in parallel. The SIMD processor executes a single instruction on all the data elements in a vector simultaneously, performing the same operation on each element concurrently.

Single Instruction, Multiple Data (SIMD):



SIMD instructions are typically supported by specialized hardware or instruction sets found in modern CPUs. These instructions are designed to perform arithmetic, logical, and other operations on vectors efficiently. SIMD instructions are commonly used in multimedia applications, scientific simulations, image and signal processing, and other computationally intensive tasks.

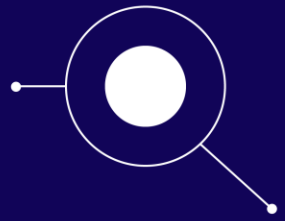
SIMD can be used in .NET through the System.Numerics and System.Runtime.Intrinsics namespaces. In .NET Core 1.0 and later versions, you can use the System.Numerics.Vector<T> class. This class provides SIMD support for a wide range of data types, including integers and floating-point numbers.

You can perform SIMD operations using Vector<T> to efficiently process large sets of data in parallel. For example, you can create Vector<T> instances, perform arithmetic or logical operations on them, and access the individual elements of the vector using familiar array-like syntax. Starting from .NET Core 3.0 and later versions, the System.Runtime.Intrinsics namespace provides access to lower-level SIMD capabilities.

Single Instruction, Multiple Data (SIMD):

The Vector128 and Vector256 structures in this namespace represent SIMD vector types for specific hardware instruction sets, such as SSE (Streaming SIMD Extensions) or AVX (Advanced Vector Extensions). These types allow you to perform more fine-grained control over SIMD operations and take advantage of the full capabilities of the underlying hardware.

Questions for the study of the topic



1. Describe the major components of the .NET Framework and their roles.
2. Explain the difference between .NET Framework, .NET Core, and .NET. What factors influence the choice of platform for application development?
3. What is .NET Standard and what benefits does it provide for library portability between different versions of .NET?
4. What is MSIL (CIL) and how does it relate to .NET applications?
5. What is the Common Language Runtime (CLR) in .NET and what tasks does it perform?
6. What is Common Type System (CTS) in the context of .NET and what are its purposes?
7. What process happens during JIT compilation in .NET and why is it important for .NET code execution?
8. What advantages and disadvantages of JIT compilation over AOT compilation do you see?

Homework

1. What is being written in .NET Core?
2. What is MSIL and what tasks does it perform in .NET?
3. What programming languages are supported in the .NET Framework? Which language would you prefer to use and why?
4. What role does MSIL have in ensuring code portability between different platforms?
5. How does the CLR provide type safety and memory management in .NET applications?
6. What are the basic principles of the Common Language Specification (CLS) and what is it used for?
7. How do CTS and CLS promote interoperability between code written in different .NET languages?
8. What is Profile-Guided Optimisation (PGO) and how can it improve the performance of .NET applications?How can SIMD technology be used to optimise computation in .NET applications?

Thank you for your attention.

.NET Academy