.NET is a free, cross-platform, [open-source developer platform](https://github.com/dotnet/core) for building [many kinds of applications](https://learn.microsoft.com/en-us/dotnet/core/apps). It can run programs written in [multiple languages](https://learn.microsoft.com/en-us/dotnet/fundamentals/languages), with [C#](https://learn.microsoft.com/en-us/dotnet/csharp/) being the most popular. It relies on a [high-performance](https://devblogs.microsoft.com/dotnet/category/performance/) runtime that is used in production by many [high-scale apps](https://devblogs.microsoft.com/dotnet/category/developer-stories/).

The .NET platform has been designed to deliver productivity, performance, security, and reliability. It provides automatic memory management via a [garbage collector (GC)](https://learn.microsoft.com/en-us/dotnet/standard/automatic-memory-management). It is type-safe and memory-safe, due to using a GC and strict language compilers. It offers [concurrency](https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/) via async/await and Task primitives. It includes a large set of libraries that have broad functionality and have been optimized for performance on multiple operating systems and chip architectures.

**Components**

.NET includes the following components:

* Runtime -- executes application code.
* Libraries -- provides utility functionality like [JSON parsing](https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/overview).
* Compiler -- compiles C# (and other languages) source code into (runtime) executable code.
* SDK and other tools -- enable building and monitoring apps with modern workflows.
* App stacks -- like ASP.NET Core and Windows Forms, that enable writing apps.

**NET ecosystem**

There are multiple variants of .NET, each supporting a different type of app. The reason for multiple variants is part historical, part technical.

.NET implementations:

* **.NET Framework** -- The original .NET. It provides access to the broad capabilities of Windows and Windows Server. It is actively supported, in maintenance.
* **Mono** -- The original community and open source .NET. A cross-platform implementation of .NET Framework. Actively supported for Android, iOS, and WebAssembly.
* **.NET (Core)** -- Modern .NET. A cross-platform and open source implementation of .NET, rethought for the cloud age while remaining significantly compatible with .NET Framework. Actively supported for Linux, macOS, and Windows.

.NET 9 is the latest version of this .NET implementation.

**.NET Framework**

.NET Framework is the original .NET implementation that has existed since 2002. Versions 4.5 and later implement .NET Standard, so code that targets .NET Standard can run on those versions of .NET Framework. It contains additional Windows-specific APIs, such as APIs for Windows desktop development with Windows Forms and WPF. .NET Framework is optimized for building Windows desktop applications.

**.NET class libraries**

Class libraries are the [shared library](https://en.wikipedia.org/wiki/Library_%28computing%29#Shared_libraries) concept for .NET. They enable you to componentize useful functionality into modules that can be used by multiple applications. They can also be used as a means of loading functionality that is not needed or not known at application startup. Class libraries are described using the [.NET Assembly file format](https://learn.microsoft.com/en-us/dotnet/standard/assembly/file-format).

There are three types of class libraries that you can use:

* **Platform-specific** class libraries have access to all the APIs in a given platform (for example, .NET Framework on Windows), but can only be used by apps and libraries that target that platform.
* **Portable** class libraries have access to a subset of APIs, and can be used by apps and libraries that target multiple platforms.
* **.NET Standard** class libraries are a merger of the platform-specific and portable library concept into a single model that provides the best of both.

**.NET Standard**

.NET Standard is a formal specification of .NET APIs that are available on multiple .NET implementations. The motivation behind .NET Standard was to establish greater uniformity in the .NET ecosystem. .NET 5 and later versions adopt a different approach to establishing uniformity that eliminates the need for .NET Standard in most scenarios. However, if you want to share code between .NET Framework and any other .NET implementation, such as .NET Core, your library should target .NET Standard 2.0. [No new versions of .NET Standard will be released](https://devblogs.microsoft.com/dotnet/the-future-of-net-standard/), but .NET 5 and all later versions will continue to support .NET Standard 2.1 and earlier.

**CLI**

This is a specification that defines a platform for building, deploying, and running applications written in different programming languages. It’s part of the Microsoft .NET Framework and provides a runtime environment that enables interoperability among various languages.

The **CLI** includes key components like:

1. **Common Language Runtime (CLR)**: This is the execution engine of the CLI that manages memory, thread execution, garbage collection, and more. It’s similar to the Java Virtual Machine (JVM) in the Java ecosystem.
2. **Metadata and Common Type System (CTS)**: These help define how data types are represented and used across different languages, ensuring that objects created in one language can be used in another.
3. **Assemblies**: These are compiled code libraries, which contain metadata and code that the CLR can execute.
4. **Common Intermediate Language (CIL)**: When you write code in languages like C# or VB.NET, it is first compiled into CIL, which is a low-level, platform-neutral representation. The CLR then compiles this intermediate code into machine code at runtime.

**AOT**

Ahead-of-time compiler. AOT compilation happens before the application is executed and is usually performed on a different machine. Because AOT tool chains don't compile at run time, they don't have to minimize time spent compiling. That means they can spend more time optimizing. Since the context of AOT is the entire application, the AOT compiler also performs cross-module linking and whole-program analysis, which means that all references are followed and a single executable is produced.

**BCL**

Base Class Library.

A set of libraries that comprise the System.\* (and to a limited extent Microsoft.\*) namespaces. The BCL is a general purpose, lower-level framework that higher-level application frameworks, such as ASP.NET Core, build on.

**CLR**

Common Language Runtime.

The exact meaning depends on the context. Common Language Runtime usually refers to the runtime of [.NET Framework](https://learn.microsoft.com/en-us/dotnet/standard/glossary#net-framework) or the runtime of [.NET](https://learn.microsoft.com/en-us/dotnet/standard/glossary#net).

A CLR handles memory allocation and management. A CLR is also a virtual machine that not only executes apps but also generates and compiles code on-the-fly using a [JIT](https://learn.microsoft.com/en-us/dotnet/standard/glossary#jit) compiler.

The CLR implementation for .NET Framework is Windows only.

The CLR implementation for .NET (also known as the Core CLR) is built from the same code base as the .NET Framework CLR. Originally, the Core CLR was the runtime of Silverlight and was designed to run on multiple platforms, specifically Windows and OS X. It's still a [cross-platform](https://learn.microsoft.com/en-us/dotnet/standard/glossary#cross-platform) runtime, now including support for many Linux distributions.

**CoreRT**

In contrast to the [CLR](https://learn.microsoft.com/en-us/dotnet/standard/glossary#clr), CoreRT is not a virtual machine, which means it doesn't include the facilities to generate and run code on-the-fly because it doesn't include a [JIT](https://learn.microsoft.com/en-us/dotnet/standard/glossary#jit). It does, however, include the [GC](https://learn.microsoft.com/en-us/dotnet/standard/glossary#gc) and the ability for run-time type identification (RTTI) and reflection. However, its type system is designed so that metadata for reflection isn't required. Not requiring metadata enables having an [AOT](https://learn.microsoft.com/en-us/dotnet/standard/glossary#aot) tool chain that can link away superfluous metadata and (more importantly) identify code that the app doesn't use. CoreRT is in development.

**JIT**

Just-in-time compiler.

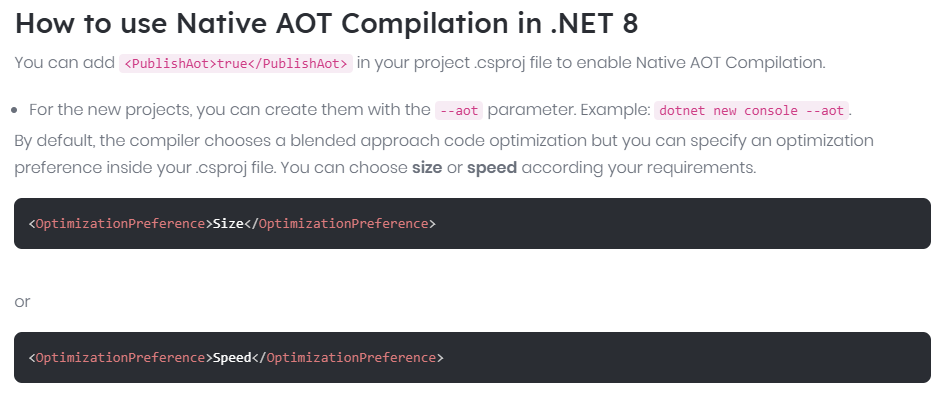
Similar to [AOT](https://learn.microsoft.com/en-us/dotnet/standard/glossary#aot), this compiler translates [IL](https://learn.microsoft.com/en-us/dotnet/standard/glossary#il) to machine code that the processor understands. Unlike AOT, JIT compilation happens on demand and is performed on the same machine that the code needs to run on. Since JIT compilation occurs during execution of the application, the compile time is part of the run time. Thus, JIT compilers have to balance time spent optimizing code against the savings that the resulting code can produce. But a JIT knows the actual hardware and can free developers from having to ship different implementations.

**Mono**

An open source, [cross-platform](https://learn.microsoft.com/en-us/dotnet/standard/glossary#cross-platform) [.NET implementation](https://learn.microsoft.com/en-us/dotnet/standard/glossary#implementation-of-net) that's used when a small runtime is required. It's the runtime that powered Xamarin applications (unsupported as of May 2024) on Android, Mac, iOS, tvOS, and watchOS. Mono focuses primarily on apps that require a small footprint.

**Native AOT**

A deployment mode where the app is self-contained and is [ahead-of-time](https://learn.microsoft.com/en-us/dotnet/standard/glossary#aot) compiled to native code at the time of publish. Native AOT apps don't use a [JIT](https://learn.microsoft.com/en-us/dotnet/standard/glossary#jit) compiler at run time. They can run on machines that don't have the .NET runtime installed.



**NET**

*.NET* has two meanings, and the one that is intended depends on the context:

* *.NET* can be used as the umbrella term for [.NET Standard](https://learn.microsoft.com/en-us/dotnet/standard/glossary#net-standard) and all [.NET implementations](https://learn.microsoft.com/en-us/dotnet/standard/glossary#implementation-of-net) and workloads.
* *.NET* more frequently refers to the cross-platform, high-performance, open-source implementation of .NET that used to be called .NET Core. It can also be referred to as *.NET 5 (and .NET Core) and later versions* or just *.NET 5+*.

## NET CLI

A cross-platform toolchain for developing applications and libraries for [.NET](https://learn.microsoft.com/en-us/dotnet/standard/glossary#net). Also known as the .NET Core CLI.

**.NET Native**

A compiler tool chain that produces native code ahead-of-time ([AOT](https://learn.microsoft.com/en-us/dotnet/standard/glossary#aot)), as opposed to just-in-time ([JIT](https://learn.microsoft.com/en-us/dotnet/standard/glossary#jit)).

Compilation happens on the developer's machine similar to the way a C++ compiler and linker works. It removes unused code and spends more time optimizing it. It extracts code from libraries and merges them into the executable. The result is a single module that represents the entire app.

UWP is the application framework supported by .NET Native.

**.NET SDK**

A set of libraries and tools that enable developers to create applications and libraries for [.NET](https://learn.microsoft.com/en-us/dotnet/standard/glossary#net). Also known as the *.NET Core SDK*.

Includes the [.NET CLI](https://learn.microsoft.com/en-us/dotnet/standard/glossary#net-cli) for building apps, .NET libraries and runtime for building and running apps, and the dotnet executable (*dotnet.exe*) that runs CLI commands and runs applications.

**.NET Standard**

A formal specification of .NET APIs that are available in each [.NET implementation](https://learn.microsoft.com/en-us/dotnet/standard/glossary#implementation-of-net).

The .NET Standard specification is sometimes called a library. Because a library includes API implementations, not only specifications (interfaces), it's misleading to call .NET Standard a "library."

**NGen**

Native (image) generation.

You can think of this technology as a persistent [JIT](https://learn.microsoft.com/en-us/dotnet/standard/glossary#jit) compiler. It usually compiles code on the machine where the code is executed, but compilation typically occurs at install time.

**runtime**

In general, the execution environment for a managed program. The OS is part of the runtime environment but is not part of the .NET runtime. Here are some examples of .NET runtimes in this sense of the word:

* Common Language Runtime ([CLR](https://learn.microsoft.com/en-us/dotnet/standard/glossary#clr))
* .NET Native (for UWP)
* Mono runtime

## target framework

The collection of APIs that a .NET app or library relies on.

An app or library can target a version of [.NET Standard](https://learn.microsoft.com/en-us/dotnet/standard/glossary#net-standard) (for example, .NET Standard 2.0), which is a specification for a standardized set of APIs across all [.NET implementations](https://learn.microsoft.com/en-us/dotnet/standard/glossary#implementation-of-net). An app or library can also target a version of a specific .NET implementation, in which case it gets access to implementation-specific APIs.

**TFM**

Target framework moniker.

A standardized token format for specifying the [target framework](https://learn.microsoft.com/en-us/dotnet/standard/glossary#target-framework) of a .NET app or library. Target frameworks are typically referenced by a short name, such as net462. Long-form TFMs (such as .NETFramework,Version=4.6.2) exist but aren't generally used to specify a target framework.

**UWP**

Universal Windows Platform.

An [implementation of .NET](https://learn.microsoft.com/en-us/dotnet/standard/glossary#implementation-of-net) that is used for building touch-enabled Windows applications and software for the Internet of Things (IoT). It's designed to unify the different types of devices that you might want to target, including PCs, tablets, phones, and even the Xbox.

The managed execution process includes the following steps, which are discussed in detail later in this topic:

1. Choosing a compiler. To obtain the benefits provided by the common language runtime, you must use one or more language compilers that target the runtime.
2. Compiling your code to intermediate language. Compiling translates your source code into common intermediate language (CIL) and generates the required metadata.
3. Compiling CIL to native code. At execution time, a just-in-time (JIT) compiler translates the CIL into native code. During this compilation, code must pass a verification process that examines the CIL and metadata to find out whether the code can be determined to be type safe.
4. Running code. The common language runtime provides the infrastructure that enables execution to take place and services that can be used during execution.

The common type system 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. The common type system performs the following functions:

* Establishes a framework that helps enable cross-language integration, type safety, and high-performance code execution.
* Provides an object-oriented model that supports the complete implementation of many programming languages.
* Defines rules that languages must follow, which helps ensure that objects written in different languages can interact with each other.
* Provides a library that contains the primitive data types (such as [Boolean](https://learn.microsoft.com/en-us/dotnet/api/system.boolean), [Byte](https://learn.microsoft.com/en-us/dotnet/api/system.byte), [Char](https://learn.microsoft.com/en-us/dotnet/api/system.char), [Int32](https://learn.microsoft.com/en-us/dotnet/api/system.int32), and [UInt64](https://learn.microsoft.com/en-us/dotnet/api/system.uint64)) used in application development.

.NET 8

 **Native AOT (Ahead-of-Time Compilation)**: .NET 8 introduces support for native AOT compilation, which significantly reduces the size of applications and improves startup times. This is useful for scenarios like cloud applications, microservices, and low-resource environments.

 **Improvements to Minimal APIs**: Minimal APIs, introduced in .NET 6, have been further optimized in .NET 8 for better performance and ease of use.

 **Cross-platform support improvements**: Enhanced support for ARM64 and other cross-platform development improvements.

 **Hot Reload Improvements**: Better support for hot reload across more types of projects, making it easier for developers to see changes in real-time without needing to restart the app.

 **Enhanced Blazor Performance**: Significant improvements to Blazor, such as better WebAssembly performance and reduced memory usage.

 **Cloud-Native Features**: Enhanced features to support building cloud-native applications, including better integration with Kubernetes and other cloud technologies.