C#과 .NET

C#, .NET and CLR

2025-01-21

Abstract

"C#은 마이크로소프트가 2000년대 초반에 .NET 플랫폼과 함께 선보인 현대적인 객체 지향 언어로, 자바의 문법과 유사하면서도 강력한 기능 확장을 통해 생산성과 성능을 동시에 추구합니다. 초기에는 Windows 중심의 .NET 프레임워크 상에서만 사용되었으나, 점차 .NET Core와 .NET 5 이상으로 발전하며 크로스 플랫폼 환경에서 동등하게 실행될 수 있도록 확장되었습니다. 자동 메모리 관리(GC), 예외 처리, 강력한 타입 시스템 같은 기본 기능을 제공함으로써 안정적이고 유지보수하기 쉬운 코드를 작성할 수 있으며, 최근 들어서는 레코드, 패턴 매칭, 원시 문자열 리터럴, 정적 추상 멤버처럼 다양한 최신 문법을 도입해 개발자 경험을 지속적으로 개선하고 있습니다."

C#

C#은 Microsoft에서 1999년에 개발한 프로그래밍 언어입니다. 현대적이고 객체 지향적인 언어로, .NET 프레임워크에서 실행되도록 설계되었습니다. C#의 문법은 C++ 및 Java와 유사합니다. C#은 출시된 이후 여러 번의 개정을 거쳐 새로운 기능을 추가했습니다. 오늘날 C#은 Windows 데스크톱 애플리케이션, 웹 애플리케이션, 게임 및 모바일 앱 개발에 활용할 수 있습니다.

C# 10의 주요 변경사항

- 파일 범위 네임스페이스(File-scoped namespace), 기존에는 네임스페이스를 감싸는 중괄호 {} 를 사용해야 했습니다. C# 10에서는 파일 최상단에 한 줄로 선언이 가능합니다. 코드 중첩이 줄어들어 가독성과 유지보수성이 향상됩니다.

```
// after
namespace MyApp
{
    class Program
    {
        static void Main() { }
    }
}

// before
```

```
namespace MyApp;

class Program
{
    static void Main() { }
}
```

- 전역 using 지시문(Global using directives), using 지시문을 전역으로 선언하여, 모든 파일에서 공통으로 사용할 수 있도록 해줍니다. 프로젝트 전역에서 공통 사용되는 네임스페이스를 한 곳에 모아 정리할 수 있어, 코드 중복을 줄입니다.

```
// GlobalUsings.cs (프로젝트 내 한 곳에만 작성)
global using System;
global using System.Collections.Generic;

// 다른 파일 어디서든 System, System.Collections.Generic를 별도의 using 없이 사용 가능
namespace MyApp;

public class Demo
{
    public List<string> Names { get; set; } = new();
}
```

- Record structs, C# 9에서 소개된 record는 기본적으로 참조 타입이었으나, C# 10부터는 값 타입인 구조체 형태도 지원합니다. 구조체의 값 복사 특성과 레코드의 불변성 및 편의 기능을 동시에 활용할 수 있습니다.

```
public record struct Point(int X, int Y);
var p1 = new Point(3, 4);
var p2 = p1 with { Y = 10 };
// 구조체지만 record로 선언했기 때문에 with 식 사용 가능
Console.WriteLine(p2); // Point { X = 3, Y = 10 }
```

- 람다 식 개선(Lambda improvements), 람다 식에서 반환 형식을 컴파일러가 추론할 수 있으며, 람다 식에 속성(Attribute)을 붙일 수 있습니다. 매개변수가 없는 람다 식에서 _(디스카드)를 사용해도 됩니다.

```
Func<int, int> square = [MyAttribute] x => x * x;
// 반환형 추론, Attribute 적용 가능
var print = () => Console.WriteLine("Hello");
print();
```

- 상수 문자열 보간(Constant interpolated strings), 상수(const) 필드에서도 문자열 보간이 가능해졌습니다.

```
const string Greeting = "Hello";
const string Message = $"{Greeting}, World!";
// C# 9까지는 불가, C# 10에서는 가능
```

C# 11의 주요 변경사항

- 원시 문자열 리터럴(Raw string literals), 문자열을 """로 둘러싸, 특수문자나 줄바꿈을 이스케이프 없이 그대로 표현할 수 있습니다. JSON, XML, Markdown 등의 멀티라인 문자열을 가독성 좋게 표현할 수 있습니다.

```
string raw = """
    이것은 "원시" 문자열입니다.
    \n, \t 등을 이스케이프 처리하지 않아도 됩니다.
""";
Console.WriteLine(raw);
```

- UTF-8 문자열 리터럴(UTF-8 string literals), 문자열 리터럴 뒤에 u8 접미사를 붙여 UTF-8 인코딩된 바이트 배열을 직접 생성할 수 있습니다. 별도의 인코딩 과정 없이 UTF-8 바이트 배열을 얻을 수 있어 성능상 이점이 있습니다.

```
ReadOnlySpan<byte> utf8Bytes = "안녕하세요"u8;
Console.WriteLine(utf8Bytes.Length);  // UTF-8로 인코딩된 길이
```

- required, 객체나 레코드가 생성될 때 반드시 초기화해야 하는 필드를 지정할 수 있습니다. 중요한 필드의 누락을 컴파일 시간에 미리 방지할 수 있습니다.

```
public class Person
{
    public required string Name { get; init; }
    public int Age { get; init; }
}
var p = new Person { Name = "Alice", Age = 30 };
// Name은 반드시 설정, Age는 선택
```

- 정적 추상 멤버(Static abstract members in interfaces), 인터페이스에 정적 추상 멤버를 정의할 수 있어, 제네릭 수학 연산 등 다양한 패턴을 지원합니다. +, - 연산자나 수학 관련 함수를 제네릭에서 간단히 이용할 수 있어, 수치 라이브러리 구현이 용이해집니다.

```
public interface IAddition<TSelf> where TSelf : IAddition<TSelf>
{
    static abstract TSelf operator +(TSelf left, TSelf right);
}

public struct MyInt : IAddition<MyInt>
{
    public int Value { get; set; }
    public MyInt(int value) => Value = value;

    public static MyInt operator +(MyInt left, MyInt right)
        => new MyInt(left.Value + right.Value);
}
```

- 리스트 패턴(List patterns), 배열이나 리스트를 패턴 매칭으로 검사할 수 있습니다. 배열의 특정 형태를 깔끔하게 검사할 수 있습니다.

```
int[] numbers = { 1, 2, 3 };
// 완벽 일치
if (numbers is [1, 2, 3]) Console.WriteLine("numbers는 1,2,3으로 구성");
if (numbers is [1, ..]) Console.WriteLine("첫 번째 요소가 1");
```

- file 접근 한정자(File-scoped types), file 한정자를 사용해 해당 파일 내에서만 유효한 클래스(또는 구조체, 열거형 등)를 선언할 수 있습니다. 특정 파일에서만 사용하는 헬퍼 타입 등을 은닉하여, 네임스페이스 오염을 줄일 수 있습니다.

```
file class MyFileScopedClass
{
    // 이 파일 바깥에서는 접근 불가
}
```

C# 12의 주요 변경사항

- 클래스/구조체용 기본 생성자(Primary constructors for non-record types), 레코드가 아닌 일반 클래스나 구조체에서도 생성자 매개변수를 클래스 선언부에 직접 선언할 수 있습니다. 레코드에서 사용하던 간결한 생성자 문법을 일반 클래스/구조체에도 적용 가능해집니다.

```
class Person(string name, int age)
{
    public string Name { get; } = name;
    public int Age { get; } = age;
}
var p = new Person("Alice", 30);
Console.WriteLine(p.Name); // "Alice"
```

- using 별칭 지시문(Using alias directive) 확장, 임의의 타입이나 멤버에 대해 using 별칭을 지정할 수 있도록 확장될 예정입니다. 긴 네임스페이스나 자주 쓰는 멤버를 짧게 별칭화하여 가독성을 높입니다.

```
using MyInt = System.Int32;
using static System.Math;
Console.WriteLine(Sqrt(25)); // using static System.Math 덕분에 바로 Sqrt 사용
```

- 람다 파라미터 기본값(Default values for lambda parameters), 람다 식의 매개변수에 기본값을 지정할 수 있게 됩니다. 메서드와 동일한 방식으로 매개변수의 기본값을 설정할 수 있어, 코드가 더 간결해집니다.

```
Func<int, int, int> add = (x = 0, y = 0) => x + y;
Console.WriteLine(add());      // 0
Console.WriteLine(add(5));     // 5
Console.WriteLine(add(5, 10)); // 15
```

.NET

.NET은 Microsoft에서 개발한 소프트웨어 개발 플랫폼입니다. 2002년 처음 출시된 이후 윈도우 애플리케이션 개발을 위한 강력한 도구로 자리매김했으며, 현재는 웹, 모바일, 게임, IoT, AI 등 다양한 분야에서 활용되고 있습니다. .NET은 개발자가 다양한 프로그래밍 언어를 사용하여 애플리케이션을 구축할 수 있도록 지원하며, 풍부한 라이브러리와 도구를 제공하여 개발 생산성을 향상시킵니다. 은 아래와 같습니다.

.NET의 주요 특징

- 다양한 언어 지원, C#, F#, Visual Basic 등 다양한 프로그래밍 언어를 지원
- 크로스 플랫폼, .NET Core 이후 버전부터는 윈도우, macOS, Linux 등 다양한 운영체제에서 실행 가능
- 풍부한 라이브러리: 다양한 작업을 위한 풍부한 클래스 라이브러리를 제공
- 뛰어난 성능: Just-In-Time (JIT) 컴파일을 통해 높은 성능을 제공

.NET의 장점

- 생산성 향상, 풍부한 라이브러리와 도구를 통해 개발 시간을 단축하고 생산성을 향상시킬 수 있음
- 유지 보수 용이성, 객체 지향 프로그래밍 및 모듈식 설계를 통해 코드를 쉽게 유지보수할 수 있음
- 안정성, 강력한 형식 검사 및 예외 처리 메커니즘을 통해 안정적인 애플리케이션을 개발할 수 있음
- 확장성, .NET 애플리케이션은 높은 확장성을 제공하여 대규모 시스템 구축에 적합

.NET의 변화: 과거와 현재

.NET은 처음 출시된 이후 지속적인 발전을 거듭해왔습니다. 초기에는 윈도우 데스크톱 애플리케이션 개발에 중점을 두었지만, 현재는 웹, 모바일, 클라우드, 게임 등 다양한 분야에서 활용될 수 있도록 진화했습니다. 이러한 변화는 .NET Framework에서 .NET Core로의 전환, 그리고 .NET 5, 6, 7, 8, 9로 이어지는 통합 및 발전 과정을 통해 이루어졌습니다.

.NET Framework 시대 (2002년 ~ 2016년)

.NET Framework는 2002년 처음 출시되었습니다. 윈도우 운영체제에서 애플리케이션을 개발하고 실행하기 위한 프레임워크로, C#, Visual Basic, F# 등 다양한 언어를 지원했습니다. .NET Framework는 윈도우 데스크톱 애플리케이션, 웹 애플리케이션, 웹 서비스 등 다양한 애플리케이션 개발에 널리 사용되었습니다. 당시에는 윈도우 운영체제에 종속적이었습니다. ASP.NET 웹 프레임워크를 통해 웹 애플리케이션 개발을 지원하였고, Windows Forms 및 WPF를 통해 데스크톱 애플리케이션 개발을 지원하였습니다. .NET Framework는 윈도우 애플리케이션 개발에 큰 영향을 미쳤지만, 윈도우 운영체제에 종속적이라는 한계를 가지고 있었습니다. 이러한 한계를 극복하고 크로스 플랫폼 지원을 위해 .NET Core가 등장하게 됩니다.

.NET Core 시대 (2016년 ~ 2020년)

.NET Framework의 한계를 극복하고 크로스 플랫폼 지원, 오픈 소스 생태계 확장, 성능 향상을 목표로 2016년 .NET Core가 출시되었습니다. .NET Core는 모듈식 디자인을 채택하여 개발자가 필요한 구성 요소만 선택적으로 사용할 수 있도록 하였고, 이는 애플리케이션의 크기를 줄이고 배포를 용이하게 했습니다. 또한, 명령줄 인터페이스(CLI)를 통해 개발자가 다양한 플랫폼에서 애플리케이션을 빌드하고 실행할 수 있도록 지원했습니다. 윈도우, macOS, Linux 등 다양한 운영체제에서 실행 가능하도록 개선하였고, .NET Core는 오픈 소스 프로젝트로, 개발자 커뮤니티의 참여를 통해 빠르게 발전했습니다. 필요한 구성 요소만 선택적으로 사용할 수 있어 애플리케이션 크기를 줄이고 배포를 용이하게 합니다. .NET Framework 대비 성능이 크게 향상되었습니다. CLI를 통해 다양한 플랫폼에서 애플리케이션을 빌드하고 실행할 수 있습니다. .NET Core는 크로스 플랫폼 지원 및 오픈 소스 생태계 확장을 통해 .NET 플랫폼의 새로운 가능성을 열었습니다. 이후 .NET 5를 통해 .NET Framework와 .NET Core는 하나의 통합된 플랫폼으로 발전하게 됩니다.

통합과 발전 (2020년 ~ 현재)

2020년 11월, Microsoft는 .NET 5를 출시하며 .NET Framework와 .NET Core를 하나의 통합된 플랫폼으로 발전시켰습니다. .NET 5는 단일 코드 베이스로 윈도우, macOS, Linux, iOS, Android 등 다양한 플랫폼에서 실행 가능한 애플리케이션을 개발할 수 있도록 지원합니다. 또한, 성능 향상, 새로운 언어 기능 추가, 클라우드 네이티브 지원 강화 등 다양한 개선 사항을 포함했습니다. .NET 5 이후, Microsoft는 매년 새로운 버전의 .NET을 출시하며 플랫폼을 지속적으로 발전시키고 있습니다. .NET 6는 .NET MAUI를 통해 크로스 플랫폼 모바일 및 데스크톱 애플리케이션 개발을 지원하고, C# 10의 새로운 기능을 도입했습니다. .NET 7은 성능 향상과 클라우드 네이티브 지원 강화에 중점을 두었으며, .NET 8은 Blazor United를 통해 웹 UI 개발을 개선하고, .NET MAUI를 통해 윈도우 데스크톱 애플리케이션 개발을 지원합니다. .NET 9는 C# 12의 새로운 기능을 도입하고, 플랫폼 전반의 성능과 안정성을 향상시켰습니다.

버전	출시일	지원 유형 (대략)	주요 특징	지원 상태
.NET 8	2023년 11월 예정 (계획)	LTS	- **Native AOT** 개선 - 컨테이너 지원 확대 - 지속적인 성능 및 언어 기능 개선	미출시 (프리뷰 진행 중)
.NET 7	2022년 11월 8일	단기 지원 (STS, 약 18개월)	- **Native AOT** 기본 지원 - 컨테이너 성능 최적화 - **C# 11** 등 최신 언어 기능 - 전반적 성능 개선	지원 중
.NET 6	2021년 11월 8일	장기 지원 (LTS, 3년)	- **C# 10**, **F# 6** 지원 - **Minimal APIs** - **.NET MAUI** - 핫 리로드, 성능 개선 등	지원 중
.NET 5	2020년 11월 10일	단기 지원 (STS, 약 18개월)	- **단일 파일 배포** - 성능 및 컨테이너 개선 - .NET Core와 .NET Framework의 통합 시도	지원 종료 (2022년 5월 10일)

버전	출시일	지원 유형 (대략)	주요 특징	지원 상태
.NET Core 3.1	2019년 12월 3일	장기 지원 (LTS, 3년)	- **Windows 데스크톱** (WinForms, WPF) 공식 지원 - 전반적 성능 및 기능 개선	지원 종료 (2022년 12월 13일)
.NET Core 3.0	2019년 9월 23일	단기 지원 (STS, 약 18개월)	- **C# 8** 지원 - Blazor 서버 - Worker Service 템플릿 - 성능 개선	지원 종료 (2020년 3월 3일)
.NET Core 2.2	2018년 12월 4일	단기 지원 (STS, 약 18개월)	- ASP.NET Core 강화 - 추가 성능 최적화	지원 종료 (2019년 12월 23일)
.NET Core 2.1	2018년 5월 30일	장기 지원 (LTS, 3년)	- Razor 클래스 라이브러리 - SignalR, gRPC 등 - Entity Framework Core 개선	지원 종료 (2021년 8월 21일)
.NET Core 2.0	2017년 8월 14일	단기 지원 (STS, 약 18개월)	- 성능 개선 및 **self-contained** 배포 - 새로운 `csproj` 구조	지원 종료 (2018년 10월 1일)
.NET Core 1.1	2016년 11월 16일	단기 지원 (STS, 약 18개월)	- 신규 API 추가 - 툴링 및 안정성 개선	지원 종료 (2019년 6월 27일)
.NET Core 1.0	2016년 6월 27일	장기 지원 (LTS, 3년)	- **초기 크로스 플랫폼** 지원 - CLI 기반 개발 환경 도입	지원 종료 (2019년 6월 27일)

CLR

CLR(Common Language Runtime)은 .NET에서 프로그램이 실행될 때 동작을 관리해 주는 엔진(또는 환경)입니다. 마치 자동차 엔진이 자동차의 작동을 책임지듯, .NET 프로그램이 안전하고 일관된 방식으로 실행될 수 있도록 다양한 기능을 제공합니다. C# 같은 고수준 언어는 사람에게 읽기 쉽지만, 컴퓨터가 직접 이해하기엔 적합하지 않습니다. 그래서 중간 단계인 IL(Intermediate Language) 코드를 거친 뒤, CLR이 이를 실제 기계어(컴퓨터가 바로 실행할 수 있는 언어)로 바꿔주어야 프로그램이 동작합니다. 이렇게 하면 여러 운영 체제나 하드웨어에서도 공통된 방식으로 실행될 수 있습니다.

안전성과 편의성: 메모리 관리, 예외 처리와 같은 자잘하지만 중요한 부분들을 자동으로 처리해 줍니다. 덕분에 프로그래머는 주로 프로그램의 로직에만 집중할 수 있습니다.

CLR의 주요 기능

- 가비지 수집(Garbage Collection), 가비지 수집(Garbage Collection, GC) 은 프로그램이 사용했던 메모리 중 더 이상 사용하지 않는 부분을 CLR이 알아서 정리해 주는 기능입니다.수동으로 메모리를 관리하려면 '어디서 할당하고 어디서 해제해야 할까?'를 계속 생각해야 하는데, GC 덕분에 그 부담을 줄일 수 있습니다. 메모리를 잘못 정리하면 프로그램이 오작동하거나 멈출 수 있는데, GC가 자동으로 해 주므로 안전성이 높아집니다.

- JIT 컴파일(Just-In-Time Compilation), C# 코드 → IL 코드 → (실행 시점에) → 기계어 코드이 순서로 변환되는데, IL 코드를 최종적으로 기계어로 바꿔 주는 역할을 CLR이 맡습니다. 이 과정을 JIT(Just-In-Time) 컴파일 이라고 합니다. "Just-In-Time"이라는 이름대로, 미리 전부 변환해 두는 것이 아니라 실행에 필요한 순간에 해당 코드를 컴파일합니다. 이렇게 하면 시작 시점의 부담을 줄이고, 실행되는 부분만 빠르게 기계어로 변환할 수 있어 효율적입니다.
- 어셈블리(Assembly) 로딩 및 관리, .NET에서 코드를 묶어 배포하는 단위를 어셈블리(Assembly) 라고 부릅니다. (보통 .dll 또는 .exe 확장자를 가집니다.) CLR은 필요한 어셈블리를 찾아서 프로그램이 잘 쓸 수 있도록 관리하고, 실행 도중에 보안이나 버전 충돌을 점검하기도 합니다.
- 타입 시스템 관리, .NET에는 다양한 자료형(int, string, double 등)이 있는데, 이들을 한데 묶어 통합적으로 정의한 것을 Common Type System(CTS) 라고 합니다. CLR은 CTS를 기반으로 모든 언어(C#, VB.NET 등)에서 동일한 자료형을 다룰 수 있게 해 줍니다. 예를 들어, C#에서 int 타입은 VB.NET의 Integer 타입과 사실상 동일한 구조를 갖습니다. CLR이 내부적으로 같은 구조를 공유하도록 관리하기 때문이죠.
- 예외 처리 및 디버깅 지원, 프로그램이 실행될 때 오류(예외)가 발생하면, CLR이 예외를 인식하고 이를 프로그램에게 알려줄 수 있도록 구조를 갖추고 있습니다. 또한 CLR은 디버깅(문제 해결을 위해 프로그램 내부를 조사하는 과정)과 프로파일링(프로그램의 성능을 측정하고 분석하는 기능)을 지원해 줍니다.

CLR의 이점

- 안전한 메모리 관리, GC 덕분에 메모리 누수를 일으키기 쉬운 실수를 줄일 수 있습니다.
- 높은 호환성, IL 코드를 기반으로 JIT 컴파일되므로, 여러 플랫폼에서 동일한 프로그램을 실행할 수 있습니다(플랫폼 전용 .NET 런타임을 사용).
- 개발 생산성, 예외 처리, 디버깅, 형식 안전성, 풍부한 라이브러리 지원 덕분에 개발 속도가 빨라지고 오류도 줄일 수 있습니다.
- 다양한 언어 지원, CLR 덕분에 C#, VB.NET, F#, Python(.NET용 IronPython) 등 다양한 언어가 서로 어셈블리를 공유하면서 .NET 위에서 돌아갈 수 있습니다.

C#과 .NET의 관계

C#은 프로그래밍 언어이고, .NET은 애플리케이션을 개발하고 실행하기 위한 플랫폼입니다. C#은 .NET 플랫폼에서 실행되는 여러 언어 중 하나이며, .NET은 C# 코드를 실행하는 데 필요한 환경을 제공합니다. C# 언어 자체는 변수, 데이터 유형, 연산자, 제어문, 함수 등 프로그래밍의 기본적인 구성 요소를 제공합니다. 또한 객체 지향 프로그래밍, 제네릭, LINQ 등과 같은 고급 기능도 제공합니다. 반면 .NET은 CLR(Common Language Runtime), 기본 클래스 라이브러리(BCL), 그리고 다양한 API와 프레임워크를 제공합니다. CLR은 C# 코드를 실행하는 런타임 환경이며, BCL은 문자열 처리, 파일 입출력, 네트워킹 등과 같은 일반적인 작업을 위한 클래스 라이브러리를 제공합니다. .NET은 또한

ASP.NET, Windows Forms, WPF, Xamarin 등과 같은 다양한 애플리케이션 프레임워크를 제공합니다. C#은 .NET 플랫폼에서 애플리케이션을 개발하기 위한 도구이며, .NET은 C# 코드를 실행하고 다양한 기능을 제공하는 환경입니다. 둘은 서로 협력하여 강력하고 다양한 애플리케이션을 개발할 수 있도록 지원합니다.