

# C# 설명서

.NET 플랫폼에서 C# 프로그래밍 언어를 사용하여 애플리케이션을 작성하는 방법에 대해 알아봅니다.

## C#의 프로그램에 대해 알아보기

### 시작하기

[C#에 학습 | 자습서, 학습 과정, 동영상 등](#)

### VIDEO

[C# 초급 동영상 시리즈](#)

### 자습서

[셀프 안내 자습서](#)

[브라우저 내 자습서](#)

### 참조

[C# 질문 및 답변](#)

[.NET Tech Community 포럼의 언어](#)

[Stack Overflow의 C#](#)

[Discord의 C#](#)

## C# 기초

### 개요

[C# 둘러보기](#)

[C# 프로그램 내부](#)

[C# 하이라이트 동영상 시리즈](#)

### 개념

[형식 시스템](#)

개체 지향 프로그래밍

기능 기술

예외

코딩 스타일

---

### 자습서

명령줄 표시

클래스 소개

개체 지향 C#

형식 변환

패턴 일치

LINQ를 사용하여 데이터 쿼리

---

## 새로운 기능

---

### 새로운 기능

C# 13의 새로운 기능

C# 12의 새로운 기능

C# 11의 새로운 기능

C# 10의 새로운 기능

---

### 자습서

레코드 종류 탐색

최상위 문 탐색

새 패턴 탐색

사용자 지정 문자열 보간 처리기 작성

---

### 참조

C# 컴파일러의 호환성이 손상되는 변경

버전 호환성

## 주요 개념

### 개념

[C# 언어 전략](#)

[프로그래밍 개념](#)

### 개념

[LINQ\(Language-Integrated Query\)](#)

[비동기 프로그래밍](#)

## 고급 개념

### 참조

[리플렉션 및 특성](#)

[식 트리](#)

[기본 상호 운용성](#)

[성능 엔지니어링](#)

[.NET Compiler Platform SDK](#)

## 연결 유지하기

### 참조

[.NET 개발자 커뮤니티 ↗](#)

[YouTube ↗](#)

[Twitter ↗](#)

# C# 언어 둘러보기

아티클 • 2024. 05. 12.

C# 언어는 무료 플랫폼 간 오픈 소스 개발 환경인 [.NET 플랫폼](#)에서 가장 널리 사용되는 언어입니다. C# 프로그램은 IoT(사물 인터넷) 디바이스에서 클라우드에 이르기까지 다양한 디바이스에서 실행될 수 있습니다. 휴대폰, 데스크톱, 랩톱 컴퓨터 및 서버용 앱을 작성할 수 있습니다.

C#은 성능이 뛰어난 코드를 작성하면서 개발자의 생산성을 높이는 플랫폼 간 범용 언어입니다. 수백만 명의 개발자가 있는 C#은 가장 인기 있는 .NET 언어입니다. C#은 에코시스템 및 모든 .NET 워크로드를 광범위하게 지원합니다. 개체 지향 원칙에 기반하여 함수형 프로그래밍을 비롯한 다른 패러다임의 많은 기능을 통합합니다. 하위 수준 기능은 안전하지 않은 코드를 작성하지 않고도 고효율 시나리오를 지원합니다. 대부분의 .NET 라이브러리는 C#으로 작성되며 C#의 발전은 모든 .NET 개발자에게 도움이 되는 경우가 많습니다.

## Hello World

"Hello, World" 프로그램은 프로그래밍 언어를 소개하는 데 일반적으로 사용됩니다. C#에서는 다음과 같습니다.

```
C#  
  
// This line prints "Hello, World"  
Console.WriteLine("Hello, World");
```

//로 시작하는 줄은 한 줄 주석입니다. C# 한 줄 주석은 //로 시작하여 현재 줄 끝까지 계속됩니다. C#은 여러 줄 주석도 지원합니다. 여러 줄 주석은 /\*로 시작하고 \*/로 끝납니다. System 네임스페이스에 있는 Console 클래스의 WriteLine 메서드는 프로그램의 출력을 생성합니다. 이 클래스는 기본적으로 모든 C# 프로그램에서 자동으로 참조되는 표준 클래스 라이브러리에서 제공됩니다.

앞의 예는 [최상위 문](#)을 사용하는 "Hello, World" 프로그램의 한 형태를 보여 줍니다. 이전 버전의 C#에서는 메서드에서 프로그램의 진입점을 정의해야 했습니다. 이 형식은 여전히 유효하며 많은 기존 C# 샘플에서 볼 수 있습니다. 다음 예에 표시된 것처럼 이 형식에도 익숙해야 합니다.

```
C#  
  
using System;  
  
class Hello
```

```
{  
    static void Main()  
    {  
        // This line prints "Hello, World"  
        Console.WriteLine("Hello, World");  
    }  
}
```

이 버전은 프로그램에서 사용하는 구성 요소를 보여 줍니다. “Hello, World” 프로그램은 `System` 네임스페이스를 참조하는 `using` 지시문으로 시작합니다. 네임스페이스는 계층적으로 C# 프로그램 및 라이브러리를 구성하는 방법을 제공합니다. 네임스페이스에는 형식 및 기타 네임스페이스가 포함됩니다. 예를 들어, `System` 네임스페이스에는 프로그램에서 참조되는 `Console` 클래스와 같은 많은 형식과 `IO` 및 `Collections` 와 같은 기타 많은 네임스페이스가 포함되어 있습니다. 지정된 네임스페이스를 참조하는 `using` 지시문을 사용하여 해당 네임스페이스의 멤버인 형식을 정규화되지 않은 방식으로 사용할 수 있습니다. `using` 지시문 때문에, 프로그램은 `Console.WriteLine` 을 `System.Console.WriteLine`의 약식으로 사용할 수 있습니다. 이전 예에서는 해당 네임스페이스가 [암시적으로 포함되었습니다](#).

“Hello, World” 프로그램에서 선언된 `Hello` 클래스에는 단일 멤버인 `Main` 메서드가 있습니다. `Main` 메서드는 `static` 한정자로 선언됩니다. 인스턴스 메서드는 키워드 `this`를 사용하여 특정 바깥쪽 개체 인스턴스를 참조할 수 있지만 정적 메서드는 특정 개체에 대한 참조 없이 작동합니다. 관례적으로 최상위 문이 없으면 `Main`이라는 정적 메서드가 C# 프로그램의 [진입점](#) 역할을 합니다.

두 진입점 형식 모두 동등한 코드를 생성합니다. 최상위 문을 사용하면 컴파일러는 프로그램 진입점에 대한 포함 클래스와 메서드를 합성합니다.

### 💡 팁

이 문서의 예에서는 C# 코드를 처음으로 살펴봅니다. 일부 샘플에는 익숙하지 않은 C# 요소가 표시될 수 있습니다. C#을 알아볼 준비가 되면 [초보자 자습서](#)부터 시작하거나 각 섹션의 링크를 살펴봅니다. [Java](#), [JavaScript](#), [TypeScript](#) 또는 [Python](#) 사용 환경이 있는 경우 C#을 빠르게 배우는 데 필요한 정보를 찾는 데 도움이 되는 팀을 읽어보세요.

## 익숙한 C# 기능

C#은 초보자가 접근하기 쉬우면서도 특수 애플리케이션을 작성하는 숙련된 개발자를 위한 유용한 기능을 제공합니다. 빠르게 생산성을 높일 수 있습니다. 사용자의 응용 분야에 필요하다면 보다 전문적인 기술을 알아볼 수 있습니다.

C# 앱은 .NET 런타임의 [자동 메모리 관리](#)를 활용합니다. C# 앱은 .NET SDK에서 제공하는 광범위한 [런타임 라이브러리](#)도 사용합니다. 파일 시스템 라이브러리, 데이터 수집, 수학 라이브러리와 같은 일부 구성 요소는 플랫폼 독립적입니다. 다른 것들은 ASP.NET Core 웹 라이브러리 또는 .NET MAUI UI 라이브러리와 같은 단일 워크로드에만 해당됩니다. [NuGet](#)의 풍부한 오픈 소스 에코시스템은 런타임의 일부인 라이브러리를 강화합니다. 이러한 라이브러리는 사용할 수 있는 더 많은 구성 요소를 제공합니다.

C#은 C 언어 계열에 속합니다. C, C++, JavaScript 또는 Java를 사용해 본 적이 있다면 [C# 구문](#)이 익숙할 것입니다. C 계열의 모든 언어와 마찬가지로 세미콜론(;)은 문의 끝을 정의합니다. C# 식별자는 대/소문자를 구분합니다. C#에서는 중괄호, {} 및 if, else 및 switch와 같은 제어 문, for 및 while과 같은 반복 구문을 동일하게 사용합니다. C#에는 모든 컬렉션 형식에 대한 foreach 문도 있습니다.

C#은 강력한 형식의 언어입니다. 선언하는 모든 변수에는 컴파일 시간에 알려진 형식이 있습니다. 컴파일러나 편집 도구는 해당 형식을 잘못 사용하고 있는지 알려 줍니다. 프로그램을 실행하기 전에 이러한 오류를 수정할 수 있습니다. [기본 데이터 형식](#)은 언어 및 런타임에 기본 제공되어 있습니다. 즉, int, double, char와 같은 값 형식, string과 같은 참조 형식, 배열 및 기타 컬렉션이 있습니다. 프로그램을 작성하면서 고유의 형식을 만들게 됩니다. 이러한 형식은 값의 경우 struct 형식이거나 개체 지향 동작을 정의하는 class 형식일 수 있습니다. 컴파일러가 동등 비교를 위해 코드를 합성하도록 struct 또는 class 형식에 record 한정자를 추가할 수 있습니다. 해당 인터페이스를 구현하는 형식이 제공해야 하는 계약 또는 멤버 집합을 정의하는 interface 정의를 만들 수도 있습니다. 제네릭 형식 및 메서드를 정의할 수도 있습니다. [제네릭](#)은 형식 매개 변수를 사용하여 사용 시 실제 형식에 대한 자리 표시자를 제공합니다.

코드를 작성할 때 [메서드](#)라고도 하는 함수를 struct 및 class 형식의 멤버로 정의합니다. 이러한 메서드는 형식의 동작을 정의합니다. 다양한 수 또는 형식의 매개 변수를 사용하여 메서드를 오버로드할 수 있습니다. 메서드는 선택적으로 값을 반환할 수 있습니다. 메서드 외에도 C# 형식에는 접근자라는 함수가 지원하는 데이터 요소인 [속성](#)이 있을 수 있습니다. C# 형식은 구독자에게 중요한 작업을 알릴 수 있는 [이벤트](#)를 정의할 수 있습니다. C#은 class 형식에 대한 상속 및 다형성과 같은 개체 지향 기술을 지원합니다.

C# 앱은 예외를 사용하여 오류를 보고하고 처리합니다. C++ 또는 Java를 사용해 본 적이 있다면 이 방법에 익숙할 것입니다. 의도한 대로 수행할 수 없는 경우 코드에서 예외가 throw됩니다. 다른 코드는 호출 스택의 수준에 관계없이 try - catch 블록을 사용하여 선택적으로 복구할 수 있습니다.

## 독특한 C# 기능

C#의 일부 요소는 익숙하지 않을 수 있습니다. LINQ([언어 통합 쿼리](#))는 모든 데이터 컬렉션을 쿼리하거나 변환하는 일반적인 패턴 기반 구문을 제공합니다. LINQ는 메모리 내 컬

렉션, XML 또는 JSON과 같은 구조화된 데이터, 데이터베이스 스토리지, 심지어 클라우드 기반 데이터 API를 쿼리하기 위한 구문을 통합합니다. 하나의 구문 집합을 학습하면 스토리지에 관계없이 데이터를 검색하고 조작할 수 있습니다. 다음 쿼리는 평점 평균이 3.5보다 큰 모든 학생을 찾습니다.

C#

```
var honorRoll = from student in Students
                 where student.GPA > 3.5
                 select student;
```

앞의 쿼리는 `Students`로 표시되는 다양한 스토리지 유형에 대해 작동합니다. 개체 컬렉션, 데이터베이스 테이블, 클라우드 스토리지 Blob 또는 XML 구조일 수 있습니다. 모든 스토리지 유형에 동일한 쿼리 구문이 적용됩니다.

[작업 기반 비동기 프로그래밍 모델](#)을 사용하면 비동기적으로 실행되더라도 동기적으로 실행되는 것처럼 읽는 코드를 작성할 수 있습니다. 비동기식 메서드와 식이 비동기식으로 평가되는 경우를 설명하기 위해 `async` 및 `await` 키워드를 활용합니다. 다음 샘플은 비동기 웹 요청을 기다립니다. 비동기 작업이 완료되면 메서드는 응답 길이를 반환합니다.

C#

```
public static async Task<int> GetPageLengthAsync(string endpoint)
{
    var client = new HttpClient();
    var uri = new Uri(endpoint);
    byte[] content = await client.GetByteArrayAsync(uri);
    return content.Length;
}
```

C#은 GraphQL 페이지 API와 같은 비동기 작업으로 지원되는 컬렉션을 반복하는 `await foreach` 문도 지원합니다. 다음 샘플은 데이터를 청크로 읽고, 사용 가능한 경우 각 요소에 대한 액세스를 제공하는 반복기를 반환합니다.

C#

```
public static async IAsyncEnumerable<int> ReadSequence()
{
    int index = 0;
    while (index < 100)
    {
        int[] nextChunk = await GetNextChunk(index);
        if (nextChunk.Length == 0)
        {
            yield break;
        }
        foreach (var item in nextChunk)
```

```
        {
            yield return item;
        }
        index++;
    }
}
```

호출자는 `await foreach` 문을 사용하여 컬렉션을 반복할 수 있습니다.

C#

```
await foreach (var number in ReadSequence())
{
    Console.WriteLine(number);
}
```

C#은 [패턴 일치](#)를 제공합니다. 이러한 식을 사용하면 데이터를 검사하고 해당 특성에 따라 결정을 내릴 수 있습니다. 패턴 일치는 데이터 기반 제어 흐름에 대한 훌륭한 구문을 제공합니다. 다음 코드는 패턴 일치 구문을 사용하여 부울 *and*, *or* 및 *xor* 연산의 메서드를 표현하는 방법을 보여 줍니다.

C#

```
public static bool Or(bool left, bool right) =>
(left, right) switch
{
    (true, true) => true,
    (true, false) => true,
    (false, true) => true,
    (false, false) => false,
};

public static bool And(bool left, bool right) =>
(left, right) switch
{
    (true, true) => true,
    (true, false) => false,
    (false, true) => false,
    (false, false) => false,
};

public static bool Xor(bool left, bool right) =>
(left, right) switch
{
    (true, true) => false,
    (true, false) => true,
    (false, true) => true,
    (false, false) => false,
};
```

패턴 일치 식은 모든 값에 대한 catch all로 `_`을 사용하여 간소화할 수 있습니다. 다음 예에서는 `and` 메서드를 간소화하는 방법을 보여 줍니다.

C#

```
public static bool ReducedAnd(bool left, bool right) =>
    (left, right) switch
    {
        (true, true) => true,
        (_, _) => false,
    };
```

마지막으로, .NET 에코시스템의 일부로 [Visual Studio](#) 또는 [C# DevKit](#)과 함께 [Visual Studio Code](#)를 사용할 수 있습니다. 이러한 도구는 사용자가 작성하는 코드를 포함하여 C#에 대한 풍부한 이해를 제공합니다. 또한 디버깅 기능도 제공합니다.

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

ଓ 설명서 문제 열기

ଓ 제품 사용자 의견 제공

# C#을 배우는 Java 개발자를 위한 로드맵

아티클 • 2024. 04. 11.

C#과 Java는 많은 유사성을 가지고 있습니다. C#을 배우면서 이미 Java 프로그래밍을 통해 얻은 많은 지식을 적용할 수 있습니다.

- 유사한 구문:** Java와 C#은 모두 C 언어 계열에 속합니다. 이러한 유사성은 이미 C#을 읽고 이해할 수 있음을 의미합니다. 약간의 차이점은 있지만 대부분의 구문은 Java, C와 동일합니다. 중괄호와 세미콜론이 동일하게 사용됩니다. `if`, `else`, `switch`와 같은 제어문은 동일합니다. `for`, `while` 및 `do...while`의 반복 문은 동일합니다. `class`, `struct` 및 `interface`에 대한 동일한 키워드가 두 언어 모두에 있습니다. `public`에서 `private`까지의 액세스 한정자는 동일합니다. 많은 내장 형식도 동일한 키워드 (`int`, `string` 및 `double`)를 사용합니다.
- 개체 지향 패러다임:** Java와 C#은 모두 개체 지향 언어입니다. 다형성, 추상화 및 캡슐화의 개념은 두 언어 모두에 적용됩니다. 둘 다 새로운 구문을 추가했지만 핵심 기능은 여전히 관련이 있습니다.
- 강력한 형식의 언어:** Java와 C#은 모두 강력한 형식의 언어입니다. 명시적으로 또는 암시적으로 변수의 데이터 형식을 선언합니다. 컴파일러는 형식 안전성을 강화합니다. 컴파일러는 코드를 실행하기 전에 코드에서 형식 관련 오류를 `catch`합니다.
- 플랫폼 간:** Java와 C#은 모두 플랫폼 간에 사용할 수 있습니다. 원하는 플랫폼에서 개발 도구를 실행할 수 있습니다. 사용자의 애플리케이션은 여러 플랫폼에서 실행될 수 있습니다. 개발 플랫폼이 대상 플랫폼과 일치할 필요는 없습니다.
- 예외 처리:** Java와 C# 모두 오류를 표시하기 위해 예외를 `throw`합니다. 둘 다 `try` - `catch` - `finally` 블록을 사용하여 예외를 처리합니다. `Exception` 클래스는 비슷한 이름과 상속 계층 구조를 갖습니다. 한 가지 차이점은 C#에는 확인된 예외라는 개념이 없다는 것입니다. 이론상 모든 메서드에서 예외가 `throw`될 수 있습니다.
- 표준 라이브러리:** .NET 런타임 및 JSL(Java Standard Library)은 일반 작업을 지원합니다. 둘 다 다른 오픈 소스 패키지를 위한 광범위한 에코시스템을 갖추고 있습니다. C#에서 패키지 관리자는 [NuGet](#)입니다. Maven과 유사합니다.
- 가비지 수집:** 두 언어 모두 가비지 수집을 통해 자동 메모리 관리를 사용합니다. 런타임은 참조되지 않은 개체에서 메모리를 회수합니다. 한 가지 차이점은 C#을 사용하면 값 형식을 `struct` 형식으로 만들 수 있다는 것입니다.

유사성으로 인해 C#에서는 거의 즉시 생산성을 발휘할 수 있습니다. 진행하면서 Java에서는 사용할 수 없는 C#의 기능과 관용구를 배워야 합니다.

- 패턴 일치:** 패턴 일치를 사용하면 복잡한 데이터 구조의 형태를 기반으로 간결한 조건문과 식을 사용할 수 있습니다. `is` 문은 변수가 어떤 패턴인지 확인합니다. 패턴 기반 `switch` 식은 변수를 검사하고 해당 특성에 따라 결정을 내릴 수 있는 풍부한 구문을 제공합니다.

2. **문자열 보간 및 원시 문자열 리터럴**: 문자열 보간을 사용하면 위치 식별자를 사용하는 대신 평가된 식을 문자열에 삽입할 수 있습니다. 원시 문자열 리터럴은 텍스트의 이스케이프 시퀀스를 최소화하는 방법을 제공합니다.
3. **nullable 형식 및 null 허용 불가 형식**: C#은 형식에 ? 접미사를 추가하여 null 허용 값 형식과 null 허용 참조 형식을 지원합니다. nullable 형식의 경우 식을 역참조하기 전에 null을 확인하지 않으면 컴파일러에서 경고를 표시합니다. Null을 허용하지 않는 형식의 경우 컴파일러는 해당 변수에 null 값을 할당할 수 있는지 경고합니다. null을 허용하지 않는 참조 형식은 System.NullReferenceException을 throw하는 프로그래밍 오류를 최소화합니다.
4. **확장 메서드**: C#에서는 클래스나 인터페이스를 확장하는 메서드를 만들 수 있습니다. 확장 메서드는 라이브러리에서 형식의 동작을 확장하거나 지정된 인터페이스를 구현하는 모든 형식을 확장합니다.
5. **LINQ**: LINQ(언어 통합 쿼리)는 스토리지에 관계없이 데이터를 쿼리하고 변환하는 공통 구문을 제공합니다.
6. **로컬 함수**: C#에서는 메서드 내부에 함수를 중첩하거나 다른 로컬 함수를 중첩할 수 있습니다. 로컬 함수는 또 다른 캡슐화 계층을 제공합니다.

C#에는 Java에는 없는 다른 기능이 있습니다. `async` 및 `await`와 같은 기능과 비메모리 리소스를 자동으로 해제하는 `using` 문을 볼 수 있습니다.

C#과 Java에는 몇 가지 유사한 기능도 있으며, 여기에는 미묘하지만 중요한 차이점이 있습니다.

1. **속성 및 인덱서**: 속성 및 인덱서(배열 또는 사전과 같이 클래스 처리)는 언어를 지원합니다. Java에서는 `get` 및 `set`로 시작하는 메서드에 대한 명명 규칙입니다.
2. **레코드**: C#에서 레코드는 `class`(참조) 또는 `struct`(값) 형식일 수 있습니다. C# 레코드는 변경할 수 없지만 변경 불가능할 필요는 없습니다.
3. **튜플**은 C#과 Java에서 서로 다른 구문을 사용합니다.
4. **특성**은 Java 주석과 유사합니다.

마지막으로 C#에서는 사용할 수 없는 Java 언어 기능이 있습니다.

1. **확인된 예외**: C#에서는 이론적으로 모든 메서드에서 예외가 throw될 수 있습니다.
2. **확인된 배열 공분산**: C#에서 배열은 안전한 공분산이 아닙니다. 공변 구조가 필요한 경우 제네릭 컬렉션 클래스와 인터페이스를 사용해야 합니다.

전반적으로 Java 환경이 있는 개발자라면 C#을 배우는 것이 원활할 것입니다. 빠르게 생산성을 높일 수 있을 만큼 친숙한 관용어를 찾을 수 있으며, 새로운 관용어를 빠르게 알아볼 수 있습니다.

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# C#을 배우는 JavaScript 및 TypeScript 개발자를 위한 로드맵

아티클 • 2024. 04. 11.

C#, TypeScript 및 JavaScript는 모두 C 언어 계열의 멤버입니다. 두 언어 간의 유사성은 C#에서 빠르게 생산성을 높이는 데 도움이 됩니다.

- 유사한 구문:** JavaScript, TypeScript 및 C#은 C 언어 계열에 속합니다. 이러한 유사성은 이미 C#을 읽고 이해할 수 있음을 의미합니다. 약간의 차이점은 있지만 대부분의 구문은 JavaScript, C와 동일합니다. 중괄호와 세미콜론은 익숙합니다. `if`, `else`, `switch`와 같은 제어문은 동일합니다. `for`, `while` 및 `do...while`의 반복 문은 동일합니다. `class`, `struct` 및 `interface`에 대한 동일한 키워드는 C#과 TypeScript에 모두 있습니다. TypeScript와 C#의 액세스 한정자는 `public`부터 `private`까지 동일합니다.
- => 토큰:** 모든 언어는 경량 함수 정의를 지원합니다. C#에서는 [람다 식](#)이라고 하며, JavaScript에서는 일반적으로 [화살표 함수](#)라고 합니다.
- 함수 계층 구조:** 세 가지 언어 모두 다른 함수에 정의된 함수인 [로컬 함수](#)를 지원합니다.
- 비동기/대기:** 세 가지 언어 모두 비동기 프로그래밍에 대해 동일한 `async` 및 `await` 키워드를 공유합니다.
- 가비지 수집:** 세 가지 언어 모두 자동 메모리 관리를 위해 가비지 수집기를 사용합니다.
- 이벤트 모델:** C#의 [event](#) 구문은 DOM(문서 개체 모델) 이벤트에 대한 JavaScript 모델과 유사합니다.
- 패키지 관리자:** [NuGet](#)은 JavaScript 애플리케이션용 npm과 유사한 C# 및 .NET용 가장 일반적인 패키지 관리자입니다. C# 라이브러리는 [어셈블리](#)로 제공됩니다.

C#을 계속 배우면서 JavaScript의 일부가 아닌 개념을 배우게 됩니다. TypeScript를 사용하는 경우 다음 개념 중 일부가 익숙할 수 있습니다.

- C# 형식 시스템:** C#은 강력한 형식의 언어입니다. 모든 변수에는 형식이 있으며 해당 형식은 변경할 수 없습니다. `class` 또는 `struct` 형식을 정의합니다. 다른 형식으로 구현되는 동작을 정의하는 `interface` 정의를 정의할 수 있습니다. TypeScript에는 이러한 개념이 많이 포함되어 있지만 TypeScript는 JavaScript를 기반으로 빌드되었기 때문에 형식 시스템이 그다지 엄격하지 않습니다.
- 패턴 일치:** 패턴 일치를 사용하면 복잡한 데이터 구조의 형태를 기반으로 간결한 조건문과 식을 사용할 수 있습니다. `is` 식은 변수가 어떤 패턴인지 확인합니다. 패턴 기반 `switch` 식은 변수를 검사하고 해당 특성에 따라 결정을 내릴 수 있는 풍부한 구문을 제공합니다.

3. **문자열 보간 및 원시 문자열 리터럴**: 문자열 보간을 사용하면 위치 식별자를 사용하는 대신 평가된 식을 문자열에 삽입할 수 있습니다. 원시 문자열 리터럴은 텍스트의 이스케이프 시퀀스를 최소화하는 방법을 제공합니다.
4. **nullable 형식 및 non-nullable 형식**: C#은 유형에 ? 접미사를 추가하여 null 허용 값 유형과 null 허용 참조 유형을 지원합니다. nullable 형식의 경우 식을 역참조하기 전에 null 을 확인하지 않으면 컴파일러에서 경고를 표시합니다. non-nullable 형식의 경우 컴파일러는 해당 변수에 null 값을 할당할 수 있는지 경고합니다. 이러한 기능은 애플리케이션이 System.NullReferenceException 을 throw하는 것을 최소화할 수 있습니다. 구문은 TypeScript가 선택적 속성에 대해 ? 을 사용하는 것과 유사할 수 있습니다.
5. **LINQ**: LINQ(언어 통합 쿼리)는 스토리지에 관계없이 데이터를 쿼리하고 변환하는 공통 구문을 제공합니다.

더 많이 배우면 다른 차이점이 분명해 지지만 이러한 차이점 중 상당수는 범위가 더 작습니다.

JavaScript 및 TypeScript의 일부 친숙한 기능과 관용구는 C#에서 사용할 수 없습니다.

1. **동적 형식**: C#에서는 정적 형식 지정을 사용합니다. 변수 선언에는 형식이 포함되며 해당 형식은 변경할 수 없습니다. C#에는 런타임 바인딩을 제공하는 dynamic 형식이 있습니다.
2. **프로토타입 상속**: C# 상속은 형식 선언의 일부입니다. C# class 선언은 기본 클래스를 명시합니다. JavaScript에서는 \_\_proto\_\_ 속성을 설정하여 모든 인스턴스에 기본 형식을 설정할 수 있습니다.
3. **해석된 언어**: C# 코드를 실행하기 전에 컴파일해야 합니다. JavaScript 코드는 브라우저에서 직접 실행할 수 있습니다.

또한 C#에서는 몇 가지 추가 TypeScript 기능을 사용할 수 없습니다.

1. **공용 구조체 형식**: C#은 공용 구조체 형식을 지원하지 않습니다. 그러나 디자인 제안은 진행 중입니다.
2. **데코레이터**: C#에는 데코레이터가 없습니다. @sealed 와 같은 일부 일반적인 데코레이터는 C#에서 예약된 키워드입니다. 다른 일반 데코레이터에는 해당하는 특성이 있을 수 있습니다. 다른 데코레이터의 경우 고유한 특성을 만들 수 있습니다.
3. **더 관대해진 구문**: C# 컴파일러는 JavaScript에서 요구하는 것보다 더 엄격하게 코드를 구문 분석합니다.

웹 애플리케이션을 빌드하는 경우 Blazor 를 사용하여 애플리케이션을 빌드하는 것을 고려해야 합니다. Blazor는 .NET 및 C#용으로 빌드된 전체 스택 웹 프레임워크입니다.

Blazor 구성 요소는 서버에서 .NET 어셈블리로 실행되거나 WebAssembly 를 사용하여 클라이언트에서 실행될 수 있습니다. Blazor는 즐겨 사용하는 JavaScript 또는 TypeScript 라이브러리와의 interop 을 지원합니다.

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# C#을 배우는 Python 개발자를 위한 로드맵

아티클 • 2024. 04. 15.

C#과 Python은 비슷한 개념을 공유합니다. Python을 이미 아는 경우 다음과 같은 친숙한 구성은 C#을 학습하는 데 도움이 됩니다.

- 개체 지향:** Python과 C#은 모두 개체 지향 언어입니다. Python의 클래스에 대한 모든 개념은 구문이 다르더라도 C#에 적용됩니다.
- 크로스 플랫폼:** Python과 C#은 모두 크로스 플랫폼 언어입니다. 두 언어 중 하나로 작성된 앱은 많은 플랫폼에서 실행될 수 있습니다.
- 가비지 수집:** 두 언어 모두 가비지 수집을 통해 자동 메모리 관리를 사용합니다. 런타임은 참조되지 않은 개체에서 메모리를 회수합니다.
- 강력한 형식:** Python과 C#은 모두 강력한 형식의 언어입니다. 형식 강제 변환은 암시적으로 발생하지 않습니다. C#은 정적으로 형식화되지만 Python은 동적으로 형식화되는 차이점이 있으며, 이는 이후에 설명합니다.
- Async/ Await:** Python의 `async` 및 `await` 기능은 C#의 `async` 및 `await` 지원에서 직접 영감을 받았습니다.
- 패턴 일치:** Python의 `match` 식 및 패턴 일치는 C#의 `패턴 일치 switch` 식과 유사합니다. 이를 사용하여 복잡한 데이터 식을 검사하여 패턴과 일치하는지 확인할 수 있습니다.
- 문 키워드:** Python과 C#은 `if`, `else`, `while`, `for` 등 많은 키워드를 공유합니다. 두 언어는 모든 구문이 동일하지는 않지만, Python을 알고 있는 경우 C#을 읽을 수 있을 만큼 유사합니다.

C#을 배우기 시작하면서 Python과는 다른 다음과 같은 중요한 개념을 알게 됩니다.

- 들여쓰기 대 토큰:** Python에서 줄 바꿈 및 들여쓰기는 일류 구문 요소입니다. C#에서는 공백이 중요하지 않습니다. ; 개별 문과 같은 토큰과 기타 토큰 {} 및 ()은(는) if 및 기타 블록 문에 대한 블록 범위를 제어합니다. 그러나 가독성을 위해 대부분의 코딩 스타일(이러한 문서에 사용된 스타일 포함)은 들여쓰기를 사용하여 {}과(와) {}이(가) 선언한 블록 범위를 강화합니다.
- 정적 형식 지정:** C#에서는 변수 선언에 해당 형식이 포함됩니다. 변수를 다른 형식의 개체에 다시 할당하면 컴파일러 오류가 발생합니다. python에서는 다시 할당할 때 형식이 변경될 수 있습니다.
- nullable 형식:** C# 변수는 `null` 허용 또는 `null`을 허용하지 않는 것일 수 있습니다. `null`을 허용하지 않는 형식은 nullable(또는 아무것도) 될 수 없는 형식입니다. 이 형식은 항상 유효한 개체를 참조합니다. 반면 nullable 형식은 유효한 개체 또는 `null`을 참조할 수 있습니다.

4. **LINQ**: LINQ(Language-Integrated Query)를 구성하는 쿼리 식 키워드는 Python의 키워드는 아닙니다. 그러나 `itertools`, `more-itertools`, `py-linq`과(와) 같은 Python 라이브러리가 비슷한 기능을 제공합니다.
5. **제네릭**: C# 제네릭은 C# 정적 형식 지정을 사용하여 형식 매개 변수에 제공된 인수에 대한 어설션을 만듭니다. 제네릭 알고리즘은 인수 형식이 충족해야 하는 제약 조건을 지정해야 할 수 있습니다.

마지막으로 C#에서는 사용할 수 없는 Python 언어 기능이 있습니다.

1. **구조적(적) 형식 지정**: C#에서는 형식에 이름과 선언이 있습니다. `tuples`를 제외하고 구조가 동일한 형식은 서로 바꿔 사용할 수 없습니다.
2. **REPL**: C#에는 솔루션을 신속하게 프로토타입하는 REPL(read–eval–print loop)이 없습니다.
3. **중요 공백**: 블록 범위를 기록하려면 중괄호 `{` 및 `}` 을(를) 올바르게 사용해야 합니다.

Python을 알고 있다면 C# 배우기는 순조로운 여정입니다. 두 언어에서는 비슷한 개념과 관용어가 사용됩니다.

### ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

💡 설명서 문제 열기

↗️ 제품 사용자 의견 제공

# 주석이 추가된 C# 전략

아티클 • 2024. 03. 13.

개발자의 변화하는 요구를 충족하고 최첨단 프로그래밍 언어로 남도록 C#을 계속 발전시킬 것입니다. 우리는 .NET 라이브러리, 개발자 도구 및 워크로드 지원을 담당하는 팀과 협력하여 열렬하고 광범위하게 혁신하면서 언어의 정신을 유지하도록 주의를 기울일 것입니다. C#이 사용되는 영역의 다양성을 인식하여 모든 또는 대부분의 개발자에게 도움이 되는 언어 및 성능 개선을 선호하고 이전 버전과의 호환성에 대한 강한 의지를 유지할 것입니다. Microsoft는 계속해서 더 광범위한 .NET 에코시스템에 권한을 부여하고 C#의 미래에서 그 역할을 확대하는 동시에 디자인 결정에 대한 책임감을 유지할 것입니다.

## 전략이 C#을 안내하는 방법

C# 전략은 C# 진화에 대한 결정을 안내하며, 이러한 주석은 핵심 문에 대해 어떻게 생각하는지에 대한 인사이트를 제공합니다.

"우리는 열성적이고 광범위하게 혁신할 것입니다."

C# 커뮤니티는 계속 성장하고 있으며, C# 언어는 커뮤니티의 요구와 기대에 맞게 계속 진화하고 있습니다. 다양한 소스에서 영감을 받아 C# 개발자의 큰 세그먼트에 도움이 되며 생산성, 가독성 및 성능을 일관되게 개선하는 기능을 선택합니다.

"언어의 정신을 유지하도록 주의를 기울입니다."

C# 언어의 정신과 역사에서 새로운 아이디어를 평가합니다. 대부분의 기존 C# 개발자가 이해하는 혁신의 우선순위를 지정합니다.

"모든 또는 대부분의 개발자에게 혜택을 주는 개선"

개발자는 웹 프런트 엔드 및 백 엔드, 클라우드 네이티브 개발, 데스크톱 개발 및 플랫폼 간 애플리케이션 빌드와 같은 모든 .NET 워크로드에서 C#을 사용합니다. 직접 또는 공통 라이브러리의 개선을 강화하여 가장 큰 영향을 주는 새로운 기능에 초점을 맞춥니다. 언어 기능 개발에는 개발자 도구 및 학습 리소스에 대한 통합이 포함됩니다.

"이전 버전과의 호환성에 대한 강한 의지"

현재 방대한 야의 C# 코드가 사용되고 있음을 존중합니다. 잠재적인 호환성이 손상되는 변경은 C# 커뮤니티에 대한 중단의 규모 및 영향에 대해 신중하게 고려됩니다.

## "관리 유지"

C# 언어 디자인 [은](#) 커뮤니티 참여와 함께 공개적으로 진행됩니다. 누구나 GitHub 리포지토리 [에서](#) 새로운 C# 기능을 제안할 수 있습니다. 언어 디자인 팀 [은](#) 커뮤니티 입력을 계량한 후 최종 결정을 내립니다.

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# C# 소개

아티클 • 2024. 04. 11.

C# 소개 자습서를 시작합니다. 이 단원은 브라우저에서 실행할 수 있는 대화형 코드로 시작됩니다. 해당 대화형 단원을 시작하기 전에 [C# 101 동영상 시리즈](#)에서 C#의 기본 사항을 학습할 수 있습니다.

<https://docs.microsoft.com/shows/CSharp-101/What-is-C/player>

첫 번째 단원에서는 작은 코드 조각을 사용하여 C# 개념을 설명합니다. C# 구문의 기본 사항과 문자열, 숫자 및 부울과 같은 데이터 형식을 사용하는 방법에 대해 학습합니다. 모두 대화형이며, 몇 분 내에 코드를 작성하여 실행할 수 있습니다. 이 첫 번째 단원에서는 프로그래밍이나 C# 언어에 대한 사전 지식이 없다고 가정합니다.

다양한 환경에서 이 자습서를 사용해 볼 수 있습니다. 학습할 개념은 같습니다. 차이점은 선호하는 환경입니다.

- [브라우저의 docs 플랫폼](#). 이 환경에는 실행 가능한 C# 코드 창이 docs 페이지에 포함되어 있습니다. 브라우저에서 C# 코드를 작성하고 실행합니다.
- [Microsoft Learn 환경](#). 이 학습 경로에는 C#의 기본 사항을 설명하는 여러 모듈이 포함되어 있습니다.
- [Binder의 Jupyter](#). Binder의 Jupyter Notebook에서 C# 코드를 시험해 볼 수 있습니다.
- [로컬 머신](#). 온라인에서 살펴본 후 .NET SDK를 [다운로드](#)하고 컴퓨터에 프로그램을 빌드할 수 있습니다.

Hello World 단원 다음에 나오는 모든 소개 자습서는 온라인 브라우저 환경이나 [자체로컬 개발 환경](#)을 사용하여 볼 수 있습니다. 각 자습서가 끝날 때 다음 단원을 온라인으로 진행할지, 사용자 머신에서 진행할지 결정합니다. 환경을 설정하고 사용자 머신에서 다음 자습서를 진행하는 데 유용한 링크가 있습니다.

## Hello World

Hello World 자습서에서는 가장 기본적인 C# 프로그램을 만듭니다. `string` 형식을 살펴보고 텍스트를 사용하는 방법을 살펴봅니다. 또한 [Microsoft Learn](#) 또는 [Binder의 Jupyter](#) 경로를 사용할 수 있습니다.

## C#의 숫자

[C#의 숫자](#) 자습서에서는 컴퓨터가 숫자를 저장하는 방법과 여러 숫자 형식으로 계산을 수행하는 방법을 알아봅니다. 반올림의 기본 사항과 C#을 사용하여 수학 계산을 수행하

는 방법에 대해 학습합니다. 이 자습서는 [머신에서 로컬로 실행](#)할 수도 있습니다.

이 자습서에서는 [Hello World](#) 단원을 완료했다고 가정합니다.

## 분기 및 루프

[분기 및 루프](#) 자습서에서는 변수에 저장된 값에 따라 코드 실행의 여러 경로를 선택하는 기본 사항을 설명합니다. 프로그램에서 결정하고 여러 작업을 선택하는 방법에 대한 기본 사항인 제어 흐름의 기본 사항에 대해 알아봅니다. 이 자습서는 [머신에서 로컬로 실행](#)할 수도 있습니다.

이 자습서에서는 [Hello World](#) 및 [C#의 숫자](#) 단원을 완료했다고 가정합니다.

## 목록 컬렉션

[목록 컬렉션](#) 단원에서는 데이터 시퀀스를 저장하는 목록 컬렉션 형식을 살펴봅니다. 항목을 추가 및 제거하고, 항목을 검색하고, 목록을 정렬하는 방법을 배웁니다. 여러 종류의 목록을 살펴봅니다. 이 자습서는 [머신에서 로컬로 실행](#)할 수도 있습니다.

이 자습서에서는 위에 나열된 단원을 완료했다고 가정합니다.

## 101 LINQ 샘플 ↗

이 샘플을 사용하려면 [dotnet-try](#) 전역 도구가 필요합니다. 도구를 설치하고 [try-samples](#) 리포지토리를 복제한 후에는 대화형으로 실행할 수 있는 101개 샘플의 세트를 통해 LINQ(Language Integrated Query)를 배울 수 있습니다. 데이터 시퀀스를 쿼리, 탐색 및 변환하는 다양한 방법을 탐색할 수 있습니다.

---

## 피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) ↗

# 로컬 환경 설정

아티클 • 2023. 05. 10.

머신에서 자습서를 실행하는 첫 번째 단계는 개발 환경을 설정하는 것입니다.

- Windows 또는 Mac용 [Visual Studio](#) 를 사용하는 것이 좋습니다. [Visual Studio 다운로드 페이지에서 무료 버전을 다운로드할 수 있습니다](#). Visual Studio에는 .NET SDK가 포함되어 있습니다.
- [Visual Studio Code](#) 편집기를 사용할 수도 있습니다. 최신 [.NET SDK](#) 를 별도로 설치해야 합니다.
- 다른 편집기를 선호하는 경우 최신 [.NET SDK](#) 를 설치해야 합니다.

## 기본 애플리케이션 개발 흐름

이러한 자습서의 지침에서는 .NET CLI를 사용하여 애플리케이션을 만들고, 빌드하고, 실행한다고 가정합니다. 다음 명령을 사용합니다.

- [dotnet new](#)는 애플리케이션 로더를 만듭니다. 이 명령은 애플리케이션에 필요한 파일과 자산을 생성합니다. C# 소개 자습서에서는 모두 `console` 애플리케이션 유형을 모두 사용합니다. 기본 사항을 알고 나면 다른 애플리케이션 유형으로 확장할 수 있습니다.
- [dotnet build](#)는 실행 파일을 빌드합니다.
- [dotnet run](#)은 실행 파일을 실행합니다.

이러한 자습서를 위해 Visual Studio 2019을 사용하는 경우 자습서에서 다음 CLI 명령 중 하나를 실행하도록 지시하는 경우 Visual Studio 메뉴 선택을 선택합니다.

- **파일>새로 만들기>프로젝트**는 애플리케이션을 만듭니다.
  - `Console Application` 프로젝트 템플릿을 권장합니다.
  - 대상 프레임워크를 지정하는 옵션이 제공됩니다. 아래 자습서는 .NET 5 이상을 대상으로 지정할 때 가장 잘 작동합니다.
- **빌드>솔루션 빌드**는 실행 파일을 빌드합니다.
- **디버그>디버깅하지 않고 시작**은 실행 파일을 실행합니다.

## 자습서 선택

다음과 같은 자습서 중 하나를 시작할 수 있습니다.

## C#의 숫자

[C#의 숫자](#) 자습서에서는 컴퓨터가 숫자를 저장하는 방법과 여러 숫자 형식으로 계산을 수행하는 방법을 알아봅니다. 반올림의 기본 사항과 C#을 사용하여 수학 계산을 수행하는 방법을 알아봅니다.

이 자습서에서는 [Hello World](#) 단원을 완료했다고 가정합니다.

## 분기 및 루프

[분기 및 루프](#) 자습서에서는 변수에 저장된 값에 따라 코드 실행의 여러 경로를 선택하는 기본 사항을 설명합니다. 프로그램에서 결정하고 여러 작업을 선택하는 방법에 대한 기본 사항인 제어 흐름의 기본 사항에 대해 알아봅니다.

이 자습서에서는 [Hello World](#) 및 [C#의 숫자](#) 단원을 완료했다고 가정합니다.

## 목록 컬렉션

[목록 컬렉션](#) 단원에서는 데이터 시퀀스를 저장하는 목록 컬렉션 형식을 살펴봅니다. 항목을 추가 및 제거하고, 항목을 검색하고, 목록을 정렬하는 방법을 배웁니다. 여러 종류의 목록을 살펴봅니다.

이 자습서에서는 위에 나열된 단원을 완료했다고 가정합니다.

# C에서 정수 및 부동 소수점 숫자를 사용하는 방법 #

아티클 • 2023. 04. 08.

이 자습서에서는 C#의 숫자 형식에 대해 설명합니다. 작은 양의 코드를 작성한 다음 해당 코드를 컴파일하고 실행합니다. 이 자습서에는 C#의 숫자 및 수학 연산을 살펴보는 일련의 단원이 포함되어 있습니다. 이러한 단원에서는 C# 언어의 기본 사항을 설명합니다.

## 💡 팁

**포커스 모드** 내에 코드 조각을 붙여넣으려면 바로 가기 키(`Ctrl` + `V` 또는 `Cmd` + `V`)를 사용해야 합니다.

## 사전 준비 사항

이 자습서에서는 컴퓨터가 로컬 개발용으로 설정되어 있다고 가정합니다. 설치 지침 및 .NET의 애플리케이션 개발 개요는 [로컬 환경 설정을 참조하세요](#).

로컬 환경을 설정하지 않으려면 [이 자습서의 브라우저 내 대화형 버전을 참조하세요](#).

## 정수 계산 살펴보기

`numbers-quickstart`라는 디렉터리를 만듭니다. 현재 디렉터리로 만들고 다음 명령을 실행합니다.

.NET CLI

```
dotnet new console -n NumbersInCSharp -o .
```

## ⓘ 중요

.NET 6용 C# 템플릿은 '최상위 문'을 사용합니다. .NET 6으로 이미 업그레이드한 경우 애플리케이션이 이 문서의 코드와 일치하지 않을 수 있습니다. 자세한 내용은 [최상위 문을 생성하는 새 C# 템플릿](#)을 참조하세요.

.NET 6 SDK는 다음 SDK를 사용하는 프로젝트에 대한 암시적 `global using` 지시문 집합도 추가합니다.

- Microsoft.NET.Sdk

- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

이러한 암시적 `global using` 지시문에는 해당 프로젝트 형식의 가장 일반적인 네임 스페이스가 포함됩니다.

원하는 편집기에서 `Program.cs`를 열고 파일의 콘텐츠를 다음 코드로 바꿉니다.

C#

```
int a = 18;
int b = 6;
int c = a + b;
Console.WriteLine(c);
```

명령 창에 `dotnet run`을 입력하여 이 코드를 실행합니다.

정수를 사용하는 기본 수학 연산 중 하나를 살펴봤습니다. `int` 형식은 정수(0, 양의 정수 또는 음의 정수)를 나타냅니다. 더하기의 경우 `+` 기호를 사용합니다. 정수에 대해 다른 일반적인 수학 연산은 다음과 같습니다.

- 빼기의 경우 `-`
- 곱하기의 경우 `*`
- 나누기의 경우 `/`

다른 연산을 살펴보세요. `c`의 값을 쓰는 줄 뒤에 다음 줄을 추가합니다.

C#

```
// subtraction
c = a - b;
Console.WriteLine(c);

// multiplication
c = a * b;
Console.WriteLine(c);

// division
c = a / b;
Console.WriteLine(c);
```

명령 창에 `dotnet run`을 입력하여 이 코드를 실행합니다.

원하는 경우 동일한 줄에서 여러 수학 연산을 작성하여 실험할 수도 있습니다. 예를 들어 `c = a + b - 12 * 17;`을 사용해 보세요. 변수와 상수를 혼합해서 사용할 수 있습니다.

## 💡 팁

C# (또는 다른 프로그래밍 언어)를 살펴보면서 코드를 작성할 때 실수를 하게 될 것입니다. 컴파일러는 그러한 오류를 찾아 사용자에게 보고합니다. 출력에 오류 메시지가 포함되어 있으면 예제 코드와 창의 코드를 자세히 살펴보고 수정 사항을 확인하세요. 이 연습은 C# 코드의 구조를 학습하는 데 도움이 됩니다.

첫 번째 단계를 완료했습니다. 다음 섹션을 시작하기 전에 현재 코드를 별도의 메서드로 이동합니다. 메서드는 함께 그룹화되고 이름이 지정된 일련의 문입니다. 메서드 이름 뒤에 `()`를 써서 메서드를 호출합니다. 코드를 메서드로 구성하면 새 예제 작업을 쉽게 시작할 수 있습니다. 작업을 마치면 코드가 다음과 같이 됩니다.

C#

```
WorkWithIntegers();  
  
void WorkWithIntegers()  
{  
    int a = 18;  
    int b = 6;  
    int c = a + b;  
    Console.WriteLine(c);  
  
    // subtraction  
    c = a - b;  
    Console.WriteLine(c);  
  
    // multiplication  
    c = a * b;  
    Console.WriteLine(c);  
  
    // division  
    c = a / b;  
    Console.WriteLine(c);  
}
```

줄 `WorkWithIntegers();`는 메서드를 호출합니다. 다음 코드는 메서드를 선언하고 정의합니다.

## 연산 순서 알아보기

`WorkingWithIntegers()`에 대한 호출을 주석으로 처리합니다. 그러면 이 섹션에서 작업할 때 출력이 덜 복잡해집니다.

C#

```
//WorkWithIntegers();
```

//는 C#에서 주석을 시작합니다. 주석은 소스 코드에 유지하되 코드로 실행하지는 않으려는 모든 텍스트입니다. 컴파일러는 주석에서 실행 코드를 생성하지 않습니다.

WorkWithIntegers()는 메서드이므로 한 줄만 주석으로 처리해야 합니다.

C# 언어는 수학에서 배운 규칙과 일치하는 규칙으로 여러 가지 수학 연산의 우선 순위를 정의합니다. 곱하기와 나누기는 더하기와 빼기보다 우선 순위가 높습니다. 다음 코드를 WorkWithIntegers() 호출 뒤에 추가하고 dotnet run을 실행하여 살펴봅니다.

C#

```
int a = 5;
int b = 4;
int c = 2;
int d = a + b * c;
Console.WriteLine(d);
```

출력에서는 곱하기가 수행된 후 더하기가 수행되었음을 보여 줍니다.

먼저 수행하려는 연산 주위에 괄호를 추가하여 다른 연산 순서를 적용할 수 있습니다. 다음 줄을 추가하고 다시 실행합니다.

C#

```
d = (a + b) * c;
Console.WriteLine(d);
```

여러 다른 연산을 결합하여 자세히 살펴보세요. 다음과 같은 줄을 추가합니다. dotnet run을 다시 시도해 봅니다.

C#

```
d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
Console.WriteLine(d);
```

정수에 대해 흥미로운 동작을 이미 알고 있을 수 있습니다. 정수 나누기는 결과에 소수 또는 소수 부분이 포함될 것으로 예상되는 경우에도 항상 정수 결과를 생성합니다.

이러한 동작을 본 적이 없다면 다음 코드를 시도해 보세요.

C#

```
int e = 7;
int f = 4;
int g = 3;
int h = (e + f) / g;
Console.WriteLine(h);
```

`dotnet run`을 다시 입력하여 결과를 확인합니다.

넘어가기 전에 이 섹션에서 작성한 모든 코드를 새 메서드에 배치해 보겠습니다. 이러한 새 메서드의 이름을 `OrderPrecedence`라고 하겠습니다. 코드는 다음과 비슷합니다.

C#

```
// WorkWithIntegers();
OrderPrecedence();

void WorkWithIntegers()
{
    int a = 18;
    int b = 6;
    int c = a + b;
    Console.WriteLine(c);

    // subtraction
    c = a - b;
    Console.WriteLine(c);

    // multiplication
    c = a * b;
    Console.WriteLine(c);

    // division
    c = a / b;
    Console.WriteLine(c);
}

void OrderPrecedence()
{
    int a = 5;
    int b = 4;
    int c = 2;
    int d = a + b * c;
    Console.WriteLine(d);

    d = (a + b) * c;
    Console.WriteLine(d);

    d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
    Console.WriteLine(d);
```

```
    int e = 7;
    int f = 4;
    int g = 3;
    int h = (e + f) / g;
    Console.WriteLine(h);
}
```

## 정수 전체 자릿수 및 한도 살펴보기

마지막 샘플에서는 정수 나누기가 결과를 자르는 것을 보여 줍니다. **modulo** 연산자(% 문자)를 사용하여 나머지를 얻을 수 있습니다. `OrderPrecedence()` 메서드 호출 뒤에 다음 코드를 시도합니다.

C#

```
int a = 7;
int b = 4;
int c = 3;
int d = (a + b) / c;
int e = (a + b) % c;
Console.WriteLine($"quotient: {d}");
Console.WriteLine($"remainder: {e}");
```

C# 정수 형식은 한 가지 다른 면에서 수학의 정수와 다릅니다. 즉 `int` 형식에는 최소 한도와 최대 한도가 있습니다. 이 코드를 추가하여 해당 한도를 확인합니다.

C#

```
int max = int.MaxValue;
int min = int.MinValue;
Console.WriteLine($"The range of integers is {min} to {max}");
```

계산이 해당 한도를 초과하는 값을 생성하는 경우 **언더플로** 또는 **오버플로** 조건이 발생합니다. 답은 한 한도에서 다른 한도로 래핑하는 것으로 나타납니다. 다음 두 줄을 추가하여 예제를 확인합니다.

C#

```
int what = max + 3;
Console.WriteLine($"An example of overflow: {what}");
```

답은 최소 (음의) 정수와 아주 가깝습니다. `min + 2`와 같습니다. 더하기 연산은 정수에 대해 허용된 값을 **오버플로했습니다**. 오버플로가 가능한 가장 큰 정수에서 가장 작은 정수로 “래핑”하기 때문에 답은 아주 큰 음수입니다.

`int` 형식이 요구 사항을 충족하지 않을 때 사용하는 여러 한도와 전체 자릿수가 있는 다른 숫자 형식이 있습니다. 다음으로 다른 형식을 살펴보겠습니다. 다음 섹션이 시작하기 전에 이 섹션에서 작성한 코드를 별도의 메서드로 옮깁니다. 이 `EventHandler`의 이름을 `TestLimits`로 지정합니다.

## double 형식 작업

`double` 숫자 형식은 배정밀도 부동 소수점 수를 나타냅니다. 이러한 용어는 생소할 수 있습니다. **부동 소수점** 수는 아주 크거나 작은 정수가 아닌 수를 나타낼 때 유용합니다. **배정밀도**는 값을 저장하는 데 사용되는 이진 자릿수를 설명하는 상대 용어입니다. **배정밀도** 숫자의 이진 자릿수는 **단정밀도**의 두 배입니다. 최신 컴퓨터에서는 단정밀도 숫자보다 배정밀도를 더 많이 사용합니다. **단정밀도** 숫자는 `float` 키워드를 사용하여 선언됩니다. 지금 살펴보세요. 다음 코드를 추가하고 결과를 확인합니다.

C#

```
double a = 5;
double b = 4;
double c = 2;
double d = (a + b) / c;
Console.WriteLine(d);
```

답에 몫의 소수 부분이 포함되어 있습니다. `double`을 사용하여 약간 더 복잡한 식을 사용해 보세요.

C#

```
double e = 19;
double f = 23;
double g = 8;
double h = (e + f) / g;
Console.WriteLine(h);
```

`double` 값의 범위는 정수 값보다 훨씬 큽니다. 지금까지 작성한 코드 아래에 다음 코드를 사용해 봅니다.

C#

```
double max = double.MaxValue;
double min = double.MinValue;
Console.WriteLine($"The range of double is {min} to {max}");
```

이 값은 과학적 표기법으로 인쇄됩니다. `E`의 왼쪽에 있는 숫자는 유효 숫자입니다. 오른쪽의 숫자는 지수이며 10의 배수입니다. 수학의 10진수 숫자와 마찬가지로, C#에서

double에는 반올림 오류가 발생할 수 있습니다. 다음 코드를 사용해 보세요.

C#

```
double third = 1.0 / 3.0;
Console.WriteLine(third);
```

한정된 횟수를 반복하는 것은 과정화히 동일  $1/3$ 하지 않다는 것을 알고  $0.3$  있습니다.

## 과제

double 형식을 사용하여 큰 숫자, 작은 숫자, 곱하기 및 나누기로 다른 계산을 수행해 보세요. 더 복잡한 계산을 수행해 보세요. 과제를 하느라 약간의 시간을 보낸 후 작성한 코드를 새 메서드에 배치합니다. 이러한 새 메서드의 이름을 WorkWithDoubles로 지정합니다.

# 10진 형식으로 작업

C#의 기본적인 숫자 형식인 정수 형식과 double 형식을 살펴봤습니다. 학습할 또 다른 형식이 있습니다. 바로 decimal 형식입니다. decimal 형식은 범위가 작지만 double보다 전체 자릿수가 큽니다. 이 형식에 대해 살펴보겠습니다.

C#

```
decimal min = decimal.MinValue;
decimal max = decimal.MaxValue;
Console.WriteLine($"The range of the decimal type is {min} to {max}");
```

범위가 double 형식보다 작습니다. 다음 코드를 사용하여 소수점이 있는 더 큰 전체 자릿수를 확인할 수 있습니다.

C#

```
double a = 1.0;
double b = 3.0;
Console.WriteLine(a / b);

decimal c = 1.0M;
decimal d = 3.0M;
Console.WriteLine(c / d);
```

숫자의 M 접미사는 상수가 decimal 형식을 사용해야 함을 나타내는 방법입니다. 형식을 지정하지 않으면 컴파일러는 double 형식으로 간주합니다.

## ① 참고

문자 `m`은 `double` 키워드와 `decimal` 키워드 사이에서 가장 시각적으로 고유한 문자로 선택되었습니다.

소수점 형식을 사용하는 수학에는 소수점 오른쪽에 더 많은 숫자가 있습니다.

## 과제

이제 여러 가지 숫자 형식을 살펴봤으므로 반지름이 2.50센티미터인 원의 면적을 계산하는 코드를 작성하세요. 원의 면적은 반지름 제곱 곱하기 PI입니다. 힌트: .NET에는 PI의 상수가 포함되어 있습니다. 즉 해당 값에 사용할 수 있는 `Math.PI`입니다. `System.Math` 네임 스페이스에 선언된 모든 상수와 마찬가지로 `Math.PI`는 `double` 값입니다. 이러한 이유로 이 과제에는 `decimal` 값 대신 `double`을 사용해야 합니다.

19에서 20 사이의 답을 받아야 합니다. [GitHub에서 완성된 샘플 코드를 보고 ↗](#) 답을 확인할 수 있습니다.

원하는 경우 다른 수식을 사용해 보세요.

"C#의 숫자" 빠른 시작을 완료했습니다. 자체 개발 환경에서 [분기 및 루프](#) 빠른 시작을 계속할 수 있습니다.

다음 문서에서는 C#의 숫자에 대해 더 자세히 알아볼 수 있습니다.

- 정수 숫자 형식
- 부동 소수점 숫자 형식
- 기본 제공 숫자 변환

# C# if 문 및 루프 - 조건부 논리 자습서

아티클 • 2023. 04. 08.

이 자습서에서는 변수를 검사하고 해당 변수에 따라 실행 경로를 변경하는 C# 코드를 작성하는 방법을 설명합니다. C# 코드를 작성하고 컴파일 및 실행 결과를 확인합니다. 이 자습서에는 C#에서 분기 및 루프 구문을 살펴보는 일련의 단원이 포함되어 있습니다. 이러한 단원에서는 C# 언어의 기본 사항을 설명합니다.

## 💡 팁

포커스 모드 내에 코드 조각을 붙여넣으려면 바로 가기 키(`Ctrl` + `V` 또는 `cmd` + `V`)를 사용해야 합니다.

## 사전 준비 사항

이 자습서에서는 컴퓨터가 로컬 개발용으로 설정되어 있다고 가정합니다. 설치 지침 및 .NET의 애플리케이션 개발 개요는 [로컬 환경 설정을 참조하세요](#).

로컬 환경을 설정하지 않고 코드를 실행하려는 경우 [이 자습서의 브라우저 내 대화형 버전을 참조하세요](#).

## if 문을 사용하여 결정하기

`branches-tutorial`이라는 디렉터리를 만듭니다. 현재 디렉터리로 만들고 다음 명령을 실행합니다.

.NET CLI

```
dotnet new console -n BranchesAndLoops -o .
```

### ⓘ 중요

.NET 6용 C# 템플릿은 '최상위 문'을 사용합니다. .NET 6으로 이미 업그레이드한 경우 애플리케이션이 이 문서의 코드와 일치하지 않을 수 있습니다. 자세한 내용은 [최상위 문을 생성하는 새 C# 템플릿을 참조하세요](#).

.NET 6 SDK는 다음 SDK를 사용하는 프로젝트에 대한 암시적 `global using` 지시문 집합도 추가합니다.

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

이러한 암시적 `global using` 지시문에는 해당 프로젝트 형식의 가장 일반적인 네임 스페이스가 포함됩니다.

이 명령은 현재 디렉터리에 새 .NET 콘솔 애플리케이션을 만듭니다. 원하는 편집기에서 `Program.cs`를 열고 콘텐츠를 다음 코드로 대체합니다.

C#

```
int a = 5;
int b = 6;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10.");
```

콘솔 창에 `dotnet run`을 입력하여 이 코드를 사용해 봅니다. 콘솔에 "답변이 10보다 큽니다."라는 메시지가 표시됩니다. 합계가 10보다 작도록 `b`의 선언을 수정합니다.

C#

```
int b = 3;
```

`dotnet run`을 다시 입력합니다. 답이 10보다 작기 때문에 아무것도 출력되지 않습니다. 테스트하는 조건은 `false`입니다. `if` 문에 대해 가능한 분기 중 하나(`true` 분기)만 작성했기 때문에 실행할 코드가 없습니다.

### 💡 팁

C# (또는 다른 프로그래밍 언어)를 살펴보면서 코드를 작성할 때 실수를 하게 될 것입니다. 컴파일러가 오류를 찾아 보고합니다. 오류 출력 및 오류를 생성한 코드를 자세히 살펴봅니다. 일반적으로 컴파일러 오류는 문제를 찾는데 도움이 될 수 있습니다.

이 첫 번째 샘플에서는 `if`의 기능과 부울 형식을 보여 줍니다. 부울은 `true` 또는 `false`의 두 값 중 하나를 가질 수 있는 변수입니다. C#은 부울 변수에 대한 특수 형식 `bool`을 정의합니다. `if` 문은 `bool`의 값을 확인합니다. 값이 `true`인 경우 `if` 뒤의 문이 실행됩니다. 그렇지 않으면 건너뜁니다. 조건을 확인하고 해당 조건에 따라 문을 실행하는 이 프로세스는 강력합니다.

# if와 else를 함께 사용하기

true 분기와 false 분기의 여러 코드를 실행하려면 조건이 false일 때 실행되는 `else` 분기 를 생성합니다. `else` 분기를 시도합니다. 아래 코드의 마지막 두 줄을 추가합니다(처음 네 줄은 이미 있음).

C#

```
int a = 5;
int b = 3;
if (a + b > 10)
    Console.WriteLine("The answer is greater than 10");
else
    Console.WriteLine("The answer is not greater than 10");
```

`else` 키워드 뒤의 문은 테스트하는 조건이 `false`인 경우에만 실행됩니다. `if` 및 `else`를 부울 조건과 결합하면 `true`와 `false` 조건을 모두 처리하는 데 필요한 모든 기능이 제공 됩니다.

## ① 중요

`if` 및 `else` 문 아래의 들여쓰기는 사용자가 보기 편하도록 하기 위함입니다. C# 언어는 들여쓰기 또는 공백을 중요하게 취급하지 않습니다. `if` 또는 `else` 키워드 뒤의 문은 조건에 따라 실행됩니다. 이 자습서의 모든 샘플에서는 문의 제어 흐름을 기준 으로 줄을 들여쓰는 일반적인 방법을 따릅니다.

들여쓰기는 중요하지 않기 때문에 `{` 및 `}`를 사용하여 두 개 이상의 문이 조건부로 실행 되는 블록의 일부가 되는 시기를 나타내야 합니다. C# 프로그래머는 일반적으로 모든 `if` 및 `else` 절에서 중괄호를 사용합니다. 다음 예제는 앞서 작성한 코드와 같습니다. 다음 코드와 일치하도록 위의 코드를 수정합니다.

C#

```
int a = 5;
int b = 3;
if (a + b > 10)
{
    Console.WriteLine("The answer is greater than 10");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
}
```

## 💡 팁

이 자습서의 나머지 부분에서 코드 샘플에는 일반적인 방법에 따라 모두 중괄호가 포함되어 있습니다.

더 복잡한 조건을 테스트할 수 있습니다. 지금까지 작성한 코드 뒤에 다음 코드를 추가합니다.

C#

```
int c = 4;
if ((a + b + c > 10) && (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("And the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("Or the first number is not equal to the second");
}
```

== 기호는 같음을 테스트합니다. ==을 사용하면 같음 테스트가 a = 5에서 확인한 할당과 구분됩니다.

&&는 "and"를 나타냅니다. true 분기에서 문을 실행하려면 두 조건이 모두 true여야 합니다. 이러한 예제에서는 { 및 }로 문을 묶으면 각 조건부 분기에 여러 문을 가질 수 있음도 보여 줍니다. 를 사용하여 || "or"를 나타낼 수도 있습니다. 지금까지 작성한 코드 뒤에 다음 코드를 추가합니다.

C#

```
if ((a + b + c > 10) || (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("Or the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("And the first number is not equal to the second");
}
```

a, b 및 c의 값을 수정하고 && 및 || 간에 전환하여 살펴봅니다. && 및 || 연산자가 어떻게 작동하는지 더 잘 이해할 수 있습니다.

첫 번째 단계를 완료했습니다. 다음 섹션을 시작하기 전에 현재 코드를 별도의 메서드로 이동합니다. 이렇게 하면 새 예제 작업을 쉽게 시작할 수 있습니다. `ExploreIf()`라는 메서드에 기존 코드를 배치합니다. 프로그램 위에서 호출합니다. 변경을 완료하면 코드가 다음과 같이 표시됩니다.

C#

```
ExploreIf();  
  
void ExploreIf()  
{  
    int a = 5;  
    int b = 3;  
    if (a + b > 10)  
    {  
        Console.WriteLine("The answer is greater than 10");  
    }  
    else  
    {  
        Console.WriteLine("The answer is not greater than 10");  
    }  
  
    int c = 4;  
    if ((a + b + c > 10) && (a > b))  
    {  
        Console.WriteLine("The answer is greater than 10");  
        Console.WriteLine("And the first number is greater than the  
second");  
    }  
    else  
    {  
        Console.WriteLine("The answer is not greater than 10");  
        Console.WriteLine("Or the first number is not greater than the  
second");  
    }  
  
    if ((a + b + c > 10) || (a > b))  
    {  
        Console.WriteLine("The answer is greater than 10");  
        Console.WriteLine("Or the first number is greater than the second");  
    }  
    else  
    {  
        Console.WriteLine("The answer is not greater than 10");  
        Console.WriteLine("And the first number is not greater than the  
second");  
    }  
}
```

`ExploreIf()`에 대한 호출을 주석으로 처리합니다. 그러면 이 섹션에서 작업할 때 출력이 덜 복잡해집니다.

C#

```
//ExploreIf();
```

//는 C#에서 주석을 시작합니다. 주석은 소스 코드에 유지하되 코드로 실행하지는 않으려는 모든 텍스트입니다. 컴파일러는 주석에서 실행 코드를 생성하지 않습니다.

## 루프를 사용하여 작업 반복

이 섹션에서는 루프를 사용하여 문을 반복합니다. ExploreIf 호출 후 이 코드를 추가합니다.

C#

```
int counter = 0;
while (counter < 10)
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
}
```

while 문은 조건을 검사하고 while 뒤에 있는 문 또는 문 블록을 실행합니다. 조건이 false가 될 때까지 해당 문을 실행하여 조건을 반복적으로 확인합니다.

이 예제에서는 다른 새 연산자가 하나 있습니다. counter 변수 뒤의 ++는 증가 연산자입니다. 이 연산자는 counter의 값에 1을 더하고 해당 값을 counter 변수에 저장합니다.

### ⓘ 중요

코드를 실행할 때 while 루프 조건이 false로 바뀌는지 확인합니다. 그러하지 않으면 프로그램이 종료되지 않는 무한 루프를 생성합니다. 이러한 내용이 이 샘플에 설명되어 있지는 않은데, CTRL-C 또는 다른 방법을 사용하여 프로그램을 강제 종료해야 하기 때문입니다.

while 루프는 while 뒤에 코드를 실행하기 전에 조건을 테스트합니다. do ... while 루프는 코드를 먼저 실행한 후 조건을 확인합니다. do while 루프는 다음 코드에 나와 있습니다.

C#

```
int counter = 0;
do
{
```

```
Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
} while (counter < 10);
```

이 `do` 루프 및 이전 `while` 루프는 같은 출력을 생성합니다.

## for 루프 작업

C#에서는 일반적으로 `for` 루프가 사용됩니다. 다음 코드를 사용해 보세요.

C#

```
for (int index = 0; index < 10; index++)
{
    Console.WriteLine($"Hello World! The index is {index}");
}
```

이전 코드는 `while` 루프 및 이미 사용한 `do` 루프와 동일한 작업을 수행합니다. `for` 문에는 작동 방식을 제어하는 세 부분이 있습니다.

첫 번째 부분은 **for 이니셜라이저입니다**. `int index = 0;` 은 `index`가 루프 변수임을 선언하고 첫 번째 값을 `0`으로 설정합니다.

중간 부분은 **for 조건입니다**. `index < 10`은 이 `for` 루프가 카운터 값이 10보다 작으면 계속 실행됨을 선언합니다.

마지막 부분은 **for 반복기입니다**. `index++`는 `for` 문 다음의 블록을 실행한 후 루프 변수를 수정하는 방법을 지정합니다. 여기서 `index`는 블록이 실행될 때마다 1씩 증가하도록 지정합니다.

직접 실험해 보세요. 다음 변형을 각각 시도합니다.

- 다른 값으로 시작하도록 이니셜라이저를 변경합니다.
- 다른 값에서 중지하도록 조건을 변경합니다.

완료하면, 학습한 내용을 토대로 직접 코드를 작성해 보겠습니다.

이 자습서에서 다루지 않은 다른 반복 문이 하나 있는데, `foreach` 문이 그것입니다.

`foreach` 문은 항목 시퀀스의 모든 항목에 대해 해당 문을 반복합니다. 컬렉션과 함께 사용되는 경우가 가장 많으므로 다음 자습서에서 설명합니다.

## 중첩 루프 만들기

`while`, `do` 또는 `for` 루프를 다른 루프 내에 중첩하여 내부 루프에 있는 각 항목과 외부 루프에 있는 각 항목의 조합을 사용하여 행렬을 만들 수 있습니다. 행과 열을 나타내는 영 숫자 쌍 세트를 작성하겠습니다.

하나의 `for` 루프가 행을 생성할 수 있습니다.

C#

```
for (int row = 1; row < 11; row++)
{
    Console.WriteLine($"The row is {row}");
}
```

다른 루프는 열을 생성할 수 있습니다.

C#

```
for (char column = 'a'; column < 'k'; column++)
{
    Console.WriteLine($"The column is {column}");
}
```

한 루프를 다른 루프 안에 중첩하여 쌍을 구성할 수 있습니다.

C#

```
for (int row = 1; row < 11; row++)
{
    for (char column = 'a'; column < 'k'; column++)
    {
        Console.WriteLine($"The cell is ({row}, {column})");
    }
}
```

내부 루프의 전체 실행마다 외부 루프가 한 번씩 증가하는 것을 볼 수 있습니다. 행과 열 중첩을 반대로 바꾸고 변경 내용을 직접 확인하세요. 완료되면 `ExploreLoops()`라는 메서드에 이 섹션의 코드를 배치합니다.

## 분기 및 루프 결합

이제 C# 언어로 된 `if` 문과 루프 구조를 확인했습니다. C# 코드를 작성하여 3으로 나눌 수 있는, 1에서 20까지의 모든 정수의 합계를 찾을 수 있는지 확인해 보세요. 다음은 몇 가지 힌트입니다.

- `%` 연산자는 나누기 연산의 나머지를 제공합니다.

- `if` 문은 숫자가 합계의 일부여야 하는지를 확인하는 조건을 제공합니다.
- `for` 루프는 1에서 20까지의 모든 숫자에 대해 일련의 단계를 반복하는 데 도움이 됩니다.

직접 시도해 보세요. 그런 다음 어떻게 했는지 확인하세요. 답으로 63을 받아야 합니다. [GitHub에서 완성된 코드를 보고](#) 가능한 한 가지 답을 확인할 수 있습니다.

“분기 및 루프” 자습서를 완료했습니다.

자체 개발 환경에서 [배열 및 컬렉션](#) 자습서를 계속할 수 있습니다.

다음 문서에서 이러한 개념을 더 자세히 알아볼 수 있습니다.

- [선택 문](#)
- [반복 문](#)

# C에서 목록<T>를 사용하여 데이터 수집을 관리하는 방법 알아보기 #

아티클 • 2023. 04. 08.

이 소개 자습서에서는 C# 언어 및 [List<T>](#) 클래스의 기본 사항을 소개합니다.

## 사전 준비 사항

이 자습서에서는 컴퓨터가 로컬 개발용으로 설정되어 있다고 가정합니다. 설치 지침 및 .NET의 애플리케이션 개발 개요는 [로컬 환경 설정을 참조하세요](#).

로컬 환경을 설정하지 않고 코드를 실행하려는 경우 [이 자습서의 브라우저 내 대화형 버전을 참조하세요](#).

## 기본 목록 예제

*list-tutorial*이라는 디렉터리를 만듭니다. 현재 디렉터리로 지정하고 `dotnet new console`을 실행합니다.

### ① 중요

.NET 6 용 C# 템플릿은 '최상위 문'을 사용합니다. .NET 6으로 이미 업그레이드한 경우 애플리케이션이 이 문서의 코드와 일치하지 않을 수 있습니다. 자세한 내용은 [최상위 문을 생성하는 새 C# 템플릿을 참조하세요](#).

.NET 6 SDK는 다음 SDK를 사용하는 프로젝트에 대한 암시적 `global using` 지시문 집합도 추가합니다.

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

이러한 암시적 `global using` 지시문에는 해당 프로젝트 형식의 가장 일반적인 네임 스페이스가 포함됩니다.

편집기에서 *Program.cs*를 열고 기존 코드를 다음으로 바꿉니다.

C#

```
var names = new List<string> { "<name>", "Ana", "Felipe" };
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

<name>을 사용자의 이름으로 바꿉니다. *Program.cs*를 저장합니다. 콘솔 창에 `dotnet run`을 입력하여 시도해 보세요.

문자열 목록을 만들고, 해당 목록에 세 개의 이름을 추가하고, 모든 CAPS에 이름을 인쇄했습니다. 이전 자습서에서 학습한 개념을 사용하여 목록을 반복합니다.

이름을 표시하는 코드는 [문자열 보간](#) 기능을 사용합니다. `string` 앞에 `$` 문자를 넣으면 문자열 선언에 C# 코드를 포함할 수 있습니다. 실제 문자열은 C# 코드를 생성하는 값으로 바꿉니다. 이 예제에서는 `ToUpper` 메서드를 호출했기 때문에 `{name.ToUpper()}`를 대문자로 변환된 각 이름으로 바꿉니다.

계속해서 살펴보겠습니다.

## 목록 콘텐츠 수정

생성한 컬렉션은 `List<T>` 형식을 사용합니다. 이 형식은 요소의 시퀀스를 저장합니다. 꺾쇠 괄호 사이의 요소 형식을 지정합니다.

이 `List<T>` 형식은 놀리거나 줄일 수 있어 요소를 추가하거나 제거할 수 있습니다. 이 코드를 프로그램 끝에 추가합니다.

C#

```
Console.WriteLine();
names.Add("Maria");
names.Add("Bill");
names.Remove("Ana");
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

목록 끝에 이름을 두 개 더 추가했습니다. 또한 이름을 하나 제거했습니다. 파일을 저장하고 `dotnet run`을 입력하여 시도해 보세요.

`List<T>`를 사용하면 [인덱스별로](#) 각 항목을 참조할 수도 있습니다. 목록 이름 뒤 [ 와 ] 토큰 사이에 인덱스를 배치합니다. C#은 첫 번째 인덱스에 0을 사용합니다. 방금 추가한 코드 바로 아래에 이 코드를 추가하여 시도합니다.

C#

```
Console.WriteLine($"My name is {names[0]}");
Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");
```

목록의 끝을 벗어나는 인덱스에 액세스할 수 없습니다. 인덱스는 0부터 시작하므로, 가장 큰 유효 인덱스는 목록의 항목 수보다 하나 작습니다. [Count 속성을 사용하여 목록의 길이를 확인할 수 있습니다.](#) 프로그램 끝에 다음 코드를 추가합니다.

C#

```
Console.WriteLine($"The list has {names.Count} people in it");
```

파일을 저장하고 `dotnet run`을 다시 입력하여 결과를 확인합니다.

## 목록 검색 및 정렬

샘플에서는 상대적으로 작은 목록을 사용하지만 애플리케이션에서는 수천에 달하는 많은 요소가 포함된 목록을 작성할 수 있습니다. 이러한 큰 컬렉션에서 요소를 찾으려면 여러 항목의 목록을 검색해야 합니다. [IndexOf 메서드는 항목을 검색하고 항목의 인덱스를 반환합니다.](#) 목록에 항목이 없으면 `IndexOf`가 `-1`을 반환합니다. 프로그램 맨 아래에 이 코드를 추가합니다.

C#

```
var index = names.IndexOf("Felipe");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns
{index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");
}

index = names.IndexOf("Not Found");
if (index == -1)
{
    Console.WriteLine($"When an item is not found, IndexOf returns
{index}");
}
else
{
    Console.WriteLine($"The name {names[index]} is at index {index}");
}
```

목록의 항목도 정렬할 수 있습니다. [Sort](#) 메서드는 일반적인 순서(문자열의 경우 사전순)로 목록의 모든 항목을 정렬합니다. 프로그램 맨 아래에 이 코드를 추가합니다.

C#

```
names.Sort();
foreach (var name in names)
{
    Console.WriteLine($"Hello {name.ToUpper()}!");
}
```

파일을 저장하고 `dotnet run`을 입력하여 이 최신 버전을 사용해 보세요.

다음 섹션을 시작하기 전에 현재 코드를 별도의 메서드로 이동합니다. 이렇게 하면 새 예제 작업을 쉽게 시작할 수 있습니다. 작성한 모든 코드를 `WorkWithStrings()`라는 새 메서드에 넣습니다. 프로그램 위에서 해당 메서드를 호출합니다. 작업을 마치면 코드가 다음과 같이 됩니다.

C#

```
WorkWithStrings();

void WorkWithStrings()
{
    var names = new List<string> { "<name>", "Ana", "Felipe" };
    foreach (var name in names)
    {
        Console.WriteLine($"Hello {name.ToUpper()}!");
    }

    Console.WriteLine();
    names.Add("Maria");
    names.Add("Bill");
    names.Remove("Ana");
    foreach (var name in names)
    {
        Console.WriteLine($"Hello {name.ToUpper()}!");
    }

    Console.WriteLine($"My name is {names[0]}");
    Console.WriteLine($"I've added {names[2]} and {names[3]} to the list");

    Console.WriteLine($"The list has {names.Count} people in it");

    var index = names.IndexOf("Felipe");
    if (index == -1)
    {
        Console.WriteLine($"When an item is not found, IndexOf returns
{index}");
    }
}
```

```

    }

    else
    {
        Console.WriteLine($"The name {names[index]} is at index {index}");

    }

    index = names.IndexOf("Not Found");
    if (index == -1)
    {
        Console.WriteLine($"When an item is not found, IndexOf returns
{index}");
    }
    else
    {
        Console.WriteLine($"The name {names[index]} is at index {index}");

    }

    names.Sort();
    foreach (var name in names)
    {
        Console.WriteLine($"Hello {name.ToUpper()}!");
    }
}

```

## 다른 형식 목록

지금까지 목록에 `string` 형식을 사용했습니다. 다른 형식을 사용하여 `List<T>`를 만들어 보겠습니다. 숫자 집합을 빌드하겠습니다.

`WorkWithStrings()`를 호출한 후 다음을 프로그램에 추가합니다.

C#

```
var fibonacciNumbers = new List<int> {1, 1};
```

정수 목록을 만들고 처음 두 정수를 값 1로 설정합니다. 숫자 시퀀스인 **피보나치 시퀀스**의 첫 번째 두 값입니다. 다음 각 피보나치 수는 이전의 두 수의 합계를 사용하여 찾습니다. 이 코드를 추가합니다.

C#

```

var previous = fibonacciNumbers[fibonacciNumbers.Count - 1];
var previous2 = fibonacciNumbers[fibonacciNumbers.Count - 2];

fibonacciNumbers.Add(previous + previous2);

foreach (var item in fibonacciNumbers)
{

```

```
        Console.WriteLine(item);
    }
```

파일을 저장하고 `dotnet run`을 입력하여 결과를 확인합니다.

### 💡 팁

이 섹션에만 집중하려면 `WorkWithStrings();`를 호출하는 코드를 주석으로 처리할 수 있습니다. 두 `/` 문자를 다음과 같이 `// WorkWithStrings();` 호출 앞에 놓습니다.

## 과제

이 단원과 이전 단원에서 학습한 개념을 함께 적용할 수 있는지 확인하세요. 피보나치 수를 사용하여 지금까지 빌드한 내용을 확장합니다. 코드를 작성하여 시퀀스에서 처음 20 개 수를 생성합니다. (힌트: 20번째 피보나치 수는 6765입니다.)

## 과제 완료

[GitHub에서 완료된 샘플 코드를 보고](#) 예제 솔루션을 확인할 수 있습니다.

루프의 각 반복을 통해 목록의 마지막 두 정수를 사용하고, 더하고, 해당 값을 목록에 추가합니다. 목록에 20개의 항목이 추가될 때까지 루프가 반복됩니다.

축하합니다. 목록 자습서를 완료했습니다. 사용자의 개발 환경에서 [추가](#) 자습서를 계속 진행할 수 있습니다.

.NET 기본 사항 문서의 [컬렉션](#)에서 `List` 형식 작업에 대해 자세히 알아볼 수 있습니다. 다른 많은 컬렉션 형식에 대해서도 학습합니다.

# C# 프로그램의 일반적인 구조체

아티클 • 2024. 02. 15.

C# 프로그램은 하나 이상의 파일로 구성됩니다. 각 파일은 0개 이상의 네임스페이스가 포함합니다. 네임스페이스는 클래스, 구조체, 인터페이스, 열거형 및 대리자와 같은 형식이나 다른 네임스페이스를 포함합니다. 다음 예제는 이러한 모든 요소를 포함하는 C# 프로그램의 기본 구조입니다.

```
C#  
  
// A skeleton of a C# program  
using System;  
  
// Your program starts here:  
Console.WriteLine("Hello world!");  
  
namespace YourNamespace  
{  
    class YourClass  
    {  
    }  
  
    struct YourStruct  
    {  
    }  
  
    interface IYourInterface  
    {  
    }  
  
    delegate int YourDelegate();  
  
    enum YourEnum  
    {  
    }  
  
    namespace YourNestedNamespace  
    {  
        struct YourStruct  
        {  
        }  
    }  
}
```

앞의 예제에서는 프로그램의 진입점에 대해 `최상위` 문을 사용합니다. 다음 예제와 같이 프로그램의 진입점으로 `Main(이)`라는 정적 메서드를 만들 수도 있습니다.

```
C#
```

```

// A skeleton of a C# program
using System;
namespace YourNamespace
{
    class YourClass
    {
    }

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }

    namespace YourNestedNamespace
    {
        struct YourStruct
        {
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            //Your program starts here...
            Console.WriteLine("Hello world!");
        }
    }
}

```

## 관련 섹션

기본 사항 가이드의 형식 섹션에서 이러한 프로그램 요소에 대해 알아봅니다.

- [클래스](#)
- [구조체](#)
- [네임스페이스](#)
- [인터페이스](#)
- [열거형](#)
- [대리자](#)

# C# 언어 사양

자세한 내용은 [C# 언어 사양의 기본 개념](#)을 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

💡 설명서 문제 열기

↗️ 제품 사용자 의견 제공

# Main()과 명령줄 인수

아티클 • 2024. 03. 14.

`Main` 메서드는 C# 애플리케이션의 진입점입니다. (라이브러리와 서비스에는 `Main` 메서드가 진입점으로 필요하지 않습니다.) 애플리케이션이 시작될 때 `Main` 메서드는 호출되는 첫 번째 메서드입니다.

C# 프로그램에는 하나의 진입점만 있을 수 있습니다. `Main` 메서드가 있는 클래스가 둘 이상 있는 경우 `StartupObject` 컴파일러 옵션으로 프로그램을 컴파일하여 진입점으로 사용할 `Main` 메서드를 지정해야 합니다. 자세한 내용은 [StartupObject\(C# 컴파일러 옵션\)](#)를 참조하세요.

C#

```
class TestClass
{
    static void Main(string[] args)
    {
        // Display the number of command line arguments.
        Console.WriteLine(args.Length);
    }
}
```

한 파일의 [최상위 문](#)을 애플리케이션의 진입점으로 사용할 수도 있습니다.

C#

```
using System.Text;

StringBuilder builder = new();
builder.AppendLine("Hello");
builder.AppendLine("World!");

Console.WriteLine(builder.ToString());
```

## 개요

- `Main` 메서드는 실행 가능한 프로그램의 진입점으로, 프로그램의 제어가 시작되고 끝나는 위치합니다.
- `Main`은 클래스 또는 구조체 내부에 선언됩니다. `Main`은 `static`이어야 하며 `public`일 필요는 없습니다. (이전 예제에서는 .의 `private`기본 액세스를 받습니다.) 은 뒤에 `class` 수 있습니다 `static`.
- `Main`은 `void`, `int`, `Task` 또는 `Task<int>` 반환 형식을 가질 수 있습니다.

- `Main`에서 `Task` 또는 `Task<int>`을 반환하는 경우에만 `Main` 선언에 `async` 한정자가 포함될 수 있습니다. 이는 특히 `async void Main` 메서드를 제외합니다.
- `Main` 메서드는 명령줄 인수를 포함하는 `string[]` 매개 변수 사용 여부에 관계 없이 선언될 수 있습니다. Visual Studio를 사용하여 Windows 애플리케이션을 만드는 경우 매개 변수를 수동으로 추가하거나 `GetCommandLineArgs()` 메서드를 사용하여 명령줄 인수를 가져올 수 있습니다. 매개 변수는 0부터 시작하는 명령줄 인수로 읽힙니다. C 및 C++와 달리, 프로그램의 이름이 `args` 배열의 첫 번째 명령줄 인수로 처리되지 않지만, `GetCommandLineArgs()` 메서드의 첫 번째 요소입니다.

다음 목록은 유효한 `Main` 서명을 보여 줍니다.

C#

```
public static void Main() { }
public static int Main() { }
public static void Main(string[] args) { }
public static int Main(string[] args) { }
public static async Task Main() { }
public static async Task<int> Main() { }
public static async Task Main(string[] args) { }
public static async Task<int> Main(string[] args) { }
```

앞의 예에서는 모두 `public` 접근자 한정자를 사용합니다. 이는 일반적이지만 필수는 아닙니다.

`async` 및 `Task`, `Task<int>` 반환 형식을 추가하면 콘솔 애플리케이션을 시작해야 하고 비동기 작업을 `Main`에서 `await`해야 하는 경우에 프로그램 코드가 간소화됩니다.

## Main() 반환 값

다음 방법 중 하나로 메서드를 정의하여 `Main` 메서드에서 `int`를 반환할 수 있습니다.

[+] 테이블 확장

Main 메서드 코드	Main 서명
<code>args</code> 또는 <code>await</code> 사용 안 함	<code>static int Main()</code>
<code>args</code> 사용, <code>await</code> 사용 안 함	<code>static int Main(string[] args)</code>
<code>args</code> 사용 안 함, <code>await</code> 사용	<code>static async Task&lt;int&gt; Main()</code>
<code>args</code> 및 <code>await</code> 사용	<code>static async Task&lt;int&gt; Main(string[] args)</code>

`Main`의 반환 값을 사용하지 않는 경우 `void` 또는 `Task`를 반환하면 코드가 다소 단순해집니다.

## ☰ 테이블 확장

Main 메서드 코드	Main 서명
<code>args</code> 또는 <code>await</code> 사용 안 함	<code>static void Main()</code>
<code>args</code> 사용, <code>await</code> 사용 안 함	<code>static void Main(string[] args)</code>
<code>args</code> 사용 안 함, <code>await</code> 사용	<code>static async Task Main()</code>
<code>args</code> 및 <code>await</code> 사용	<code>static async Task Main(string[] args)</code>

그러나 `int` 또는 `Task<int>`를 반환하면 프로그램이 실행 파일을 호출하는 다른 프로그램이나 스크립트에 상태 정보를 전달할 수 있습니다.

다음 예제에서는 프로세스의 종료 코드에 액세스할 수 있는 방법을 보여줍니다.

이 예제에서는 [.NET Core 명령줄 도구](#)를 사용합니다. .NET Core 명령줄 도구에 대해 잘 모르는 경우 이 [시작 문서](#)에서 알아볼 수 있습니다.

`dotnet new console`을 실행하여 새 애플리케이션을 만듭니다. `Program.cs`에서 `Main` 메서드를 다음과 같이 수정합니다.

```
C#  
  
// Save this program as MainRetValTest.cs.  
class MainRetValTest  
{  
    static int Main()  
    {  
        //...  
        return 0;  
    }  
}
```

Windows에서 프로그램을 실행하는 경우 `Main` 함수에서 반환된 값은 환경 변수에 저장됩니다. 이 환경 변수는 배치 파일에서 `ERRORLEVEL`을 사용하거나 PowerShell에서 `$LastExitCode`를 사용하여 검색할 수 있습니다.

`dotnet CLI` `dotnet build` 명령을 사용하여 애플리케이션을 빌드할 수 있습니다.

다음으로 애플리케이션을 실행하고 결과를 표시하는 PowerShell 스크립트를 만듭니다. 다음 코드를 텍스트 파일에 붙여넣고 이 파일을 프로젝트가 포함된 폴더에 `test.ps1`로

저장합니다. PowerShell 프롬프트에 `test.ps1`을 입력하여 PowerShell 스크립트를 실행합니다.

코드에서 0을 반환하기 때문에 배치 파일이 성공했다고 보고합니다. 그러나 0이 아닌 값을 반환하도록 `MainReturnValTest.cs`를 변경한 다음 프로그램을 다시 컴파일하면 다음에 PowerShell 스크립트를 실행할 때 오류가 보고됩니다.

PowerShell

```
dotnet run
if ($LastExitCode -eq 0) {
    Write-Host "Execution succeeded"
} else {
    Write-Host "Execution Failed"
}
Write-Host "Return value = " $LastExitCode
```

출력

```
Execution succeeded
Return value = 0
```

## 비동기 Main 반환 값

`Main`에 대한 `async` 반환 값을 선언하면 컴파일러는 `Main`에서 비동기 메서드를 호출하기 위한 상용구 코드를 생성합니다. `async` 키워드를 지정하지 않으면 다음 예와 같이 해당 코드를 직접 작성해야 합니다. 예의 코드는 비동기 작업이 완료될 때까지 프로그램이 실행되도록 보장합니다.

C#

```
class AsyncMainReturnValTest
{
    public static void Main()
    {
        AsyncConsoleWork().GetAwaiter().GetResult();
    }

    private static async Task<int> AsyncConsoleWork()
    {
        // Main body here
        return 0;
    }
}
```

이 상용구 코드는 다음으로 바뀔 수 있습니다.

C#

```
class Program
{
    static async Task<int> Main(string[] args)
    {
        return await AsyncConsoleWork();
    }

    private static async Task<int> AsyncConsoleWork()
    {
        // main body here
        return 0;
    }
}
```

Main 을 `async`로 선언하면 컴파일러가 항상 올바른 코드를 생성한다는 이점이 있습니다.

애플리케이션 진입점에서 `Task` 또는 `Task<int>`를 반환하는 경우 컴파일러는 애플리케이션 코드에서 선언된 진입점 메서드를 호출하는 새 진입점을 생성합니다. 이 진입점이 `$GeneratedMain`이라고 가정하면 컴파일러는 이러한 진입점에 대해 다음 코드를 생성합니다.

- `static Task Main()` - 컴파일러에서 `private static void $GeneratedMain() => Main().GetAwaiter().GetResult();`에 해당하는 코드를 내보냅니다.
- `static Task Main(string[])` - 컴파일러에서 `private static void $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`에 해당하는 코드를 내보냅니다.
- `static Task<int> Main()` - 컴파일러에서 `private static int $GeneratedMain() => Main().GetAwaiter().GetResult();`에 해당하는 코드를 내보냅니다.
- `static Task<int> Main(string[])` - 컴파일러에서 `private static int $GeneratedMain(string[] args) => Main(args).GetAwaiter().GetResult();`에 해당하는 코드를 내보냅니다.

### ① 참고

예제에서 `Main` 메서드에 `async` 키워드를 사용하더라도 컴파일러는 동일한 코드를 생성합니다.

## 명령줄 인수

다음 방법 중 하나로 메서드를 정의하여 인수를 `Main` 메서드에 보낼 수 있습니다.

#### □ 테이블 확장

Main 메서드 코드	Main 서명
반환 값 없음, <code>await</code> 사용 없음	<code>static void Main(string[] args)</code>
반환 값, <code>await</code> 사용 없음	<code>static int Main(string[] args)</code>
반환 값 없음, <code>await</code> 사용	<code>static async Task Main(string[] args)</code>
반환 값, <code>await</code> 사용	<code>static async Task&lt;int&gt; Main(string[] args)</code>

인수가 사용되지 않는 경우 약간 더 간단한 코드를 위해 메서드 서명에서 `args`를 생략할 수 있습니다.

#### □ 테이블 확장

Main 메서드 코드	Main 서명
반환 값 없음, <code>await</code> 사용 없음	<code>static void Main()</code>
반환 값, <code>await</code> 사용 없음	<code>static int Main()</code>
반환 값 없음, <code>await</code> 사용	<code>static async Task Main()</code>
반환 값, <code>await</code> 사용	<code>static async Task&lt;int&gt; Main()</code>

#### ① 참고

`Environment.CommandLine` 또는 `Environment.GetCommandLineArgs`를 사용하여 콘솔 또는 Windows Forms 애플리케이션의 임의 지점에서 명령줄 인수에 액세스할 수 있습니다. Windows Forms 애플리케이션의 `Main` 메서드 서명에서 명령줄 인수를 사용하도록 설정하려면 `Main`의 서명을 수동으로 수정해야 합니다. Windows Forms 디자이너에서 생성된 코드는 입력 매개 변수 없이 `Main`을 만듭니다.

`Main` 메서드의 매개 변수는 명령줄 인수를 나타내는 `String` 배열입니다. 일반적으로 다음과 같이 `Length` 속성을 테스트하여 인수가 있는지 확인합니다.

C#

```
if (args.Length == 0)
{
    System.Console.WriteLine("Please enter a numeric argument.");
```

```
    return 1;  
}
```

### 💡 팁

`args` 배열은 null일 수 없습니다. 따라서 null 검사 없이 `Length` 속성에 액세스하는 것이 안전합니다.

`Convert` 클래스 또는 `Parse` 메서드를 사용하여 문자열 인수를 숫자 형식으로 변환할 수도 있습니다. 예를 들어 다음 문은 `Parse` 메서드를 사용하여 `string`을 `long` 숫자로 변환합니다.

C#

```
long num = Int64.Parse(args[0]);
```

`Int64`의 별칭을 지정하는 C# 형식 `long`을 사용할 수도 있습니다.

C#

```
long num = long.Parse(args[0]);
```

`Convert` 클래스 메서드 `ToInt64`를 사용하여 같은 작업을 수행할 수도 있습니다.

C#

```
long num = Convert.ToInt64(s);
```

자세한 내용은 `Parse` 및 `Convert`을 참조하세요.

### 💡 팁

명령줄 인수 구문 분석이 복잡할 수 있습니다. `System.CommandLine`[라이브러리\(현재 베타\)를 사용하여](#) 프로세스를 간소화하는 것이 좋습니다.

다음 예제에서는 콘솔 애플리케이션에서 명령줄 인수를 사용하는 방법을 보여 줍니다. 애플리케이션은 런타임에 하나의 인수를 사용하고, 인수를 정수로 변환하고, 숫자의 계승을 계산합니다. 인수가 제공되지 않으면 애플리케이션에서는 프로그램의 올바른 사용법을 설명하는 메시지를 표시합니다.

명령 프롬프트에서 애플리케이션을 컴파일 및 실행하려면 다음 단계를 수행합니다.

1. 다음 코드를 텍스트 편집기에 붙여넣고 이를 *Factorial.cs*를 사용하여 파일을 텍스트 파일로 저장합니다.

```
C#  
  
public class Functions  
{  
    public static long Factorial(int n)  
    {  
        // Test for invalid input.  
        if ((n < 0) || (n > 20))  
        {  
            return -1;  
        }  
  
        // Calculate the factorial iteratively rather than recursively.  
        long tempResult = 1;  
        for (int i = 1; i <= n; i++)  
        {  
            tempResult *= i;  
        }  
        return tempResult;  
    }  
}  
  
class MainClass  
{  
    static int Main(string[] args)  
    {  
        // Test if input arguments were supplied.  
        if (args.Length == 0)  
        {  
            Console.WriteLine("Please enter a numeric argument.");  
            Console.WriteLine("Usage: Factorial <num>");  
            return 1;  
        }  
  
        // Try to convert the input arguments to numbers. This will  
        throw  
        // an exception if the argument is not a number.  
        // num = int.Parse(args[0]);  
        int num;  
        bool test = int.TryParse(args[0], out num);  
        if (!test)  
        {  
            Console.WriteLine("Please enter a numeric argument.");  
            Console.WriteLine("Usage: Factorial <num>");  
            return 1;  
        }  
  
        // Calculate factorial.  
        long result = Functions.Factorial(num);  
  
        // Print result.  
    }  
}
```

```
        if (result == -1)
            Console.WriteLine("Input must be >= 0 and <= 20.");
        else
            Console.WriteLine($"The Factorial of {num} is {result}.");

        return 0;
    }
}

// If 3 is entered on command line, the
// output reads: The factorial of 3 is 6.
```

2. 시작 화면이나 시작 메뉴에서 Visual Studio 개발자 명령 프롬프트 창을 열고 만든 파일이 포함된 폴더로 이동합니다.
3. 다음 명령을 입력하여 애플리케이션을 컴파일합니다.

```
dotnet build
```

애플리케이션에 컴파일 오류가 없으면 *Factorial.exe*라는 실행 파일이 만들어집니다.

4. 다음 명령을 입력하여 3의 계승을 계산합니다.

```
dotnet run -- 3
```

5. 이 명령은 다음 출력을 생성합니다. *The factorial of 3 is 6.*

#### ① 참고

Visual Studio에서 애플리케이션을 실행할 경우 [프로젝트 디자이너, 디버그 페이지](#)에서 명령줄 인수를 지정할 수 있습니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)을 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

## 참고 항목

- [System.Environment](#)
- [명령줄 인수를 표시하는 방법](#)

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 최상위 문 - Main 메서드가 없는 프로그램

아티클 • 2024. 03. 01.

콘솔 애플리케이션 프로젝트에 `Main` 메서드를 명시적으로 포함할 필요가 없습니다. 대신 최상위 문 기능을 사용하여 작성해야 하는 코드를 최소화할 수 있습니다.

최상위 문을 사용하면 파일의 루트에 직접 실행 코드를 작성할 수 있으므로 클래스 또는 메서드에서 코드를 래핑할 필요가 없습니다. 즉, 클래스와 메서드의 `Program` 의식 없이 프로그램을 만들 수 있습니다 `Main`. 이 경우 컴파일러는 애플리케이션에 대한 진입점 메서드를 사용하여 `Program` 클래스를 생성합니다. 생성된 메서드의 이름은 `Main` 이(가) 아니며, 코드에서 직접 참조할 수 없는 구현 세부 정보입니다.

다음은 C# 10의 완전한 C# 프로그램인 `Program.cs` 파일입니다.

C#

```
Console.WriteLine("Hello World!");
```

최상위 문을 사용하면 Azure Functions 및 GitHub Actions와 같은 소규모 유틸리티에 대한 간단한 프로그램을 작성할 수 있습니다. 또한 새로운 C# 프로그래머가 코드를 학습하고 작성하는 작업을 더 쉽게 수행할 수 있습니다.

다음 섹션에서는 최상위 문으로 수행할 수 있는 작업과 수행할 수 없는 작업에 대한 규칙을 설명합니다.

## 하나의 최상위 파일만

애플리케이션에는 진입점이 하나만 있어야 합니다. 프로젝트에는 최상위 문이 있는 파일이 하나만 있을 수 있습니다. 프로젝트의 두 개 이상의 파일에 최상위 문을 넣으면 다음과 같은 컴파일러 오류가 발생합니다.

CS8802 하나의 컴파일 단위만 최상위 문을 포함할 수 있습니다.

프로젝트에는 최상위 문이 없는 추가 소스 코드 파일이 얼마든지 있을 수 있습니다.

## 다른 진입점 없음

`Main` 메서드를 명시적으로 작성할 수 있지만 진입점으로 작동할 수는 없습니다. 컴파일러에서 다음과 같은 경고가 발생합니다.

CS7022 프로그램의 진입점은 전역 코드이며 'Main()' 진입점은 무시됩니다.

최상위 문이 있는 프로젝트에서는 프로젝트에 하나 이상의 `Main` 메서드가 있는 경우에도 `-main` 컴파일러 옵션을 사용하여 진입점을 선택할 수 없습니다.

## using 지시문

using 지시문을 포함하는 경우 다음 예제와 같이 파일에서 먼저 제공되어야 합니다.

C#

```
using System.Text;

StringBuilder builder = new();
builder.AppendLine("Hello");
builder.AppendLine("World!");

Console.WriteLine(builder.ToString());
```

## 전역 네임스페이스

최상위 문은 전역 네임스페이스에서 암시적으로 사용할 수 있습니다.

## 네임스페이스 및 형식 정의

최상위 문이 있는 파일에는 네임스페이스 및 형식 정의도 포함될 수 있지만 최상위 문 뒤에 와야 합니다. 예시:

C#

```
MyClass.TestMethod();
MyNamespace.MyClass.MyMethod();

public class MyClass
{
    public static void TestMethod()
    {
        Console.WriteLine("Hello World!");
    }
}

namespace MyNamespace
{
    class MyClass
    {
```

```
    public static void MyMethod()
    {
        Console.WriteLine("Hello World from
MyNamespace.MyClass.MyMethod!");
    }
}
```

## args

최상위 문은 `args` 변수를 참조하여 입력된 명령줄 인수에 액세스할 수 있습니다. `args` 변수는 `null`이 아니지만 명령줄 인수가 제공되지 않은 경우 `Length`는 0입니다. 예시:

C#

```
if (args.Length > 0)
{
    foreach (var arg in args)
    {
        Console.WriteLine($"Argument={arg}");
    }
}
else
{
    Console.WriteLine("No arguments");
}
```

## await

`await`를 사용하여 비동기 메서드를 호출할 수 있습니다. 예시:

C#

```
Console.Write("Hello ");
await Task.Delay(5000);
Console.WriteLine("World!");
```

# 프로세스의 종료 코드

애플리케이션 종료될 때 `int` 값을 반환하려면 `int`를 반환하는 `Main` 메서드에서와 같이 `return` 문을 사용합니다. 예시:

C#

```
string? s = Console.ReadLine();

int returnValue = int.Parse(s ?? "-1");
return returnValue;
```

## 암시적 진입점 메서드

컴파일러는 최상위 문이 있는 프로젝트의 프로그램 진입점 역할을 하는 메서드를 생성합니다. 메서드의 서명은 최상위 문에 `await` 키워드 또는 `return` 문이 포함되어 있는지 여부에 따라 달라집니다. 다음 표에서는 편의를 위해 표에 있는 메서드 이름 `Main`을 사용하여 메서드 서명이 어떻게 표시되는지 보여줍니다.

[+] 테이블 확장

최상위 코드에는 다음이 포함됩니다.	암시적 <code>Main</code> 서명
<code>await</code> 및 <code>return</code>	<code>static async Task&lt;int&gt; Main(string[] args)</code>
<code>await</code>	<code>static async Task Main(string[] args)</code>
<code>return</code>	<code>static int Main(string[] args)</code>
<code>await</code> 또는 <code>return</code> 없음	<code>static void Main(string[] args)</code>

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)을 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

### 기능 사양 - 최상위 문

#### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

#### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

[설명서 문제 열기](#)

[제품 사용자 의견 제공](#)

# C# 형식 시스템

아티클 • 2024. 02. 15.

C#은 강력한 형식의 언어입니다. 모든 변수 및 상수에는 값으로 계산되는 모든 식을 실행하는 형식이 있습니다. 모든 메서드 선언은 각 입력 매개 변수와 반환 값의 이름, 형식, 종류(값, 참조 또는 출력)를 지정합니다. .NET 클래스 라이브러리는 기본 제공 숫자 형식과 다양한 구문을 나타내는 복합 형식을 정의합니다. 여기에는 파일 시스템, 네트워크 연결, 개체의 컬렉션과 배열, 날짜가 포함됩니다. 일반 C# 프로그램에서는 클래스 라이브러리의 형식 및 프로그램의 문제 도메인에 관련된 개념을 모델링하는 사용자 정의 형식을 사용합니다.

형식에 저장된 정보에는 다음 항목이 포함될 수 있습니다.

- 형식 변수에 필요한 스토리지 공간.
- 형식이 나타낼 수 있는 최대값 및 최소값.
- 형식에 포함되는 멤버(메서드, 필드, 이벤트 등).
- 형식이 상속하는 기본 형식.
- 구현하는 인터페이스.
- 허용되는 작업 유형.

컴파일러는 형식 정보를 사용하여 코드에서 수행되는 모든 작업의 형식이 안전한지 확인합니다. 예를 들어 `int` 형식의 변수를 선언하는 경우 컴파일러를 통해 더하기 및 빼기 작업에서 변수를 사용할 수 있습니다. `bool` 형식의 변수에 대해 같은 작업을 수행하려고 하면 컴파일러는 다음 예제와 같이 오류를 생성합니다.

C#

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and
// 'bool'.
int c = a + test;
```

## ① 참고

C 및 C++ 개발자는 C#에서 `bool` 이(가) `int`(으)로 변환될 수 없음을 알고 있습니다.

컴파일러는 형식 정보를 실행 파일에 메타데이터로 포함합니다. CLR(공용 언어 런타임)는 런타임에 이 메타데이터를 사용하여 메모리를 할당 및 회수할 때 형식 안정성을 추가로

보장합니다.

## 변수 선언에서 형식 지정

프로그램에서 변수나 상수를 선언할 때 컴파일러가 형식을 유추하게 하려면 형식을 지정하거나 `var` 키워드를 사용해야 합니다. 다음 예제에서는 기본 제공 숫자 형식 및 복잡한 사용자 정의 형식을 둘 다 사용하는 일부 변수 선언을 보여 줍니다.

C#

```
// Declaration only:  
float temperature;  
string name;  
MyClass myClass;  
  
// Declaration with initializers (four examples):  
char firstLetter = 'C';  
var limit = 3;  
int[] source = [0, 1, 2, 3, 4, 5];  
var query = from item in source  
            where item <= limit  
            select item;
```

메서드 매개 변수 및 반환 값의 형식은 메서드 선언에서 지정됩니다. 다음 시그니처는 입력 인수로 `int`(이)가 필요하고 문자열을 반환하는 메서드를 보여줍니다.

C#

```
public string GetName(int ID)  
{  
    if (ID < names.Length)  
        return names[ID];  
    else  
        return String.Empty;  
}  
private string[] names = ["Spencer", "Sally", "Doug"];
```

변수를 선언한 후에는 새 형식으로 다시 선언할 수 없으며 선언된 형식과 호환되지 않는 값을 할당할 수 없습니다. 예를 들어 `int`을(를) 선언한 다음 여기에 `true` 부울 값을 할당할 수 없습니다. 그러나 같은 새 변수에 할당되거나 메서드 인수로 전달될 경우 다른 형식으로 변환할 수 있습니다. 데이터 손실을 일으키지 않는 형식 변환은 컴파일러에서 자동으로 수행됩니다. 데이터 손실을 일으킬 수 있는 변환의 경우 소스 코드에 `캐스트`가 있어야 합니다.

자세한 내용은 [캐스팅 및 형식 변환](#)을 참조하세요.

# 기본 제공 형식

C#은 기본 제공 형식의 표준 집합을 제공합니다. 이러한 표준 집합은 정수, 부동 소수점 값, 부울 식, 텍스트 문자, 10진수 값, 기타 데이터 형식을 나타냅니다. 이 밖에도 기본 제공 `string` 및 `object` 형식이 있습니다. 이러한 형식을 이러한 형식을 모든 C# 프로그램에서 사용할 수 있습니다. 기본 제공 형식의 전체 목록은 [기본 제공 형식](#)을 참조하세요.

## 사용자 지정 형식

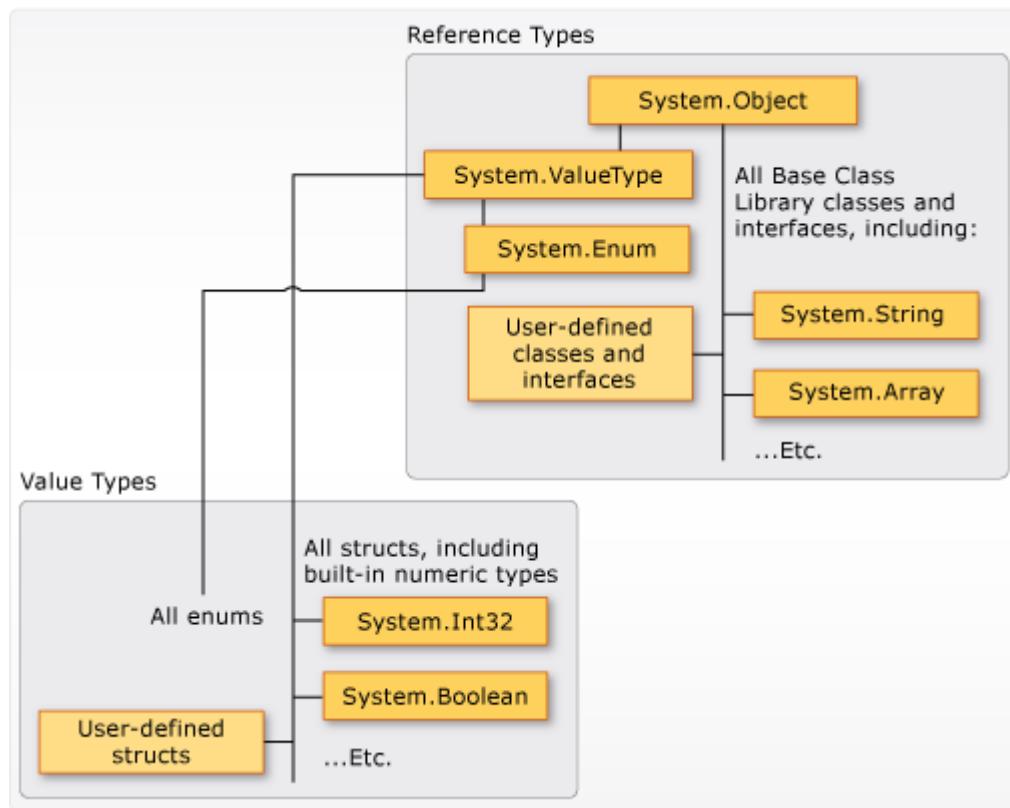
`struct`, `class`, `interface`, `enum`, `record` 구문을 사용하여 자체 사용자 지정 형식을 만듭니다. .NET 클래스 라이브러리 자체는 자체 애플리케이션에서 사용할 수 있는 사용자 지정 형식의 컬렉션입니다. 기본적으로 클래스 라이브러리의 가장 자주 사용되는 형식을 모든 C# 프로그램에서 사용할 수 있습니다. 기타 형식은 해당 형식을 정의하는 어셈블리에 프로젝트 참조를 명시적으로 추가하는 경우에만 사용할 수 있습니다. 컴파일러에 어셈블리에 대한 참조가 포함된 후에는 소스 코드에서 해당 어셈블리에 선언된 형식의 변수(및 상수)를 선언할 수 있습니다. 자세한 내용은 [.NET 클래스 라이브러리](#)를 참조하세요.

## CTS(공용 형식 시스템)

.NET의 형식 시스템에 대한 다음과 같은 두 가지 기초 사항을 이해해야 합니다.

- 형식 시스템은 상속 원칙을 지원합니다. 형식은 [기본 형식](#)이라는 다른 형식에서 파생될 수 있습니다. 파생 형식은 기본 형식의 메서드, 속성 및 기타 멤버를 상속합니다(몇 가지 제한 사항 있음). 기본 형식이 다른 형식에서 파생될 수도 있습니다. 이 경우 파생 형식은 상속 계층 구조에 있는 두 기본 형식의 멤버를 상속합니다.  
[System.Int32](#)(C# 키워드: `int`)과 같은 기본 제공 숫자 형식을 포함한 모든 형식은 궁극적으로 단일 기본 형식 [System.Object](#)(C# 키워드: `object`)에서 파생됩니다. 이 통합 형식 계층 구조를 CTS([공용 형식 시스템](#))라고 합니다. C#의 상속에 대한 자세한 내용은 [상속](#)을 참조하세요.
- CTS의 각 형식은 값 형식 또는 참조 형식으로 정의됩니다. 이러한 형식에는 .NET 클래스 라이브러리의 모든 사용자 지정 형식과 자체 사용자 정의 형식도 포함됩니다. `struct` 을(를) 사용하여 정의한 형식은 값 형식이고, 모든 기본 제공 숫자 형식은 `structs`입니다. `class` 또는 `record` 키워드를 사용하여 정의한 형식은 참조 형식입니다. 참조 형식과 값 형식의 컴파일 시간 규칙 및 런타임 동작은 서로 다릅니다.

다음 그림에서는 CTS에서 값 형식과 참조 형식 간의 관계를 보여 줍니다.



## ① 참고

가장 일반적으로 사용되는 형식은 모두 System 네임스페이스에 구성되어 있다는 사실을 알 수 있습니다. 그러나 형식이 포함된 네임스페이스는 형식이 값 형식인지 또는 참조 형식인지와 관련이 없습니다.

클래스 및 구조체는 .NET의 CTS(공용 형식 시스템)의 기본 구문 중 두 가지입니다. 각각은 기본적으로 하나의 논리 단위에 속하는 데이터 및 동작 집합을 캡슐화하는 데이터 구조입니다. 데이터와 동작은 클래스, 구조체 또는 레코드의 '멤버'입니다. 멤버는 이 문서의 뒷부분에 나열된 대로 해당 메서드, 속성, 이벤트 등을 포함합니다.

클래스, 구조체 또는 레코드 선언은 런타임에 인스턴스 또는 개체를 만드는데 사용되는 청사진과도 같습니다. `Person`이라는 클래스, 구조체 또는 레코드를 정의하는 경우 `Person`이 형식의 이름입니다. 형식 `Person`의 변수 `p`를 선언하고 초기화하면 `p`는 `Person`의 개체 또는 인스턴스로 지칭됩니다. 같은 `Person` 형식의 여러 인스턴스를 만들 수 있으며 각 인스턴스는 속성 및 필드에 서로 다른 값을 가질 수 있습니다.

클래스는 참조 형식입니다. 이 형식의 개체가 만들어지면 개체가 할당되는 변수는 해당 메모리에 대한 참조만 보유합니다. 개체 참조가 새 변수에 할당되면 새 변수는 원래 개체를 나타냅니다. 모두 동일한 데이터를 참조하므로 한 변수의 변경 내용이 다른 변수에도 반영됩니다.

구조체는 값 형식입니다. 구조체가 만들어지면 해당 구조체가 할당되는 변수에 구조체의 실제 데이터가 포함됩니다. 구조체를 새 변수에 할당하면 구조체가 복사됩니다. 따라서

새 변수와 원래 변수에 동일한 데이터의 두 가지 별도 복사본이 포함됩니다. 한 복사본의 변경 내용은 다른 복사본에 영향을 주지 않습니다.

레코드 형식은 참조 형식(record class) 또는 값 형식(record struct)일 수 있습니다. 레코드 형식에는 값 같음을 지원하는 메서드가 포함되어 있습니다.

일반적으로 클래스는 더 복잡한 동작을 모델링하는 데 사용됩니다. 클래스는 일반적으로 클래스 개체가 만들어진 후 수정할 데이터를 저장합니다. 구조체는 작은 데이터 구조에 가장 적합합니다. 구조체는 일반적으로 구조체가 만들어진 후 수정하지 않을 데이터를 저장합니다. 레코드 형식은 추가 컴파일러 합성 멤버가 있는 데이터 구조입니다. 레코드는 일반적으로 개체가 만들어진 후 수정하지 않을 데이터를 저장합니다.

## 값 형식

값 형식은 [System.Object](#)에서 파생되는 [System.ValueType](#)에서 파생됩니다.

[System.ValueType](#)에서 파생되는 형식에는 CLR의 특수 동작이 있습니다. 값 형식 변수에는 해당 값이 직접 포함됩니다. 구조체의 메모리는 변수가 선언된 컨텍스트(무엇이든지)에서 인라인으로 할당됩니다. 값 형식 변수에 대한 별도 힙 할당이나 가비지 수집 오버헤드는 없습니다. 값 형식인 record struct 형식을 선언하고 [레코드](#)의 합성 멤버를 포함할 수 있습니다.

값 형식에는 struct 및 enum의 두 가지 범주가 있습니다.

기본 제공 숫자 형식은 구조체이며, 액세스할 수 있는 필드와 메서드가 있습니다.

C#

```
// constant field on type byte.  
byte b = byte.MaxValue;
```

하지만 단순 비집계 형식처럼 값을 선언하고 변수에 할당합니다.

C#

```
byte num = 0xA;  
int i = 5;  
char c = 'Z';
```

값 형식은 sealed입니다. 값 형식(예: [System.Int32](#))에서 형식을 파생할 수 없습니다. 구조체는 [System.ValueType](#)에서만 상속할 수 있기 때문에 사용자 정의 클래스 또는 구조체에서 상속하는 구조체를 정의할 수 없습니다. 그러나 구조체는 하나 이상의 인터페이스를 구현할 수 있습니다. 구조체 형식을 구현하는 인터페이스 형식으로 캐스팅할 수 있습니다. 이 캐스트로 인해 *boxing* 작업은 관리되는 힙의 참조 형식 개체 내에 구조체를 래핑합

니다. Boxing 작업은 [System.Object](#) 또는 인터페이스 형식을 입력 매개 변수로 사용하는 메서드에 값 형식을 전달할 때 발생합니다. 자세한 내용은 [boxing 및 unboxing](#)을 참조하세요.

[struct](#) 키워드를 사용하여 고유한 사용자 지정 값 형식을 만듭니다. 일반적으로 구조체는 다음 예제와 같이 소규모 관련 변수 집합의 컨테이너로 사용됩니다.

C#

```
public struct Coords
{
    public int x, y;

    public Coords(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

구조체에 대한 자세한 내용은 [구조 형식](#)을 참조하세요. 값 형식에 대한 자세한 내용은 [값 형식](#)을 참조하세요.

값 형식의 다른 범주는 [enum](#)입니다. 열거형은 명명된 정수 상수 집합을 정의합니다. 예를 들어, .NET 클래스 라이브러리의 [System.IO.FileMode](#) 열거형에는 파일을 여는 방법을 지정하는 명명된 상수 정수 집합이 포함됩니다. 이 패턴은 다음 예제와 같이 정의됩니다.

C#

```
public enum FileMode
{
    CreateNew = 1,
    Create = 2,
    Open = 3,
    OpenOrCreate = 4,
    Truncate = 5,
    Append = 6,
}
```

[System.IO.FileMode.Create](#) 상수 값은 2입니다. 그러나 이 이름은 소스 코드를 읽는 사람에게 훨씬 더 의미가 있습니다. 따라서 상수 리터럴 숫자 대신 열거형을 사용하는 것이 더 좋습니다. 자세한 내용은 [System.IO.FileMode](#)를 참조하세요.

모든 열거형은 [System.ValueType](#)에서 상속받는 [System.Enum](#)에서 상속됩니다. 구조체에 적용되는 모든 규칙이 열거형에도 적용됩니다. 열거형에 대한 자세한 내용은 [열거형 형식](#)을 참조하세요.

## 참조 형식

`class`, `record`, `delegate`, 배열 또는 `interface`로 정의된 형식은 `reference type`입니다.

`reference type`의 변수를 선언하는 경우 해당 형식의 인스턴스로 할당하거나 `new` 연산자를 사용해 생성할 때까지 값 `null`을 포함합니다. 다음 예제에서는 클래스의 생성 및 할당을 보여줍니다.

C#

```
MyClass myClass = new MyClass();
MyClass myClass2 = myClass;
```

`interface`는 `new` 연산자를 사용하여 직접 인스턴스화할 수 없습니다. 대신, 인터페이스를 구현하는 클래스의 인스턴스를 만들고 할당합니다. 다음 예제를 참조하세요.

C#

```
MyClass myClass = new MyClass();

// Declare and assign using an existing value.
IMyInterface myInterface = myClass;

// Or create and assign a value in a single statement.
IMyInterface myInterface2 = new MyClass();
```

개체가 만들어지면 관리되는 힙에 메모리가 할당됩니다. 변수는 개체의 위치에 대한 참조만 포함합니다. 관리되는 힙의 형식은 할당될 때와 회수될 때 모두 오버헤드가 필요합니다. ‘가비지 수집’은 회수를 수행하는 CLR의 자동 메모리 관리 기능입니다. 그러나 가비지 수집은 고도로 최적화되고 대부분 시나리오에서 성능 문제를 일으키지 않습니다. 가비지 수집에 대한 자세한 내용은 [자동 메모리 관리](#)를 참조하세요.

모든 배열은 해당 요소가 값 형식이더라도 참조 형식입니다. 배열은 `System.Array` 클래스에서 암시적으로 파생됩니다. 다음 예제와 같이 C#에서 제공하는 단순한 구문을 사용하여 배열을 선언하고 사용할 수 있습니다.

C#

```
// Declare and initialize an array of integers.
int[] nums = [1, 2, 3, 4, 5];

// Access an instance property of System.Array.
int len = nums.Length;
```

참조 형식은 상속을 완벽하게 지원합니다. 클래스를 만들 때 `sealed`로 정의되지 않은 기타 인터페이스 또는 클래스에서 상속할 수 있습니다. 기타 클래스는 직접 만든 클래스에서 상속되고 가상 메서드를 재정의할 수 있습니다. 클래스를 직접 만드는 방법에 대한 자세한 내용은 [클래스, 구조체, 레코드를 참조하세요](#). 상속 및 가상 메서드에 대한 자세한 내용은 [상속을 참조하세요](#).

## 리터럴 값 형식

C#에서는 리터럴 값이 컴파일러에서 형식을 받습니다. 숫자의 끝에 문자를 추가하여 숫자 리터럴의 입력 방법을 지정할 수 있습니다. 예를 들어 값 `4.56` 이 `float`로 처리되도록 지정하려면 숫자 뒤에 "f" 또는 "F"를 추가합니다(`4.56f`). 문자를 추가하지 않으면 컴파일러가 리터럴의 형식을 유추합니다. 문자 접미사를 사용하여 지정할 수 있는 형식에 대한 자세한 내용은 [정수 숫자 형식](#) 및 [부동 소수점 숫자 형식](#)을 참조하세요.

리터럴은 형식화되고 모든 형식이 궁극적으로 `System.Object`에서 파생되기 때문에 다음과 같은 코드를 작성하고 컴파일할 수 있습니다.

```
C#  
  
string s = "The answer is " + 5.ToString();  
// Outputs: "The answer is 5"  
Console.WriteLine(s);  
  
Type type = 12345.GetType();  
// Outputs: "System.Int32"  
Console.WriteLine(type);
```

## 제네릭 형식

실제 형식('구체적 형식')에 대한 자리 표시자로 사용되는 하나 이상의 '형식 매개 변수'를 사용하여 형식을 선언할 수 있습니다. 클라이언트 코드는 형식의 인스턴스를 만들 때 구체적 형식을 제공합니다. 해당 형식을 **제네릭 형식**이라고 합니다. 예를 들어 .NET 형식 `System.Collections.Generic.List<T>`에는 변환을 통해 이름 `T` 가 제공되는 하나의 형식 매개 변수가 있습니다. 형식의 인스턴스를 만들 때 목록에 포함될 개체의 형식(예: `string`)을 지정합니다.

```
C#  
  
List<string> stringList = new List<string>();  
stringList.Add("String example");  
// compile time error adding a type other than a string:  
stringList.Add(4);
```

형식 매개 변수를 사용하면 각 요소를 개체로 변환할 필요 없이 같은 클래스를 재사용하여 요소 형식을 포함할 수 있습니다. 컴파일러는 컬렉션 요소의 특정 형식을 인식하며, 예를 들어 이전 예제에서 `stringList` 개체에 정수를 추가하려는 경우 컴파일 시간에 오류를 발생시킬 수 있기 때문에 제네릭 컬렉션 클래스를 강력한 형식의 컬렉션이라고 합니다. 자세한 내용은 [제네릭](#)을 참조하세요.

## 암시적 형식, 무명 형식 및 nullable 값 형식

`var` 키워드를 사용하여 클래스 멤버가 아닌 로컬 변수를 암시적으로 형식화할 수 있습니다. 이 변수는 컴파일 타임에 형식을 받지만 형식은 컴파일러에서 제공됩니다. 자세한 내용은 [암시적으로 형식화된 지역 변수](#)를 참조하세요.

저장하거나 메서드 경계 외부로 전달할 의도가 없는 관련 값의 단순 집합에 대한 명명된 형식을 만드는 것이 불편할 수 있습니다. 이 목적으로는 [무명 형식](#)을 만들 수 있습니다. 자세한 내용은 [무명 형식](#)을 참조하세요.

일반적인 값 형식은 `null` 값을 가질 수 없습니다. 그러나 형식 뒤에 `?`를 추가하면 `null` 허용 값 형식을 만들 수 있습니다. 예를 들어 `int?`는 `null` 값을 가질 수도 있는 `int` 형식입니다. nullable 값 형식은 제네릭 구조체 형식 `System.Nullable<T>`의 인스턴스입니다. `null` 허용 값 형식은 특히 숫자 값이 `null` 일 수 있는 데이터베이스에 데이터를 전달하는 경우에 유용합니다. 자세한 내용은 [nullable 값 형식](#)을 참조하세요.

## 컴파일 시간 형식 및 런타임 형식

변수의 컴파일 시간과 런타임 형식은 서로 다를 수 있습니다. 컴파일 시간 형식은 소스 코드에서 선언되거나 유추되는 변수의 형식입니다. 런타임 형식은 해당 변수에서 참조하는 인스턴스의 형식입니다. 다음 예제에서는 이와 같은 두 가지 유형이 동일한 경우가 많습니다.

C#

```
string message = "This is a string of characters";
```

하지만 다음 두 가지 예에서 보듯 컴파일 시간 형식이 다른 경우도 있습니다.

C#

```
object anotherMessage = "This is another string of characters";
IEnumerable<char> someCharacters = "abcdefghijklmnopqrstuvwxyz";
```

앞선 두 예제에서 런타임 형식은 `string`입니다. 컴파일 시간 형식은 첫 번째 줄에서 `object`, 두 번째 줄에서 `IEnumerable<char>`입니다.

두 형식이 변수에 대해 다른 경우 컴파일 시간 형식과 런타임 형식이 적용되는 경우를 이해하는 것이 중요합니다. 컴파일 시간 형식에 따라 컴파일러가 수행하는 모든 작업이 결정됩니다. 이러한 컴파일러 동작으로는 메서드 호출 확인, 오버로드 확인 및 사용 가능한 암시적 및 명시적 캐스트가 있습니다. 런타임 형식에 따라 런타임에서 확인되는 모든 작업이 결정됩니다. 이러한 런타임 동작에는 가상 메서드 호출 디스패치, `is` 및 `switch` 식 계산, 기타 형식 테스트 API가 포함됩니다. 코드가 형식과 상호 작용하는 방식을 보다 효과적으로 이해하려면 어떤 작업이 어떤 형식에 적용되는지를 알아야 합니다.

## 관련 단원

자세한 내용은 다음 문서를 참조하세요.

- [Builtin 형식](#)
- [값 형식](#)
- [참조 형식](#)

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)을 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# 형식을 구성하도록 네임스페이스 선언

아티클 • 2024. 02. 13.

네임스페이스는 C# 프로그래밍에서 두 가지 방법으로 많이 사용됩니다. 먼저 .NET은 다음과 같이 네임스페이스를 사용하여 여러 클래스를 구성합니다.

C#

```
System.Console.WriteLine("Hello World!");
```

[System](#)은 네임스페이스이고 [Console](#)은 해당 네임스페이스의 클래스입니다. 다음 예제와 같이 전체 이름이 필요하지 않도록 `using` 키워드를 사용할 수 있습니다.

C#

```
using System;
```

C#

```
Console.WriteLine("Hello World!");
```

자세한 내용은 [using 지시문](#)을 참조하세요.

## ⓘ 중요

.NET 6 용 C# 템플릿은 ‘최상위 문’을 사용합니다. .NET 6으로 이미 업그레이드한 경우 애플리케이션이 이 문서의 코드와 일치하지 않을 수 있습니다. 자세한 내용은 [최상위 문을 생성하는 새 C# 템플릿](#)을 참조하세요.

.NET 6 SDK는 다음 SDK를 사용하는 프로젝트에 대한 [암시적 global using](#) 지시문 집합도 추가합니다.

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

이러한 [암시적 global using](#) 지시문에는 해당 프로젝트 형식의 가장 일반적인 네임스페이스가 포함됩니다.

자세한 내용은 [암시적 using 지시문](#)에 대한 문서를 참조하세요.

둘째, 고유한 네임스페이스를 선언하면 대규모 프로그래밍 프로젝트에서 클래스 및 메서드 이름의 범위를 제어할 수 있습니다. 다음 예와 같이 [네임스페이스](#) 키워드를 사용하여 네임스페이스를 선언합니다.

C#

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
                "SampleMethod inside SampleNamespace");
        }
    }
}
```

네임스페이스 이름은 유효한 C# [식별자 이름](#)이어야 합니다.

C# 10부터 다음 예제와 같이 해당 파일에 정의된 모든 형식에 대한 네임스페이스를 선언 할 수 있습니다.

C#

```
namespace SampleNamespace;

class AnotherSampleClass
{
    public void AnotherSampleMethod()
    {
        System.Console.WriteLine(
            "SampleMethod inside SampleNamespace");
    }
}
```

이 새로운 구문의 장점은 가로 공간과 중괄호를 절약하여 더 간단하다는 것입니다. 이렇게 하면 코드를 더 쉽게 읽을 수 있습니다.

## 네임스페이스 개요

네임스페이스에는 다음과 같은 속성이 있습니다.

- 대규모 코드 프로젝트를 구성합니다.
- `.` 연산자를 사용하여 구분됩니다.
- `using` 지시문은 모든 클래스에 대해 네임스페이스 이름을 지정할 필요가 없습니다.

- `global` 네임스페이스는 “루트” 네임스페이스입니다. `global::System`은 항상 .NET [System](#) 네임스페이스를 가리킵니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 네임스페이스](#) 섹션을 참조하세요.

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 클래스 소개

아티클 • 2024. 02. 20.

## 참조 형식

`class`(으)로 정의된 형식은 참조 형식입니다. 런타임에 참조 형식의 변수를 선언할 때 변수는 `new` 연산자를 사용하여 클래스의 인스턴스를 명시적으로 만들거나 다음 예제와 같이 다른 곳에서 생성되었을 수 있는 호환되는 형식의 개체를 할당할 때까지 `null` 값을 포함합니다.

C#

```
//Declaring an object of type MyClass.  
MyClass mc = new MyClass();  
  
//Declaring another object of the same type, assigning it the value of the  
//first object.  
MyClass mc2 = mc;
```

개체가 만들어지면 해당 특정 개체에 대해 관리되는 힙에 충분한 메모리가 할당되고 변수에는 개체 위치에 대한 참조만 포함됩니다. 개체에서 사용하는 메모리는 가비지 수집으로 알려진 CLR의 자동 메모리 관리 기능에 의해 회수됩니다. 가비지 수집에 대한 자세한 내용은 [자동 메모리 관리 및 가비지 수집](#)을 참조하세요.

## 클래스 선언

클래스는 다음 예제와 같이 `class` 키워드와 뒤에 고유 식별자를 사용하여 선언됩니다.

C#

```
//[access modifier] - [class] - [identifier]  
public class Customer  
{  
    // Fields, properties, methods and events go here...  
}
```

선택적 액세스 한정자는 `class` 키워드 앞에 옵니다. 이 경우 `public`(이)가 사용되므로 누구나 이 클래스의 인스턴스를 만들 수 있습니다. 클래스 이름은 `class` 키워드 뒤에 옵니다. 클래스의 이름은 유효한 C# 식별자 이름이어야 합니다. 정의의 나머지 부분은 동작과 데이터가 정의되는 클래스 본문입니다. 클래스의 필드, 속성, 메서드 및 이벤트를 모두 `클래스 멤버`라고 합니다.

# 개체 만들기

서로 같은 의미로 사용되는 경우도 있지만 클래스와 개체는 서로 다릅니다. 클래스는 개체의 형식을 정의하지만 개체 자체는 아닙니다. 개체는 클래스에 기반을 둔 구체적 엔터티이고 클래스의 인스턴스라고도 합니다.

다음과 같이 `new` 키워드와 뒤에 클래스 이름을 사용하여 개체를 만들 수 있습니다.

C#

```
Customer object1 = new Customer();
```

클래스 인스턴스가 만들어질 때 개체에 대한 참조가 다시 프로그래머에게 전달됩니다. 이전 예제에서 `object1`은 `Customer`에 기반을 둔 개체에 대한 참조입니다. 이 참조는 새 개체를 참조하지만 개체 데이터 자체는 포함하지 않습니다. 실제로 개체를 만들지 않고도 개체 참조를 만들 수 있습니다.

C#

```
Customer object2;
```

이러한 참조를 통해 개체에 액세스하려고 하면 런타임에 실패하므로 개체를 참조하지 않는 개체 참조를 만들지 않는 것이 좋습니다. 새 개체를 만들거나 다음과 같은 기존 개체를 할당하여 개체를 참조하도록 참조를 생성할 수 있습니다.

C#

```
Customer object3 = new Customer();
Customer object4 = object3;
```

이 코드에서는 같은 개체를 참조하는 두 개의 개체 참조를 만듭니다. 따라서 `object3`을 통해 이루어진 모든 개체 변경 내용은 이후 `object4` 사용 시 반영됩니다. 클래스에 기반을 둔 개체는 참조를 통해 참조되므로 클래스를 참조 형식이라고 합니다.

## 생성자 및 초기화

이전 섹션에서는 클래스 형식을 선언하고 해당 형식의 인스턴스를 만드는 구문을 소개했습니다. 형식의 인스턴스를 만들 때 해당 필드와 속성이 유용한 값으로 초기화되었는지 확인하려고 합니다. 값을 초기화하는 방법에는 여러 가지가 있습니다.

- 기본값 수용
- 필드 초기화

- 생성자 매개 변수
- 개체 이니셜라이저

모든 .NET 형식에는 기본값이 있습니다. 일반적으로 이 값은 숫자 형식의 경우 0이고 모든 참조 형식에 대해서는 `null`입니다. 앱에서 적절한 경우 해당 기본값을 사용할 수 있습니다.

.NET 기본값이 올바른 값이 아닌 경우 필드 *이니셜라이저*를 사용하여 초기 값을 설정할 수 있습니다.

C#

```
public class Container
{
    // Initialize capacity field to a default value of 10:
    private int _capacity = 10;
}
```

호출자가 초기 값을 설정하는 생성자를 정의하여 초기 값을 제공하도록 요구할 수 있습니다.

C#

```
public class Container
{
    private int _capacity;

    public Container(int capacity) => _capacity = capacity;
}
```

C# 12부터 클래스 선언의 일부로 기본 생성자를 정의할 수 있습니다.

C#

```
public class Container(int capacity)
{
    private int _capacity = capacity;
}
```

클래스 이름에 매개 변수를 추가하면 기본 생성자가 정의됩니다. 이러한 매개 변수는 해당 멤버를 포함하는 클래스 본문에서 사용할 수 있습니다. 이를 사용하여 필드 또는 필요한 다른 곳을 초기화할 수 있습니다.

속성에서 `required` 한정자를 사용하고 호출자가 *개체 이니셜라이저*를 사용하여 속성의 초기 값을 설정하도록 허용할 수도 있습니다.

C#

```
public class Person
{
    public required string LastName { get; set; }
    public required string FirstName { get; set; }
}
```

`required` 키워드를 추가하면 호출자가 해당 속성을 `new` 식의 일부로 설정해야 합니다.

C#

```
var p1 = new Person(); // Error! Required properties not set
var p2 = new Person() { FirstName = "Grace", LastName = "Hopper" };
```

## 클래스 상속

클래스는 개체 지향 프로그래밍의 기본적인 특성인 '상속'을 완전히 지원합니다. 클래스를 만들 때 `sealed`(으)로 정의되지 않은 다른 클래스에서 상속할 수 있습니다. 다른 클래스는 클래스에서 상속하고 클래스 가상 메서드를 재정의할 수 있습니다. 또한 하나 이상의 인터페이스를 구현할 수 있습니다.

상속은 **파생**을 통해 수행합니다. 즉, 클래스는 데이터와 동작을 상속하는 소스 **기본 클래스**를 사용하여 선언됩니다. 다음과 같이 파생 클래스 이름 뒤에 콜론 및 기본 클래스 이름을 추가하여 기본 클래스를 지정합니다.

C#

```
public class Manager : Employee
{
    // Employee fields, properties, methods and events are inherited
    // New Manager fields, properties, methods and events go here...
}
```

클래스 선언에 기본 클래스가 포함된 경우 생성자를 제외한 기본 클래스의 모든 멤버를 상속합니다. 자세한 내용은 [상속](#)을 참조하세요.

C#의 클래스는 하나의 기본 클래스에서만 직접 상속할 수 있습니다. 그러나 기본 클래스 자체가 다른 클래스에서 상속될 수 있으므로 클래스는 여러 기본 클래스를 간접적으로 상속할 수 있습니다. 또한 클래스는 하나 이상의 인터페이스를 직접 구현할 수 있습니다. 자세한 내용은 [인터페이스](#)를 참조하세요.

클래스는 `abstract`(으)로 선언할 수 있습니다. 추상 클래스에는 시그니처 정의가 있지만 구현이 없는 추상 메서드가 포함됩니다. 추상 클래스는 인스턴스화할 수 없습니다. 추상

클래스는 추상 메서드를 구현하는 파생 클래스를 통해서만 사용할 수 있습니다. 반면, [봉인된](#) 클래스는 다른 클래스에서 파생되는 것을 허용하지 않습니다. 자세한 내용은 [Abstract 및 Sealed 클래스와 클래스 멤버](#)를 참조하세요.

클래스 정의는 여러 소스 파일로 분할될 수 있습니다. 자세한 내용은 참조 [Partial 클래스 및 메서드](#)합니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)을 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# C#의 레코드 형식 소개

아티클 • 2024. 02. 19.

C#의 [레코드](#)는 데이터 모델 작업에 특별한 구문과 동작을 제공하는 [클래스](#) 또는 [구조체](#)입니다. `record` 한정자는 주 역할이 데이터를 저장하는 형식에 유용한 멤버를 합성하도록 컴파일러에 지시합니다. 이러한 멤버에는 값 같음을 지원하는 `ToString()` 및 멤버의 오버로드가 포함됩니다.

## 레코드를 사용하는 경우

다음 시나리오에서 클래스 또는 구조체 대신 레코드를 사용하는 것이 좋습니다.

- [값 같음](#) 여부에 따라 달라지는 데이터 모델을 정의하려는 경우
- 변경할 수 없는 개체의 형식을 정의하려고 합니다.

## 값 같음

레코드의 경우 값 같음은 형식이 일치하고 모든 속성 및 필드 값이 일치하는 경우 레코드 형식의 두 변수가 같다는 것을 의미합니다. 클래스와 같은 다른 참조 형식의 경우 같음은 [참조 같음](#)을 의미합니다. 즉, 클래스 형식의 두 변수는 같은 개체를 참조하는 경우 같습니다. 두 레코드 인스턴스의 같음을 확인하는 메서드와 연산자에서 값 같음을 사용합니다.

일부 데이터 모델은 값 같음을 사용하지 않습니다. 예를 들어, [Entity Framework Core](#)는 참조 같음을 사용하여 개념적으로 하나의 엔터티에 해당하는 엔터티 형식의 인스턴스를 하나만 사용하는지 확인합니다. 이러한 이유로 레코드 형식은 Entity Framework Core에서 엔터티 형식으로 사용하기에 적절하지 않습니다.

## 불변성

변경할 수 없는 형식은 인스턴스화된 후 개체의 속성 또는 필드 값을 변경하는 것을 방지하는 형식입니다. 불변성은 형식이 스레드로부터 안전해야 하거나 해시 테이블에서 동일하게 남아 있는 해시 코드를 사용하는 경우에 유용할 수 있습니다. 레코드는 변경할 수 없는 형식을 만들고 사용하기 위한 간결한 구문을 제공합니다.

불변성은 모든 데이터 시나리오에 적합하지 않습니다. 예를 들어, [Entity Framework Core](#)는 변경할 수 없는 엔터티 형식을 사용한 업데이트를 지원하지 않습니다.

## 레코드가 클래스 및 구조체와 다른 방식

클래스 또는 구조체를 선언하고 인스턴스화하는 동일한 구문을 레코드와 함께 사용할 수 있습니다. `class` 키워드를 `record`(으)로 대체하거나 `struct` 대신 `record struct`(을)를 사용합니다. 마찬가지로, 상속 관계를 표현하기 위한 동일한 구문은 레코드 클래스에서 지원됩니다. 레코드가 클래스와 다른 점은 다음과 같습니다.

- 기본 생성자에서 위치 매개 변수를 사용하여 변경할 수 없는 속성을 사용하여 형식을 만들고 인스턴스화할 수 있습니다.
- 클래스에서 참조 같음 또는 같지 않음(예: `Object.Equals(Object)` 및 `==`)을 나타내는 동일한 메서드와 연산자가 레코드에서 값 같음 또는 같지 않음을 나타냅니다.
- `with` 식을 사용하여 선택된 속성에 새 값을 포함하는 변경할 수 없는 개체의 복사본을 만들 수 있습니다.
- 레코드의 `ToString` 메서드는 개체 형식 이름과 모든 퍼블릭 속성의 이름과 값을 표시하는 형식 문자열을 만듭니다.
- 레코드는 다른 레코드에서 상속될 수 있습니다. 레코드는 클래스에서 상속될 수 없으며 클래스는 레코드에서 상속될 수 없습니다.

레코드 구조체는 컴파일러가 같음과 `ToString` 을(를) 위한 메서드를 합성한다는 점에서 구조체와 다릅니다. 컴파일러는 위치 레코드 구조체에 대한 `Deconstruct` 메서드를 합성합니다.

컴파일러는 `record class` 의 각 기본 생성자 매개 변수에 대한 공개 초기화 전용 속성을 합성합니다. `record struct` 에서 컴파일러는 공개 읽기/쓰기 속성을 합성합니다. 컴파일러는 `record` 한정자를 포함하지 않는 `class` 및 `struct` 형식에서 기본 생성자 매개 변수에 대한 속성을 만들지 않습니다.

## 예제

다음 예제에서는 위치 매개 변수를 사용하여 레코드를 선언하고 인스턴스화하는 퍼블릭 레코드를 정의합니다. 그런 다음 형식 이름 및 속성 값을 출력합니다.

C#

```
public record Person(string FirstName, string LastName);

public static class Program
{
    public static void Main()
    {
        Person person = new("Nancy", "Davolio");
        Console.WriteLine(person);
        // output: Person { FirstName = Nancy, LastName = Davolio }
    }
}
```

```
}
```

다음 예제에서는 레코드에서 값 같음을 보여 줍니다.

```
C#
```

```
public record Person(string FirstName, string LastName, string[]
PhoneNumbers);
public static class Program
{
    public static void Main()
    {
        var phoneNumbers = new string[2];
        Person person1 = new("Nancy", "Davolio", phoneNumbers);
        Person person2 = new("Nancy", "Davolio", phoneNumbers);
        Console.WriteLine(person1 == person2); // output: True

        person1.PhoneNumbers[0] = "555-1234";
        Console.WriteLine(person1 == person2); // output: True

        Console.WriteLine(ReferenceEquals(person1, person2)); // output:
False
    }
}
```

다음 예제에서는 `with` 식을 사용하여 변경할 수 없는 개체를 복사하고 속성 중 하나를 변경하는 방법을 보여 줍니다.

```
C#
```

```
public record Person(string FirstName, string LastName)
{
    public required string[] PhoneNumbers { get; init; }
}

public class Program
{
    public static void Main()
    {
        Person person1 = new("Nancy", "Davolio") { PhoneNumbers = new
string[1] };
        Console.WriteLine(person1);
        // output: Person { FirstName = Nancy, LastName = Davolio,
PhoneNumbers = System.String[] }

        Person person2 = person1 with { FirstName = "John" };
        Console.WriteLine(person2);
        // output: Person { FirstName = John, LastName = Davolio,
PhoneNumbers = System.String[] }
        Console.WriteLine(person1 == person2); // output: False
    }
}
```

```
person2 = person1 with { PhoneNumbers = new string[1] };
Console.WriteLine(person2);
// output: Person { FirstName = Nancy, LastName = Davolio,
PhoneNumbers = System.String[] }
Console.WriteLine(person1 == person2); // output: False

person2 = person1 with { };
Console.WriteLine(person1 == person2); // output: True
}
}
```

자세한 내용은 [레코드\(C# 참조\)](#)를 참조하세요.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)을 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 인터페이스 - 여러 형식에 대한 동작 정의

아티클 • 2023. 03. 18.

인터페이스에는 비 추상 `class` 또는 가 구현해야 하는 관련 기능 그룹에 대한 정의가 `struct` 포함되어 있습니다. 인터페이스에서는 구현이 있어야 하는 `static` 메서드를 정의 할 수 있습니다. 인터페이스는 멤버에 대한 기본 구현을 정의할 수 있습니다. 인터페이스에서는 필드, 자동 구현 속성, 속성과 유사한 이벤트 등과 같은 인스턴스 데이터를 선언할 수 없습니다.

예를 들어 인터페이스를 사용하면 여러 소스의 동작을 클래스에 포함할 수 있습니다. 해당 기능은 언어가 클래스의 여러 상속을 지원하지 않기 때문에 C#에서 중요합니다. 또한 구조체는 다른 구조체나 클래스에서 실제로 상속할 수 없기 때문에 구조체에 대한 상속 을 시뮬레이트하려는 경우 인터페이스를 사용해야 합니다.

다음 예제와 같이 키워드를 `interface` 사용하여 인터페이스를 정의합니다.

C#

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

인터페이스 이름은 유효한 C# [식별자 이름](#)이어야 합니다. 규칙에 따라 인터페이스 이름 은 대문자 `I`로 시작합니다.

`IEquatable<T>` 인터페이스를 구현하는 모든 클래스나 구조체에는 인터페이스에서 지정 한 서명과 일치하는 `Equals` 메서드에 대한 정의가 포함되어 있어야 합니다. 따라서 `IEquatable<T>`를 구현하는 클래스를 계산하여 클래스의 인스턴스에서 동일한 클래스의 다른 인스턴스와 동일한지 여부를 확인할 수 있는 `Equals` 메서드를 포함할 수 있습니다.

`IEquatable<T>`의 정의에서는 `Equals`에 대한 구현을 제공하지 않습니다. 클래스 또는 구 조체는 여러 인터페이스를 구현할 수 있지만 클래스는 단일 클래스에서만 상속할 수 있습니다.

추상 클래스에 대한 자세한 내용은 [추상 및 봉인 클래스와 클래스 멤버](#)를 참조하세요.

인터페이스에는 인스턴스 메서드, 속성, 이벤트, 인덱서 또는 이러한 네 가지 멤버 형식의 조합이 포함될 수 있습니다. 인터페이스에는 정적 생성자, 필드, 상수 또는 연산자가 포함 될 수 있습니다. C# 11부터 필드가 아닌 인터페이스 멤버는 일 수 있습니다 `static` `abstract`. 인터페이스에는 인스턴스 필드, 인스턴스 생성자 또는 종료자가 포함될 수 없습니다. 인터페이스 멤버는 기본적으로 공용이며, `public`, `protected`, `internal`, `private`,

`protected internal` 또는 `private protected` 등의 접근성 한정자를 명시적으로 지정할 수 있습니다. `private` 멤버에는 기본 구현이 있어야 합니다.

인터페이스 멤버를 구현하려면 구현 클래스의 해당 멤버가 공용이고 비정적이어야 하며 인터페이스 멤버와 동일한 이름 및 서명을 사용해야 합니다.

## ① 참고

인터페이스가 정적 멤버를 선언하는 경우 해당 인터페이스를 구현하는 형식은 동일한 서명으로 정적 멤버를 선언할 수도 있습니다. 멤버를 선언하는 형식으로 고유하고 고유하게 식별됩니다. 형식에서 선언된 정적 멤버는 인터페이스에 선언된 정적 멤버를 재정 의하지 않습니다.

인터페이스를 구현하는 클래스 또는 구조체는 인터페이스에서 제공하는 기본 구현 없이 선언된 모든 멤버에 대한 구현을 제공해야 합니다. 그러나 기본 클래스에서 인터페이스를 구현하는 경우에는 기본 클래스에서 파생되는 모든 클래스가 해당 구현을 상속합니다.

다음 예제에서는 `IEquatable<T>` 인터페이스의 구현을 보여 줍니다. 구현 클래스 `Car`는 `Equals` 메서드의 구현을 제공해야 합니다.

C#

```
public class Car : IEquatable<Car>
{
    public string? Make { get; set; }
    public string? Model { get; set; }
    public string? Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car? car)
    {
        return (this.Make, this.Model, this.Year) ==
               (car?.Make, car?.Model, car?.Year);
    }
}
```

클래스의 속성 및 인덱서는 인터페이스에 정의된 속성이나 인덱서에 대해 추가 접근자를 정의할 수 있습니다. 예를 들어 인터페이스는 `get` 접근자가 있는 속성을 선언할 수 있습니다. 인터페이스를 구현하는 클래스는 `get` 및 `set` 접근자를 둘 다 사용하는 동일한 속성을 선언할 수 있습니다. 그러나 속성 또는 인덱서에서 명시적 구현을 사용하는 경우에는 접근자가 일치해야 합니다. 명시적 구현에 대한 자세한 내용은 [명시적 인터페이스 구현](#) 및 [인터페이스 속성\(C# 프로그래밍 가이드\)](#)를 참조하세요.

인터페이스는 하나 이상의 인터페이스에서 상속할 수 있습니다. 파생 인터페이스는 기본 인터페이스에서 멤버를 상속합니다. 파생 인터페이스를 구현하는 클래스는 파생 인터페

이스의 기본 인터페이스 멤버 모두를 포함해 파생 인터페이스의 모든 멤버를 구현해야 합니다. 이 클래스는 파생 인터페이스나 해당하는 기본 인터페이스로 암시적으로 변환될 수 있습니다. 클래스는 상속하는 기본 클래스 또는 다른 인터페이스에서 상속하는 인터페이스를 통해 인터페이스를 여러 번 포함할 수 있습니다. 그러나 클래스는 인터페이스의 구현을 한 번만 제공할 수 있으며 클래스가 인터페이스를 클래스 정의의 일부로 선언하는 경우에만 제공할 수 있습니다(`class ClassName : InterfaceName`). 인터페이스를 구현하는 기본 클래스를 상속했기 때문에 인터페이스가 상속되는 경우 기본 클래스는 인터페이스 멤버의 구현을 제공합니다. 그러나 파생 클래스는 상속된 구현을 사용하는 대신 가상 인터페이스 멤버를 다시 구현할 수 있습니다. 인터페이스가 메서드의 기본 구현을 선언하면 해당 인터페이스를 구현하는 모든 클래스가 해당 구현을 상속합니다(인터페이스 멤버의 기본 구현에 액세스하려면 클래스 인스턴스를 인터페이스 형식으로 캐스팅해야 합니다).

또한 기본 클래스는 가상 멤버를 사용하여 인터페이스 멤버를 구현할 수 있습니다. 이 경우 파생 클래스는 가상 멤버를 재정의하여 인터페이스 동작을 변경할 수 있습니다. 가상 멤버에 대한 자세한 내용은 [다형성](#)을 참조하세요.

## 인터페이스 요약

인터페이스에는 다음과 같은 속성이 있습니다.

- 8.0 이전 C# 버전에서 인터페이스는 추상 멤버만 있는 추상 기본 클래스와 유사합니다. 인터페이스를 구현하는 클래스 또는 구조체는 해당 멤버를 모두 구현해야 합니다.
- C# 8.0부터 인터페이스에서 해당 멤버 일부 또는 모두의 기본 구현을 정의할 수 있습니다. 인터페이스를 구현하는 클래스 또는 구조체는 기본 구현이 있는 멤버를 구현할 필요가 없습니다. 자세한 내용은 [기본 인터페이스 메서드](#)를 참조하세요.
- 인터페이스는 직접 인스턴스화할 수 없습니다. 해당 멤버는 인터페이스를 구현하는 클래스 또는 구조체에 의해 구현됩니다.
- 클래스 또는 구조체는 여러 인터페이스를 구현할 수 있습니다. 클래스는 기본 클래스를 상속할 수 있으며 하나 이상의 인터페이스를 제공할 수도 있습니다.

# 제네릭 클래스 및 메서드

아티클 • 2024. 03. 19.

제네릭은 .NET에 형식 매개 변수의 개념을 소개합니다. 제네릭을 사용하면 코드에서 클래스 또는 메서드를 사용할 때까지 하나 이상의 형식 매개 변수 사양을 연기하는 클래스와 메서드를 디자인할 수 있습니다. 예를 들어 제네릭 형식 매개 변수 `T`를 사용하여 여기에 표시된 것처럼, 다른 클라이언트 코드에서 런타임 캐스팅 또는 boxing 작업에 대한 비용이나 위험을 발생하지 않고 사용할 수 있는 단일 클래스를 작성할 수 있습니다.

C#

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }

    class TestGenericList
    {
        private class ExampleClass { }

        static void Main()
        {
            // Declare a list of type int.
            GenericList<int> list1 = new GenericList<int>();
            list1.Add(1);

            // Declare a list of type string.
            GenericList<string> list2 = new GenericList<string>();
            list2.Add("");

            // Declare a list of type ExampleClass.
            GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
            list3.Add(new ExampleClass());
        }
    }
}
```

제네릭 클래스와 메서드는 재사용성, 형식 안전성 및 효율성을 제네릭이 아닌 클래스에서 사용할 수 없는 방식으로 결합합니다. 제네릭 형식 매개 변수는 컴파일 중에 형식 인수로 대체됩니다. 앞의 예제에서 컴파일러는 `.int`로 바뀝니다. 제네릭은 컬렉션 및 해당 컬렉션에서 작동하는 메서드에서 가장 자주 사용됩니다. `System.Collections.Generic` 네임스페이스에는 몇 가지 제네릭 기반 컬렉션 클래스가 있습니다. 비제네릭 컬렉션(예: `ArrayList` 권장되지 않음)은 호환성 목적으로만 기본. 자세한 내용은 [.NET의 제네릭](#)을 참조하세요.

사용자 지정 제네릭 형식 및 메서드를 만들어 형식이 안전하고 효율적인 일반화된 솔루션 및 디자인 패턴을 직접 제공할 수도 있습니다. 다음 코드 예제에서는 데모용으로 간단한 제네릭 연결된 목록 클래스를 보여 줍니다. (대부분의 경우 직접 만드는 대신 .NET에서

제공하는 클래스를 사용해야 `List<T>` 합니다.) 형식 매개 변수 `T` 는 구체적인 형식이 일반적으로 목록에 저장된 항목의 형식을 나타내는 데 사용되는 여러 위치에서 사용됩니다.

- `AddHead` 메서드에서 메서드 매개 변수의 형식.
- 중첩 `Node` 클래스에서 `Data` 속성의 반환 형식.
- 중첩 클래스에서 `private` 멤버 `data`의 형식.

`T`는 중첩된 `Node` 클래스에 사용할 수 있습니다. `GenericList<T>` 예를 들어 구체적인 형식으로 `GenericList<int>` 인스턴스화되면 각 발생 `T` 항목이 `.로 int` 바뀝니다.

C#

```
// type parameter T in angle brackets
public class GenericList<T>
{
    // The nested class is also generic on T.
    private class Node
    {
        // T used in non-generic constructor.
        public Node(T t)
        {
            next = null;
            data = t;
        }

        private Node? next;
        public Node? Next
        {
            get { return next; }
            set { next = value; }
        }

        // T as private member data type.
        private T data;

        // T as return type of property.
        public T Data
        {
            get { return data; }
            set { data = value; }
        }
    }

    private Node? head;

    // constructor
    public GenericList()
    {
        head = null;
    }

    // T as method parameter type:
```

```

public void AddHead(T t)
{
    Node n = new Node(t);
    n.Next = head;
    head = n;
}

public IEnumerator<T> GetEnumerator()
{
    Node? current = head;

    while (current != null)
    {
        yield return current.Data;
        current = current.Next;
    }
}

```

다음 코드 예제에서는 클라이언트 코드에서 제네릭 `GenericList<T>` 클래스를 사용하여 정수 목록을 만드는 방법을 보여 줍니다. 형식 인수를 변경하는 경우 다음 코드는 문자열 목록 또는 다른 사용자 지정 형식을 만듭니다.

C#

```

class TestGenericList
{
    static void Main()
    {
        // int is the type argument
        GenericList<int> list = new GenericList<int>();

        for (int x = 0; x < 10; x++)
        {
            list.AddHead(x);
        }

        foreach (int i in list)
        {
            System.Console.Write(i + " ");
        }
        System.Console.WriteLine("\nDone");
    }
}

```

## ① 참고

제네릭 형식은 클래스로 제한되지 않습니다. 앞의 예제에서는 형식을 사용 `class` 하지만 형식을 포함하여 `record` 제네릭 `interface` 및 `struct` 형식을 정의할 수 있습니다.

다.

## 제네릭 개요

- 제네릭 형식을 사용하여 코드 재사용, 형식 안전성 및 성능을 최대화합니다.
- 가장 일반적으로 제네릭은 컬렉션 클래스를 만드는 데 사용됩니다.
- .NET 클래스 라이브러리에는 [System.Collections.Generic](#) 네임스페이스의 여러 제네릭 컬렉션 클래스가 포함됩니다. 제네릭 컬렉션은 가능할 때마다 [System.Collections](#) 네임스페이스의 [ArrayList](#)처럼 클래스 대신 사용되어야 합니다.
- 사용자 고유의 제네릭 인터페이스, 클래스, 메서드, 이벤트 및 대리자를 만들 수 있습니다.
- 제네릭 클래스는 특정 데이터 형식의 메서드에 액세스할 수 있도록 제한될 수 있습니다.
- 리플렉션을 사용하여 제네릭 데이터 형식에 사용되는 형식에 대한 정보를 런타임에 가져올 수 있습니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)을 참조하세요.

## 참고 항목

- [.NET의 제네릭](#)
- [System.Collections.Generic](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# 무명 형식

아티클 • 2024. 02. 17.

익명 형식을 사용하면 먼저 명시적으로 형식을 정의할 필요 없이 읽기 전용 속성 집합을 단일 개체로 편리하게 캡슐화할 수 있습니다. 형식 이름은 컴파일러에 의해 생성되며 소스 코드 수준에서 사용할 수 없습니다. 각 속성의 형식은 컴파일러에서 유추합니다.

[new](#) 연산자를 개체 이니셜라이저와 함께 사용하여 무명 형식을 만듭니다. 개체 이니셜라이저에 대한 자세한 내용은 [개체 및 컬렉션 이니셜라이저](#)를 참조하세요.

다음 예제에서는 `Amount` 및 `Message`라는 두 속성으로 초기화된 익명 형식을 보여 줍니다.

C#

```
var v = new { Amount = 108, Message = "Hello" };

// Rest the mouse pointer over v.Amount and v.Message in the following
// statement to verify that their inferred types are int and string.
Console.WriteLine(v.Amount + v.Message);
```

일반적으로 무명 형식은 소스 시퀀스에 있는 각 개체의 속성 하위 집합을 반환하기 위해 쿼리 식의 `select` 절에 사용됩니다. 쿼리에 대한 자세한 내용은 [C#의 LINQ](#)를 참조하세요.

익명 형식은 하나 이상의 `public` 읽기 전용 속성을 포함합니다. 메서드 또는 이벤트와 같은 다른 종류의 클래스 멤버는 유효하지 않습니다. 속성을 초기화하는 데 사용되는 식은 `null`, 익명 함수 또는 포인터 형식일 수 없습니다.

가장 일반적인 시나리오는 다른 형식의 속성으로 익명 형식을 초기화하는 것입니다. 다음 예제에서는 `Product`라는 클래스가 있다고 가정합니다. `Product` 클래스에는 `Color` 및 `Price` 속성뿐만 아니라 관심 없는 다른 속성도 포함되어 있습니다. `products` 변수는 `Product` 개체의 컬렉션입니다. 익명 형식 선언은 `new` 키워드로 시작합니다. 선언에서는 `Product`의 두 속성만 사용하는 새 형식을 초기화합니다. 무명 형식을 사용하면 더 적은 양의 데이터가 쿼리에 반환됩니다.

무명 형식에 멤버 이름을 지정하지 않으면 컴파일러가 무명 형식 멤버에 해당 멤버를 초기화하는 데 사용되는 속성과 동일한 이름을 제공합니다. 앞의 예제에 표시된 것처럼, 식으로 초기화되는 속성의 이름을 제공합니다. 다음 예제에서 익명 형식의 속성 이름은 `Color` 및 `Price`입니다.

C#

```
var productQuery =
    from prod in products
```

```
select new { prod.Color, prod.Price };

foreach (var v in productQuery)
{
    Console.WriteLine("Color={0}, Price={1}", v.Color, v.Price);
}
```

### 💡 팁

.NET 스타일 규칙 [IDE0037](#)을 사용하여 유추된 멤버 이름 또는 명시적 멤버 이름 중 어느 것이 선호되는지를 적용할 수 있습니다.

클래스, 구조체 또는 다른 무명 형식과 같은 다른 형식의 개체별로 필드를 정의할 수도 있습니다. 이 작업은 이미 인스턴스화된 사용자 정의 형식을 사용하여 두 개의 무명 형식이 만들어지는 다음 예제와 마찬가지로 이 개체를 보유하는 변수를 사용하여 수행됩니다. 두 경우 모두 무명 형식 `shipment` 및 `shipmentWithBonus`의 `product` 필드는 각 필드의 기본값을 포함하는 `Product` 형식이 됩니다. 그리고 `bonus` 필드는 컴파일러에서 만든 무명 형식이 됩니다.

C#

```
var product = new Product();
var bonus = new { note = "You won!" };
var shipment = new { address = "Nowhere St.", product };
var shipmentWithBonus = new { address = "Somewhere St.", product, bonus };
```

일반적으로 무명 형식을 사용하여 변수를 초기화할 때는 `var`을 사용하여 변수를 암시적 형식 지역 변수로 선언합니다. 컴파일러만 익명 형식의 기본 이름에 액세스할 수 있으므로 변수 선언에는 형식 이름을 지정할 수 없습니다. `var`에 대한 자세한 내용은 [암시적 형식 지역 변수](#)를 참조하세요.

다음 예제에 표시된 것처럼, 암시적으로 형식화된 지역 변수와 암시적으로 형식화된 배열을 결합하여 익명으로 형식화된 요소의 배열을 만들 수 있습니다.

C#

```
var anonArray = new[] { new { name = "apple", diam = 4 }, new { name =
"grape", diam = 1 } };
```

무명 형식은 `object`에서 직접 파생되고 `object`를 제외한 어떠한 형식으로도 캐스팅될 수 없는 `class` 형식입니다. 컴파일러는 애플리케이션에서 해당 익명 형식에 액세스할 수 없더라도 각 익명 형식의 이름을 제공합니다. 공용 언어 런타임의 관점에서 익명 형식은 다른 참조 형식과 다를 바가 없습니다.

어셈블리에서 둘 이상의 익명 개체 이니셜라이저가 순서와 이름 및 형식이 동일한 속성의 시퀀스를 지정하는 경우 컴파일러는 개체를 동일한 형식의 인스턴스로 처리합니다. 이러한 개체는 컴파일러에서 생성된 동일한 형식 정보를 공유합니다.

무명 형식은 [with](#) 식 형태로 비파괴적 변경을 지원합니다. 이로 인해 하나 이상의 속성에 새 값이 있는 무명 형식의 새 인스턴스를 만들 수 있습니다.

C#

```
var apple = new { Item = "apples", Price = 1.35 };
var onSale = apple with { Price = 0.79 };
Console.WriteLine(apple);
Console.WriteLine(onSale);
```

익명 형식을 가지고 있으므로 필드, 속성, 이벤트 또는 메서드의 반환 형식은 선언할 수 없습니다. 마찬가지로, 익명 형식을 가지고 있으므로 메서드, 속성, 생성자 또는 인덱서의 정식 매개 변수는 선언할 수 없습니다. 무명 형식이나 무명 형식이 포함된 컬렉션을 메서드에 인수로 전달하려면 매개 변수를 `object` 형식으로 선언하면 됩니다. 그러나 무명 형식에 `object`를 사용하면 강력한 형식화의 목적이 무효화됩니다. 쿼리 결과를 저장하거나 메서드 경계 외부로 전달해야 하는 경우 익명 형식 대신 일반적인 명명된 구조체 또는 클래스 사용을 고려하세요.

익명 형식에 대한 [Equals](#) 및 [GetHashCode](#) 메서드는 속성의 `Equals` 및 `GetHashCode` 메서드 측면에서 정의되므로 동일한 익명 형식의 두 인스턴스는 해당 속성이 모두 동일한 경우에만 동일합니다.

### ① 참고

무명 형식의 [접근성 수준](#)은 `internal` 이므로 서로 다른 어셈블리에 정의된 두 무명 형식은 동일한 형식이 아닙니다. 따라서 무명 형식의 인스턴스는 모든 속성이 같은 경우에도 서로 다른 어셈블리에 정의된 경우 서로 같을 수 없습니다.

무명 형식은 [ToString](#) 메서드를 재정의하여 중괄호로 둘러싸인 모든 속성의 이름과 `ToString` 출력을 연결합니다.

```
var v = new { Title = "Hello", Age = 24 };

Console.WriteLine(v.ToString()); // "{ Title = Hello, Age = 24 }"
```

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# C#의 클래스, 구조체 및 레코드 개요

아티클 • 2024. 02. 15.

C#에서 형식(클래스, 구조체 또는 레코드)의 정의는 형식이 수행할 수 있는 작업을 지정하는 청사진과 같습니다. 개체는 기본적으로 청사진에 따라 구성 및 할당된 메모리 블록입니다. 이 문서에서는 이러한 청사진과 해당 기능에 대한 개요를 제공합니다. [이 시리즈의 다음 문서](#)에서는 개체를 소개합니다.

## 캡슐화

캡슐화는 경우에 따라 개체 지향 프로그래밍의 첫 번째 pillar 또는 원리로 인식됩니다. 클래스 또는 구조체는 클래스 또는 구조체 외부의 코드에 각 멤버가 액세스하는 방법을 지정할 수 있습니다. 클래스 또는 어셈블리 외부에서 사용하지 않으려는 메서드 및 변수를 숨겨 코딩 오류 또는 악의적인 악용 가능성을 제한할 수 있습니다. 자세한 내용은 [개체 지향 프로그래밍](#) 자습서를 참조하세요.

## 멤버

형식의 '멤버'는 모든 메서드, 필드, 상수, 속성, 이벤트를 포함합니다. 다른 언어에는 있지만 C#에는 전역 변수 또는 메서드가 없습니다. 프로그램의 진입점인 `Main` 메서드까지도 클래스나 구조체 내에 선언되어야 합니다([최상위 문](#)을 사용하는 경우 암시적으로).

다음 목록은 클래스, 구조체 또는 레코드에서 선언될 수 있는 모든 다양한 종류의 멤버입니다.

- 필드
- 상수
- 속성
- 메서드
- 생성자
- 이벤트
- 종료자
- 인덱서
- 연산자
- 중첩 형식

자세한 내용은 [멤버](#)를 참조하세요.

## 접근성

일부 메서드 및 속성은 클라이언트 코드라고 하는 클래스 또는 구조체 외부의 코드에서 호출하거나 액세스할 수 있습니다. 다른 메서드 및 속성은 클래스 또는 구조체 자체에서만 사용할 수 있습니다. 의도된 클라이언트 코드에서만 연결될 수 있도록 코드의 액세스 가능성을 제한하는 것이 중요합니다. 어떻게 형식 및 해당 멤버가 다음 액세스 한정자를 사용하여 클라이언트 코드에 액세스할 수 있는지 지정합니다.

- [public](#)
- [protected](#)
- [internal](#)
- [protected internal](#)
- [private](#)
- [private protected](#)

기본 액세스 가능성은 [private](#)입니다.

## 상속

클래스(구조체는 아님)는 상속 개념을 지원합니다. 다른 클래스('기본 클래스'라고 함)에서 파생되는 클래스는 생성자와 종료자를 제외하고 기본 클래스의 모든 public, protected, internal 멤버를 자동으로 포함합니다.

클래스를 [abstract](#)로 선언할 수도 있습니다. 즉, 하나 이상의 해당 메서드에 구현이 없는 상태를 의미합니다. 추상 클래스는 직접 인스턴스화할 수 없지만 누락된 구현을 제공하는 다른 클래스에 대한 기본 클래스로 사용될 수 있습니다. 다른 클래스가 이 클래스에서 상속 받지 못하게 하려면 클래스를 [sealed](#)로 선언할 수도 있습니다.

자세한 내용은 [상속](#) 및 [다형성](#)을 참조하세요.

## 인터페이스

클래스, 구조체, 레코드는 여러 인터페이스를 구현할 수 있습니다. 인터페이스에서 구현하는 것은 형식이 해당 인터페이스에 정의된 모든 메서드를 구현한다는 의미입니다. 자세한 내용은 [인터페이스](#)를 참조하세요.

## 제네릭 형식

클래스, 구조체 및 레코드는 하나 이상의 형식 매개 변수로 정의할 수 있습니다. 클라이언트 코드는 형식의 인스턴스를 만들 때 형식을 제공합니다. 예를 들어 [System.Collections.Generic](#) 네임스페이스의 [List<T>](#) 클래스는 하나의 형식 매개 변수로 정의됩니다. 클라이언트 코드는 [List<string>](#) 또는 [List<int>](#)의 인스턴스를 만들어 목록에 포함될 형식을 지정합니다. 자세한 내용은 [제네릭](#)을 참조하세요.

## 정적 형식

클래스(구조체나 레코드는 아님)를 `static`(으)로 선언할 수 있습니다. `static` 클래스는 `static` 멤버만 포함할 수 있고 `new` 키워드로 인스턴스화할 수 없습니다. 프로그램이 로드될 때 클래스의 단일 복사본만 메모리에 로드되고 해당 멤버는 클래스 이름을 통해 액세스됩니다. 클래스, 구조체 및 레코드에는 `static` 멤버가 포함될 수 있습니다. 자세한 내용은 [정적 클래스 및 정적 클래스 멤버](#)를 참조하세요.

## 중첩 형식

클래스, 구조체 또는 레코드는 다른 클래스, 구조체 또는 레코드 내에 중첩될 수 있습니다. 자세한 내용은 [중첩 형식](#)을 참조하세요.

## 부분 형식(Partial Type)

하나의 코드 파일 및 별도 코드 파일의 다른 부분에서 클래스, 구조체 또는 메서드의 부분을 정의할 수 있습니다. 자세한 내용은 참조 [Partial 클래스 및 메서드](#)합니다.

## 개체 이니셜라이저

해당 속성에 값을 할당하여 클래스 또는 구조체 개체와 개체 컬렉션을 인스턴스화하고 초기화할 수 있습니다. 자세한 내용은 [개체 이니셜라이저를 사용하여 개체를 초기화하는 방법](#)을 참조하세요.

## 익명 형식

명명된 클래스를 만드는 것이 불편하거나 필요하지 않은 상황에서는 무명 형식을 사용합니다. 무명 형식은 명명된 데이터 멤버로 정의됩니다. 자세한 내용은 [무명 형식](#)을 참조하세요.

## 확장명 메서드

별도의 형식을 만들어 파생 클래스를 만들지 않고도 클래스를 “확장”할 수 있습니다. 해당 형식에는 원래 형식에 속한 것처럼 호출할 수 있는 메서드가 포함됩니다. 자세한 내용은 [확장 메서드](#)를 참조하세요.

## 암시적 형식 지역 변수

클래스 또는 구조체 메서드 내에서 암시적 형식 지정을 사용하여 컴파일러가 컴파일 시간에 변수의 형식을 결정하도록 할 수 있습니다. 자세한 내용은 [var\(C# 참조\)](#)을 참조하세요.

## 레코드

클래스 또는 구조체에 `record` 한정자를 추가할 수 있습니다. 레코드는 값 기반 같음을 위한 기본 제공 동작이 있는 형식입니다. 레코드(`record class` 또는 `record struct`)는 다음과 같은 기능을 제공합니다.

- 변경할 수 없는 속성을 사용하여 참조 형식을 만드는 간결한 구문
- 값 같음 레코드 종류의 두 변수는 레코드 종류가 동일한 경우와 모든 필드에 대해 두 레코드의 값이 같은 경우에 같습니다. 클래스는 참조 같음을 사용합니다. 한 클래스 형식의 두 변수가 같은 개체를 참조하면 두 변수가 동일합니다.
- 비파괴적 변형을 위한 간결한 구문 `with` 식을 사용하면 기존 인스턴스의 복사본이지만 지정된 속성 값이 변경된 새 레코드 인스턴스를 만들 수 있습니다.
- 표시를 위한 기본 제공 형식 `ToString` 메서드는 레코드 형식 이름과 퍼블릭 속성의 이름 및 값을 출력합니다.
- 레코드 클래스의 상속 계층 구조에 대한 지원. 레코드 클래스는 상속을 지원합니다. 레코드 구조체는 상속을 지원하지 않습니다.

자세한 내용은 [레코드](#)를 참조하세요.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)을 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.



### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

#### 설명서 문제 열기

#### 제품 사용자 의견 제공

# 개체 - 형식의 인스턴스 만들기

아티클 • 2024. 02. 13.

클래스 또는 구조체 정의는 형식이 수행할 수 있는 작업을 지정하는 청사진과 비슷합니다. 개체는 기본적으로 청사진에 따라 구성 및 할당된 메모리 블록입니다. 프로그램에서 동일한 클래스의 많은 개체를 만들 수 있습니다. 개체를 인스턴스라고도 하며, 명명된 변수나 배열 또는 컬렉션에 저장할 수 있습니다. 클라이언트 코드는 이러한 변수를 사용하여 메서드를 호출하고 개체의 공용 속성에 액세스하는 코드입니다. C#과 같은 개체 지향 언어에서 일반적인 프로그램은 동적으로 상호 작용하는 여러 개체로 구성됩니다.

## ① 참고

정적 형식은 여기에 설명된 것과 다르게 동작합니다. 자세한 내용은 [static 클래스 및 static 클래스 멤버](#)를 참조하세요.

## 구조체 인스턴스 및 클래스 인스턴스

클래스는 참조 형식이므로 클래스 개체의 변수는 관리되는 힙의 개체 주소에 대한 참조를 포함합니다. 동일한 형식의 두 번째 변수가 첫 번째 변수에 할당된 경우 두 변수 모두 해당 주소의 개체를 참조합니다. 이 점은 이 문서의 뒷부분에서 자세히 설명합니다.

클래스 인스턴스는 [new 연산자](#)를 사용하여 생성됩니다. 다음 예제에서 `Person`은 형식이고 `person1` 및 `person2`는 해당 형식의 인스턴스 또는 개체입니다.

C#

```
using System;

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
    // Other properties, methods, events...
}

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

```

        Person person1 = new Person("Leopold", 6);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name,
person1.Age);

        // Declare new person, assign person1 to it.
        Person person2 = person1;

        // Change the name of person2, and person1 also changes.
        person2.Name = "Molly";
        person2.Age = 16;

        Console.WriteLine("person2 Name = {0} Age = {1}", person2.Name,
person2.Age);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name,
person1.Age);
    }
}

/*
Output:
person1 Name = Leopold Age = 6
person2 Name = Molly Age = 16
person1 Name = Molly Age = 16
*/

```

구조체는 값 형식이므로 구조체 개체의 변수는 전체 개체의 복사본을 포함합니다. 다음 예제와 같이 `new` 연산자를 사용하여 구조체 인스턴스를 만들 수도 있지만 필수는 아닙니다.

C#

```

using System;

namespace Example
{
    public struct Person
    {
        public string Name;
        public int Age;
        public Person(string name, int age)
        {
            Name = name;
            Age = age;
        }
    }

    public class Application
    {
        static void Main()
        {
            // Create struct instance and initialize by using "new".
            // Memory is allocated on thread stack.
            Person p1 = new Person("Alex", 9);
        }
    }
}

```

```

        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

        // Create new struct object. Note that struct can be
        initialized
        // without using "new".
        Person p2 = p1;

        // Assign values to p2 members.
        p2.Name = "Spencer";
        p2.Age = 7;
        Console.WriteLine("p2 Name = {0} Age = {1}", p2.Name, p2.Age);

        // p1 values remain unchanged because p2 is copy.
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);
    }
}

/*
Output:
p1 Name = Alex Age = 9
p2 Name = Spencer Age = 7
p1 Name = Alex Age = 9
*/
}

```

`p1` 및 `p2` 둘 다에 대한 메모리가 스택에서 할당됩니다. 해당 메모리는 선언된 형식 또는 메서드와 함께 회수됩니다. 이는 할당 시 구조체가 복사되는 이유 중 하나입니다. 반면, 클래스 인스턴스에 할당된 메모리는 개체에 대한 모든 참조가 범위를 벗어날 때 공용 언어 런타임에 의해 자동으로 회수(가비지 수집)됩니다. C++에서처럼 클래스 개체를 결정적으로 삭제할 수는 없습니다. .NET의 가비지 수집에 대한 자세한 내용은 [가비지 수집](#)을 참조하세요.

### ① 참고

관리되는 힙의 메모리 할당 및 할당 취소는 공용 언어 런타임에서 고도로 최적화되어 있습니다. 대부분의 경우 힙에서 클래스 인스턴스를 할당하는 경우와 스택에서 구조체 인스턴스를 할당하는 경우의 성능 차이는 크지 않습니다.

## 개체 ID와 값 같음 비교

두 개체가 같은지를 비교하는 경우 먼저 두 변수가 메모리에서 동일한 개체를 나타내는지 또는 해당 필드 값이 하나 이상 같은지를 알고 싶은 것인지 구분해야 합니다. 값을 비교하려는 경우 개체가 값 형식(구조체) 또는 참조 형식(클래스, 대리자, 배열)의 인스턴스인지 고려해야 합니다.

- 두 클래스 인스턴스가 메모리의 동일한 위치를 참조하는지 확인하려면(즉, *ID*가 같음) 정적 [Object.Equals](#) 메서드를 사용합니다. [System.Object](#)은 사용자 정의 구조체 및 클래스를 포함하여 모든 값 형식 및 참조 형식에 대한 암시적 기본 클래스입니다.
- 두 구조체 인스턴스의 인스턴스 필드 값이 같은지 확인하려면 [ValueType.Equals](#) 메서드를 사용합니다. 모든 구조체가 [System.ValueType](#)에서 암시적으로 상속하기 때문에 다음 예제와 같이 개체에서 직접 메서드를 호출합니다.

C#

```
// Person is defined in the previous example.

//public struct Person
//{
//    public string Name;
//    public int Age;
//    public Person(string name, int age)
//    {
//        Name = name;
//        Age = age;
//    }
//}

Person p1 = new Person("Wallace", 75);
Person p2 = new Person("", 42);
p2.Name = "Wallace";
p2.Age = 75;

if (p2.Equals(p1))
    Console.WriteLine("p2 and p1 have the same values.");

// Output: p2 and p1 have the same values.
```

[Equals](#)의 [System.ValueType](#) 구현에서는 경우에 따라 boxing 및 리플렉션을 사용합니다. 형식에 따라 효율적인 같음 알고리즘을 제공하는 방법에 대한 자세한 내용은 [형식의 값 같음을 정의하는 방법](#)을 참조하세요. 레코드는 같음을 위해 값 의미 체계를 사용하는 참조 형식입니다.

- 두 클래스 인스턴스의 필드 값이 같은지 확인하기 위해 [Equals](#) 메서드 또는 [== 연산자](#)를 사용할 수 있습니다. 그러나 클래스가 해당 형식의 개체에 대해 "같음"이 무엇을 의미하는지의 사용자 지정 정의를 제공하도록 재정의 또는 오버로드한 경우에만 사용합니다. 클래스는 [IEquatable<T>](#) 인터페이스 또는 [IEqualityComparer<T>](#) 인터페이스도 구현할 수 있습니다. 두 인터페이스 모두 값이 같은지를 테스트하는 데 사용할 수 있는 메서드를 제공합니다. [Equals](#)를 재정의하는 고유한 클래스를 디자인하는 경우 [형식의 값 같음을 정의하는 방법](#) 및 [Object.Equals\(Object\)](#)에 명시된 지침을 따라야 합니다.

# 관련 섹션

자세한 내용은 다음에서 확인합니다.

- 클래스
- 생성자
- 종료자
- 이벤트
- object
- 상속
- class
- 구조체 형식
- new 연산자
- 공용 형식 시스템

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

ଓ 설명서 문제 열기

ଓ 제품 사용자 의견 제공

# 상속 - 형식을 파생하여 보다 특수화된 동작을 만듭니다.

아티클 • 2023. 04. 07.

캡슐화 및 다형성과 함께 상속은 개체 지향 프로그래밍의 세 가지 주요 특징 중 하나입니다. 상속을 사용하면 다른 클래스에 정의된 동작을 다시 사용, 확장 및 수정하는 새 클래스를 만들 수 있습니다. 멤버가 상속되는 클래스를 **기본 클래스**라고 하고 해당 멤버를 상속하는 클래스를 **파생 클래스**라고 합니다. 파생 클래스에는 직접적인 기본 클래스가 하나만 있을 수 있습니다. 그러나 상속은 전이됩니다. `ClassC`가 `ClassB`에서 파생된 클래스이고 `ClassB`가 `ClassA`에서 파생된 클래스이면 `ClassC`는 `ClassB`와 `ClassA`에서 선언된 멤버를 상속합니다.

## ① 참고

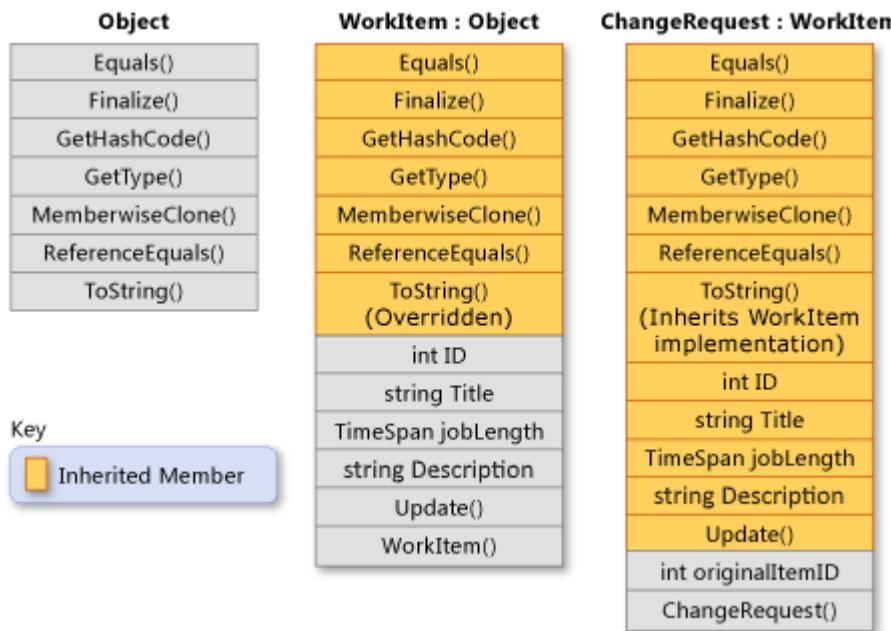
구조체는 상속을 지원하지 않지만 인터페이스를 구현할 수 있습니다.

파생 클래스는 개념적 측면에서 기본 클래스의 특수화입니다. 예를 들어 기본 클래스 `Animal`이 있는 경우 `Mammal`이라는 하나의 파생 클래스와 `Reptile`이라는 다른 파생 클래스가 있을 수 있습니다. `Mammal`은 `Animal`이고 `Reptile`은 `Animal`이지만 각 파생 클래스는 기본 클래스의 서로 다른 특수화를 나타냅니다.

인터페이스 선언은 그 멤버의 기본 구현을 정의할 수 있습니다. 이러한 구현은 파생된 인터페이스에 의해, 그리고 해당 인터페이스를 구현하는 클래스에 의해 상속됩니다. 기본 인터페이스 메서드에 대한 자세한 내용은 인터페이스 문서를 참조 [하세요](#).

다른 클래스에서 파생할 클래스를 정의하는 경우 파생 클래스는 해당 생성자와 종료자를 제외하고 기본 클래스의 모든 멤버를 암시적으로 얻게 됩니다. 파생 클래스는 기본 클래스에서 코드를 다시 구현하지 않고 다시 사용합니다. 파생 클래스에서 더 많은 멤버를 추가할 수 있습니다. 파생 클래스는 기본 클래스의 기능을 확장합니다.

다음 그림은 일부 비즈니스 프로세스의 작업 항목을 나타내는 `WorkItem` 클래스를 보여 줍니다. 모든 클래스와 마찬가지로, `System.Object`에서 파생되고 해당 메서드를 모두 상속합니다. `WorkItem`는 6명의 멤버를 추가합니다. 생성자는 상속되지 않으므로 이러한 멤버에는 생성자도 포함됩니다. `ChangeRequest` 클래스는 `WorkItem`에서 상속되며 특정 종류의 작업 항목을 나타냅니다. `ChangeRequest`는 `WorkItem` 및 `Object`에서 상속하는 멤버에 둘 이상의 멤버를 추가합니다. 고유한 생성자를 추가해야 하며, `originalItemID`도 추가합니다. `originalItemID` 속성을 사용하면 `ChangeRequest` 인스턴스를 변경 요청이 적용되는 원래 `WorkItem`에 연결할 수 있습니다.



다음 예제에서는 앞의 그림에서 보여 주는 클래스 관계가 C#에서 어떻게 표현되는지를 보여 줍니다. 또한 이 예제에서 `WorkItem`은 가상 메서드 `Object.ToString`을 재정의하는 방법과 `ChangeRequest` 클래스가 메서드의 `WorkItem` 구현을 상속하는 방법을 보여 줍니다. 첫 번째 블록은 클래스를 정의합니다.

C#

```
// WorkItem implicitly inherits from the Object class.
public class WorkItem
{
    // Static field currentID stores the job ID of the last WorkItem that
    // has been created.
    private static int currentID;

    //Properties.
    protected int ID { get; set; }
    protected string Title { get; set; }
    protected string Description { get; set; }
    protected TimeSpan jobLength { get; set; }

    // Default constructor. If a derived class does not invoke a base-
    // class constructor explicitly, the default constructor is called
    // implicitly.
    public WorkItem()
    {
        ID = 0;
        Title = "Default title";
        Description = "Default description.";
        jobLength = new TimeSpan();
    }

    // Instance constructor that has three parameters.
    public WorkItem(string title, string desc, TimeSpan joblen)
    {
        this.ID = GetNextID();
```

```
        this.Title = title;
        this.Description = desc;
        this.jobLength = joblen;
    }

    // Static constructor to initialize the static member, currentID. This
    // constructor is called one time, automatically, before any instance
    // of WorkItem or ChangeRequest is created, or currentID is referenced.
    static WorkItem() => currentID = 0;

    // currentID is a static field. It is incremented each time a new
    // instance of WorkItem is created.
    protected int GetNextID() => ++currentID;

    // Method Update enables you to update the title and job length of an
    // existing WorkItem object.
    public void Update(string title, TimeSpan joblen)
    {
        this.Title = title;
        this.jobLength = joblen;
    }

    // Virtual method override of the ToString method that is inherited
    // from System.Object.
    public override string ToString() =>
        $"{this.ID} - {this.Title}";
}

// ChangeRequest derives from WorkItem and adds a property (originalItemID)
// and two constructors.
public class ChangeRequest : WorkItem
{
    protected int originalItemID { get; set; }

    // Constructors. Because neither constructor calls a base-class
    // constructor explicitly, the default constructor in the base class
    // is called implicitly. The base class must contain a default
    // constructor.

    // Default constructor for the derived class.
    public ChangeRequest() { }

    // Instance constructor that has four parameters.
    public ChangeRequest(string title, string desc, TimeSpan jobLen,
                         int originalID)
    {
        // The following properties and the GetNexID method are inherited
        // from WorkItem.
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = jobLen;

        // Property originalItemID is a member of ChangeRequest, but not
        // of WorkItem.
    }
}
```

```
    this.originalItemID = originalID;
}
}
```

다음 블록은 기본 클래스와 파생 클래스를 사용하는 방법을 보여 줍니다.

C#

```
// Create an instance of WorkItem by using the constructor in the
// base class that takes three arguments.
WorkItem item = new WorkItem("Fix Bugs",
                             "Fix all bugs in my code branch",
                             new TimeSpan(3, 4, 0, 0));

// Create an instance of ChangeRequest by using the constructor in
// the derived class that takes four arguments.
ChangeRequest change = new ChangeRequest("Change Base Class Design",
                                         "Add members to the class",
                                         new TimeSpan(4, 0, 0),
                                         1);

// Use the ToString method defined in WorkItem.
Console.WriteLine(item.ToString());

// Use the inherited Update method to change the title of the
// ChangeRequest object.
change.Update("Change the Design of the Base Class",
             new TimeSpan(4, 0, 0));

// ChangeRequest inherits WorkItem's override of ToString.
Console.WriteLine(change.ToString());
/* Output:
   1 - Fix Bugs
   2 - Change the Design of the Base Class
*/
```

## 추상 메서드와 가상 메서드

기본 클래스가 메서드를 [virtual](#)로 선언하는 경우 파생 클래스가 자체 구현으로 메서드를 [override](#)할 수 있습니다. 기본 클래스가 멤버를 [abstract](#)로 선언하는 경우 해당 클래스에서 직접 상속되는 모든 비추상 클래스에서 메서드를 재정의해야 합니다. 파생 클래스 자체가 [abstract](#)인 경우 직접 구현하지 않고 추상 멤버를 상속합니다. 추상 멤버 및 가상 멤버는 개체 지향 프로그래밍의 두 번째 주요 특징인 다형성의 기초가 됩니다. 자세한 내용은 [다형성](#)을 참조하세요.

## 추상 기본 클래스

`new` 연산자를 사용한 직접 인스턴스화를 방지하려는 경우 클래스를 `abstract`로 선언할 수 있습니다. 추상 클래스는 해당 클래스에서 새 클래스가 파생되는 경우에만 사용할 수 있습니다. 추상 클래스에는 그 자체가 `abstract`로 선언된 메서드 시그니처가 하나 이상 포함될 수 있습니다. 이러한 시그니처는 매개 변수와 반환 값을 지정하지만 구현(메서드 본문)이 없습니다. 추상 클래스는 추상 멤버를 포함하지 않아도 됩니다. 그러나 클래스에 추상 멤버가 포함되지 않은 경우 클래스 자체를 `abstract`로 선언해야 합니다. 그 자체가 추상이 아닌 파생 클래스는 추상 기본 클래스에서 모든 추상 메서드에 대한 구현을 제공해야 합니다.

## 인터페이스

'인터페이스'는 멤버 집합을 정의하는 참조 형식입니다. 인터페이스를 구현하는 모든 클래스와 구조체는 해당 멤버 집합을 구현해야 합니다. 인터페이스는 임의 또는 모든 구성 원에 대한 기본 구현을 정의할 수 있습니다. 하나의 직접 기본 클래스에서만 파생할 수 있는 경우에도 클래스에서 여러 인터페이스를 구현할 수 있습니다.

인터페이스는 "is a" 관계가 없을 수도 있는 클래스에 대해 특정 기능을 정의하는 데 사용됩니다. 예를 들어 `System.IEquatable<T>` 인터페이스는 모든 클래스 또는 구조체에 의해 구현되어 해당 형식의 두 개체가 동일한지 여부를(해당 형식에서 정의하는 동일성 기준에 따라) 확인할 수 있습니다. `IEquatable<T>`은 기본 클래스와 파생 클래스 간에 존재하는 것과 같은 "is a" 관계(예: `Mammal` is an `Animal`)를 암시하지 않습니다. 자세한 내용은 [인터페이스](#)를 참조하세요.

## 추가 파생 방지

클래스는 자신이나 멤버를 `sealed`로 선언하여 다른 클래스가 해당 클래스나 그 멤버에서 상속할 수 없도록 할 수 있습니다.

## 파생 클래스의 기본 클래스 멤버 숨기기

파생 클래스는 동일한 이름과 시그니처로 멤버를 선언하여 기본 클래스 멤버를 숨길 수 있습니다. `new` 한정자를 사용하여 멤버가 기본 멤버를 재정의하지 않음을 명시적으로 나타낼 수 있습니다. `new`의 사용은 필수가 아니지만 `new`를 사용하지 않을 경우 컴파일러 경고가 생성됩니다. 자세한 내용은 [Override 및 New 키워드를 사용하여 버전 관리](#) 및 [Override 및 New 키워드를 사용해야 하는 경우](#)를 참조하세요.

# 다형성

아티클 • 2023. 04. 07.

다형성은 흔히 캡슐화와 상속의 뒤를 이어 개체 지향 프로그래밍의 세 번째 특징으로 일컬어집니다. 다형성은 "여러 형태"를 의미하는 그리스어 단어이며 다음과 같은 두 가지 고유한 측면을 가집니다.

- 런타임에 파생 클래스의 개체가 메서드 매개 변수 및 컬렉션 또는 배열과 같은 위치에서 기본 클래스의 개체로 처리될 수 있습니다. 이러한 다형성이 발생하면 개체의 선언된 형식이 더 이상 해당 런타임 형식과 같지 않습니다.
- 기본 클래스는 [가상 메서드](#)를 정의 및 구현할 수 있으며, 파생 클래스는 이러한 가상 메서드를 [재정의](#)할 수 있습니다. 즉, 파생 클래스는 고유한 정의 및 구현을 제공합니다. 런타임에 클라이언트 코드에서 메서드를 호출하면 CLR은 개체의 런타임 형식을 조회하고 가상 메서드의 해당 재정의를 호출합니다. 소스 코드에서 기본 클래스에 대해 메서드를 호출하여 메서드의 파생 클래스 버전이 실행되도록 할 수 있습니다.

가상 메서드를 사용하면 동일한 방식으로 관련 개체 그룹에 대한 작업을 수행할 수 있습니다. 예를 들어, 사용자가 그리기 화면에서 다양한 종류의 도형을 만들 수 있는 그리기 애플리케이션이 있다고 가정합니다. 컴파일 시 사용자가 만들 특정 유형의 셰이프를 알 수 없습니다. 그러나 애플리케이션은 만들어지는 다양한 모든 형식의 도형을 추적해야 하며, 사용자의 마우스 작업에 따라 이러한 도형을 업데이트해야 합니다. 다음과 같은 기본 두 단계로 다형성을 사용하여 이 문제를 해결할 수 있습니다.

1. 각 특정 도형 클래스가 공통 기본 클래스에서 파생되는 클래스 계층 구조를 만듭니다.
2. 가상 메서드를 사용하여 기본 클래스 메서드에 대한 단일 호출을 통해 모든 파생 클래스에 대해 적절한 메서드를 호출합니다.

먼저, `Shape`라는 기본 클래스를 만들고 `Rectangle`, `Circle` 및 `Triangle`과 같은 파생 클래스를 만듭니다. `Shape` 클래스에 `Draw`라는 가상 메서드를 제공하고, 각 파생 클래스에서 이를 재정의하여 클래스가 나타내는 특정 도형을 그립니다. `List<Shape>` 개체를 만들고 이 개체에 `Circle`, `Triangle` 및 `Rectangle`을 추가합니다.

C#

```
public class Shape
{
    // A few example members
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }
```

```

// Virtual method
public virtual void Draw()
{
    Console.WriteLine("Performing base class drawing tasks");
}
}

public class Circle : Shape
{
    public override void Draw()
    {
        // Code to draw a circle...
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}

public class Rectangle : Shape
{
    public override void Draw()
    {
        // Code to draw a rectangle...
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}

public class Triangle : Shape
{
    public override void Draw()
    {
        // Code to draw a triangle...
        Console.WriteLine("Drawing a triangle");
        base.Draw();
    }
}

```

그리기 화면을 업데이트하려면 `foreach` 루프를 사용하여 목록을 반복하고 목록의 각 `Shape` 개체에 대해 `Draw` 메서드를 호출합니다. 목록의 각 개체에 선언된 형식 `Shape` 가 있더라도 이는 호출될 런타임 형식(각 파생 클래스에 있는 메서드의 재정의된 버전)입니다.

C#

```

// Polymorphism at work #1: a Rectangle, Triangle and Circle
// can all be used wherever a Shape is expected. No cast is
// required because an implicit conversion exists from a derived
// class to its base class.
var shapes = new List<Shape>
{
    new Rectangle(),
    new Triangle(),
    new Circle()
};

```

```

// Polymorphism at work #2: the virtual method Draw is
// invoked on each of the derived classes, not the base class.
foreach (var shape in shapes)
{
    shape.Draw();
}
/* Output:
   Drawing a rectangle
   Performing base class drawing tasks
   Drawing a triangle
   Performing base class drawing tasks
   Drawing a circle
   Performing base class drawing tasks
*/

```

C#에서 모든 형식은 사용자 정의 형식을 포함한 모든 형식이 [Object](#)에서 파생되므로 다른 형식입니다.

## 다형성 개요

### 가상 멤버

파생 클래스가 기본 클래스에서 상속되는 경우 기본 클래스의 모든 멤버가 포함됩니다. 기본 클래스에 선언된 모든 동작은 파생 클래스의 일부입니다. 이렇게 하면 파생 클래스의 개체를 기본 클래스의 개체로 처리할 수 있습니다. 액세스 한정자(`public`, `protected` 등 `private`)는 파생 클래스 구현에서 해당 멤버에 액세스할 수 있는지 여부를 결정합니다. 가상 메서드는 파생 클래스의 동작에 대해 디자이너에 다른 선택 항목을 제공합니다.

- 파생 클래스가 기본 클래스의 가상 멤버를 재정의하여 새로운 동작을 정의할 수 있습니다.
- 파생 클래스는 재정의하지 않고 가장 가까운 기본 클래스 메서드를 상속하여 기존 동작을 유지하지만 추가 파생 클래스가 메서드를 재정의할 수 있도록 할 수 있습니다.
- 파생 클래스가 기본 클래스 구현을 숨기는 멤버의 새로운 비가상 구현을 정의할 수 있습니다.

파생 클래스는 기본 클래스 멤버가 `virtual` 또는 `abstract`로 선언된 경우에만 기본 클래스 멤버를 재정의할 수 있습니다. 파생 멤버는 `override` 키워드를 사용하여 메서드가 가상 호출에 참여하도록 되어 있음을 명시적으로 나타내야 합니다. 다음 코드는 예제를 제공합니다.

```

public class BaseClass
{
    public virtual void DoWork() { }
    public virtual int WorkProperty
    {
        get { return 0; }
    }
}
public class DerivedClass : BaseClass
{
    public override void DoWork() { }
    public override int WorkProperty
    {
        get { return 0; }
    }
}

```

필드는 가상일 수 없습니다. 메서드, 속성, 이벤트 및 인덱서만 가상일 수 있습니다. 파생 클래스가 가상 멤버를 재정의하면 해당 멤버는 해당 클래스의 인스턴스가 기본 클래스의 인스턴스로 액세스되는 경우에도 호출됩니다. 다음 코드는 예제를 제공합니다.

C#

```

DerivedClass B = new DerivedClass();
B.DoWork(); // Calls the new method.

BaseClass A = B;
A.DoWork(); // Also calls the new method.

```

가상 메서드 및 속성을 통해 파생 클래스는 메서드의 기본 클래스 구현을 사용할 필요 없이 기본 클래스를 확장할 수 있습니다. 자세한 내용은 [Override 및 New 키워드를 사용하여 버전 관리](#)를 참조하세요. 인터페이스는 구현이 파생 클래스에 남겨진 메서드 또는 메서드 집합을 정의하는 또 다른 방법을 제공합니다.

## 새 멤버로 기본 클래스 멤버 숨기기

파생 클래스가 기본 클래스의 멤버와 동일한 이름을 갖는 멤버를 갖도록 하려면 `new` 키워드를 사용하여 기본 클래스 멤버를 숨길 수 있습니다. `new` 키워드는 바꿀 클래스 멤버의 반환 형식 앞에 배치됩니다. 다음 코드는 예제를 제공합니다.

C#

```

public class BaseClass
{
    public void DoWork() { WorkField++; }
    public int WorkField;
    public int WorkProperty

```

```

    {
        get { return 0; }
    }

public class DerivedClass : BaseClass
{
    public new void DoWork() { WorkField++; }
    public new int WorkField;
    public new int WorkProperty
    {
        get { return 0; }
    }
}

```

파생 클래스의 인스턴스를 기본 클래스의 인스턴스로 캐스팅하여 숨겨진 기본 클래스 멤버를 클라이언트 코드에서 액세스할 수 있습니다. 예를 들어:

C#

```

DerivedClass B = new DerivedClass();
B.DoWork(); // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork(); // Calls the old method.

```

## 파생 클래스가 가상 멤버를 재정의하지 못하도록 설정

가상 멤버는 가상 멤버와 원래 가상 멤버를 선언한 클래스에서 선언된 클래스의 개수와 관계없이 가상으로 유지됩니다. 클래스 A가 가상 멤버를 선언하고, 클래스 B가 A에서 파생되며, 클래스 C가 B에서 파생되면 클래스 C는 클래스 B가 해당 멤버에 대한 재정의를 선언했는지 여부와 관계없이 가상 멤버를 상속하며, 해당 가상 멤버를 재정의할 수 있습니다. 다음 코드는 예제를 제공합니다.

C#

```

public class A
{
    public virtual void DoWork() { }
}

public class B : A
{
    public override void DoWork() { }
}

```

파생 클래스는 재정의를 sealed로 선언하여 가상 상속을 중지할 수 있습니다. 가상 상속을 중지하려면 클래스 멤버 선언에서 override 키워드 앞에 sealed 키워드를 배치해야

합니다. 다음 코드는 예제를 제공합니다.

C#

```
public class C : B
{
    public sealed override void DoWork() { }
```

앞의 예제에서 `DoWork` 메서드는 `c`에서 파생된 모든 클래스에 대해 더 이상 가상이 아닙니다. 그러나 이 메서드는 `c`의 인스턴스가 형식 `B` 또는 형식 `A`로 캐스팅되더라도 여전히 `C`의 인스턴스에 대해서는 가상입니다. `sealed` 메서드는 다음 예제에서처럼 `new` 키워드를 사용하여 파생 클래스로 바꿀 수 있습니다.

C#

```
public class D : C
{
    public new void DoWork() { }
```

이 경우 형식 `D` 변수를 사용하여 `D`에 대해 `DoWork`을 호출하면 새 `DoWork`이 호출됩니다. 형식 `C`, `B` 또는 `A` 변수를 사용하여 `D`의 인스턴스에 액세스하면 `DoWork`에 대한 호출은 가상 상속의 규칙을 따라 해당 호출을 클래스 `C`에 대한 `DoWork`의 구현으로 라우팅합니다.

## 파생 클래스에서 기본 클래스 가상 멤버에 액세스

메서드 또는 속성을 바꾸었거나 재정의한 파생 클래스는 계속해서 `base` 키워드를 사용하여 기본 클래스에 대한 메서드 또는 속성에 액세스할 수 있습니다. 다음 코드는 예제를 제공합니다.

C#

```
public class Base
{
    public virtual void DoWork() /*...*/
}
public class Derived : Base
{
    public override void DoWork()
    {
        //Perform Derived's work here
        //...
        // Call DoWork on base class
        base.DoWork();
```

```
    }  
}
```

자세한 내용은 [base](#)를 참조하세요.

### ① 참고

가상 멤버는 `base`를 사용하여 자체 구현에서 해당 멤버의 기본 클래스 구현을 호출하는 것이 좋습니다. 기본 클래스 동작이 발생하도록 하면 파생 클래스가 파생 클래스에 대한 동작 구현에 집중할 수 있습니다. 기본 클래스 구현이 호출되지 않는 경우 해당 동작이 기본 클래스의 동작과 호환되도록 하는 것은 파생 클래스의 책임입니다.

# 패턴 일치 개요

아티클 • 2024. 03. 20.

'패턴 일치'는 식에 특정 특징이 있는지 확인하기 위해 테스트하는 기법입니다. C# 패턴 일치는 식을 테스트하고 식이 일치하는 경우 작업을 수행하기 위한 보다 간결한 구문을 제공합니다. "is expression"은 패턴 일치를 지원하여 식을 테스트하고 해당 식의 결과에 새 변수를 조건부로 선언합니다. "switch 식"을 사용하면 식의 처음 일치 패턴을 기준으로 작업을 수행할 수 있습니다. 이 두 식은 다양한 '패턴' 어휘를 지원합니다.

이 문서에서는 패턴 일치를 사용할 수 있는 시나리오를 살펴봅니다. 이러한 기법은 코드의 가독성과 정확성을 향상하는 데 사용할 수 있습니다. 적용할 수 있는 모든 패턴에 대해 알아보려면 언어 참조에서 [패턴](#)에 관한 문서를 살펴보세요.

## null 검사

패턴 일치에서 가장 널리 사용되는 시나리오는 값이 `null`이 아닌지 확인하는 것입니다. 다음 예제를 사용하여 `null`이 아닌지를 테스트할 때 `null` 허용 값 형식을 테스트하고 기본 형식으로 변환할 수 있습니다.

C#

```
int? maybe = 12;

if (maybe is int number)
{
    Console.WriteLine($"The nullable int 'maybe' has the value {number}");
}
else
{
    Console.WriteLine("The nullable int 'maybe' doesn't hold a value");
}
```

위 코드는 변수의 형식을 테스트하고 여기에 새 값을 할당하는 '선언 패턴'입니다. 이 언어의 규칙 덕분에 다른 언어보다 이 기법을 더 안전하게 사용할 수 있습니다. 변수 `number`는 액세스만 가능하며 `if` 절의 실제 부분에 할당됩니다. `else` 절에서나 `if` 블록 뒤에서 등 다른 곳에서 액세스하려고 하면 컴파일러가 오류를 발생시킵니다. 이 패턴은 `==` 연산자를 사용하지 않으므로 형식이 `==` 연산자를 오버로드하는 경우에도 작동합니다. 따라서 이것은 `not` 패턴을 추가하여 `null` 참조를 확인하는 이상적인 방법이 됩니다.

C#

```
string? message = ReadMessageOrDefault();
```

```
if (message is not null)
{
    Console.WriteLine(message);
}
```

앞에 나온 예제에서는 '[상수 패턴](#)'을 사용하여 변수를 `null`에 비교했습니다. `not`은 부정된 패턴이 일치하지 않는 경우 일치하는 '[논리 패턴](#)'입니다.

## 형식 테스트

패턴 일치를 위해 사용되는 또 다른 일반적인 방법은 변수가 지정된 형식과 일치하는지 테스트하는 것입니다. 예를 들어, 다음 코드는 변수가 `null`이 아닌 값인지 테스트하며 `System.Collections.Generic.IList<T>` 인터페이스를 구현합니다. 이 경우 해당 목록의 `ICollection<T>.Count` 속성을 사용하여 중간 인덱스 찾기를 수행합니다. 선언 패턴은 변수의 컴파일 시간 형식과 관계없이 `null` 값과 일치하지 않습니다. 아래 코드는 `IList`을 구현하지 않는 형식으로부터 보호하는 것 외에도 `null`로부터 보호합니다.

C#

```
public static T MidPoint<T>(IEnumerable<T> sequence)
{
    if (sequence is IList<T> list)
    {
        return list[list.Count / 2];
    }
    else if (sequence is null)
    {
        throw new ArgumentNullException(nameof(sequence), "Sequence can't be null.");
    }
    else
    {
        int halfLength = sequence.Count() / 2 - 1;
        if (halfLength < 0) halfLength = 0;
        return sequence.Skip(halfLength).First();
    }
}
```

`switch` 식에 동일한 테스트를 적용하여 변수를 여러 형식에 대해 테스트할 수 있습니다. 이 정보는 특정 런타임 형식을 기반으로 더 나은 알고리즘을 만드는 데 사용할 수 있습니다.

## 불연속 값 비교

변수를 테스트하여 특정 값에 대한 일치 항목을 찾을 수도 있습니다. 다음 코드는 열거형에서 선언된 모든 가능한 값에 대해 값을 테스트하는 예를 보여 줍니다.

C#

```
public State PerformOperation(Operation command) =>
    command switch
    {
        Operation.SystemTest => RunDiagnostics(),
        Operation.Start => StartSystem(),
        Operation.Stop => StopSystem(),
        Operation.Reset => ResetToReady(),
        _ => throw new ArgumentException("Invalid enum value for command",
            nameof(command)),
    };
}
```

위의 예제에서는 열거형의 값을 기준으로 디스패치된 메서드를 보여 주었습니다. 마지막 케이스는 모든 값과 일치하는 ‘무시 패턴’입니다. 이 케이스는 값이 정의된 enum 값 중 어느 것과도 일치하지 않는 경우의 오류 조건을 처리합니다. 해당 스위치 암을 생략하면 컴파일러는 패턴 식이 가능한 모든 입력 값을 처리하지 않는다고 경고합니다. 런타임에는 검사 대상 개체가 switch 암(arm) 중 어느 것과도 일치하지 않는 경우 switch 식이 예외를 throw합니다. 일련의 열거형 값 대신 숫자형 상수를 사용할 수 있습니다. 명령을 나타내는 상수 문자열에 대해서도 이와 비슷한 기법을 사용할 수 있습니다.

C#

```
public State PerformOperation(string command) =>
    command switch
    {
        "SystemTest" => RunDiagnostics(),
        "Start" => StartSystem(),
        "Stop" => StopSystem(),
        "Reset" => ResetToReady(),
        _ => throw new ArgumentException("Invalid string value for command",
            nameof(command)),
    };
}
```

위의 예제에서는 동일한 알고리즘을 보여 주지만 열거형 대신 문자열 값을 사용합니다. 이 시나리오는 애플리케이션이 일반적인 데이터 형식이 아닌 텍스트 명령에 응답하는 경우에 사용할 수 있습니다. C# 11부터 다음 샘플과 같이 Span<char> 또는 ReadOnlySpan<char> 을(를) 사용하여 상수 문자열 값을 테스트할 수도 있습니다.

C#

```
public State PerformOperation(ReadOnlySpan<char> command) =>
    command switch
    {
```

```

    "SystemTest" => RunDiagnostics(),
    "Start" => StartSystem(),
    "Stop" => StopSystem(),
    "Reset" => ResetToReady(),
    _ => throw new ArgumentException("Invalid string value for command",
        nameof(command)),
};

}

```

위의 모든 예제에서 '무시 패턴'으로 모든 입력이 처리되었는지 확인할 수 있습니다. 컴파일러는 모든 가능한 입력값이 처리되었는지 확인함으로써 이 작업을 도와줍니다.

## 관계형 패턴

'관계형 패턴'을 사용하여 값이 상수와 비교되는 방식을 테스트할 수 있습니다. 예를 들어, 다음 코드는 화씨 온도를 기준으로 물의 상태를 반환합니다.

C#

```

string WaterState(int tempInFahrenheit) =>
    tempInFahrenheit switch
    {
        (> 32) and (< 212) => "liquid",
        < 32 => "solid",
        > 212 => "gas",
        32 => "solid/liquid transition",
        212 => "liquid / gas transition",
    };

```

위 코드는 두 관계형 패턴이 모두 일치하는지 확인하는 결합 `and` [논리 패턴](#)도 보여 줍니다. 분리 `or` 패턴을 사용하여 둘 중 어느 패턴이 일치하는지 확인할 수도 있습니다. 두 관계형 패턴은 괄호로 묶여 있습니다. 어떤 패턴도 명확성을 위해 괄호로 묶을 수 있습니다. 마지막 두 개의 `switch` 암(arm)은 녹는점 케이스와 끓는점 케이스를 처리합니다. 이 두 암(arm)이 없으면 모든 가능한 입력이 처리되지 않았다는 경고가 발생합니다.

앞의 코드는 또한 컴파일러가 패턴 일치 식에 대해 제공하는 또 다른 중요한 기능을 보여 줍니다. 모든 입력 값을 처리하지 않으면 컴파일러가 경고합니다. 또한 스위치 암에 대한 패턴이 이전 패턴에서 적용되는 경우에도 컴파일러에서 경고를 발생합니다. 이를 통해 스위치 식을 자유롭게 리팩터링하고 순서를 조정할 수 있습니다. 동일한 식을 작성하는 또 다른 방법은 다음과 같습니다.

C#

```

string WaterState2(int tempInFahrenheit) =>
    tempInFahrenheit switch
    {
        < 32 => "solid",

```

```
    32 => "solid/liquid transition",
    < 212 => "liquid",
    212 => "liquid / gas transition",
    _ => "gas",
};
```

이전 샘플의 주요 단원과 다른 리팩터링 또는 다시 정렬은 컴파일러가 코드가 가능한 모든 입력을 처리하는지 유효성을 검사한다는 것입니다.

## 여러 입력

지금까지 다룬 모든 패턴은 하나의 입력을 검사. 개체의 여러 속성을 검사하는 패턴을 작성할 수도 있습니다. 다음 `Order` 레코드를 살펴보겠습니다.

C#

```
public record Order(int Items, decimal Cost);
```

위의 위치 지정 레코드 형식은 명시적 위치에 두 개의 멤버를 선언합니다. 먼저 `Items` 가 표시되고 그다음에 주문의 `Cost` 가 표시됩니다. 자세한 내용은 [레코드](#)를 참조하세요.

다음 코드는 품목의 개수와 주문의 금액을 검사하여 할인 가격을 계산합니다.

C#

```
public decimal CalculateDiscount(Order order) =>
    order switch
    {
        { Items: > 10, Cost: > 1000.00m } => 0.10m,
        { Items: > 5, Cost: > 500.00m } => 0.05m,
        { Cost: > 250.00m } => 0.02m,
        null => throw new ArgumentNullException(nameof(order), "Can't
calculate discount on null order"),
        var someObject => 0m,
    };
}
```

처음 두 개의 암(arm)은 `Order`의 두 속성을 검사합니다. 세 번째 암(arm)은 비용만 검사합니다. 그다음 암(arm)은 `null`을 검사하고 마지막 암(arm)은 다른 모든 값의 일치 여부를 확인합니다. `Order` 형식이 적합한 `Deconstruct` 메서드를 정의하는 경우, 패턴에서 속성 이름을 생략하고 분해를 사용하여 속성을 검사할 수 있습니다.

C#

```
public decimal CalculateDiscount(Order order) =>
    order switch
```

```

{
    (> 10, > 1000.00m) => 0.10m,
    (> 5, > 50.00m) => 0.05m,
    { Cost: > 250.00m } => 0.02m,
    null => throw new ArgumentNullException(nameof(order), "Can't
calculate discount on null order"),
    var someObject => 0m,
}

```

위의 코드는 속성이 식에 대해 분해되는 ‘[위치 지정 패턴](#)’을 보여 줍니다.

## 목록 패턴

목록 패턴을 사용하여 목록 또는 배열의 요소를 확인할 수 있습니다. [목록 패턴](#)은 시퀀스의 모든 요소에 패턴을 적용하는 방법을 제공합니다. 또한 무시 항목 패턴(`_`)을 적용하여 모든 요소와 일치시키거나 슬라이스 패턴을 적용하여 0개 이상의 요소와 일치시킬 수 있습니다.

목록 패턴은 데이터가 일반 구조를 따르지 않는 경우 유용한 도구입니다. 패턴 일치를 사용하여 개체 집합으로 변환하는 대신 데이터의 세이프와 값을 테스트할 수 있습니다.

은행 거래가 포함된 텍스트 파일에서 발췌한 다음을 고려합니다.

출력

04-01-2020, DEPOSIT,	Initial deposit,	2250.00
04-15-2020, DEPOSIT,	Refund,	125.65
04-18-2020, DEPOSIT,	Paycheck,	825.65
04-22-2020, WITHDRAWAL,	Debit, Groceries,	255.73
05-01-2020, WITHDRAWAL,	#1102, Rent, apt,	2100.00
05-02-2020, INTEREST,		0.65
05-07-2020, WITHDRAWAL,	Debit, Movies,	12.57
04-15-2020, FEE,		5.55

CSV 형식이지만 일부 행에는 다른 행보다 더 많은 열이 있습니다. 처리의 경우 더 나쁜 경우 형식의 `WITHDRAWAL` 한 열에 사용자가 생성한 텍스트가 포함되며 텍스트에 쉼표가 포함될 수 있습니다. 값 프로세스 데이터를 이 형식으로 캡처하기 위한 `dis` 카드 패턴, 상수 패턴 및 `var` 패턴을 포함하는 목록 패턴입니다.

C#

```

decimal balance = 0m;
foreach (string[] transaction in ReadRecords())
{
    balance += transaction switch
    {
        [_, "DEPOSIT", _, var amount] => decimal.Parse(amount),

```

```

        [_, "WITHDRAWAL", ..., var amount] => -decimal.Parse(amount),
        [_, "INTEREST", var amount]      => decimal.Parse(amount),
        [_, "FEE", var fee]            => -decimal.Parse(fee),
        _                                => throw new
    InvalidOperationException($"Record {string.Join(", ", transaction)} is not
    in the expected format!"),
};

Console.WriteLine($"Record: {string.Join(", ", transaction)}, New
balance: {balance:C}");
}

```

앞의 예제에서는 각 요소가 행의 한 필드인 문자열 배열을 가져옵니다. 두 번째 필드의 `switch` 식 키로, 트랜잭션의 종류와 나머지 열 수를 결정합니다. 각 행은 데이터가 올바른 형식인지 확인합니다. 취소 패턴(`_`)은 트랜잭션 날짜와 함께 첫 번째 필드를 건너뜁니다. 두 번째 필드는 트랜잭션 형식과 일치합니다. 나머지 요소 일치 항목은 크기가 있는 필드로 건너뜁니다. 마지막 일치 항목은 `var` 패턴을 사용하여 양의 문자열 표현을 캡처합니다. 식은 잔액을 추가하거나 빼는 양을 계산합니다.

목록 패턴을 사용하면 데이터 요소 시퀀스의 모양과 일치시킬 수 있습니다. 요소의 위치와 일치하도록 무시 항목 및 조각 패턴을 사용합니다. 다른 패턴을 사용하여 개별 요소에 대한 특성을 일치시킬 수 있습니다.

이 문서에서는 C#의 패턴 일치를 사용하여 작성할 수 있는 여러 종류의 코드를 살펴보았습니다. 다음 문서에서는 시나리오에서 패턴을 사용하는 더 많은 예제와 사용할 수 있는 패턴의 전체 어휘를 보여 줍니다.

## 참고 항목

- 패턴 일치를 사용하여 'is' 검사 이후 캐스트 방지(스타일 규칙 IDE0020 및 IDE0038)
- 탐색: 패턴 일치를 사용하여 코드를 개선하는 클래스 동작 빌드
- 자습서: 패턴 일치를 사용하여 형식 기반 및 데이터 기반 알고리즘 빌드
- 참조: 패턴 일치

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.



### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

#### 설명서 문제 열기

#### 제품 사용자 의견 제공

# 무시 항목 - C# 기본 사항

아티클 • 2024. 02. 19.

무시 항목은 애플리케이션 코드에서 의도적으로 사용을 하지 않는 자리 표시자입니다. 무시 항목은 할당되지 않은 변수에 해당하므로 값을 가지지 않습니다. 무시 항목은 식의 결과를 무시하려고 한 사용자 의도를 사용자 코드를 읽는 다른 사용자와 컴파일러에 전달합니다. 식의 결과, 하나 이상의 튜플 식 멤버, 메서드에 대한 `out` 매개 변수 또는 패턴 일치 식의 대상을 무시해야 할 수 있습니다.

무시 항목은 코드의 의도를 명확하게 합니다. 무시 항목은 코드에서 변수를 사용하지 않는다는 것을 나타냅니다. 또한 코드의 가독성 및 유지 관리를 향상시킵니다.

변수가 무시 항목임을 지정하려면 변수에 밑줄(`_`)을 이름으로 할당합니다. 예를 들어 다음 메서드 호출은 첫 번째 및 두 번째 값이 무시 항목인 튜플을 반환합니다. `area`는 이전에 선언된 변수로, `GetCityInformation`에서 반환된 세 번째 구성 요소로 설정됩니다.

C#

```
(_, _, area) = city.GetCityInformation(cityName);
```

무시 항목을 사용하여 람다 식의 사용하지 않는 입력 매개 변수를 지정할 수 있습니다. 자세한 내용은 [람다 식 문서의 람다 식 입력 매개 변수 섹션](#)을 참조하세요.

`_`이(가) 유효한 무시 항목인 경우, 이 값을 검색하거나 할당 작업에서 사용하려고 하면 “이름 '\_' 이 현재 컨텍스트에 없습니다”라는 컴파일러 오류 CS0103이 생성됩니다. 이 오류는 `_`에 값이 할당되어 있지 않고 스토리지 위치도 할당되어 있지 않을 수 있기 때문입니다. 실제 변수인 경우에는 이전 예제에서처럼 2개 이상의 값을 무시할 수 없습니다.

## 튜플 및 개체 분해

무시 항목은 튜플을 작업할 때 애플리케이션 코드에 튜플 요소 중 일부만 사용하고 일부는 무시하는 경우에 유용합니다. 예를 들어, 다음 `QueryCityDataForYears` 메서드는 도시의 이름, 도시의 면적, 연도, 해당 연도의 도시 인구, 두 번째 연도, 해당 두 번째 연도의 도구 인구를 포함하는 튜플을 반환합니다. 이 예제는 이러한 두 연도 사이의 인구 변화를 보여 줍니다. 튜플에서 사용 가능한 데이터 중 도시 면적에는 관심이 없고 디자인 타임에 도시 이름과 두 날짜를 알고 있습니다. 따라서 튜플에 저장된 두 가지 인구 값에만 관심이 있고 나머지 값은 무시 항목으로 처리할 수 있습니다.

C#

```

var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City", 1960,
2010);

Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");

static (string, double, int, int, int, int) QueryCityDataForYears(string
name, int year1, int year2)
{
    int population1 = 0, population2 = 0;
    double area = 0;

    if (name == "New York City")
    {
        area = 468.48;
        if (year1 == 1960)
        {
            population1 = 7781984;
        }
        if (year2 == 2010)
        {
            population2 = 8175133;
        }
        return (name, area, year1, population1, year2, population2);
    }

    return ("", 0, 0, 0, 0, 0);
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149

```

무시 항목을 사용한 튜플 분해에 대한 자세한 내용은 [튜플 및 기타 형식 분해](#)를 참조하세요.

클래스, 구조체 또는 인터페이스의 `Deconstruct` 메서드로도 개체에서 특정 데이터 집합을 검색 및 분해할 수 있습니다. 분해된 값의 하위 집합만으로 작업하려는 경우 무시 항목을 사용할 수 있습니다. 다음 예제에서는 `Person` 개체를 4개의 문자열(이름, 성, 도시 및 주)로 분해하지만 성과 주는 무시합니다.

C#

```

using System;

namespace Discards
{
    public class Person
    {
        public string FirstName { get; set; }
        public string MiddleName { get; set; }
        public string LastName { get; set; }
        public string City { get; set; }
    }
}

```

```

        public string State { get; set; }

        public Person(string fname, string mname, string lname,
                      string cityName, string stateName)
    {
        FirstName = fname;
        MiddleName = mname;
        LastName = lname;
        City = cityName;
        State = stateName;
    }

    // Return the first and last name.
    public void Deconstruct(out string fname, out string lname)
    {
        fname = FirstName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string mname, out
string lname)
    {
        fname = FirstName;
        mname = MiddleName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string lname,
                           out string city, out string state)
    {
        fname = FirstName;
        lname = LastName;
        city = City;
        state = State;
    }
}

class Example
{
    public static void Main()
    {
        var p = new Person("John", "Quincy", "Adams", "Boston", "MA");

        // Deconstruct the person object.
        var (fName, _, city, _) = p;
        Console.WriteLine($"Hello {fName} of {city}!");
        // The example displays the following output:
        //      Hello John of Boston!
    }
}

```

무시 항목을 사용한 사용자 정의 형식 분해에 대한 자세한 내용은 [튜플 및 기타 형식 분해](#)를 참조하세요.

## switch를 사용한 패턴 일치

'무시 패턴'은 `switch` 식을 사용한 패턴 일치에서 사용할 수 있습니다. `null`을 포함한 모든 식은 무시 패턴과 항상 일치합니다.

다음 예제에서는 `switch` 식을 사용하여 개체가 `IFormatProvider` 구현을 제공하고 개체가 `null`인지 테스트하는지를 결정하는 `ProvidesFormatInfo` 메서드를 정의합니다. 또한 무시 패턴을 사용하여 다른 형식의 `null`이 아닌 개체도 처리합니다.

C#

```
object?[] objects = [CultureInfo.CurrentCulture,
                      CultureInfo.CurrentCulture.DateTimeFormat,
                      CultureInfo.CurrentCulture.NumberFormat,
                      new ArgumentException(), null];
foreach (var obj in objects)
    ProvidesFormatInfo(obj);

static void ProvidesFormatInfo(object? obj) =>
    Console.WriteLine(obj switch
    {
        IFormatProvider fmt => $"{fmt.GetType()} object",
        null => "A null object reference: Its use could result in a
NullReferenceException",
        _ => "Some object type without format information"
    });
// The example displays the following output:
// System.Globalization.CultureInfo object
// System.Globalization.DateTimeFormatInfo object
// System.Globalization.NumberFormatInfo object
// Some object type without format information
// A null object reference: Its use could result in a
NullReferenceException
```

## out 매개 변수를 사용한 메서드 호출

`Deconstruct` 메서드를 호출하여 사용자 정의 형식(클래스, 구조체 또는 인터페이스의 인스턴스)을 분해할 때 개별 `out` 인수의 값을 무시할 수 있습니다. 하지만 어느 메서드든 `out` 매개 변수를 사용하여 호출할 때 `out` 인수의 값을 무시할 수도 있습니다.

다음 예제에서는 `DateTime.TryParse(String, out DateTime)` 메서드를 호출하여 날짜의 문자열 표현이 현재 문화권에 유효한지 확인합니다. 이 예제에서는 날짜 문자열의 유효성 검사에만 관심이 있고 이 문자열의 구문 검사를 통한 날짜 추출에는 관심이 없으므로 메서드의 `out` 인수는 무시 항목입니다.

C#

```

string[] dateStrings = ["05/01/2018 14:57:32.8", "2018-05-01 14:57:32.8",
                      "2018-05-01T14:57:32.8375298-04:00", "5/01/2018",
                      "5/01/2018 14:57:32.80 -07:00",
                      "1 May 2018 2:57:32.8 PM", "16-05-2018 1:00:32 PM",
                      "Fri, 15 May 2018 20:10:57 GMT"];
foreach (string dateString in dateStrings)
{
    if (DateTime.TryParse(dateString, out _))
        Console.WriteLine($"'{dateString}': valid");
    else
        Console.WriteLine($"'{dateString}': invalid");
}
// The example displays output like the following:
//      '05/01/2018 14:57:32.8': valid
//      '2018-05-01 14:57:32.8': valid
//      '2018-05-01T14:57:32.8375298-04:00': valid
//      '5/01/2018': valid
//      '5/01/2018 14:57:32.80 -07:00': valid
//      '1 May 2018 2:57:32.8 PM': valid
//      '16-05-2018 1:00:32 PM': invalid
//      'Fri, 15 May 2018 20:10:57 GMT': invalid

```

## 독립 실행형 무시 항목

독립 실행형 무시 항목을 사용하여 무시할 변수를 지정할 수 있습니다. 한 가지 일반적인 용도는 인수가 null이 아니도록 할당을 사용하는 것입니다. 다음 코드에서는 무시 항목을 사용하여 할당을 적용합니다. 할당의 오른쪽은 [null 병합 연산자](#)를 사용하여 인수가 null 일 때 [System.ArgumentNullException](#)을 throw합니다. 코드에 할당 결과가 필요하지 않으므로 할당이 무시됩니다. 식에서 null 검사를 강제로 수행합니다. 무시 항목은 할당 결과가 필요하지 않거나 사용되지 않는다는 의도를 명확하게 합니다.

C#

```

public static void Method(string arg)
{
    _ = arg ?? throw new ArgumentNullException(paramName: nameof(arg),
message: "arg can't be null");

    // Do work with arg.
}

```

다음 예제에서는 독립 실행형 무시 항목을 사용하여 비동기 작업에서 반환되는 [Task](#)개체를 무시합니다. 작업을 할당하면 작업이 완료되려고 할 때 throw되는 예외가 표시되지 않습니다. [Task](#)을(를) 무시하고 해당 비동기 작업에서 생성되는 모든 오류를 무시하려고 하는 의도가 명확히 드러납니다.

C#

```
private static async Task ExecuteAsyncMethods()
{
    Console.WriteLine("About to launch a task...");
    _ = Task.Run(() =>
    {
        var iterations = 0;
        for (int ctr = 0; ctr < int.MaxValue; ctr++)
            iterations++;
        Console.WriteLine("Completed looping operation...");
        throw new InvalidOperationException();
    });
    await Task.Delay(5000);
    Console.WriteLine("Exiting after 5 second delay");
}
// The example displays output like the following:
//     About to launch a task...
//     Completed looping operation...
//     Exiting after 5 second delay
```

작업을 무시 항목에 할당하지 않으면 다음 코드는 컴파일러 경고를 생성합니다.

C#

```
private static async Task ExecuteAsyncMethods()
{
    Console.WriteLine("About to launch a task...");
    // CS4014: Because this call is not awaited, execution of the current
    method continues before the call is completed.
    // Consider applying the 'await' operator to the result of the call.
    Task.Run(() =>
    {
        var iterations = 0;
        for (int ctr = 0; ctr < int.MaxValue; ctr++)
            iterations++;
        Console.WriteLine("Completed looping operation...");
        throw new InvalidOperationException();
    });
    await Task.Delay(5000);
    Console.WriteLine("Exiting after 5 second delay");
```

## ① 참고

디버거를 사용하여 위의 두 샘플 중 하나를 실행하면 예외가 throw될 때 디버거가 프로그램을 중지합니다. 연결된 디버거가 없으면 두 경우 모두 예외가 자동으로 무시됩니다.

`_`은 유효한 식별자이기도 합니다. 지원되는 컨텍스트 외부에서 사용하면 `_`은 무시 항목이 아니라 유효한 변수로 처리됩니다. `_`이라는 식별자가 이미 범위 내에 있는 경우 `_`을 독립 실행형 무시 항목으로 사용하면 다음과 같은 결과가 발생할 수 있습니다.

- 범위 내 `_` 변수 값을 실수로 수정하여 의도한 무시 항목의 값 할당. 예시:

```
C#  
  
private static void ShowValue(int _)  
{  
    byte[] arr = [0, 0, 1, 2];  
    _ = BitConverter.ToInt32(arr, 0);  
    Console.WriteLine(_);  
}  
// The example displays the following output:  
//      33619968
```

- 형식 안전성 위반으로 인한 컴파일러 오류. 예시:

```
C#  
  
private static bool RoundTrips(int _)  
{  
    string value = _.ToString();  
    int newValue = 0;  
    _ = Int32.TryParse(value, out newValue);  
    return _ == newValue;  
}  
// The example displays the following compiler error:  
//      error CS0029: Cannot implicitly convert type 'bool' to 'int'
```

- 컴파일러 오류 CS0136, “이름이 ‘\_’인 지역 또는 매개 변수는 이 범위에서 선언될 수 없습니다. 해당 이름이 지역 또는 매개 변수를 정의하기 위해 바깥쪽 지역 범위에서 사용되었습니다.” 예를 들어:

```
C#  
  
public void DoSomething(int _)  
{  
    var _ = GetValue(); // Error: cannot declare local _ when one is  
    already in scope  
}  
// The example displays the following compiler error:  
// error CS0136:  
//      A local or parameter named '_' cannot be declared in this  
//      scope  
//      because that name is used in an enclosing local scope  
//      to define a local or parameter
```

# 참고 항목

- 불필요한 식 값 제거(스타일 규칙 IDE0058)
- 불필요한 값 할당 제거(스타일 규칙 IDE0059)
- 사용되지 않는 매개 변수 제거(스타일 규칙 IDE0060)
- 튜플 및 기타 형식 분해
- is 연산자
- switch 식

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

⌚ 설명서 문제 열기

☒ 제품 사용자 의견 제공

# 튜플 및 기타 형식 분해

아티클 • 2024. 02. 21.

튜플은 메서드 호출에서 여러 값을 검색할 수 있는 간단한 방법을 제공합니다. 하지만 튜플을 검색한 후 튜플의 개별 요소를 처리해야 합니다. 다음 예에서 볼 수 있듯이 요소별로 작업하는 것은 번거롭습니다. `QueryCityData` 메서드는 3개의 튜플을 반환하고 각 요소는 별도의 작업을 통해 변수에 할당됩니다.

C#

```
public class Example
{
    public static void Main()
    {
        var result = QueryCityData("New York City");

        var city = result.Item1;
        var pop = result.Item2;
        var size = result.Item3;

        // Do something with the data.
    }

    private static (string, int, double) QueryCityData(string name)
    {
        if (name == "New York City")
            return (name, 8175133, 468.48);

        return ("", 0, 0);
    }
}
```

개체에서 여러 필드 및 속성 값을 검색하는 작업도 똑같이 번거로울 수 있습니다. 멤버별로 변수에 필드 또는 속성 값을 할당해야 하기 때문입니다.

단일 ~~분해~~ 작업으로 튜플에서 여러 요소를 검색하거나 개체에서 여러 필드, 속성 및 계산된 값을 검색할 수 있습니다. 튜플을 분해하려면 해당 요소를 개별 변수에 할당합니다. 개체를 분해할 때는 선택한 값을 개별 변수에 할당합니다.

## 튜플

C#에서는 튜플 분해를 기본적으로 지원하므로 한 작업에서 튜플의 모든 항목을 패키지 해제할 수 있습니다. 튜플을 분해하는 일반 구문은 정의하는 구문과 유사합니다. 즉, 대입 문 왼쪽에서 각 요소가 할당되는 변수를 괄호로 묶습니다. 예를 들어, 다음 문은 4-튜플의 요소를 4개의 개별 변수에 할당합니다.

C#

```
var (name, address, city, zip) = contact.GetAddressInfo();
```

다음과 같은 세 가지 방법으로 튜플을 분해합니다.

- 괄호 안에 각 필드의 형식을 명시적으로 선언할 수 있습니다. 다음 예에서는 이 방법을 사용하여 `QueryCityData` 메서드에서 반환된 3-튜플을 분해합니다.

C#

```
public static void Main()
{
    (string city, int population, double area) = QueryCityData("New
    York City");

    // Do something with the data.
}
```

- C#에서 각 변수의 형식을 유추하도록 `var` 키워드를 사용할 수 있습니다. `var` 키워드는 괄호 밖에 놓습니다. 다음 예에서는 `QueryCityData` 메서드에서 반환된 3-튜플을 분해할 때 형식 유추를 사용합니다.

C#

```
public static void Main()
{
    var (city, population, area) = QueryCityData("New York City");

    // Do something with the data.
}
```

괄호 안에 일부 또는 모든 변수 선언에 `var` 키워드를 개별적으로 사용할 수도 있습니다.

C#

```
public static void Main()
{
    (string city, var population, var area) = QueryCityData("New York
    City");

    // Do something with the data.
}
```

이 방법은 번거로우며 권장되지 않습니다.

- 마지막으로, 튜플을 이미 선언된 변수로 분해할 수 있습니다.

C#

```
public static void Main()
{
    string city = "Raleigh";
    int population = 458880;
    double area = 144.8;

    (city, population, area) = QueryCityData("New York City");

    // Do something with the data.
}
```

- C# 10부터 분해 시 변수 선언과 할당을 혼합할 수 있습니다.

C#

```
public static void Main()
{
    string city = "Raleigh";
    int population = 458880;

    (city, population, double area) = QueryCityData("New York City");

    // Do something with the data.
}
```

튜플의 모든 필드가 동일한 형식을 가지더라도 괄호 외부에 특정 형식을 지정할 수 없습니다. 그렇게 하면 컴파일러 오류 CS8136, "'var (...)'" 형식 분해는 'var'에 대한 특정 형식을 허용하지 않습니다."가 생성됩니다.

튜플의 각 요소를 변수에 할당해야 합니다. 요소를 생략하면 컴파일러에서 오류 CS8132, "'x' 요소의 튜플을 'y' 변수로 분해할 수 없습니다."를 생성합니다.

## 무시 항목이 있는 튜플 요소

튜플을 분해할 때 일부 요소 값에만 관심이 있는 경우가 종종 있습니다. 값을 무시하도록 선택한 쓰기 전용 변수인 무시 항목에 대한 C#의 지원을 활용할 수 있습니다. 무시 항목은 할당에서 밑줄 문자("\_")로 선택됩니다. 원하는 수의 값을 모두 하나의 무시 항목 \_로 표시하여 무시할 수 있습니다.

다음 예제에서는 무시 항목과 함께 튜플을 사용하는 방법을 보여 줍니다.

`QueryCityDataForYears` 메서드는 도시 이름, 지역, 연도, 해당 연도의 도시 모집단, 두 번째 해, 두 번째 해의 도시 모집단이 포함된 6개의 튜플을 반환합니다. 이 예제는 이러한 두

연도 사이의 인구 변화를 보여 줍니다. 튜플에서 사용 가능한 데이터 중 도시 면적에는 관심이 없고 디자인 타임에 도시 이름과 두 날짜를 알고 있습니다. 따라서 튜플에 저장된 두 가지 인구 값에만 관심이 있고 나머지 값은 무시 항목으로 처리할 수 있습니다.

```
C#  
  
using System;  
  
public class ExampleDiscard  
{  
    public static void Main()  
    {  
        var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York  
City", 1960, 2010);  
  
        Console.WriteLine($"Population change, 1960 to 2010: {pop2 -  
pop1:N0}");  
    }  
  
    private static (string, double, int, int, int, int)  
QueryCityDataForYears(string name, int year1, int year2)  
    {  
        int population1 = 0, population2 = 0;  
        double area = 0;  
  
        if (name == "New York City")  
        {  
            area = 468.48;  
            if (year1 == 1960)  
            {  
                population1 = 7781984;  
            }  
            if (year2 == 2010)  
            {  
                population2 = 8175133;  
            }  
            return (name, area, year1, population1, year2, population2);  
        }  
  
        return ("", 0, 0, 0, 0, 0);  
    }  
}  
// The example displays the following output:  
//      Population change, 1960 to 2010: 393,149
```

## 사용자 정의 형식

C#은 [record](#) 및 [DictionaryEntry](#) 형식 이외의 튜플이 아닌 형식을 분해하기 위한 기본 지원을 제공하지 않습니다. 그러나 클래스, 구조체 또는 인터페이스의 만든 이는 하나 이상의 [Deconstruct](#) 메서드를 구현하여 형식의 인스턴스를 분해하도록 허용할 수 있습니다.

이 메서드는 void를 반환하며 분해할 각 값은 메서드 시그니처에서 `out` 매개 변수로 표시됩니다. 예를 들어 다음 `Person` 클래스의 `Deconstruct` 메서드는 이름, 중간 이름 및 성을 반환합니다.

C#

```
public void Deconstruct(out string fname, out string mname, out string lname)
```

그리고 다음 코드와 같은 할당을 사용하여 `p`라는 `Person` 클래스의 인스턴스를 분해할 수 있습니다.

C#

```
var (fName, mName, lName) = p;
```

다음 예제에서는 `Deconstruct` 메서드를 오버로드하여 `Person` 개체의 속성을 다양한 조합으로 반환합니다. 개별 오버로드는 다음을 반환합니다.

- 이름 및 성
- 성, 중간 이름, 성
- 이름, 성, 도시 이름 및 주 이름

C#

```
using System;

public class Person
{
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public string City { get; set; }
    public string State { get; set; }

    public Person(string fname, string mname, string lname,
                  string cityName, string stateName)
    {
        FirstName = fname;
        MiddleName = mname;
        LastName = lname;
        City = cityName;
        State = stateName;
    }

    // Return the first and last name.
    public void Deconstruct(out string fname, out string lname)
    {
```

```

        fname = FirstName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string mname, out string
lname)
    {
        fname = FirstName;
        mname = MiddleName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string lname,
                           out string city, out string state)
    {
        fname = FirstName;
        lname = LastName;
        city = City;
        state = State;
    }
}

public class ExampleClassDeconstruction
{
    public static void Main()
    {
        var p = new Person("John", "Quincy", "Adams", "Boston", "MA");

        // Deconstruct the person object.
        var (fName, lName, city, state) = p;
        Console.WriteLine($"Hello {fName} {lName} of {city}, {state}!");
    }
}
// The example displays the following output:
// Hello John Adams of Boston, MA!

```

매개 변수 수가 같은 여러 `Deconstruct` 메서드는 모호합니다. 매개 변수 수, 즉 "인자"가 다른 `Deconstruct` 메서드를 정의하도록 주의해야 합니다. 오버로드 확인 중에 동일한 수의 매개 변수를 가진 `Deconstruct` 메서드를 구분할 수 없습니다.

## 무시 항목이 포함된 사용자 정의 형식

[튜플](#)에서와 마찬가지로 무시 항목을 사용하여 `Deconstruct` 메서드에서 반환된 항목 중 선택한 항목을 무시할 수 있습니다. 각 무시 항목은 "\_"라는 변수로 정의하며 단일 분해 작업에 여러 무시 항목을 포함할 수 있습니다.

다음 예제에서는 `Person` 개체를 4개의 문자열(이름, 성, 도시 및 주)로 분해하지만 성과 주는 무시합니다.

C#

```
// Deconstruct the person object.  
var (fName, _, city, _) = p;  
Console.WriteLine($"Hello {fName} of {city}!");  
// The example displays the following output:  
//      Hello John of Boston!
```

## 사용자 정의 형식의 확장 메서드

클래스, 구조체 또는 인터페이스의 만든 이가 아니더라도 하나 이상의 **Deconstruct** 확장 메서드 구현을 통해 해당 형식의 개체를 분해하여 관심 있는 값을 반환할 수 있습니다.

다음 예제에서는 [System.Reflection.PropertyInfo](#) 클래스에 대한 두 개의 **Deconstruct** 확장 메서드를 정의합니다. 첫 번째 메서드는 속성의 형식, 정적 속성인지 인스턴스 속성인지 여부, 읽기 전용인지 여부, 인덱싱되었는지 여부 등 속성의 특성을 나타내는 값 집합을 반환합니다. 두 번째 메서드는 속성의 접근성을 나타냅니다. get 및 set 접근자의 접근성이 다를 수 있으므로 부울 값은 속성에 별도의 get 및 set 접근자가 있는지 여부와 있는 경우 접근성이 동일한지 여부를 나타냅니다. 접근자가 하나만 있거나 get 및 set 접근자 모두 동일한 접근성을 갖는 경우 **access** 변수는 속성의 접근성을 전체적으로 나타냅니다. 그러지 않으면 get 및 set 접근자의 접근성이 **getAccess** 및 **setAccess** 변수로 표시됩니다.

C#

```
using System;  
using System.Collections.Generic;  
using System.Reflection;  
  
public static class ReflectionExtensions  
{  
    public static void Deconstruct(this PropertyInfo p, out bool isStatic,  
                                  out bool isReadOnly, out bool isIndexed,  
                                  out Type propertyType)  
    {  
        var getter = p.GetMethod();  
  
        // Is the property read-only?  
        isReadOnly = !p.CanWrite;  
  
        // Is the property instance or static?  
        isStatic = getter.IsStatic;  
  
        // Is the property indexed?  
        isIndexed = p.GetIndexParameters().Length > 0;  
  
        // Get the property type.
```

```
        propertyType = p.PropertyType;
    }

    public static void Deconstruct(this PropertyInfo p, out bool
hasGetAndSet,
                                out bool sameAccess, out string access,
                                out string getAccess, out string
setAccess)
{
    hasGetAndSet = sameAccess = false;
    string getAccessTemp = null;
    string setAccessTemp = null;

    MethodInfo getter = null;
    if (p.CanRead)
        getter = p.GetMethod;

    MethodInfo setter = null;
    if (p.CanWrite)
        setter = p.SetMethod;

    if (setter != null && getter != null)
        hasGetAndSet = true;

    if (getter != null)
    {
        if (getter.IsPublic)
            getAccessTemp = "public";
        else if (getter.IsPrivate)
            getAccessTemp = "private";
        else if (getter.IsAssembly)
            getAccessTemp = "internal";
        else if (getter.IsFamily)
            getAccessTemp = "protected";
        else if (getter.IsFamilyOrAssembly)
            getAccessTemp = "protected internal";
    }

    if (setter != null)
    {
        if (setter.IsPublic)
            setAccessTemp = "public";
        else if (setter.IsPrivate)
            setAccessTemp = "private";
        else if (setter.IsAssembly)
            setAccessTemp = "internal";
        else if (setter.IsFamily)
            setAccessTemp = "protected";
        else if (setter.IsFamilyOrAssembly)
            setAccessTemp = "protected internal";
    }

    // Are the accessibility of the getter and setter the same?
    if (setAccessTemp == getAccessTemp)
    {
```

```

        sameAccess = true;
        access = getAccessTemp;
        getAccess = setAccess = String.Empty;
    }
    else
    {
        access = null;
        getAccess = getAccessTemp;
        setAccess = setAccessTemp;
    }
}

public class ExampleExtension
{
    public static void Main()
    {
        Type dateType = typeof(DateTime);
        PropertyInfo prop = dateType.GetProperty("Now");
        var (isStatic, isRO, isIndexed, propType) = prop;
        Console.WriteLine($"\\nThe {dateType.FullName}.{prop.Name}
property:");
        Console.WriteLine($"    PropertyType: {propType.Name}");
        Console.WriteLine($"    Static:      {isStatic}");
        Console.WriteLine($"    Read-only:   {isRO}");
        Console.WriteLine($"    Indexed:     {isIndexed}");

        Type listType = typeof(List<>);
        prop = listType.GetProperty("Item",
                                    BindingFlags.Public |
        BindingFlags.NonPublic | BindingFlags.Instance | BindingFlags.Static);
        var (hasGetAndSet, sameAccess, accessibility, getAccessibility,
setAccessibility) = prop;
        Console.Write($"\\nAccessibility of the {listType.FullName}.
{prop.Name} property: ");

        if (!hasGetAndSet | sameAccess)
        {
            Console.WriteLine(accessibility);
        }
        else
        {
            Console.WriteLine($"\\n    The get accessor: {getAccessibility}");
            Console.WriteLine($"    The set accessor: {setAccessibility}");
        }
    }
}

// The example displays the following output:
//     The System.DateTime.Now property:
//         PropertyType: DateTime
//         Static:      True
//         Read-only:   True
//         Indexed:     False
//

```

```
// Accessibility of the System.Collections.Generic.List`1.Item  
property: public
```

## 시스템 형식의 확장 메서드

일부 시스템 형식은 편의를 위해 `Deconstruct` 메서드를 제공합니다. 예를 들어, `System.Collections.Generic.KeyValuePair< TKey, TValue >` 형식은 이 기능을 제공합니다. `System.Collections.Generic.Dictionary< TKey, TValue >`를 반복할 때 각 요소는 `KeyValuePair< TKey, TValue >`이며 분해될 수 있습니다. 다음 예제를 참조하세요.

C#

```
Dictionary<string, int> snapshotCommitMap =  
    new(StringComparer.OrdinalIgnoreCase)  
{  
    ["https://github.com/dotnet/docs"] = 16_465,  
    ["https://github.com/dotnet/runtime"] = 114_223,  
    ["https://github.com/dotnet/installer"] = 22_436,  
    ["https://github.com/dotnet/roslyn"] = 79_484,  
    ["https://github.com/dotnet/aspnetcore"] = 48_386  
};  
  
foreach (var (repo, commitCount) in snapshotCommitMap)  
{  
    Console.WriteLine(  
        $"The {repo} repository had {commitCount:N0} commits as of November  
        10th, 2021.");  
}
```

`Deconstruct` 메서드가 없는 시스템 형식에 추가할 수 있습니다. 다음 확장 메서드를 고려합니다.

C#

```
public static class NullableExtensions  
{  
    public static void Deconstruct<T>(  
        this T? nullable,  
        out bool hasValue,  
        out T value) where T : struct  
    {  
        hasValue = nullable.HasValue;  
        value = nullable.GetValueOrDefault();  
    }  
}
```

이 확장 메서드를 사용하면 모든 `Nullable<T>` 형식을 `(bool hasValue, T value)`의 튜플로 분해할 수 있습니다. 다음 예에서는 이 확장 메서드를 사용하는 코드를 보여 줍니다.

C#

```
DateTime? questionableDateTime = default;
var (hasValue, value) = questionableDateTime;
Console.WriteLine(
    $"{{ HasValue = {hasValue}, Value = {value} }}");

questionableDateTime = DateTime.Now;
(hasValue, value) = questionableDateTime;
Console.WriteLine(
    $"{{ HasValue = {hasValue}, Value = {value} }}");

// Example outputs:
// { HasValue = False, Value = 1/1/0001 12:00:00 AM }
// { HasValue = True, Value = 11/10/2021 6:11:45 PM }
```

## record 형식

둘 이상의 위치 매개 변수를 사용하여 `record` 형식을 선언하는 경우 컴파일러는 `record` 선언의 각 위치 매개 변수에 대해 `out` 매개 변수를 사용하여 `Deconstruct` 메서드를 만듭니다. 자세한 내용은 [속성 정의의 위치 구문](#) 및 [파생 레코드의 분해자 동작](#)을 참조하세요.

## 참고 항목

- 변수 선언 분해(스타일 규칙 IDE0042)
- 무시 항목
- 튜플 형식

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

#### 설명서 문제 열기

#### 제품 사용자 의견 제공

# 예외 및 예외 처리

아티클 • 2024. 02. 21.

C# 언어의 예외 처리 기능은 프로그램이 실행 중일 때 발생하는 예기치 않은 문제나 예외 상황을 처리하는 데 도움이 됩니다. 예외 처리는 `try`, `catch` 및 `finally` 키워드를 사용하여 실패했을 수 있는 작업을 시도하고, 실패를 처리하는 것이 적절하다고 판단될 때 처리하고, 리소스를 정리합니다. 예외는 CLR(공용 언어 런타임), .NET, 타사 라이브러리 또는 애플리케이션 코드에서 생성될 수 있습니다. 예외는 `throw` 키워드를 사용하여 생성됩니다.

대부분의 경우 코드에서 직접 호출한 메서드가 아니라 호출 스택에서 추가로 작동 중단된 다른 메서드에 의해 예외가 `throw`될 수 있습니다. 예외가 `throw`되는 경우 CLR은 스택을 해제하고 특정 예외 형식에 대해 `catch` 블록이 있는 메서드를 찾으며 해당하는 첫 번째 `catch` 블록을 실행합니다. 호출 스택에서 적절한 `catch` 블록을 찾지 못하면 프로세스를 종료하고 사용자에게 메시지를 표시합니다.

이 예제에서 메서드는 0으로 나누기를 테스트하고 오류를 `catch`합니다. 예외 처리를 사용하지 않을 경우 이 프로그램은 `DivideByZeroException`이(가) 처리되지 않았습니다. 오류를 나타내며 종료됩니다.

C#

```
public class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new DivideByZeroException();
        return x / y;
    }

    public static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result;

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

```
    }  
}
```

## 예외 개요

예외는 다음과 같은 속성을 갖습니다.

- 모든 예외는 궁극적으로 `System.Exception`에서 파생되는 형식입니다.
- 예외를 `throw`할 수 있는 문 주위에 `try` 블록을 사용합니다.
- `try` 블록에서 예외가 발생하면 제어 흐름이 호출 스택에 있는 첫 번째 관련 예외 처리기로 이동됩니다. C#에서 `catch` 키워드는 예외 처리기를 정의하는 데 사용됩니다.
- 지정된 예외에 대한 예외 처리기가 없으면 프로그램은 오류 메시지를 나타내며 실행을 중지합니다.
- 예외를 처리하고 애플리케이션을 알려진 상태로 둘 수 없으면 예외를 `catch`하지 마세요. `System.Exception`을 `catch`하는 경우 `catch` 블록 끝에서 `throw` 키워드를 사용하여 다시 `throw`하세요.
- `catch` 블록이 예외 변수를 정의하는 경우 이 변수를 사용하여 발생한 예외 형식에 대한 추가 정보를 얻을 수 있습니다.
- `throw` 키워드를 사용하여 프로그램에서 명시적으로 예외를 생성할 수 있습니다.
- 예외 개체는 호출 스택의 상태 및 오류에 대한 텍스트 설명 같은 오류에 대한 자세한 정보를 포함합니다.
- `finally` 블록의 코드는 예외가 `throw`되는지와 상관없이 실행됩니다. `finally` 블록을 사용하여 `try` 블록에서 열려 있는 스트림이나 파일을 닫는 것처럼 리소스를 해제합니다.
- .NET의 관리되는 예외는 Win32 구조적 예외 처리 메커니즘을 토대로 구현됩니다. 자세한 내용은 [구조적 예외 처리\(C/C++\)](#) 및 [Win32 구조적 예외 처리에 대한 집중 과정](#)을 참조하세요.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 예외를 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

## 참고 항목

- [System.Exception](#)
- [예외 처리문](#)
- [예외](#)

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 예외 사용

아티클 • 2024. 02. 23.

C#에서는 런타임 시 프로그램의 오류가 예외라는 메커니즘을 사용하여 프로그램 전체에 전파됩니다. 오류가 발생하는 코드에서 예외를 throw하고, 오류를 수정할 수 있는 코드에서 예외를 catch합니다. .NET 런타임이나 프로그램의 코드에서 예외를 throw할 수 있습니다. 예외가 throw되면 예외에 대한 `catch` 문이 발견될 때까지 호출 스택이 전파됩니다. Catch되지 않은 예외는 대화 상자를 표시하는 시스템에서 제공하는 제네릭 예외 처리기에 의해 처리됩니다.

예외는 `Exception`에서 파생된 클래스로 표현됩니다. 이 클래스는 예외의 형식을 식별하며 예외에 대한 세부 정보가 들어 있는 속성을 포함합니다. 예외를 throw하는 것에는 예외에서 파생된 클래스의 인스턴스를 만드는 것, 선택적으로 예외의 속성을 구성하는 것, 그리고 `throw` 키워드를 사용하여 개체를 throw하는 것이 포함됩니다. 예시:

C#

```
class CustomException : Exception
{
    public CustomException(string message)
    {
    }
}
private static void TestThrow()
{
    throw new CustomException("Custom exception in TestThrow()");
}
```

예외가 throw되면 런타임은 현재 문을 확인하여 `try` 블록 내에 있는지 알아봅니다. 있는 경우, `try` 블록과 연결된 `catch` 블록을 확인하여 예외를 catch할 수 있는지 알아봅니다. `Catch` 블록은 일반적으로 예외 형식을 지정합니다. `catch` 블록의 형식이 예외와 동일한 형식이거나 예외의 기본 클래스인 경우 `catch` 블록이 메서드를 처리할 수 있습니다. 예시:

C#

```
try
{
    TestThrow();
}
catch (CustomException ex)
{
    System.Console.WriteLine(ex.ToString());
}
```

예외를 throw하는 문이 try 블록 내에 없거나 이를 감싸는 try 블록에 일치하는 catch 블록이 없는 경우 런타임은 호출 메서드에 try 문과 catch 블록이 있는지 확인합니다. 런타임은 호출 스택까지 계속 확인하여 호환되는 catch 블록을 검색합니다. catch 블록을 찾아서 실행한 후에는 해당 catch 블록 다음의 문으로 제어가 전달됩니다.

try 문은 catch 블록을 둘 이상 포함할 수 있습니다. 예외를 처리할 수 있는 첫 번째 catch 문이 실행됩니다. 그 뒤의 catch 문은 호환되더라도 무시됩니다. 가장 구체적인(또는 가장 많이 발생된) 예외부터 가장 덜 구체적인 예외 순으로 catch 블록 순서를 지정합니다. 예시:

C#

```
using System;
using System.IO;

namespace Exceptions
{
    public class CatchOrder
    {
        public static void Main()
        {
            try
            {
                using (var sw = new StreamWriter("./test.txt"))
                {
                    sw.WriteLine("Hello");
                }
            }
            // Put the more specific exceptions first.
            catch (DirectoryNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            catch (FileNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            // Put the least specific exception last.
            catch (IOException ex)
            {
                Console.WriteLine(ex);
            }
            Console.WriteLine("Done");
        }
    }
}
```

catch 블록이 실행되기 전에 런타임은 finally 블록을 확인합니다. 프로그래머는 Finally 블록을 사용해 중단된 try 블록으로부터 남겨질 수 있는 모호한 상태를 정리하

거나, 런타임 시 가비지 수집기를 기다리지 않은 채 외부 리소스(예: 그래픽 핸들, 데이터 베이스 연결 또는 파일 스트림)를 해제하여 개체를 마무리할 수 있습니다. 예시:

C#

```
static void TestFinally()
{
    FileStream? file = null;
    //Change the path to something that works on your machine.
    FileInfo fileInfo = new System.IO.FileInfo("./file.txt");

    try
    {
        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    finally
    {
        // Closing the file allows you to reopen it immediately - otherwise
        IOException is thrown.
        file?.Close();
    }

    try
    {
        file = fileInfo.OpenWrite();
        Console.WriteLine("OpenWrite() succeeded");
    }
    catch (IOException)
    {
        Console.WriteLine("OpenWrite() failed");
    }
}
```

`WriteByte()`에서 예외를 throw한 경우, `file.Close()`가 호출되지 않으면 파일을 다시 열려고 시도하는 두 번째 `try` 블록이 실패하고 파일이 잠긴 상태로 유지됩니다. `finally` 블록은 예외가 throw되도 실행되므로, 이전 예제의 `finally` 블록을 통해 파일을 정확히 닫고 오류를 방지할 수 있습니다.

예외가 throw된 후 호출 스택에서 호환되는 `catch` 블록을 찾지 못하면 다음 세 가지 중 하나가 발생합니다.

- 예외가 종료자 내부에 있으면 종료자가 중단되고 기본 종료자(있는 경우)가 호출됩니다.
- 호출 스택에 정적 생성자 또는 정적 필드 이니셜라이저가 포함된 경우 새 예외의 `InnerException` 속성에 할당된 원래 예외와 함께 `TypeInitializationException`이 throw됩니다.
- 스레드의 시작에 도달하면 스레드가 종료됩니다.

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 예외 처리(C# 프로그래밍 가이드)

아티클 • 2024. 02. 21.

`try` 블록은 C# 프로그래머가 예외의 영향을 받을 수 있는 코드를 분할하는 데 사용됩니다. 연결된 `catch` 블록은 결과 예외를 처리하는 데 사용됩니다. `finally` 블록에는 `try` 블록에서 할당되는 리소스 해제와 같이 `try` 블록에서 예외가 throw되는지 여부와 관계없이 실행되는 코드가 포함됩니다. `try` 블록에는 하나 이상의 연결된 `catch` 블록, `finally` 블록 또는 둘 다가 필요합니다.

다음 예제에서는 `try-catch` 문, `try-finally` 문 및 `try-catch-finally` 문을 보여 줍니다.

C#

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
    // Only catch exceptions that you know how to handle.
    // Never catch base class System.Exception without
    // rethrowing it at the end of the catch block.
}
```

C#

```
try
{
    // Code to try goes here.
}
finally
{
    // Code to execute after the try block goes here.
}
```

C#

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
}
finally
```

```
{  
    // Code to execute after the try (and possibly catch) blocks  
    // goes here.  
}
```

`try` 블록에 `catch` 또는 `finally` 블록이 없으면 컴파일러 오류가 발생합니다.

## catch 블록

`catch` 블록에서는 `catch`할 예외의 형식을 지정할 수 있습니다. 형식 사양을 예외 필터라고 합니다. 예외 형식은 `Exception`에서 파생되어야 합니다. 일반적으로 `Exception` 블록에서 `throw`될 수 있는 모든 예외를 처리하는 방법을 알고 있거나 `try` 블록의 끝에 `throw` 문을 포함한 경우가 아니라면 `catch`를 예외 필터로 지정하지 마세요.

다른 예외 클래스를 사용하는 여러 `catch` 블록을 함께 연결할 수 있습니다. `catch` 블록은 코드의 위에서 아래로 계산되지만 `throw`되는 각 예외에 대해 하나의 `catch` 블록만 실행됩니다. `throw`된 예외의 정확한 형식이나 기본 클래스를 지정하는 첫 번째 `catch` 블록이 실행됩니다. `catch` 블록에서 일치하는 예외 클래스를 지정하지 않으면 `catch` 블록이 문에 있는 경우 어느 형식도 없는 블록이 선택됩니다. 먼저 가장 구체적인(즉, 최다 파생) 예외 클래스를 사용하여 `catch` 블록을 배치해야 합니다.

다음 조건을 충족하는 경우 예외를 `catch`합니다.

- 예외가 `throw`되는 이유를 충분히 이해하고 있으면 `FileNotFoundException` 개체를 `catch`할 때 새 파일 이름을 입력하라는 메시지를 사용자에게 표시하는 경우처럼 구체적인 복구를 구현할 수 있습니다.
- 보다 구체적인 새 예외를 생성하고 `throw`할 수 있습니다.

C#

```
int GetInt(int[] array, int index)  
{  
    try  
    {  
        return array[index];  
    }  
    catch (IndexOutOfRangeException e)  
    {  
        throw new ArgumentException(  
            "Parameter index is out of range.", e);  
    }  
}
```

- 추가 처리를 위해 예외를 전달하기 전에 예외를 부분적으로 처리하려고 합니다. 다음 예제에서 `catch` 블록은 예외를 다시 `throw`하기 전에 오류 로그에 항목을 추가하

는 데 사용됩니다.

```
C#  
  
try  
{  
    // Try to access a resource.  
}  
catch (UnauthorizedAccessException e)  
{  
    // Call a custom error logging procedure.  
    LogError(e);  
    // Re-throw the error.  
    throw;  
}
```

'예외 필터'를 지정하여 catch 절에 부울 식을 추가할 수도 있습니다. 예외 필터는 해당 조건이 true인 경우에만 특정 catch 절이 일치함을 나타냅니다. 다음 예에서는 두 catch 절 모두 동일한 예외 클래스를 사용하지만 다른 오류 메시지를 만들기 위해 추가 조건을 확인합니다.

```
C#  
  
int GetInt(int[] array, int index)  
{  
    try  
{  
        return array[index];  
    }  
    catch (IndexOutOfRangeException e) when (index < 0)  
    {  
        throw new ArgumentException(  
            "Parameter index cannot be negative.", e);  
    }  
    catch (IndexOutOfRangeException e)  
    {  
        throw new ArgumentException(  
            "Parameter index cannot be greater than the array size.", e);  
    }  
}
```

항상 `false`를 반환하는 예외 필터를 사용하여 모든 예외를 검사하지만 처리하지는 않을 수 있습니다. 일반적인 용도는 예외를 기록하는 것입니다.

```
C#  
  
public class ExceptionFilter  
{  
    public static void Main()  
}
```

```

{
    try
    {
        string? s = null;
        Console.WriteLine(s.Length);
    }
    catch (Exception e) when (LogException(e))
    {
    }
    Console.WriteLine("Exception must have been handled");
}

private static bool LogException(Exception e)
{
    Console.WriteLine($"\\tIn the log routine. Caught {e.GetType()}");
    Console.WriteLine($"\\tMessage: {e.Message}");
    return false;
}
}

```

`LogException` 메서드는 항상 `false`를 반환하므로 이 예외 필터를 사용하는 `catch` 절 중 어느 것도 일치하지 않습니다. `catch` 절은 `System.Exception`을 사용하는 범용일 수 있으며 이후 절이 더 구체적인 예외 클래스를 처리할 수 있습니다.

## Finally 블록

`finally` 블록에서는 `try` 블록에서 수행된 작업을 정리할 수 있습니다. `finally` 블록이 있는 경우 `try` 블록 및 일치하는 모든 `catch` 블록 다음에 마지막으로 실행됩니다.

`finally` 블록은 예외가 `throw`되었는지 또는 예외 형식과 일치하는 `catch` 블록이 있는지와 관계없이 항상 실행됩니다.

`finally` 블록은 런타임의 가비지 수집기가 개체를 종료할 때까지 기다리지 않고 파일 스트림, 데이터베이스 연결, 그래픽 핸들 등의 리소스를 해제하는 데 사용됩니다.

다음 예제에서는 `finally` 블록을 사용하여 `try` 블록에서 연 파일을 닫습니다. 파일을 닫기 전에 파일 핸들의 상태를 확인해야 합니다. `try` 블록에서 파일을 열 수 없는 경우 파일 핸들은 값이 `null`이며 `finally` 블록은 파일을 닫지 않습니다. 대신 `try` 블록에서 파일을 연 경우 `finally` 블록에서 열려 있는 파일을 닫습니다.

C#

```

FileStream? file = null;
FileInfo fileinfo = new System.IO.FileInfo("./file.txt");
try
{
    file = fileinfo.OpenWrite();
    file.WriteByte(0xF);
}
finally
{
    if (file != null)
        file.Close();
}

```

```
    }
    finally
    {
        // Check for null because OpenWrite might have failed.
        file?.Close();
    }
}
```

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 예외](#) 및 [try 문](#)을 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

## 참고 항목

- [C# 참조](#)
- [예외 처리문](#)
- [using 문](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# 예외 만들기 및 throw

아티클 • 2023. 12. 21.

예외는 프로그램을 실행하는 동안 오류가 발생했음을 나타내는 데 사용됩니다. 오류를 설명하는 예외 객체가 생성되고 문 또는 식과 `throw` 함께 `throw`됩니다. 그런 다음 런타임에 가장 호환성이 높은 예외 처리기를 검색합니다.

프로그래머는 다음 조건 중 하나 이상에 해당할 경우 예외를 `throw`해야 합니다.

- 메서드가 정의된 기능을 완료할 수 없는 경우. 예를 들어 메서드에 대한 매개 변수에 잘못된 값이 포함된 경우입니다.

C#

```
static void CopyObject(SampleClass original)
{
    _ = original ?? throw new ArgumentException("Parameter cannot be
null", nameof(original));
}
```

- 개체 상태에 따라 개체에 대한 부적절한 호출이 이루어진 경우. 한 가지 예는 읽기 전용 파일에 쓰려고 시도하는 경우입니다. 개체 상태가 작업을 허용하지 않을 경우 이 클래스의 파생에 따라 `InvalidOperationException`의 인스턴스 또는 개체를 `throw`합니다. 다음 코드는 `InvalidOperationException` 개체를 `throw`하는 메서드의 예제입니다.

C#

```
public class ProgramLog
{
    FileStream logFile = null!;
    public void OpenLog(FileInfo fileName, FileMode mode) { }

    public void WriteLog()
    {
        if (!logFile.CanWrite)
        {
            throw new InvalidOperationException("Logfile cannot be
read-only");
        }
        // Else write data to the log and return.
    }
}
```

- 메서드에 대한 인수가 예외를 일으키는 경우. 이 경우 원래 예외가 `catch`되고 `ArgumentException` 인스턴스가 만들어져야 합니다. 원래 예외를

[ArgumentException](#)의 생성자에 [InnerException](#) 매개 변수로 전달해야 합니다.

C#

```
static int GetValueFromArray(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index is out of range.", e);
    }
}
```

### ① 참고

앞의 예제에서는 속성을 사용하는 방법을 보여 있습니다 [InnerException](#) . 의 도적으로 간소화되었습니다. 실제로 인덱스는 사용하기 전에 범위 내의 인덱스 임을 검사 합니다. 매개 변수의 멤버가 멤버를 호출하기 전에 예상할 수 없는 예외를 throw할 때 예외를 래핑하는 이 기술을 사용할 수 있습니다.

예외에 이름이 [StackTrace](#)인 속성이 포함되어 있습니다. 이 문자열에는 현재 콜 스택에 대한 메서드의 이름과 각 메서드에 대해 예외가 throw된 파일 이름 및 줄 번호가 포함됩니다. [StackTrace](#) 객체는 [throw](#) 문의 지점에서 CLR(공용 언어 런타임)에 의해 자동으로 만들어지므로 해당 예외는 스택 추적이 시작되는 지점에서 throw되어야 합니다.

모든 예외에 이름이 [Message](#)인 속성이 포함되어 있습니다. 예외의 이유를 설명하려면 이 문자열을 설정해야 합니다. 보안이 중요한 정보는 메시지 텍스트에 넣으면 안 됩니다. [Message](#) 외에 [ArgumentException](#)에는 예외를 throw한 인수의 이름으로 설정해야 하는 [ParamName](#) 속성이 포함되어 있습니다. 속성 setter에서는 [ParamName](#)을 [value](#)로 설정해야 합니다.

public 및 protected 메서드는 의도한 함수를 완료할 수 없을 때마다 예외를 throw합니다. throw된 예외 클래스는 오류 조건에 맞을 수 있는 가장 구체적인 예외입니다. 이러한 예외는 클래스 기능의 일부로 문서화해야 하고 파생 클래스 또는 원래 클래스의 업데이트는 이전 버전과의 호환성을 위해 같은 동작을 유지해야 합니다.

## 예외를 throw할 때 피해야 하는 작업

다음 목록은 예외를 throw할 때 피해야 할 사례를 나타냅니다.

- 프로그램의 흐름을 일반 실행의 일부로 변경하는 데는 예외를 사용하지 마세요. 오류 조건을 보고하고 처리하는 데 예외를 사용합니다.
- 예외는 throw하는 대신 반환 값 또는 매개 변수로 반환하면 안 됩니다.
- 고유한 소스 코드에서 의도적으로 `System.Exception`, `System.SystemException`, `System.NullReferenceException` 또는 `System.IndexOutOfRangeException`을 throw하지 마세요.
- 릴리스 모드가 아닌 디버그 모드에서 throw될 수 있는 예외를 만들지 마세요. 개발 단계에서 런타임 오류를 식별하려면 대신 디버그 어설션을 사용하세요.

## 작업 반환 메서드의 예외

한정자를 사용하여 `async` 선언된 메서드는 예외와 관련하여 몇 가지 특별한 고려 사항이 있습니다. 메서드에서 `async` throw된 예외는 반환된 작업에 저장되며 작업이 대기될 때 까지 나타나지 않습니다. 저장된 예외에 대한 자세한 내용은 비동기 예외를 참조 [하세요](#).

메서드의 비동기 부분을 입력하기 전에 인수의 유효성을 검사하고 `ArgumentNullException` 해당하는 예외(예: `ArgumentException` 및)를 throw하는 것이 좋습니다. 즉, 이러한 유효성 검사 예외는 작업이 시작되기 전에 동기적으로 나타나야 합니다. 다음 코드 조각은 예외가 throw `ArgumentException` 되면 예외가 동기적으로 나타나는 반면 `InvalidOperationException` 반환된 작업에 저장되는 예제를 보여 줍니다.

C#

```
// Non-async, task-returning method.
// Within this method (but outside of the local function),
// any thrown exceptions emerge synchronously.
public static Task<Toast> ToastBreadAsync(int slices, int toastTime)
{
    if (slices is < 1 or > 4)
    {
        throw new ArgumentException(
            "You must specify between 1 and 4 slices of bread.",
            nameof(slices));
    }

    if (toastTime < 1)
    {
        throw new ArgumentException(
            "Toast time is too short.", nameof(toastTime));
    }

    return ToastBreadAsyncCore(slices, toastTime);

    // Local async function.
    // Within this function, any thrown exceptions are stored in the task.
    static async Task<Toast> ToastBreadAsyncCore(int slices, int time)
    {
```

```

        for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("Putting a slice of bread in the toaster");
    }
    // Start toasting.
    await Task.Delay(time);

    if (time > 2_000)
    {
        throw new InvalidOperationException("The toaster is on fire!");
    }

    Console.WriteLine("Toast is ready!");

    return new Toast();
}
}

```

## 예외 클래스 정의

프로그램에서는 [System](#) 네임스페이스의 미리 정의된 예외 클래스를 `throw`하거나(이전에 언급한 위치 제외) [Exception](#)에서 파생시켜 자체 예외 클래스를 만들 수 있습니다. 파생 클래스는 매개 변수가 없는 생성자 1개, 메시지 속성을 설정하는 생성자, 속성과 [InnerException](#) 속성을 둘 다 [Message](#) 설정하는 생성자 등 세 개 이상의 생성자를 정의해야 합니다. 예시:

C#

```

[Serializable]
public class InvalidDepartmentException : Exception
{
    public InvalidDepartmentException() : base() { }
    public InvalidDepartmentException(string message) : base(message) { }
    public InvalidDepartmentException(string message, Exception inner) :
    base(message, inner) { }
}

```

새로운 속성이 제공하는 데이터가 예외 확인에 유용할 경우 해당 속성을 예외 클래스에 추가합니다. 새 속성이 파생 예외 클래스에 추가되면 추가된 정보를 반환하기 위해 `ToString()`을 재정의해야 합니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 예외](#) 및 `throw` 문을 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

# 참고 항목

- 예외 계층

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

ଓ 설명서 문제 열기

¤ 제품 사용자 의견 제공

# 컴파일러 생성 예외

아티클 • 2024. 02. 14.

기본 작업이 실패하면 .NET 런타임에 의해 자동으로 일부 예외가 throw됩니다. 이러한 예외와 관련 오류 조건이 다음 표에 나와 있습니다.

[+] 테이블 확장

예외	설명
ArithmeticsException	<code>DivideByZeroException</code> , <code>OverflowException</code> 등의 산술 연산 중에 발생하는 예외에 대한 기본 클래스입니다.
ArrayTypeMismatchException	제공된 요소의 실제 형식이 배열의 실제 형식과 호환되지 않아 배열이 요소를 저장할 수 없는 경우 throw됩니다.
DivideByZeroException	정수 값을 0으로 나누려고 시도할 경우 throw됩니다.
IndexOutOfRangeException	인덱스가 0보다 작거나 배열 경계를 벗어날 때 배열을 인덱싱하고 시도할 경우 throw됩니다.
InvalidCastException	기본 형식에서 인터페이스 또는 파생 형식으로의 명시적 변환이 런타임에 실패할 경우 throw됩니다.
NullReferenceException	값이 <code>null</code> 인 개체를 참조하려고 시도할 경우 throw됩니다.
OutOfMemoryException	<code>new</code> 연산자를 사용한 메모리 할당 시도가 실패할 경우 throw됩니다. 이 예외는 공용 언어 런타임에 사용할 수 있는 메모리가 모두 사용되었음을 나타냅니다.
OverflowException	<code>checked</code> 컨텍스트의 산술 연산이 오버플로될 경우 throw됩니다.
StackOverflowException	보류 중인 메서드 호출이 너무 많아 실행 스택이 모두 사용될 경우 throw됩니다. 대개 매우 깊은 재귀나 무한 재귀를 나타냅니다.
TypeInitializationException	정적 생성자가 예외를 throw하고 이 예외를 catch할 수 있는 호환되는 <code>catch</code> 절이 없는 경우 throw됩니다.

## 참고 항목

- [예외 처리문](#)

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# C# 식별자 명명 규칙 및 규칙

아티클 • 2024. 02. 21.

식별자는 형식(클래스, 인터페이스, 구조체, 대리자 또는 열거형), 멤버, 변수 또는 네임스페이스에 할당하는 이름입니다.

## 이름 지정 규칙

유효한 식별자는 다음 규칙을 따라야 합니다. C# 컴파일러는 다음 규칙을 따르지 않는 식별자에 대해 오류를 생성합니다.

- 식별자는 문자나 밑줄(\_)로 시작해야 합니다.
- 식별자에는 유니코드 문자, 10진수 문자, 유니코드 연결 문자, 유니코드 조합 문자 또는 유니코드 형식 문자가 포함될 수 있습니다. 유니코드 범주에 대한 자세한 내용은 [유니코드 범주 데이터베이스](#)를 참조하세요.

식별자의 @ 접두사를 사용하여 C# 키워드와 일치하는 식별자를 선언할 수 있습니다. @은 식별자 이름의 일부가 아닙니다. 예를 들어 @if는 if라는 식별자를 선언합니다. 이러한 [verbatim](#) 식별자는 주로 다른 언어로 선언된 식별자와의 상호 운용성을 위한 것입니다.

유효한 식별자에 대한 전체 정의를 보려면 [C# 언어 사양의 식별자 문서](#)를 참조하세요.

### ① 중요

[C# 언어 사양](#)에서는 문자(Lu, Li, Lt, Lm, Lo 또는 Ni), 숫자(Nd), 연결(Pc), 결합(Mn 또는 Mc) 및 서식(Cf) 범주만 허용합니다. 외부의 모든 항목은 \_을 사용하여 자동으로 바뀝니다. 이는 특정 유니코드 문자에 영향을 미칠 수 있습니다.

## 명명 규칙

규칙 외에도 식별자 이름에 대한 규칙이 .NET API 전체에서 사용됩니다. 이러한 규칙은 이름에 대한 일관성을 제공하지만 컴파일러는 이를 강제하지 않습니다. 프로젝트에서 다양한 규칙을 자유롭게 사용할 수 있습니다.

규칙에 따라 C# 프로그램은 형식 이름, 네임스페이스 및 모든 공용 멤버에 PascalCase를 사용합니다. 또한 `dotnet/docs` 팀은 [.NET 런타임 팀의 코딩 스타일](#)에서 채택한 다음 규칙을 사용합니다.

- 인터페이스 이름은 대문자 I로 시작합니다.

- 특성 유형은 `Attribute` 단어로 끝납니다.
- 열거형 형식은 플래그가 아닌 경우에는 단수 명사를 사용하고 플래그인 경우에는 복수 명사를 사용합니다.
- 식별자에는 두 개의 연속된 밑줄(`_`) 문자가 포함될 수 없습니다. 이러한 이름은 컴파일러 생성 식별자용으로 예약되어 있습니다.
- 변수, 메서드, 클래스에 의미 있고 설명이 포함된 이름을 사용합니다.
- 간결함보다 명확성이 더 중요합니다.
- 클래스 이름과 메서드 이름에는 PascalCase를 사용합니다.
- 메서드 인수, 지역 변수, 프라이빗 필드에는 camelCase를 사용합니다.
- 필드와 지역 상수 모두 상수 이름에 PascalCase를 사용합니다.
- 프라이빗 인스턴스 필드는 밑줄(`_`)로 시작합니다.
- 정적 필드는 `s`로 시작합니다. 이 규칙은 기본 Visual Studio 동작도 아니고 [프레임워크 디자인 지침](#)의 일부도 아니지만 [editorconfig](#)에서 구성할 수 있습니다.
- 널리 알려지고 인정되는 약어를 제외하고 이름에 약어나 머리글자어를 사용하지 마세요.
- 역방향 도메인 이름 표기법에 따라 의미 있고 설명이 포함된 네임스페이스를 사용합니다.
- 어셈블리의 기본 목적을 나타내는 어셈블리 이름을 선택합니다.
- 간단한 루프 카운터를 제외하고 단일 문자 이름을 사용하지 마세요. 또한 C# 구문(construct)의 구문(syntax)을 설명하는 구문(syntax) 예에서는 [C# 언어 사양](#)에 사용된 규칙과 일치하는 다음과 같은 단일 문자 이름을 사용하는 경우가 많습니다. 구문 예는 규칙의 예외입니다.
  - 구조체에는 `s`를 사용하고 클래스에는 `c`를 사용합니다.
  - 메서드에는 `m`을 사용합니다.
  - 변수에는 `v`, 매개 변수에는 `p`을 사용합니다.
  - `ref` 매개 변수에는 `r`을 사용합니다.

### 💡 팁

[코드 스타일 명명 규칙](#)을 사용하여 대문자, 접두사, 접미사 및 단어 구분 기호와 관련된 명명 규칙을 적용할 수 있습니다.

다음 예에서 `public` 으로 표시된 요소와 관련된 지침은 `protected` 및 `protected internal` 요소로 작업할 때도 적용 가능하며, 이 요소는 모두 외부 호출자에게 표시됩니다.

## 파스칼식 대/소문자

`class`, `interface`, `struct` 또는 `delegate` 형식의 이름을 지정할 때 파스칼식 대/소문자 ("PascalCasing")를 사용합니다.

C#

```
public class DataService
{
}
```

C#

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

C#

```
public struct ValueCoordinate
{
}
```

C#

```
public delegate void DelegateType(string message);
```

`interface` 이름을 지정할 때는 이름 앞에 `I` 접두사를 적용하고 파스칼식 대/소문자를 사용합니다. 이 접두사는 해당 항목이 `interface`임을 소비자에게 분명히 나타냅니다.

C#

```
public interface IWorkerQueue
{
}
```

필드, 속성, 이벤트 등 형식의 `public` 멤버 이름을 지정할 때 파스칼식 대/소문자를 사용합니다. 또한 모든 메서드와 로컬 함수에 대해 파스칼식 대/소문자 구분을 사용합니다.

C#

```
public class ExampleEvents
{
    // A public field, these should be used sparingly
    public bool IsValid;

    // An init-only property
    public IWorkerQueue WorkerQueue { get; init; }

    // An event
    public event Action EventProcessing;

    // Method
    public void StartEventProcessing()
    {
        // Local function
        static int CountQueueItems() => WorkerQueue.Count;
        // ...
    }
}
```

위치 레코드를 작성할 때는 레코드의 public 속성인 매개 변수에 파스칼식 대/소문자를 사용합니다.

C#

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

위치 레코드에 대한 자세한 내용은 [속성 정의에 대한 위치 구문](#)을 참조하세요.

## 카멜식 대/소문자

`private` 또는 `internal` 필드 이름을 지정할 때 카멜식 대/소문자("camelCasing")를 사용하고 앞에 `_`을 붙입니다. 대리자 형식의 인스턴스를 포함하여 지역 변수의 이름을 지정할 때 카멜식 대/소문자를 사용합니다.

C#

```
public class DataService
{
    private IWorkerQueue _workerQueue;
}
```

## 💡 팁

문 완성을 지원하는 IDE에서 이러한 명명 규칙을 따르는 C# 코드를 편집할 때 `_`을 입력하면 개체 범위 멤버가 모두 표시됩니다.

`private` 또는 `internal` 인 `static` 필드를 사용할 때는 `s_` 접두사를 사용하고 스레드 정적인 경우 `t_`을 사용합니다.

C#

```
public class DataService
{
    private static IWorkerQueue s_workerQueue;

    [ThreadStatic]
    private static TimeSpan t_timeSpan;
}
```

메서드 매개 변수를 작성할 때 카멜식 대/소문자를 사용합니다.

C#

```
public T SomeMethod<T>(int someNumber, bool isValid)
{ }
```

C# 명명 규칙에 대한 자세한 내용은 [.NET 런타임 팀의 코딩 스타일](#)을 참조하세요.

## 형식 매개 변수 명명 지침

다음 지침은 제네릭 형식 매개 변수의 형식 매개 변수에 적용됩니다. 형식 매개 변수는 제네릭 형식 또는 제네릭 메서드의 인수에 대한 자리 표시자입니다. C# 프로그래밍 가이드에서 [제네릭 형식 매개 변수](#)에 대해 자세히 알아볼 수 있습니다.

- **필수적** 단일 문자 이름으로도 자체 설명이 가능하여 설명적인 이름을 굳이 사용할 필요가 없는 경우가 아니면 제네릭 형식 매개 변수 이름을 설명적인 이름으로 지정하세요.

`./snippets/coding-conventions`

```
public interface ISessionChannel<TSession> { /*...*/ }
public delegate TOutput Converter<TInput, TOutput>(TInput from);
public class List<T> { /*...*/ }
```

- 선택적 단일 문자 형식 매개 변수를 사용하는 형식에는 형식 매개 변수 이름으로 `T`를 사용해 보세요.

```
./snippets/coding-conventions

public int IComparer<T>() { return 0; }
public delegate bool Predicate<T>(T item);
public struct Nullable<T> where T : struct { /*...*/ }
```

- 필수적 설명적인 형식 매개 변수 이름 앞에 “T”를 붙이세요.

```
./snippets/coding-conventions

public interface ISessionChannel<TSession>
{
    TSession Session { get; }
```

- 선택적 매개 변수 이름 안에 형식 매개 변수에 적용되는 제약 조건을 나타내 보세요. 예를 들어, `ISession`으로 제한된 매개 변수는 `TSession`이라고 불릴 수 있습니다.

코드 분석 규칙 CA1715를 사용하여 형식 매개 변수의 이름이 적절하게 지정되었는지 확인할 수 있습니다.

## 추가 명명 규칙

- `using` 지시문이 포함되지 않는 예제에서는 네임스페이스 한정을 사용합니다. 프로젝트에서 네임스페이스를 기본적으로 가져오는 경우에는 해당 네임스페이스의 이름을 정규화할 필요가 없습니다. 정규화된 이름은 한 줄에 표시하기가 너무 길면 다음 예에 나와 있는 것처럼 점(.)으로 분할할 수 있습니다.

```
C#

var currentPerformanceCounterCategory = new System.Diagnostics.
    PerformanceCounterCategory();
```

- 다른 지침에 맞도록 조정하기 위해 Visual Studio 디자이너 도구를 사용하여 만든 개체 이름을 변경할 필요는 없습니다.



GitHub에서 Microsoft와 공동 작업



## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

제공해 주세요.

💡 설명서 문제 열기

↗️ 제품 사용자 의견 제공

# 일반적인 C# 코드 규칙

아티클 • 2024. 02. 20.

코드 표준은 개발 팀 내에서 코드 가독성, 일관성 및 공동 작업을 유지하는 데 매우 중요한 역할을 합니다. 업계 사례와 설정된 지침을 따르는 코드는 이해하기 쉽고, 유지 관리하고 확장하기도 편합니다. 대부분의 프로젝트는 코드 규칙을 통해 일관된 스타일을 적용합니다. [dotnet/docs](#) 프로젝트와 [dotnet/samples](#) 프로젝트도 예외는 아닙니다. 이 문서 시리즈에서는 코딩 규칙과 이를 적용하는 데 사용하는 도구를 알아봅니다. 규칙을 있는 그대로 적용해도 되고, 팀의 요구에 맞게 수정할 수도 있습니다.

Microsoft는 다음 목표를 기준으로 규칙을 선택했습니다.

- 정확성:** Microsoft 샘플은 사용자의 애플리케이션으로 복사 및 붙여넣기 하여 사용됩니다. 따라서 Microsoft는 여러 번의 편집 후에도 복원력과 정확성을 유지하는 코드를 만들 필요가 있습니다.
- 교육 효과:** 샘플의 목적은 .NET 및 C#를 모두 가르치는 것입니다. 따라서 어떠한 언어 기능이나 API에도 제한을 두지 않습니다. 대신, 이러한 샘플은 기능 선택이 제대로 된 경우에 교육 효과가 있습니다.
- 일관성:** 독자는 콘텐츠 전체에서 일관된 환경을 기대합니다. 모든 샘플은 동일한 스타일을 따라야 합니다.
- 채택성:** Microsoft는 새 언어 기능을 사용하도록 샘플을 적극적으로 업데이트합니다. 이러한 방식은 새로운 기능이 널리 알려지고 모든 C# 개발자에게 더 친숙해지도록 만듭니다.

## ① 중요

이러한 지침은 Microsoft에서 샘플 및 설명서를 개발하는 데 사용됩니다. 지침은 [.NET 런타임](#), [C# 코딩 스타일](#) 및 [C# 컴파일러\(roslyn\)](#) 지침에서 채택되었습니다. 이러한 지침은 오랜 기간 오픈 소스 개발에서 테스트를 거쳐 선택된 것입니다. 지침은 커뮤니티 구성원이 런타임 및 컴파일러 프로젝트에 참여하는 데 도움을 주었습니다. 지침의 쓰임새는 일반적인 C# 규칙의 예이지, 신뢰할 수 있는 목록(이 경우 [프레임워크 디자인 지침](#) 참조)은 아닙니다.

문서 코딩 규칙은 **교육 효과** 및 **채택성** 목표로 인해 런타임 및 컴파일러 규칙과 차별화됩니다. 런타임과 컴파일러에는 실행 부하 과다 경로에 대한 엄격한 성능 메트릭이 있습니다. 다른 많은 애플리케이션은 그렇지 않습니다. Microsoft의 **교육 효과** 목표는 어떤 구문도 금지하지 않을 것을 의무화합니다. 대신, 구문 사용이 필요한 경우를 샘플로 보여줍니다. Microsoft는 대부분의 프로덕션 애플리케이션보다 더 적극적으로 샘플을 업데이트합니다. **채택성** 목표에 따라 Microsoft는 사용자가 오늘 작성

해야 하는 코드를 보여주며, 작년에 작성된 코드에 변경이 필요 없는 경우도 예외가 아닙니다.

이 문서는 Microsoft 지침을 설명합니다. 지침은 시간이 지남에 따라 진화했으며, 지침을 따르지 않는 샘플도 보일 수 있습니다. 그러한 샘플을 규정 준수로 인도하는 PR이나, 업데이트가 필요한 샘플로 주의를 끄는 문제 제출을 환영합니다. Microsoft 지침은 오픈 소스이며, PR과 문제 제출을 환영합니다. 하지만 제출로 권장 사항을 변경하려는 경우 먼저 논의를 위해 문제를 오픈하세요. 지침을 사용하실 수 있으며, 요구 사항에 맞게 조정할 수 있습니다.

## 도구 및 분석기

도구는 팀이 표준을 적용하도록 하는 데 도움이 될 수 있습니다. [코드 분석](#)을 사용하여 원하는 규칙을 적용할 수 있습니다. 또한 [editorconfig](#)를 만들어 Visual Studio에서 사용자의 스타일 지침을 자동으로 적용하게 할 수 있습니다. 우선 시작점에서 [dotnet/docs 리포지토리 파일](#)을 복사하여 Microsoft 스타일을 사용할 수 있습니다.

이러한 도구를 사용하면 선호하는 지침을 팀에서 더 쉽게 채택할 수 있습니다. Visual Studio는 범위 내 모든 `.editorconfig` 파일에 규칙을 적용하여 코드의 형식을 지정합니다. 여러 구성을 사용하여 회사 전체 표준, 팀 표준 및 훨씬 세분된 프로젝트 표준을 적용할 수 있습니다.

코드 분석은 활성화된 규칙이 위반될 때 경고 및 진단을 생성합니다. 프로젝트에 적용하려는 규칙을 구성합니다. 그런 다음 각 CI 빌드는 개발자가 규칙을 위반할 때 개발자에게 알립니다.

## 진단 ID

- 자체 분석기를 빌드할 때 [적절한 진단 ID 선택](#)

## 언어 지침

다음 섹션에서는 .NET 문서 팀이 코드 예제와 샘플을 준비할 때 따르는 방식을 설명합니다. 일반적으로 다음 방식을 따릅니다.

- 가능하면 최신 버전의 언어 기능과 C#을 활용합니다.
- 사용되지 않거나 오래된 언어 구문은 사용하지 않습니다.
- 적절히 처리할 수 있는 예외만 catch하며, 제네릭 예외는 catch하지 않습니다.
- 특정 예외 유형을 사용하여 의미 있는 오류 메시지를 제공합니다.
- 컬렉션 조작에 LINQ 쿼리와 메서드를 사용하여 코드 가독성을 높입니다.

- I/O 바인딩된 작업에 `async` 및 `await`를 사용한 비동기 프로그래밍을 사용합니다.
  - 교착 상태에 주의하고 적절한 경우 `Task.ConfigureAwait`을(를) 사용합니다.
  - 데이터 형식에 런타임 형식 대신 언어 키워드를 사용합니다. 예를 들어 `System.String` 대신 `string`을(를), `System.Int32` 대신 `int`을(를) 사용합니다.
  - 부호 없는 형식 대신 `int`을(를) 사용합니다. C# 전체에서 `int` 사용은 일반적이며, `int`을(를) 사용할 때 다른 라이브러리와 더 쉽게 상호 작용합니다. 부호 없는 데이터 형식과 관련된 문서의 경우는 예외입니다.
  - 독자가식에서 형식을 유추할 수 있는 경우에만 `var`을(를) 사용합니다. 독자는 문서 플랫폼에서 샘플을 봅니다. 변수 형식을 표시하는 호버 또는 도구 팁이 없습니다.
  - 명확성과 단순성을 염두에 두고 코드를 작성합니다.
  - 지나치게 복잡하고 복잡한 코드 논리는 피합니다.

다음은 좀 더 구체적인 지침입니다.

# 문자열 데이터

- 다음 코드에 나와 있는 것처럼 **문자열 보간**을 사용하여 짧은 문자열을 연결합니다.

C#

```
string displayName = $"{nameList[n].LastName},  
{nameList[n].FirstName}";
```

- 특히 많은 양의 텍스트를 사용할 때 문자열을 루프에 추가하려면 `System.Text.StringBuilder` 개체를 사용합니다.

C#

백열

- 선언 줄에서 배열을 초기화할 때는 간결한 구문을 사용합니다. 다음 예제에서는 `string[]` 대신 `var`을(를) 사용할 수 없습니다.

C#

```
string[] vowels1 = { "a", "e", "i", "o", "u" };
```

- 명시적 인스턴스화를 사용하는 경우 `var`을 사용할 수 있습니다.

C#

```
var vowels2 = new string[] { "a", "e", "i", "o", "u" };
```

## 대리자

- 대리자 형식을 정의하는 대신 `Func<>` 및 `Action<>`을 사용합니다. 클래스에서 대리자 메서드를 정의합니다.

C#

```
Action<string> actionExample1 = x => Console.WriteLine($"x is: {x}");

Action<string, string> actionExample2 = (x, y) =>
    Console.WriteLine($"x is: {x}, y is {y}");

Func<string, int> funcExample1 = x => Convert.ToInt32(x);

Func<int, int, int> funcExample2 = (x, y) => x + y;
```

- `Func<>` 또는 `Action<>` 대리자로 정의된 시그니처를 사용하여 메서드를 호출합니다.

C#

```
actionExample1("string for x");

actionExample2("string for x", "string for y");

Console.WriteLine($"The value is {funcExample1("1")}");

Console.WriteLine($"The sum is {funcExample2(1, 2)}");
```

- 대리자 형식의 인스턴스를 만드는 경우 간결한 구문을 사용합니다. 클래스에서 일치하는 시그니처가 있는 대리자 형식 및 메서드를 정의합니다.

C#

```
public delegate void Del(string message);

public static void DelMethod(string str)
```

```
{  
    Console.WriteLine("DelMethod argument: {0}", str);  
}
```

- 대리자 형식의 인스턴스를 만들고 호출합니다. 다음 선언에서는 압축된 구문을 보여 줍니다.

C#

```
Del exampleDel2 = DelMethod;  
exampleDel2("Hey");
```

- 다음 선언에서는 전체 구문을 사용합니다.

C#

```
Del exampleDel1 = new Del(DelMethod);  
exampleDel1("Hey");
```

## 예외 처리의 try-catch 및 using 문

- 대부분의 예외 처리에서는 try-catch 문을 사용합니다.

C#

```
static double ComputeDistance(double x1, double y1, double x2, double  
y2)  
{  
    try  
    {  
        return Math.Sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 -  
y2));  
    }  
    catch (System.ArithmaticException ex)  
    {  
        Console.WriteLine($"Arithmatic overflow or underflow: {ex}");  
        throw;  
    }  
}
```

- C# using 문을 사용하면 코드를 간소화할 수 있습니다. finally 블록의 코드가 Dispose 메서드 호출뿐인 try-finally 문이 있는 경우에는 using 문을 대신 사용합니다.

다음 예제에서 try-finally 문은 finally 블록의 Dispose 만 호출합니다.

C#

```
Font bodyStyle = new Font("Arial", 10.0f);
try
{
    byte charset = bodyStyle.GdiCharSet;
}
finally
{
    if (bodyStyle != null)
    {
        ((IDisposable)bodyStyle).Dispose();
    }
}
```

`using` 문을 사용하여 같은 작업을 수행할 수 있습니다.

C#

```
using (Font arial = new Font("Arial", 10.0f))
{
    byte charset2 = arial.GdiCharSet;
}
```

중괄호가 필요하지 않은 새 `using` 구문을 사용합니다.

C#

```
using Font normalStyle = new Font("Arial", 10.0f);
byte charset3 = normalStyle.GdiCharSet;
```

## && 및 || 연산자

- 다음 예제와 같이, 비교를 수행할 때 `&` 대신 `&&`을(를), `||` 대신 `|`을(를) 사용합니다.

C#

```
Console.Write("Enter a dividend: ");
int dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");
int divisor = Convert.ToInt32(Console.ReadLine());

if ((divisor != 0) && (dividend / divisor) is var result)
{
    Console.WriteLine("Quotient: {0}", result);
}
else
```

```
{  
    Console.WriteLine("Attempted division by 0 ends up here.");  
}
```

제수가 0인 경우 `if` 문의 두 번째 절을 실행하면 런타임 오류가 발생합니다. 그러나 첫 번째 식이 `false`이면 `&&` 연산자는 단락(short-circuit)됩니다. 즉, 두 번째 식을 계산하지 않습니다. `divisor`가 0인 경우 `&` 연산자는 둘 다를 계산하므로 런타임 오류가 발생합니다.

## new 연산자

- 다음 선언에 나와 있는 것처럼 간결한 형식의 개체 인스턴스화 중 하나를 사용합니다.

C#

```
var firstExample = new ExampleClass();
```

C#

```
ExampleClass instance2 = new();
```

앞의 선언은 다음 선언과 같습니다.

C#

```
ExampleClass secondExample = new ExampleClass();
```

- 다음 예제에 나와 있는 것처럼 개체 이니셜라이저를 사용하여 개체 만들기를 간소화합니다.

C#

```
var thirdExample = new ExampleClass { Name = "Desktop", ID = 37414,  
                                     Location = "Redmond", Age = 2.3 };
```

다음 예제에서는 앞의 예제와 같은 속성을 설정하지만, 이니셜라이저를 사용하지는 않습니다.

C#

```
var fourthExample = new ExampleClass();  
fourthExample.Name = "Desktop";  
fourthExample.ID = 37414;
```

```
fourthExample.Location = "Redmond";
fourthExample.Age = 2.3;
```

## 이벤트 처리

- 나중에 제거할 필요가 없는 이벤트 처리기를 정의하려는 경우 람다 식을 사용합니다.

C#

```
public Form2()
{
    this.Click += (s, e) =>
    {
        MessageBox.Show(
            ((MouseEventArgs)e).Location.ToString());
    };
}
```

람다 식은 다음과 같은 기존 정의를 줄여 줍니다.

C#

```
public Form1()
{
    this.Click += new EventHandler(Form1_Click);
}

void Form1_Click(object? sender, EventArgs e)
{
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());
}
```

## 정적 멤버

*ClassName.StaticMember*와 같이 클래스 이름을 사용하여 **static** 멤버를 호출합니다. 이렇게 하면 정적 액세스가 명확하게 표시되므로 코드를 보다 쉽게 읽을 수 있습니다. 파생 클래스 이름을 사용하여 기본 클래스에 정의된 정적 멤버를 정규화하지 않습니다. 이 코드는 컴파일되기는 하지만 가독성이 떨어지며 나중에 파생 클래스와 이름이 같은 정적 멤버를 추가하면 코드가 손상될 수도 있습니다.

## LINQ 쿼리

- 쿼리 변수에 의미 있는 이름을 사용합니다. 다음 예제에서는 Seattle 거주 고객에 대해 `seattleCustomers`를 사용합니다.

C#

```
var seattleCustomers = from customer in customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

- 별칭을 사용하여 익명 형식의 속성 이름 대/소문자를 올바르게 표시합니다(파스칼식 대/소문자 사용).

C#

```
var localDistributors =
    from customer in customers
    join distributor in distributors on customer.City equals
distributor.City
    select new { Customer = customer, Distributor = distributor };
```

- 결과의 속성 이름이 모호하면 속성 이름을 바꿉니다. 예를 들어 쿼리에서 고객 이름과 배포자 ID를 반환하는 경우 결과에서 이러한 정보를 `Name` 및 `ID`로 유지하는 대신 `Name`은 고객의 이름이고 `ID`는 배포자의 ID임을 명확하게 나타내도록 이름을 바꿉니다.

C#

```
var localDistributors2 =
    from customer in customers
    join distributor in distributors on customer.City equals
distributor.City
    select new { CustomerName = customer.Name, DistributorID =
distributor.ID };
```

- 쿼리 변수 및 범위 변수의 선언에서 암시적 형식을 사용합니다. LINQ 쿼리의 암시적 형식에 대한 이 지침은 [암시적 형식 지역 변수](#)에 대한 일반 규칙을 재정의합니다. LINQ 쿼리는 무명 형식을 만드는 프로젝션을 사용하는 경우가 많습니다. 다른 쿼리식은 중첩된 제네릭 형식으로 결과를 만듭니다. 암시적 형식 변수는 더 읽기 쉬운 경우가 많습니다.

C#

```
var seattleCustomers = from customer in customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

- 위 예제처럼 `from` 절 아래의 쿼리 절을 정렬합니다.
- `where` 절을 다른 쿼리 절 앞에 사용하여, 뒤에 있는 쿼리 절이 필터링으로 범위가 좁아진 데이터 집합에 대해 작동하게 합니다.

C#

```
var seattleCustomers2 = from customer in customers
                        where customer.City == "Seattle"
                        orderby customer.Name
                        select customer;
```

- 하나의 `join` 절 대신 여러 개의 `from` 절을 사용하여 내부 컬렉션에 액세스합니다. 예를 들어 `Student` 개체 컬렉션이 각각 테스트 점수 컬렉션을 포함하는 경우 다음 쿼리를 실행하면 90점보다 높은 각 점수와 해당 점수를 받은 학생의 성이 반환됩니다.

C#

```
var scoreQuery = from student in students
                  from score in student.Scores!
                  where score > 90
                  select new { Last = student.LastName, score };
```

## 암시적 형식 지역 변수

- 할당 오른쪽에서 변수 형식이 명확하면 지역 변수에 대해 **암시적 형식**을 사용합니다.

C#

```
var message = "This is clearly a string.";
var currentTemperature = 27;
```

- 할당 오른쪽에서 변수 형식이 명확하지 않으면 `var`를 사용하지 않습니다. 메서드 이름에서 형식이 명확하다고 가정하지 않습니다. 변수 형식이 `new` 연산자, 명시적 캐스트 또는 리터럴 값에 대한 할당인 경우 명확한 것으로 간주합니다.

C#

```
int numberOfIterations = Convert.ToInt32(Console.ReadLine());
int currentMaximum = ExampleClass.ResultSoFar();
```

- 변수 이름을 사용하여 변수의 형식을 지정하지 않습니다. 이렇게 하면 형식이 올바르게 지정되지 않을 수 있습니다. 대신 형식을 사용하여 형식을 지정하고, 변수 이름

을 사용하여 변수의 의미 체계 정보를 나타냅니다. 다음 예제에서는 형식에 `string` 을(를) 사용하고, 콘솔에서 읽은 정보의 의미를 나타내는 데 `iterations` 과(와) 같은 것을 사용해야 합니다.

C#

```
var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- **dynamic** 대신 `var`를 사용하지 않습니다. 런타임 형식 유추를 원하는 경우 `dynamic`을 사용합니다. 자세한 내용은 [dynamic 형식 사용\(C# 프로그래밍 가이드\)](#)을 참조하세요.
  - **for** 루프의 루프 변수에 암시적 형식을 사용합니다.

다음 예제에서는 `for` 문에서 암시적 형식을 사용합니다.

C#

- `foreach` 루프의 루프 변수 형식을 결정하는 데 암시적 형식을 사용하지 않습니다. 대부분의 경우 컬렉션 요소의 형식이 즉시 명확하지는 않습니다. 해당 요소의 형식을 유추하는데 컬렉션의 이름에만 의존해서는 안 됩니다.

다음 예제에서는 `foreach` 문에서 명시적 형식을 사용합니다.

C#

```
foreach (char ch in laugh)
{
    if (ch == 'h')
        Console.Write("H");
    else
        Console.Write(ch);
}
Console.WriteLine();
```

- LINQ 쿼리의 결과 시퀀스에 암시적 형식을 사용합니다. LINQ 섹션에서는 많은 LINQ 쿼리로 인해 암시적 형식을 사용해야 하는 무명 형식이 발생하는 것을 설명합니다. 다른 쿼리는 더 읽기 쉬운 중첩된 제네릭 형식 `var` 을(를) 생성합니다.

### ① 참고

반복 가능한 컬렉션의 요소 형식을 실수로 변경하지 않도록 주의해야 합니다. 예를 들어 `foreach` 문에서 `System.Linq.IQueryable`을 `System.Collections.IEnumerable`으로 전환하기 쉬운데 그러면 쿼리 실행이 변경됩니다.

일부 샘플에서는 식의 자연스러운 형식을 설명합니다. 이러한 샘플에서는 컴파일러가 자연 형식을 선택할 수 있도록 `var` 을(를) 사용해야 합니다. 이러한 예제는 덜 명확하지만 샘플에 `var` 을(를) 사용해야 합니다. 텍스트는 동작을 설명해야 합니다.

## using 지시문을 네임스페이스 선언 외부에 배치

`using` 지시문이 네임스페이스 선언 외부에 있는 경우 가져온 네임스페이스는 정규화된 이름입니다. 정규화된 이름은 더 명확합니다. `using` 지시문이 네임스페이스 내부에 있는 경우 해당 네임스페이스에 상대적이거나 정규화된 이름일 수 있습니다.

C#

```
using Azure;

namespace CoolStuff.AwesomeFeature
{
    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

`WaitUntil` 클래스에 대한 참조(직접 또는 간접)가 있다고 가정합니다.

이제 약간 변경해 보겠습니다.

C#

```
namespace CoolStuff.AwesomeFeature
{
```

```
using Azure;

public class Awesome
{
    public void Stuff()
    {
        WaitUntil wait = WaitUntil.Completed;
        // ...
    }
}
```

오늘 컴파일합니다. 또 내일 컴파일합니다. 하지만 다음 주에는 이전(손대지 않은) 코드가 두 가지 오류로 실패합니다.

#### 콘솔

```
- error CS0246: The type or namespace name 'WaitUntil' could not be found
(are you missing a using directive or an assembly reference?)
- error CS0103: The name 'WaitUntil' does not exist in the current context
```

종속성 중 하나가 네임스페이스에 이 클래스를 사용한 후 `.Azure`(으)로 끝났습니다.

#### C#

```
namespace CoolStuff.Azure
{
    public class SecretsManagement
    {
        public string FetchFromKeyVault(string vaultId, string secretId) {
            return null; }
    }
}
```

네임스페이스 내부에 배치되는 `using` 지시문은 상황에 따라 다르며 이름 확인을 복잡하게 만듭니다. 이 예제에서는 첫 번째 네임스페이스를 찾습니다.

- `CoolStuff.AwesomeFeature.Azure`
- `CoolStuff.Azure`
- `Azure`

`CoolStuff.Azure` 또는 `CoolStuff.AwesomeFeature.Azure` 과(와) 일치하는 새 네임스페이스를 추가하면 전역 `Azure` 네임스페이스보다 먼저 일치합니다. `using` 선언에 `global::` 한 정자를 추가하여 이를 해결할 수 있습니다. 그러나 네임스페이스 외부에 `using` 선언을 배치하는 것이 더 쉽습니다.

C#

```
namespace CoolStuff.AwesomeFeature
{
    using global::Azure;

    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

## 스타일 지침

일반적으로 코드 샘플에는 다음 형식을 사용합니다.

- 들여쓰기에 네 개의 공백을 사용합니다. 탭을 사용하지 않습니다.
- 가독성을 위해 코드를 일관되게 정렬합니다.
- 특히 모바일 화면에서 문서 코드의 가독성을 위해 줄을 65자로 제한합니다.
- 긴 문을 여러 줄로 나누면 명확성이 향상됩니다.
- 중괄호에는 "Allman" 스타일을 사용합니다. 여는 중괄호와 닫는 중괄호는 그 자체가 새 줄입니다. 중괄호는 현재 들여쓰기 수준에 맞춰 정렬됩니다.
- 필요한 경우 이진 연산자 앞에 줄 바꿈이 있어야 합니다.

## 주석 스타일

- 한 줄 주석(//)으로 간단하게 설명합니다.
- 여러 줄 주석(\* \*)으로 길게 설명하지 않습니다. 주석은 현지화되지 않습니다. 대신, 긴 설명은 관련 문서에 나와 있습니다.
- 메서드, 클래스, 필드 및 모든 공용 멤버는 [XML 주석](#)을 사용하여 설명합니다.
- 코드 줄의 끝이 아닌 별도의 줄에 주석을 배치합니다.
- 주석 텍스트는 대문자로 시작합니다.
- 주석 텍스트 끝에는 마침표를 붙입니다.
- 다음 예제와 같이 주석 구분 기호(//)와 주석 텍스트 사이에 공백을 하나 삽입합니다.

C#

```
// The following declaration creates a query. It does not run  
// the query.
```

## 레이아웃 규칙

효율적인 레이아웃에서는 서식을 사용하여 코드 구조를 강조하고 코드를 보다 쉽게 읽을 수 있도록 생성합니다. Microsoft 예제 및 샘플은 다음 규칙을 따릅니다.

- 기본 코드 편집기 설정(스마트 들여쓰기, 4자 들여쓰기, 탭을 공백으로 저장)을 사용 합니다. 자세한 내용은 [옵션](#), [텍스트 편집기](#), [C#, 서식](#)을 참조하세요.
- 문을 한 줄에 하나씩만 작성합니다.
- 선언을 한 줄에 하나씩만 작성합니다.
- 연속 줄이 자동으로 들여쓰기되지 않으면 탭 정지 1개(공백 4개)로 들여쓰기합니다.
- 메서드 정의와 속성 정의 간에는 빈 줄을 하나 이상 추가합니다.
- 다음 코드에 나와 있는 것처럼 괄호를 사용하여 식의 절을 명확하게 구분합니다.

C#

```
if ((startX > endX) && (startX > previousX))  
{  
    // Take appropriate action.  
}
```

샘플에서 연산자 또는 식 우선순위를 설명하는 경우는 예외입니다.

## 보안

[보안 코딩 지침](#)의 지침을 따르세요.

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 [설명서 문제 열기](#)

수도 있습니다. 자세한 내용은  
참여자 가이드를 참조하세요.

☞ 제품 사용자 의견 제공

# 명령줄 인수를 표시하는 방법

아티클 • 2023. 06. 08.

명령줄에서 실행 파일에 제공된 인수는 [최상위 문](#) 또는 `Main`에 대한 선택적 매개 변수를 통해 액세스할 수 있습니다. 인수는 문자열 배열의 형태로 제공됩니다. 배열의 각 요소에는 하나의 인수가 포함되어 있습니다. 인수 사이의 공백은 제거됩니다. 예를 들어 명령줄에서 다음과 같이 가상 실행 파일을 호출한다고 가정합니다.

명령줄 입력	Main에 전달되는 문자열 배열
<code>executable.exe a b c</code>	"a" "b" "c"
<code>executable.exe one two</code>	"one" "two"
<code>executable.exe "one two" three</code>	"one two" "three"

## ① 참고

Visual Studio에서 애플리케이션을 실행할 경우 [프로젝트 디자이너, 디버그 페이지](#)에서 명령줄 인수를 지정할 수 있습니다.

## 예제

이 예제에서는 명령줄 애플리케이션에 전달된 명령줄 인수를 표시합니다. 위의 표에서 첫 번째 항목에 대한 출력이 표시됩니다.

C#

```
// The Length property provides the number of array elements.
Console.WriteLine($"parameter count = {args.Length}");

for (int i = 0; i < args.Length; i++)
{
    Console.WriteLine($"Arg[{i}] = [{args[i]}]");
}
```

```
/* Output (assumes 3 cmd line args):  
parameter count = 3  
Arg[0] = [a]  
Arg[1] = [b]  
Arg[2] = [c]  
*/
```

## 참고 항목

- [System.CommandLine 개요](#)
- 자습서: [System.CommandLine 시작](#)

# 클래스 및 개체를 사용한 개체 지향 프로그래밍 살펴보기

아티클 • 2024. 02. 19.

이 자습서에서는 콘솔 애플리케이션을 빌드하고 C# 언어의 일부인 기본 개체 지향 기능을 확인합니다.

## 필수 조건

- Windows 또는 Mac용 [Visual Studio](#)를 사용하는 것이 좋습니다. [Visual Studio 다운로드 페이지](#)에서 무료 버전을 다운로드할 수 있습니다. Visual Studio에는 .NET SDK가 포함되어 있습니다.
- [Visual Studio Code](#) 편집기를 사용할 수도 있습니다. 최신 [.NET SDK](#)를 별도로 설치해야 합니다.
- 다른 편집기를 선호하는 경우 최신 [.NET SDK](#)를 설치해야 합니다.

## 애플리케이션 만들기

터미널 창을 사용하여 클래스라는 �렉터리를 만듭니다. 거기에 애플리케이션을 빌드할 것입니다. 해당 디렉터리로 변경하고 콘솔 창에 `dotnet new console`을 입력합니다. 이 명령은 애플리케이션을 만듭니다. `Program.cs`를 엽니다. 다음과 같이 표시됩니다.

C#

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

이 자습서에서는 은행 계좌를 나타내는 새로운 형식을 만듭니다. 일반적으로 개발자는 여러 텍스트 파일에 각 클래스를 정의합니다. 그러면 프로그램의 크기가 커질 때 쉽게 관리할 수 있습니다. 클래스 딕터리에 `BankAccount.cs`라는 이름의 새 파일을 만듭니다.

이 파일에는 **은행 계좌**의 정의가 포함됩니다. 개체 지향 프로그래밍은 **클래스** 형태로 형식을 생성하여 코드를 구성합니다. 이러한 클래스에는 특정 엔티티를 나타내는 코드가 포함됩니다. `BankAccount` 클래스는 은행 계좌를 나타냅니다. 코드는 메서드 및 속성을 통해 특정 작업을 구현합니다. 이 자습서에서 은행 계좌는 다음 동작을 지원합니다.

- 은행 계좌를 고유하게 식별하는 10자리 숫자가 있습니다.
- 소유자의 이름을 저장하는 문자열이 있습니다.
- 잔액을 검색할 수 있습니다.

4. 예금을 허용합니다.
5. 인출을 허용합니다.
6. 초기 잔액은 양수여야 합니다.
7. 인출로 인해 잔액이 음수가 될 수 없습니다.

## 은행 계좌 형식 정의

동작을 정의하는 클래스의 기본 사항을 만들어 시작할 수 있습니다. **File>New** 명령을 사용하여 새 파일을 만듭니다. 파일의 이름을 *BankAccount.cs*로 지정합니다. *BankAccount.cs* 파일에 다음 코드를 추가합니다.

C#

```
namespace Classes;

public class BankAccount
{
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance { get; }

    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
    }
}
```

계속하기 전에 빌드한 내용을 살펴보겠습니다. `namespace` 선언은 코드를 논리적으로 구성하는 방법을 제공합니다. 이 자습서는 비교적 작으므로 하나의 네임스페이스에 모든 코드를 넣습니다.

`public class BankAccount`(은)는 생성하는 클래스 또는 형식을 정의합니다. 클래스 선언 뒤에 오는 `{` 및 `}`의 모든 항목은 클래스의 상태와 동작을 정의합니다. `BankAccount` 클래스의 **멤버**가 다섯 개 있습니다. 첫 번째 세 개는 **속성**입니다. 속성은 데이터 요소이며 유효성 검사 또는 기타 규칙을 적용하는 코드가 있을 수 있습니다. 마지막 두 개는 **메서드**입니다. 메서드는 단일 함수를 수행하는 코드 블록입니다. 각 멤버의 이름을 읽으면 사용자 또는 다른 개발자가 클래스가 수행하는 작업을 이해하기에 충분한 정보를 제공해야 합니다.

## 새 계좌 개설

구현할 첫 번째 기능은 은행 계좌 개설 기능입니다. 고객이 계좌를 개설할 때 초기 잔액과 해당 계좌 소유자에 대한 정보를 제공해야 합니다.

`BankAccount` 형식의 새 개체를 생성하는 것은 해당 값을 지정하는 **생성자**를 정의하는 것입니다. **생성자**는 클래스와 이름이 같은 멤버입니다. 해당 클래스 형식의 개체를 초기화하는 데 사용됩니다. `BankAccount` 형식에 다음 생성자를 추가합니다. `MakeDeposit` 선언 위에 다음 코드를 배치합니다.

C#

```
public BankAccount(string name, decimal initialBalance)
{
    this.Owner = name;
    this.Balance = initialBalance;
}
```

앞의 코드는 `this` 한정자를 포함하여 생성되는 개체의 속성을 식별합니다. 해당 한정자는 일반적으로 선택 사항이며 생략됩니다. 다음을 작성할 수도 있습니다.

C#

```
public BankAccount(string name, decimal initialBalance)
{
    Owner = name;
    Balance = initialBalance;
}
```

`this` 한정자는 지역 변수 또는 매개 변수의 이름이 해당 필드 또는 속성과 같은 경우에만 필요합니다. 필요한 경우가 아니면 이 문서의 나머지 부분에서 `this` 한정자는 생략됩니다.

생성자는 `new`를 사용하여 개체를 만들 때 호출됩니다. `Program.cs`의 `Console.WriteLine("Hello World!");` 줄을 다음 코드로 바꿉니다(<name>을 사용자의 이름으로 바꿈).

C#

```
using Classes;

var account = new BankAccount("<name>", 1000);
Console.WriteLine($"Account {account.Number} was created for {account.Owner}
with {account.Balance} initial balance.");
```

지금까지 빌드한 코드를 실행해 보겠습니다. Visual Studio를 사용하는 경우 **디버그** 메뉴에서 **디버깅하지 않고 시작**을 선택합니다. 명령줄을 사용하는 경우 프로젝트를 만든 디렉

터리에 `dotnet run` 을 입력합니다.

계좌 번호가 공백인가요? 이를 수정해 보겠습니다. 개체가 생성될 때 계좌 번호를 지정해야 합니다. 그러나 계좌 번호 생성은 호출자의 책임이 아닙니다. `BankAccount` 클래스 코드는 새 계좌 번호를 지정하는 방법을 알아야 합니다. 간단한 방법은 10자리 숫자로 시작하는 것입니다. 새 계좌가 생성될 때마다 숫자가 늘어납니다. 마지막으로, 개체가 생성될 때 현재 계좌 번호를 저장합니다.

`BankAccount` 클래스에 멤버 선언을 추가합니다. `BankAccount` 클래스의 시작 부분에서 여는 중괄호 `{` 뒤에 다음 코드 줄을 배치합니다.

C#

```
private static int s_accountNumberSeed = 1234567890;
```

`accountNumberSeed`(은)는 데이터 멤버입니다. 이것은 `private`입니다. 즉 `BankAccount` 클래스 내의 코드로만 액세스할 수 있습니다. 이는 전용 구현(계좌 번호가 생성되는 방법)과 공공 책임(계좌 번호를 가지는 것 등)을 구분하는 방법입니다. 또한 `static`로서 모든 `BankAccount` 개체에서 공유됨을 의미합니다. 비정적 변수의 값은 `BankAccount` 개체의 각 인스턴스에 고유합니다. `accountNumberSeed`(은)는 `private static` 필드이므로 C# 명명 규칙에 따라 `s_` 접두사를 가집니다. `s`(은)는 `static`(을)를 나타내고 `_`(은)는 `private` 필드를 나타냅니다. 생성자에 다음 두 줄을 추가하여 계좌 번호를 지정합니다. `this.Balance = initialBalance` 줄 뒤에 배치합니다.

C#

```
Number = s_accountNumberSeed.ToString();
s_accountNumberSeed++;
```

`dotnet run` 을 입력하여 결과를 확인합니다.

## 예금 및 인출 만들기

은행 계좌 클래스는 제대로 작동하려면 예금과 인출을 허용해야 합니다. 계좌의 모든 트랜잭션에 대한 저널을 만들어 예금과 인출을 구현하겠습니다. 모든 트랜잭션을 추적하는 것은 단순히 각 트랜잭션의 잔액을 업데이트하는 것보다 몇 가지 이점이 있습니다. 기록을 사용하여 모든 트랜잭션을 감사하고 일별 잔액을 관리할 수 있습니다. 필요한 경우 모든 트랜잭션의 기록에서 잔액을 계산하면 고정된 단일 트랜잭션의 오류가 다음 계산의 잔액에 올바르게 반영되도록 할 수 있습니다.

트랜잭션을 나타내는 새 형식을 생성해 보겠습니다. 트랜잭션은 책임이 없는 간단한 형식입니다. 몇 가지 속성이 필요합니다. *Transaction.cs*라는 새 파일을 만듭니다. 파일에 다음 코드를 추가합니다.

```
C#  
  
namespace Classes;  
  
public class Transaction  
{  
    public decimal Amount { get; }  
    public DateTime Date { get; }  
    public string Notes { get; }  
  
    public Transaction(decimal amount, DateTime date, string note)  
    {  
        Amount = amount;  
        Date = date;  
        Notes = note;  
    }  
}
```

이제 `Transaction` 개체의 `List<T>`를 `BankAccount` 클래스에 추가하겠습니다. `BankAccount.cs` 파일의 생성자 뒤에 다음 선언을 추가합니다.

```
C#  
  
private List<Transaction> _allTransactions = new List<Transaction>();
```

이제 `Balance`를 올바르게 계산해 보겠습니다. 모든 트랜잭션의 값을 합하여 현재 잔액을 찾을 수 있습니다. 코드가 현재 상태이기 때문에 계정의 초기 잔액만 가져올 수 있으므로 `Balance` 속성을 업데이트해야 합니다. `BankAccount.cs`의 `public decimal Balance { get; }` 줄을 다음 코드로 바꿉니다.

```
C#  
  
public decimal Balance  
{  
    get  
    {  
        decimal balance = 0;  
        foreach (var item in _allTransactions)  
        {  
            balance += item.Amount;  
        }  
  
        return balance;  
    }  
}
```

```
    }  
}
```

이 예제에서는 속성의 중요한 측면을 보여 줍니다. 이제 다른 프로그래머가 값을 요청할 때 잔액을 계산합니다. 계산은 모든 트랜잭션을 열거하고 합계를 현재 잔액으로 제공합니다.

다음으로 `MakeDeposit` 및 `MakeWithdrawal` 메서드를 구현합니다. 이러한 메서드는 마지막 두 규칙을 적용합니다. 초기 잔액은 양수여야 하며 인출은 음수 잔액을 생성하면 안 됩니다.

이러한 규칙은 예외의 개념을 소개합니다. 메서드가 작업을 성공적으로 완료할 수 없음을 나타내는 표준 방법은 예외를 throw하는 것입니다. 예외 형식 및 관련 메시지는 오류를 설명합니다. 여기서 `MakeDeposit` 메서드는 입금 금액이 0보다 크지 않으면 예외를 throw합니다. `MakeWithdrawal` 메서드는 인출 금액이 0보다 크지 않거나 인출을 적용하면 음수 잔액이 발생하는 경우 예외를 throw합니다. `_allTransactions` 목록의 선언 뒤에 다음 코드를 추가합니다.

C#

```
public void MakeDeposit(decimal amount, DateTime date, string note)  
{  
    if (amount <= 0)  
    {  
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of  
deposit must be positive");  
    }  
    var deposit = new Transaction(amount, date, note);  
    _allTransactions.Add(deposit);  
}  
  
public void MakeWithdrawal(decimal amount, DateTime date, string note)  
{  
    if (amount <= 0)  
    {  
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of  
withdrawal must be positive");  
    }  
    if (Balance - amount < 0)  
    {  
        throw new InvalidOperationException("Not sufficient funds for this  
withdrawal");  
    }  
    var withdrawal = new Transaction(-amount, date, note);  
    _allTransactions.Add(withdrawal);  
}
```

`throw` 문은 예외를 **throw합니다**. 현재 블록의 실행이 종료되고 제어가 호출 스택에 있는 처음 일치하는 `catch` 블록으로 전달됩니다. `catch` 블록을 추가하여 나중에 이 코드를 테스트합니다.

생성자는 잔액을 직접 업데이트하지 않고 초기 트랜잭션을 추가하도록 변경해야 합니다. `MakeDeposit` 메서드를 이미 작성했으므로 생성자에서 호출합니다. 완성된 생성자는 다음과 같아야 합니다.

```
C#  
  
public BankAccount(string name, decimal initialBalance)  
{  
    Number = s_accountNumberSeed.ToString();  
    s_accountNumberSeed++;  
  
    Owner = name;  
    MakeDeposit(initialBalance, DateTime.Now, "Initial balance");  
}
```

`DateTime.Now`은 현재 날짜 및 시간을 반환하는 속성입니다. 새 `BankAccount`(을)를 만드는 코드에 따라 `Main` 메서드에 몇 가지 입금 및 인출을 추가하여 이 코드를 테스트합니다.

```
C#  
  
account.MakeWithdrawal(500, DateTime.Now, "Rent payment");  
Console.WriteLine(account.Balance);  
account.MakeDeposit(100, DateTime.Now, "Friend paid me back");  
Console.WriteLine(account.Balance);
```

다음으로, 잔액이 음수인 계정을 생성하여 오류 조건이 발생하는지 테스트합니다. 방금 추가한 이전 코드 뒤에 다음 코드를 추가합니다.

```
C#  
  
// Test that the initial balances must be positive.  
BankAccount invalidAccount;  
try  
{  
    invalidAccount = new BankAccount("invalid", -55);  
}  
catch (ArgumentOutOfRangeException e)  
{  
    Console.WriteLine("Exception caught creating account with negative  
balance");  
    Console.WriteLine(e.ToString());  
    return;  
}
```

`try-catch` 문(을)를 사용하여 예외를 `throw`할 수 있는 코드 블록을 표시하고 예상한 오류를 포착합니다. 동일한 기술을 사용하여 음수 잔액에 대한 예외를 `throw`하는 코드를 테스트할 수 있습니다. `Main` 메서드에서 `invalidAccount`(을)를 선언하기 전에 다음 코드를 추가합니다.

```
C#  
  
// Test for a negative balance.  
try  
{  
    account.MakeWithdrawal(750, DateTime.Now, "Attempt to overdraw");  
}  
catch (InvalidOperationException e)  
{  
    Console.WriteLine("Exception caught trying to overdraw");  
    Console.WriteLine(e.ToString());  
}
```

파일을 저장하고 `dotnet run`을 입력하여 시도해 보세요.

## 과제 - 모든 트랜잭션 기록

이 자습서를 완료하기 위해 트랜잭션 기록에 대해 `string`을 생성하는 `GetAccountHistory` 메서드를 작성할 수 있습니다. `BankAccount` 형식에 이 메서드를 추가합니다.

```
C#  
  
public string GetAccountHistory()  
{  
    var report = new System.Text.StringBuilder();  
  
    decimal balance = 0;  
    report.AppendLine("Date\t\tAmount\tBalance\tNote");  
    foreach (var item in _allTransactions)  
    {  
        balance += item.Amount;  
        report.AppendLine($"  
{item.Date.ToShortDateString()}\t{item.Amount}\t{balance}\t{item.Notes}");  
    }  
  
    return report.ToString();  
}
```

기록은 `StringBuilder` 클래스를 사용하여 각 트랜잭션에 대해 한 줄이 포함된 문자열의 형식을 지정합니다. 이러한 자습서의 앞부분에서 문자열 형식 지정 코드를 살펴보았습니다. 새 문자는 `\t`입니다. 이 새 문자는 탭을 삽입하여 출력 형식을 지정합니다.

*Program.cs*에서 테스트하려면 이 줄을 추가합니다.

C#

```
Console.WriteLine(account.GetAccountHistory());
```

프로그램을 실행하여 결과를 확인합니다.

## 다음 단계

잘 알 수 없는 경우 [GitHub 리포지토리](#)에서 이 자습서의 소스를 확인할 수 있습니다.

[개체 지향 프로그래밍](#) 자습서를 계속 진행할 수 있습니다.

다음 문서에서 이러한 개념을 더 자세히 알아볼 수 있습니다.

- [선택 문](#)
- [반복 문](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# 개체 지향 프로그래밍(C#)

아티클 • 2023. 06. 08.

C#은 개체 지향 프로그래밍 언어입니다. 개체 지향 프로그래밍의 네 가지 기본 원칙은 다음과 같습니다.

- 추상화 - 관련 특성 및 엔터티의 상호 작용을 클래스로 모델링하여 시스템의 추상적 표현을 정의합니다.
- 캡슐화 - 개체의 내부 상태와 기능을 숨기고 public 함수 세트를 통해서만 개체에 액세스할 수 있습니다.
- 상속 - 기존 추상화를 기반으로 새 추상화를 만들 수 있습니다.
- 다형성 - 여러 추상화에서 다양한 방법으로 상속된 속성 또는 메서드를 구현할 수 있습니다.

앞의 [클래스 소개](#) 자습서에서 추상화와 캡슐화를 살펴봤습니다. `BankAccount` 클래스는 은행 계좌 개념에 대한 추상화를 제공했습니다. `BankAccount` 클래스를 사용한 코드에 영향을 주지 않고 해당 구현을 수정할 수 있습니다. `BankAccount` 및 `Transaction` 클래스는 코드에서 이러한 개념을 설명하는 데 필요한 구성 요소의 캡슐화를 제공합니다.

이 자습서에서는 상속과 다형성을 활용하도록 이 애플리케이션을 확장해 새 기능을 추가합니다. 또한 `BankAccount` 클래스에 기능을 추가하여 앞의 자습서에서 배운 추상화 및 캡슐화 방법을 활용합니다.

## 다른 유형의 계좌 만들기

이 프로그램을 빌드한 후 기능 추가를 요청 받습니다. 이 프로그램은 은행 계좌 유형이 하나뿐인 상황에서는 잘 작동합니다. 시간이 지나면서 요구 사항이 바뀌고 관련된 다음과 같은 계정 유형이 요청됩니다.

- 매월말에 이자가 붙는 이자 소득 계좌.
- 잔고가 음수일 수 있지만 잔고가 있는 경우 매달 이자 비용이 발생하는 신용 한도.
- 1회 예치로 시작하고 지불만 가능한 선불 선물 카드 계좌. 이 계좌는 매월초에 한 번 잔고를 다시 채울 수 있습니다.

이 모든 다양한 계좌는 이전 자습서에서 정의한 `BankAccount` 클래스와 비슷합니다. 해당 코드를 복사하고 클래스 이름을 바꾸고 수정할 수도 있습니다. 이 방법은 단기적으로는 효과가 있지만 시간이 지남에 따라 작업이 늘어납니다. 모든 변경 내용은 영향을 받는 모든 클래스에 복사됩니다.

대신 이전 자습서에서 만든 `BankAccount` 클래스에서 메서드와 데이터를 상속하는 새 은행 계좌 유형을 만들 수 있습니다. 이러한 새 클래스는 각 유형에 필요한 특정 동작으로

`BankAccount` 클래스를 확장할 수 있습니다.

C#

```
public class InterestEarningAccount : BankAccount
{
}

public class LineOfCreditAccount : BankAccount
{
}

public class GiftCardAccount : BankAccount
{
}
```

이러한 각 클래스는 공유 기본 클래스인 `BankAccount` 클래스에서 공유 동작을 상속합니다. 파생 클래스 각각에 새롭고 다양한 기능의 구현을 작성합니다. 이러한 파생 클래스에는 이미 `BankAccount` 클래스에 정의된 동작이 모두 있습니다.

각각의 새 클래스는 서로 다른 소스 파일에 만드는 것이 좋습니다. [Visual Studio](#) 에서 프로젝트를 마우스 오른쪽 단추로 클릭하고 클래스 추가를 선택하여 새 파일에 새 클래스를 추가할 수 있습니다. [Visual Studio Code](#) 에서는 파일을 선택한 다음 새로 만들기를 선택하여 새 원본 파일을 만듭니다. 어느 도구에서나 클래스와 일치하도록 파일 이름을 지정합니다. *InterestEarningAccount.cs*, *LineOfCreditAccount.cs*, *GiftCardAccount.cs*.

위의 샘플에 나온 것처럼 클래스를 만들면 파생 클래스가 컴파일되지 않는 것을 확인할 수 있습니다. 생성자는 개체를 초기화합니다. 파생 클래스 생성자는 파생 클래스를 초기화하고 파생 클래스에 포함된 기본 클래스 개체를 초기화하는 방법에 대한 지침을 제공해야 합니다. 적절한 초기화는 일반적으로 추가 코드 없이 발생합니다. `BankAccount` 클래스는 다음 서명을 사용하여 하나의 공용 생성자를 선언합니다.

C#

```
public BankAccount(string name, decimal initialBalance)
```

컴파일러는 사용자가 직접 생성자를 정의할 때 기본 생성자를 생성하지 않습니다. 즉, 각 파생 클래스가 이 생성자를 명시적으로 호출해야 합니다. 기본 클래스 생성자에 인수를 전달할 수 있는 생성자를 선언합니다. 다음 코드는 `InterestEarningAccount`의 생성자를 보여 줍니다.

C#

```
public InterestEarningAccount(string name, decimal initialBalance) :
    base(name, initialBalance)
```

```
{  
}
```

이 새로운 생성자의 매개 변수는 기본 클래스 생성자의 매개 변수 형식 및 이름과 일치합니다. `: base()` 구문을 사용하여 기본 클래스 생성자에 대한 호출을 나타낼 수 있습니다. 일부 클래스는 여러 생성자를 정의하며, 이 구문을 사용하면 호출하는 기본 클래스 생성자를 선택할 수 있습니다. 생성자를 업데이트한 후 각 파생 클래스의 코드를 개발할 수 있습니다. 새 클래스에 대한 요구 사항은 다음과 같이 지정할 수 있습니다.

- 이자 소득 계좌:
  - 월말 잔고의 2%에 해당하는 예금을 얻게 됩니다.
- 신용 한도:
  - 음수의 잔고일 수 있지만 절대값은 대출 한도보다 클 수 없습니다.
  - 월말 잔고가 0이 아닌 경우 매달 이자 비용이 발생합니다.
  - 대출 한도를 초과하는 인출 때마다 수수료가 발생합니다.
- 선물 카드 계좌:
  - 매월 한 번 말일에 지정된 금액으로 계좌를 다시 채울 수 있습니다.

이러한 계좌 유형 세 가지 모두 월말에 발생하는 작업이 있음을 볼 수 있습니다. 하지만 계좌 유형마다 수행하는 작업은 다릅니다. 다형성을 사용하여 이 코드를 구현합니다.

`BankAccount` 클래스에서 단일 `virtual` 메서드를 만듭니다.

C#

```
public virtual void PerformMonthEndTransactions() { }
```

앞의 코드는 `virtual` 키워드를 사용하여 파생 클래스가 다른 구현을 제공할 수 있는 기본 클래스에서 메서드를 선언하는 방법을 보여 줍니다. `virtual` 메서드는 파생 클래스가 다시 구현하도록 선택할 수 있는 메서드입니다. 파생 클래스는 `override` 키워드를 사용하여 새 구현을 정의합니다. 일반적으로 이것을 “기본 클래스 구현 재정의”라고 합니다. `virtual` 키워드는 파생 클래스가 동작을 재정의할 수 있도록 지정합니다. 파생 클래스가 동작을 재정의해야 하는 `abstract` 메서드를 선언할 수도 있습니다. 기본 클래스는 `abstract` 메서드의 구현을 제공하지 않습니다. 다음으로 만든 새로운 두 클래스의 구현을 정의해야 합니다. `InterestEarningAccount`로 시작합니다.

C#

```
public override void PerformMonthEndTransactions()  
{  
    if (Balance > 500m)  
    {  
        decimal interest = Balance * 0.05m;  
        MakeDeposit(interest, DateTime.Now, "apply monthly interest");  
    }  
}
```

```
    }  
}
```

`LineOfCreditAccount`에 다음 코드를 추가합니다. 이 코드는 계좌에서 인출되는 양수의 이자 비용을 계산하기 위해 잔고를 무효화합니다.

C#

```
public override void PerformMonthEndTransactions()  
{  
    if (Balance < 0)  
    {  
        // Negate the balance to get a positive interest charge:  
        decimal interest = -Balance * 0.07m;  
        MakeWithdrawal(interest, DateTime.Now, "Charge monthly interest");  
    }  
}
```

`GiftCardAccount` 클래스가 해당 월말 기능을 구현하려면 두 가지 변경이 필요합니다. 먼저 매월 더할 선택적 금액을 포함하도록 생성자를 수정합니다.

C#

```
private readonly decimal _monthlyDeposit = 0m;  
  
public GiftCardAccount(string name, decimal initialBalance, decimal  
monthlyDeposit = 0) : base(name, initialBalance)  
    => _monthlyDeposit = monthlyDeposit;
```

생성자는 `monthlyDeposit` 값의 기본값을 제공하므로 호출자는 월별 예치금이 없는 0을 생략할 수 있습니다. 다음으로 생성자에서 0이 아닌 값으로 설정된 경우 월별 예치금을 추가하도록 `PerformMonthEndTransactions` 메서드를 재정의합니다.

C#

```
public override void PerformMonthEndTransactions()  
{  
    if (_monthlyDeposit != 0)  
    {  
        MakeDeposit(_monthlyDeposit, DateTime.Now, "Add monthly deposit");  
    }  
}
```

재정의는 생성자에서 설정된 월별 예치금을 적용합니다. `Main` 메서드에 다음 코드를 추가하여 `GiftCardAccount` 및 `InterestEarningAccount`에 대한 이러한 변경을 테스트합니다.

C#

```
var giftCard = new GiftCardAccount("gift card", 100, 50);
giftCard.MakeWithdrawal(20, DateTime.Now, "get expensive coffee");
giftCard.MakeWithdrawal(50, DateTime.Now, "buy groceries");
giftCard.PerformMonthEndTransactions();
// can make additional deposits:
giftCard.MakeDeposit(27.50m, DateTime.Now, "add some additional spending
money");
Console.WriteLine(giftCard.GetAccountHistory());

var savings = new InterestEarningAccount("savings account", 10000);
savings.MakeDeposit(750, DateTime.Now, "save some money");
savings.MakeDeposit(1250, DateTime.Now, "Add more savings");
savings.MakeWithdrawal(250, DateTime.Now, "Needed to pay monthly bills");
savings.PerformMonthEndTransactions();
Console.WriteLine(savings.GetAccountHistory());
```

결과를 확인합니다. 이제 `LineOfCreditAccount`에 대한 유사한 테스트 코드 집합을 추가합니다.

C#

```
var lineOfCredit = new LineOfCreditAccount("line of credit", 0);
// How much is too much to borrow?
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly
advance");
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for
repairs");
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on
repairs");
lineOfCredit.PerformMonthEndTransactions();
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

앞의 코드를 추가하고 프로그램을 실행하면 다음과 같은 오류가 표시됩니다.

콘솔

```
Unhandled exception. System.ArgumentOutOfRangeException: Amount of deposit
must be positive (Parameter 'amount')
   at OOProgramming.BankAccount.MakeDeposit(Decimal amount, DateTime date,
String note) in BankAccount.cs:line 42
   at OOProgramming.BankAccount..ctor(String name, Decimal initialBalance)
in BankAccount.cs:line 31
   at OOProgramming.LineOfCreditAccount..ctor(String name, Decimal
initialBalance) in LineOfCreditAccount.cs:line 9
   at OOProgramming.Program.Main(String[] args) in Program.cs:line 29
```

## ① 참고

실제 출력에는 프로젝트와 함께 폴더의 전체 경로가 포함됩니다. 간단히 하기 위해 폴더 이름이 생략되었습니다. 또한 코드 형식에 따라 줄 번호가 약간 다를 수 있습니다.

`BankAccount`는 초기 잔고가 0보다 커야 한다고 가정하기 때문에 이 코드는 실패합니다. `BankAccount` 클래스에 베이킹된 또 다른 가정은 잔고는 음수가 될 수 없다는 것입니다. 대신 계좌 잔고를 초과하는 인출은 거부됩니다. 두 가지 가정 모두 변경해야 합니다. 신용 한도 계좌는 0에서 시작하며, 일반적으로 음수의 잔고를 갖습니다. 또한 고객이 너무 많은 비용을 빌리는 경우 수수료가 발생합니다. 트랜잭션은 허용되지만 비용이 더 많이 듭니다. 첫 번째 규칙은 최소 잔고를 지정하는 `BankAccount` 생성자에 선택적 인수를 추가하여 구현할 수 있습니다. 기본값은 `0`입니다. 두 번째 규칙에는 파생 클래스가 기본 알고리즘을 수정할 수 있도록 하는 메커니즘이 필요합니다. 어떤 면에서 기본 클래스는 초과 인출이 있을 때 수행해야 하는 작업을 파생 형식에게 '물어봅니다'. 기본 동작은 예외를 `throw`하여 트랜잭션을 거부하는 것입니다.

선택적 `minimumBalance` 매개 변수를 포함하는 두 번째 생성자를 추가하여 시작해 보겠습니다. 이 새 생성자는 기존 생성자가 수행하는 모든 작업을 수행합니다. 또한 최소 잔고 속성을 설정합니다. 기존 생성자의 본문을 복사할 수 있지만 나중에 두 위치가 변경될 수 있습니다. 대신 생성자 연결을 사용하여 한 생성자가 다른 생성자를 호출하도록 할 수 있습니다. 다음 코드는 두 개의 생성자와 새 추가 필드를 보여 줍니다.

C#

```
private readonly decimal _minimumBalance;

public BankAccount(string name, decimal initialBalance) : this(name,
initialBalance, 0) { }

public BankAccount(string name, decimal initialBalance, decimal
minimumBalance)
{
    Number = s_accountNumberSeed.ToString();
    s_accountNumberSeed++;

    Owner = name;
    _minimumBalance = minimumBalance;
    if (initialBalance > 0)
        MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

앞의 코드는 두 가지 새로운 방법을 보여 줍니다. 첫째, `minimumBalance` 필드는 `readonly`로 표시됩니다. 즉, 개체가 생성된 후에는 값을 변경할 수 없습니다. `BankAccount`가 만들

어지면 `minimumBalance`를 변경할 수 없습니다. 둘째, 두 매개 변수를 취하는 생성자는 : `this(name, initialBalance, 0) { }`를 구현으로 사용합니다. `: this()` 식은 매개 변수가 세 개인 다른 생성자를 호출합니다. 이 방법을 사용하면 클라이언트 코드가 여러 생성자 중 하나를 선택할 수 있더라도 개체 초기화에 단일 구현을 사용할 수 있습니다.

이 구현은 초기 잔고가 `0`보다 큰 경우에만 `MakeDeposit`을 호출합니다. 그러면 예치금은 양수여야 한다는 규칙이 유지되지만 신용 계정이 `0`의 잔고로 열립니다.

이제 `BankAccount` 클래스에 최소 잔고에 대한 읽기 전용 필드가 있으므로 마지막 변경은 `MakeWithdrawal` 메서드에서 하드 코드를 `0`에서 `minimumBalance`로 변경하는 것입니다.

C#

```
if (Balance - amount < minimumBalance)
```

`BankAccount` 클래스를 확장한 후 다음 코드에 나온 것처럼 새 기본 생성자를 호출하도록 `LineOfCreditAccount` 생성자를 수정할 수 있습니다.

C#

```
public LineOfCreditAccount(string name, decimal initialBalance, decimal creditLimit) : base(name, initialBalance, -creditLimit)
{
}
```

`LineOfCreditAccount` 생성자는 `minimumBalance` 매개 변수의 의미와 일치하도록 `creditLimit` 매개 변수의 부호를 변경할 수 있습니다.

## 다른 초과 인출 규칙

추가할 마지막 기능을 사용하면 `LineOfCreditAccount`는 트랜잭션을 거부하는 대신 대출 한도 초과에 대해 수수료를 청구할 수 있습니다.

한 가지 방법은 필요한 동작을 구현하는 가상 함수를 정의하는 것입니다. `BankAccount` 클래스는 `MakeWithdrawal` 메서드를 두 개의 메서드로 리팩터링합니다. 새 메서드는 인출로 잔고가 최솟값보다 낮아지면 지정된 작업을 수행합니다. 기존 `MakeWithdrawal` 메서드에는 다음과 같은 코드가 있습니다.

C#

```
public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
```

```

    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of
withdrawal must be positive");
    }
    if (Balance - amount < _minimumBalance)
    {
        throw new InvalidOperationException("Not sufficient funds for this
withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    allTransactions.Add(withdrawal);
}

```

다음 코드로 바꿉니다.

C#

```

public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of
withdrawal must be positive");
    }
    Transaction? overdraftTransaction = CheckWithdrawalLimit(Balance -
amount < _minimumBalance);
    Transaction? withdrawal = new(-amount, date, note);
    _allTransactions.Add(withdrawal);
    if (overdraftTransaction != null)
        _allTransactions.Add(overdraftTransaction);
}

protected virtual Transaction? CheckWithdrawalLimit(bool isOverdrawn)
{
    if (isOverdrawn)
    {
        throw new InvalidOperationException("Not sufficient funds for this
withdrawal");
    }
    else
    {
        return default;
    }
}

```

추가된 메서드는 `protected`로, 파생 클래스에서만 호출할 수 있음을 뜻합니다. 이렇게 선언하면 다른 클라이언트가 메서드를 호출할 수 없습니다. 또한 파생 클래스가 동작을 변경할 수 있도록 `virtual`입니다. 반환 형식은 `Transaction?`입니다. `?` 주석은 메서드가 `null`을 반환할 수 있음을 나타냅니다. 인출 한도를 초과할 때 수수료를 청구하기 위해 `LineOfCreditAccount`에 다음 구현을 추가합니다.

C#

```
protected override Transaction? CheckWithdrawalLimit(bool isOverdrawn) =>
    isOverdrawn
    ? new Transaction(-20, DateTime.Now, "Apply overdraft fee")
    : default;
```

재정의는 계좌에서 초과 인출할 때 수수료 트랜잭션을 반환합니다. 인출이 한도를 초과하지 않으면 메서드는 `null` 트랜잭션을 반환합니다. 이는 수수료가 없음을 나타냅니다.

`Program` 클래스의 `Main` 메서드에 다음 코드를 추가하여 이러한 변경 내용을 테스트합니다.

C#

```
var lineOfCredit = new LineOfCreditAccount("line of credit", 0, 2000);
// How much is too much to borrow?
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly
advance");
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for
repairs");
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on
repairs");
lineOfCredit.PerformMonthEndTransactions();
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

프로그램을 실행하고 결과를 확인합니다.

## 요약

잘 알 수 없는 경우 [GitHub 리포지토리](#)에서 이 자습서의 소스를 확인할 수 있습니다.

이 자습서에서 개체 지향 프로그래밍에 사용되는 다양한 방법을 살펴봤습니다.

- 각 계좌 유형의 클래스를 정의할 때 추상화를 사용했습니다. 이러한 클래스는 해당 계좌 유형의 동작을 설명합니다.
- 각 클래스에서 많은 세부 정보를 `private`으로 유지하는 경우 캡슐화를 사용했습니다.
- `BankAccount` 클래스에서 이미 만든 구현을 활용하여 코드를 저장하는 경우 상속을 사용했습니다.
- 파생 클래스가 해당 계좌 유형의 특정 동작을 만들기 위해 재정의할 수 있는 `virtual` 메서드를 만들 때 다형성을 사용했습니다.

# C# 및 .NET의 상속

아티클 • 2023. 04. 08.

이 자습서에서는 C#의 상속에 대해 소개합니다. 상속은 특정 기능(데이터 및 동작)을 제공하는 기본 클래스를 정의하고 해당 기능을 상속하거나 재정의하는 파생 클래스를 정의할 수 있는 개체 지향 프로그래밍 언어의 기능입니다.

## 사전 요구 사항

- Windows 또는 Mac용 [Visual Studio](#) 를 사용하는 것이 좋습니다. [Visual Studio 다운로드 페이지에서 무료 버전을 다운로드할 수 있습니다](#). Visual Studio에는 .NET SDK가 포함되어 있습니다.
- [Visual Studio Code](#) 편집기를 사용할 수도 있습니다. 최신 [.NET SDK](#) 를 별도로 설치해야 합니다.
- 다른 편집기를 선호하는 경우 최신 [.NET SDK](#) 를 설치해야 합니다.

## 예제 실행

이 자습서의 예제를 만들고 실행하기 위해 명령줄에서 [dotnet](#) 유틸리티를 사용합니다. 각 예제에 대해 다음 단계를 수행합니다.

1. 이 예제를 저장할 디렉터리를 만듭니다.
2. 명령 프롬프트에 [dotnet new console](#)을 입력하여 새로운 .NET Core 프로젝트를 만듭니다.
3. 예제의 코드를 복사한 후 코드 편집기에 붙여 넣습니다.
4. 명령줄에서 [dotnet restore](#) 명령을 입력하여 프로젝트의 종속성을 로드하거나 복원합니다.

,, `dotnet publish` `dotnet build` `dotnet test` `dotnet run`, 및 `dotnet pack` 와 같이 `dotnet new` 복원이 필요한 모든 명령에서 암시적으로 실행되므로 를 실행할 `dotnet restore` 필요가 없습니다. 암시적 복원을 사용하지 않으려면 `--no-restore` 옵션을 사용합니다.

`dotnet restore` 명령은 [Azure DevOps Services의 연속 통합 빌드](#) 또는 복원 발생 시점을 명시적으로 제어해야 하는 빌드 시스템과 같이 명시적으로 복원이 가능한 특정 시나리오에서 여전히 유용합니다.

NuGet 피드를 관리하는 방법에 대한 자세한 내용은 [dotnet restore 설명서](#)를 참조하세요.

5. `dotnet run` 명령을 입력하여 예제를 컴파일하고 실행합니다.

## 배경: 상속이란?

상속은 개체 지향 프로그래밍의 기본적인 특성 중 하나입니다. 부모 클래스의 동작을 다시 사용(상속), 확장 또는 수정하는 자식 클래스를 정의할 수 있습니다. 멤버가 상속되는 클래스를 **기본 클래스**라고 합니다. 기본 클래스의 멤버를 상속하는 클래스를 **파생 클래스**라고 합니다.

C# 및 .NET은 단일 상속만 지원합니다. 즉, 하나의 클래스가 단일 클래스에서만 상속할 수 있습니다. 그러나 상속은 전이적이므로 형식 집합에 대해 상속 계층을 정의할 수 있습니다. 즉, 형식 `D`는 형식 `C`에서 상속할 수 있으며, 이 형식은 `B` 형식에서 상속하고, 이 형식은 기본 클래스 형식 `A`에서 상속합니다. 상속은 전이적이므로 형식 `A`의 멤버를 형식 `D`에서 사용할 수 있습니다.

기본 클래스의 모든 멤버가 파생 클래스에서 상속되는 것은 아닙니다. 다음 멤버는 상속되지 않습니다.

- **정적 생성자**: 클래스의 정적 데이터를 초기화합니다.
- **인스턴스 생성자**: 클래스의 새 인스턴스를 만들기 위해 호출합니다. 각 클래스는 자체 생성자를 정의해야 합니다.
- **종료자**: 클래스의 인스턴스를 삭제하기 위해 런타임의 가비지 수집기에 의해 호출됩니다.

기본 클래스의 다른 모든 멤버는 파생 클래스에서 상속되지만 표시 가능 여부는 해당 액세스 가능성에 따라 달라집니다. 멤버의 액세스 가능성은 다음과 같이 파생 클래스의 표시 여부에 영향을 미칩니다.

- **개인** 멤버는 기본 클래스에 중첩된 파생 클래스에서만 표시됩니다. 그렇지 않으면 파생 클래스에서 표시되지 않습니다. 다음 예제에서 `A.B`는 `A`에서 파생되는 중첩 클래스이고 `c`는 `A`에서 파생됩니다. 프라이빗 `A._value` 필드는 `A.B`에 표시됩니다. 그러나 메서드에서 `c.GetValue` 주석을 제거하고 예제를 컴파일하려고 하면 컴파일러 오류 CS0122가 생성됩니다. "'A.\_value'은 보호 수준으로 인해 액세스할 수 없습니다."

C#

```
public class A
{
```

```

private int _value = 10;

public class B : A
{
    public int GetValue()
    {
        return _value;
    }
}

public class C : A
{
    //    public int GetValue()
    //    {
    //        return _value;
    //    }
}

public class AccessExample
{
    public static void Main(string[] args)
    {
        var b = new A.B();
        Console.WriteLine(b.GetValue());
    }
}

// The example displays the following output:
//      10

```

- **Protected** 멤버는 파생 클래스에서만 표시됩니다.
- **Internal** 멤버는 기본 클래스와 동일한 어셈블리에 있는 파생 클래스에서만 표시됩니다. 기본 클래스와는 다른 어셈블리에 있는 파생 클래스에서는 표시되지 않습니다.
- **Public** 멤버는 파생 클래스에서 표시되고 파생 클래스의 공용 인터페이스에 속합니다. 상속된 **public** 멤버는 파생 클래스에서 정의된 것처럼 호출할 수 있습니다. 다음 예제에서 클래스 **A**는 **Method1**이라는 메서드를 정의하고 클래스 **B**는 클래스 **A**에서 상속합니다. 그런 다음 이 예제에서는 마치 **B**에 대한 인스턴스 메서드인 것처럼 **Method1**을 호출합니다.

C#

```

public class A
{
    public void Method1()
    {
        // Method implementation.
    }
}

```

```

}

public class B : A
{ }

public class Example
{
    public static void Main()
    {
        B b = new ();
        b.Method1();
    }
}

```

파생 클래스는 대체 구현을 제공하여 상속된 멤버를 재정의할 수도 있습니다. 멤버를 재정의하기 위해서는 기본 클래스의 멤버가 `virtual` 키워드로 표시되어야 합니다. 기본적으로 기본 클래스 멤버는 `virtual`로 표시되지 않으며 재정의할 수 없습니다. 다음 예제와 같이 가상이 아닌 멤버를 재정의하려고 하면 컴파일러 오류 CS0506이 생성됩니다. "<멤버>는 가상, 추상 또는 재정의로 표시되지 않으므로 상속된 멤버 <member>를 재정의할 수 없습니다."

C#

```

public class A
{
    public void Method1()
    {
        // Do something.
    }
}

public class B : A
{
    public override void Method1() // Generates CS0506.
    {
        // Do something else.
    }
}

```

일부 경우에 파생 클래스는 기본 클래스 구현을 반드시 재정의해야 합니다. `abstract` 키워드로 표시된 기본 클래스 멤버의 경우 파생 클래스에서 재정의해야 합니다. 다음 예제를 컴파일하려고 하면 클래스 `B`가 `A.Method1`에 대한 구현을 제공하지 않으므로 컴파일러 오류 CS0534, "<class>는 상속된 추상 멤버 <member>를 구현하지 않습니다."가 표시됩니다.

C#

```
public abstract class A
{
    public abstract void Method1();
}

public class B : A // Generates CS0534.
{
    public void Method3()
    {
        // Do something.
    }
}
```

상속은 클래스 및 인터페이스에만 적용됩니다. 다른 형식 범주(구조체, 대리자 및 열거형)은 상속을 지원하지 않습니다. 이러한 규칙 때문에 다음 예제와 같은 코드를 컴파일하려고 시도하면 컴파일러 오류 CS0527이 생성됩니다. "인터페이스 목록에서 'ValueType' 형식은 인터페이스가 아닙니다." 오류 메시지는 구조체가 구현하는 인터페이스를 정의할 수 있지만 상속은 지원되지 않음을 나타냅니다.

C#

```
public struct ValueStructure : ValueType // Generates CS0527.
{ }
```

## 암시적 상속

단일 상속을 통해 상속할 수 있는 형식을 제외하고, .NET 형식 시스템의 모든 형식은 [Object](#) 또는 여기에서 파생된 형식에서 암시적으로 상속합니다. [Object](#)의 공통 기능은 모든 형식에서 사용할 수 있습니다.

암시적 상속의 의미를 살펴보기 위해 빈 클래스 정의에 해당하는 새 클래스 `SimpleClass`를 정의해 보겠습니다.

C#

```
public class SimpleClass
{ }
```

그런 다음, 리플렉션(형식의 메타데이터를 검사하여 해당 형식에 대한 정보를 가져올 수 있음)을 사용하여 `SimpleClass` 형식에 속한 멤버의 목록을 가져올 수 있습니다.

`SimpleClass` 클래스에 어떤 멤버도 정의되지 않은 경우에도 예제의 출력에는 실제로 9개의 멤버가 있는 것으로 나타납니다. 이러한 멤버 중 하나는 C# 컴파일러에서 `SimpleClass`

형식에 대해 자동으로 제공하는 매개 변수가 없는(또는 기본) 생성자입니다. 나머지 8개 멤버는 .NET 형식 시스템의 모든 클래스 및 인터페이스가 마지막에 암시적으로 상속하는 형식인 [Object](#)의 멤버입니다.

C#

```
using System.Reflection;

public class SimpleClassExample
{
    public static void Main()
    {
        Type t = typeof(SimpleClass);
        BindingFlags flags = BindingFlags.Instance | BindingFlags.Static |
        BindingFlags.Public |
                BindingFlags.NonPublic |
        BindingFlags.FlattenHierarchy;
        MemberInfo[] members = t.GetMembers(flags);
        Console.WriteLine($"Type {t.Name} has {members.Length} members: ");
        foreach (MemberInfo member in members)
        {
            string access = "";
            string stat = "";
            var method = member as MethodBase;
            if (method != null)
            {
                if (method.IsPublic)
                    access = " Public";
                else if (method.IsPrivate)
                    access = " Private";
                else if (method.IsFamily)
                    access = " Protected";
                else if (method.IsAssembly)
                    access = " Internal";
                else if (method.IsFamilyOrAssembly)
                    access = " Protected Internal ";
                if (method.IsStatic)
                    stat = " Static";
            }
            string output = $"{member.Name} ({member.MemberType}): {access}
{stat}, Declared by {member.DeclaringType}";
            Console.WriteLine(output);
        }
    }
}

// The example displays the following output:
// Type SimpleClass has 9 members:
// ToString (Method): Public, Declared by System.Object
// Equals (Method): Public, Declared by System.Object
// Equals (Method): Public Static, Declared by System.Object
// ReferenceEquals (Method): Public Static, Declared by System.Object
// GetHashCode (Method): Public, Declared by System.Object
// GetType (Method): Public, Declared by System.Object
```

```
// Finalize (Method): Internal, Declared by System.Object
// MemberwiseClone (Method): Internal, Declared by System.Object
// .ctor (Constructor): Public, Declared by SimpleClass
```

`Object` 클래스에서 암시적으로 상속되므로 다음 메서드를 `SimpleClass` 클래스에서 사용할 수 있습니다.

- 공용 `ToString` 메서드: `SimpleClass` 개체를 해당 문자열 표현으로 변환하고 정규화된 형식 이름을 반환합니다. 이 경우 `ToString` 메서드는 문자열 "SimpleClass"를 반환합니다.
- 두 개체가 같은지를 테스트하는 세 가지 메서드: 공용 인스턴스 `Equals(Object)` 메서드, 공용 정적 `Equals(Object, Object)` 메서드, 공용 정적 `ReferenceEquals(Object, Object)` 메서드. 기본적으로 이러한 메서드는 참조 같음을 테스트합니다. 즉, 두 개체 변수가 같으려면 같은 개체를 참조해야 합니다.
- 공용 `GetHashCode` 메서드: 형식의 인스턴스가 해시된 컬렉션에 사용될 수 있도록 하는 값을 계산합니다.
- 공용 `GetType` 메서드: `SimpleClass` 형식을 나타내는 `Type` 개체를 반환합니다.
- 보호된 `Finalize` 메서드: 개체의 메모리를 가비지 수집기에 의해 회수되기 전에 관리되지 않는 리소스를 해제하도록 설계되었습니다.
- 보호된 `MemberwiseClone` 메서드: 현재 개체의 단순 복제를 만듭니다.

암시적 상속으로 인해 `SimpleClass` 개체에서 상속된 모든 멤버를 실제로 `SimpleClass` 클래스에 정의된 멤버인 것처럼 호출할 수 있습니다. 예를 들어 다음 예제에서는 `SimpleClass` 가 `Object`에서 상속하는 `SimpleClass.ToString` 메서드를 호출합니다.

C#

```
public class EmptyClass
{ }

public class ClassNameExample
{
    public static void Main()
    {
        EmptyClass sc = new();
        Console.WriteLine(sc.ToString());
    }
}
// The example displays the following output:
//      EmptyClass
```

다음 표에는 C#으로 만들 수 있는 형식 및 이러한 형식이 암시적으로 상속하는 형식 범주가 나와 있습니다. 각 기본 형식은 암시적으로 파생된 형식에 대한 상속을 통해 다른 멤버 집합을 사용할 수 있게 합니다.

형식 범주	다음에서 암시적으로 상속
class	Object
struct	ValueType, Object
enum	Enum, ValueType, Object
대리자(delegate)	MulticastDelegate, Delegate, Object

## 상속 및 "~이다(is a)" 관계

일반적으로 상속은 기본 클래스와 하나 이상의 파생 클래스 간 "~이다(is a)" 관계를 나타내는 데 사용됩니다. 여기서 파생 클래스는 기본 클래스의 특수화된 버전입니다. 즉, 파생 클래스는 기본 클래스의 한 종류입니다. 예를 들어 `Publication` 클래스는 임의 종류의 출판물을 나타내고 `Book` 및 `Magazine` 클래스는 특정 유형의 출판물을 나타냅니다.

### ① 참고

클래스 또는 구조체는 하나 이상의 인터페이스를 구현할 수 있습니다. 인터페이스 구현은 종종 단일 상속을 위한 해결 방법 또는 구조체에 상속을 사용하는 방법으로 제공되지만, 인터페이스 및 해당 구현 형식 사이에서 상속과는 다른 관계("~할 수 있다(can do)" 관계)를 나타내는 데 사용됩니다. 인터페이스는 해당 인터페이스를 구현 형식에서 사용 가능하게 만드는 기능 일부(예: 같은지 테스트하는 기능, 개체를 비교하거나 정렬하는 기능 또는 문화권별 구문 분석 및 서식 지정을 지원하는 기능)를 정의합니다.

"~이다(is a)"는 형식과 해당 형식의 특정 인스턴스화 사이의 관계를 나타내기도 합니다. 다음 예제에서 `Automobile`은 세 가지 고유한 읽기 전용 속성, 즉 자동차의 제조업체인 `Make`, 자동차의 종류인 `Model`, 제조 연도인 `Year`를 갖는 클래스입니다. 또한 `Automobile` 클래스에는 해당 인수가 속성 값에 할당된 생성자가 있으며, `Object.ToString` 메서드를 재정의하여 `Automobile` 클래스가 아닌 `Automobile` 인스턴스를 고유하게 식별하는 문자열을 생성합니다.

C#

```
public class Automobile
{
    public Automobile(string make, string model, int year)
```

```

    {
        if (make == null)
            throw new ArgumentNullException(nameof(make), "The make cannot
be null.");
        else if (string.IsNullOrWhiteSpace(make))
            throw new ArgumentException("make cannot be an empty string or
have space characters only.");
        Make = make;

        if (model == null)
            throw new ArgumentNullException(nameof(model), "The model cannot
be null.");
        else if (string.IsNullOrWhiteSpace(model))
            throw new ArgumentException("model cannot be an empty string or
have space characters only.");
        Model = model;

        if (year < 1857 || year > DateTime.Now.Year + 2)
            throw new ArgumentException("The year is out of range.");
        Year = year;
    }

    public string Make { get; }

    public string Model { get; }

    public int Year { get; }

    public override string ToString() => $"{Year} {Make} {Model}";
}

```

이 경우 특정 자동차 제조업체 및 모델을 나타내기 위해 상속을 사용하지 않아야 합니다. 예를 들어 Packard Motor Car Company에서 제조한 자동차임을 나타내기 위해 `Packard` 형식을 정의할 필요가 없습니다. 대신, 다음 예제와 같이 해당 클래스 생성자에 적절한 값을 사용하여 `Automobile` 객체를 만들어 이러한 속성을 나타낼 수 있습니다.

C#

```

using System;

public class Example
{
    public static void Main()
    {
        var packard = new Automobile("Packard", "Custom Eight", 1948);
        Console.WriteLine(packard);
    }
}

// The example displays the following output:
//      1948 Packard Custom Eight

```

상속을 기준으로 하는 ~이다(is a) 관계는 기본 클래스와 기본 클래스에 추가 멤버를 더하거나 기본 클래스에 없는 추가 기능을 필요로 하는 파생 클래스에 가장 잘 적용됩니다.

## 기본 클래스 및 파생 클래스 디자인

기본 클래스와 해당 파생 클래스를 디자인하는 프로세스를 살펴보겠습니다. 이 섹션에서는 책, 잡지, 신문, 저널, 기사 등과 같은 모든 종류의 출판물을 나타내는 `Publication` 기본 클래스를 정의합니다. 또한 `Publication` 클래스에서 파생되는 `Book` 클래스도 정의합니다. `Magazine`, `Journal`, `Newspaper` 및 `Article`과 같은 다른 파생 클래스를 정의하도록 예제를 쉽게 확장할 수 있습니다.

### 기본 게시 클래스

`Publication` 클래스를 디자인할 때 결정해야 하는 몇 가지 디자인은 다음과 같습니다.

- 기본 `Publication` 클래스에 포함할 멤버, `Publication` 멤버에서 메서드 구현을 제공하는지 여부 또는 `Publication`이 해당 파생 클래스에 대한 템플릿으로 사용되는 추상 기본 클래스인지 여부

이 경우 `Publication` 클래스는 메서드 구현을 제공합니다. [추상 기본 클래스 및 파생 클래스 디자인](#) 섹션에는 추상 기본 클래스를 사용하여 파생 클래스가 재정의해야 하는 메서드를 정의하는 예제가 포함되어 있습니다. 파생 클래스는 파생 형식에 적합한 모든 구현을 자유롭게 제공할 수 있습니다.

코드를 다시 사용하는 기능(즉, 여러 파생 클래스가 기본 클래스 메서드의 선언 및 구현을 공유하며 재정의할 필요가 없음)은 비추상 기본 클래스의 장점입니다. 따라서 일부 또는 대부분의 특수화된 `Publication` 형식에서 해당 코드를 공유할 가능성이 높은 경우 `Publication`에 멤버를 추가해야 합니다. 기본 클래스 구현을 효율적으로 제공하지 못하면 기본 클래스에서 단일 구현이 아니라 파생 클래스에서 거의 동일한 멤버 구현을 제공해야 합니다. 여러 위치에서 중복된 코드를 유지해야 하면 버그가 발생하기 쉬워집니다.

코드 재사용을 최대화하고 논리적이고 직관적인 상속 계층 구조를 만들려면 모두 또는 대부분의 출판물에 공통되는 데이터 및 기능만 `Publication` 클래스에 포함해야 합니다. 그러면 파생 클래스는 나타내는 특정 종류를 출판물에 고유한 멤버를 구현합니다.

- 클래스 계층 구조 확장 범위. 단순히 하나의 기본 클래스와 하나 이상의 파생 클래스가 아닌 세 개 이상의 클래스로 구성된 계층 구조를 개발하려고 하나요? 예를 들어 `Publication`은 `Magazine`, `Journal` 및 `Newspaper`의 기본 클래스인 `Periodical`의 기본 클래스일 수 있습니다.

예제에서는 `Publication` 클래스와 `Book` 파생 클래스가 각각 하나씩 구성된 작은 계층 구조를 사용합니다. 이 예제는 쉽게 확장하여 `Publication`에서 파생되는 많은 수의 추가 클래스(예: `Magazine` 및 `Article`)를 만들 수 있습니다.

- 기본 클래스의 인스턴스화가 타당한지 여부. 타당하지 않은 경우 `abstract` 키워드를 클래스에 적용해야 합니다. 그렇지 않으면 해당 클래스 생성자를 호출하여 `Publication` 클래스를 인스턴스화할 수 있습니다. 클래스 생성자에 대한 직접 호출을 통해 키워드(keyword) 표시된 `abstract` 클래스를 인스턴스화하려고 하면 C# 컴파일러는 "추상 클래스 또는 인터페이스의 instance 만들 수 없습니다."라는 오류 CS0144를 생성합니다. 리플렉션을 사용하여 클래스를 인스턴스화하려고 하면 리플렉션 메서드가 `MemberAccessException`을 `throw`합니다.

기본적으로 기본 클래스는 해당 클래스 생성자를 호출하여 인스턴스화할 수 있습니다. 클래스 생성자를 명시적으로 정의할 필요는 없습니다. 생성자가 기본 클래스의 소스 코드에 없는 경우 C# 컴파일러는 기본(매개 변수 없는) 생성자를 자동으로 제공합니다.

예를 들어 `Publication` 클래스를 인스턴스화할 수 없도록 `abstract`로 표시합니다. `abstract` 메서드가 없는 `abstract` 클래스는 이 클래스가 몇 가지 구체적인 클래스(예: `Book`, `Journal`) 간에 공유되는 추상 개념을 나타낸다는 것을 나타냅니다.

- 파생 클래스에서 특정 멤버의 기본 클래스 구현을 상속해야 하는지 여부, 파생 클래스에 기본 클래스 구현을 재정의할 수 있는 옵션이 있는지 여부 또는 파생 클래스에서 구현을 제공해야 하는지 여부. `abstract` 키워드를 사용하여 파생 클래스에서 구현을 제공하도록 적용합니다. `virtual` 키워드를 사용하여 파생 클래스에서 기본 클래스 메서드를 재정의할 수 있도록 허용합니다. 기본적으로 기본 클래스에 정의된 메서드는 재정의 가능하지 않습니다.

`Publication` 클래스에는 `abstract` 메서드가 없지만 클래스 자체는 `abstract`입니다.

- 파생 클래스가 상속 계층 구조의 최종 클래스를 나타내고 자체적으로 추가 파생 클래스에 대한 기본 클래스로 사용될 수 없는지 여부. 기본적으로 모든 클래스는 기본 클래스로 사용될 수 있습니다. `sealed` 키워드를 적용하여 클래스가 추가 클래스에 대한 기본 클래스로 사용될 수 없음을 나타낼 수 있습니다. 봉인된 클래스 생성 컴파일러 오류 CS0509에서 파생하려고 하면 "sealed typeName<>에서 파생될 수 없습니다."

예를 들어 파생 클래스를 `sealed`로 표시합니다.

다음 예제에서는 `Publication` 클래스에 대한 소스 코드와 `Publication.PublicationType` 속성이 반환하는 `PublicationType` 열거형을 보여 줍니다. `Object`에서 상속하는 멤버 외에

`Publication` 클래스는 다음과 같은 고유한 멤버 및 멤버 재정의를 정의합니다.

C#

```
public enum PublicationType { Misc, Book, Magazine, Article };

public abstract class Publication
{
    private bool _published = false;
    private DateTime _datePublished;
    private int _totalPages;

    public Publication(string title, string publisher, PublicationType type)
    {
        if (string.IsNullOrWhiteSpace(publisher))
            throw new ArgumentException("The publisher is required.");
        Publisher = publisher;

        if (string.IsNullOrWhiteSpace(title))
            throw new ArgumentException("The title is required.");
        Title = title;

        Type = type;
    }

    public string Publisher { get; }

    public string Title { get; }

    public PublicationType Type { get; }

    public string? CopyrightName { get; private set; }

    public int CopyrightDate { get; private set; }

    public int Pages
    {
        get { return _totalPages; }
        set
        {
            if (value <= 0)
                throw new ArgumentOutOfRangeException(nameof(value), "The
number of pages cannot be zero or negative.");
            _totalPages = value;
        }
    }

    public string GetPublicationDate()
    {
        if (!_published)
            return "NYP";
        else
            return _datePublished.ToString("d");
    }
}
```

```

    }

    public void Publish(DateTime datePublished)
    {
        _published = true;
        _datePublished = datePublished;
    }

    public void Copyright(string copyrightName, int copyrightDate)
    {
        if (string.IsNullOrWhiteSpace(copyrightName))
            throw new ArgumentException("The name of the copyright holder is required.");
        CopyrightName = copyrightName;

        int currentYear = DateTime.Now.Year;
        if (copyrightDate < currentYear - 10 || copyrightDate > currentYear + 2)
            throw new ArgumentOutOfRangeException($"The copyright year must be between {currentYear - 10} and {currentYear + 1}");
        CopyrightDate = copyrightDate;
    }

    public override string ToString() => Title;
}

```

- 생성자

`Publication` 클래스는 `abstract` 이므로 다음 예제와 같은 코드에서 직접 인스턴스화할 수 없습니다.

C#

```

var publication = new Publication("Tiddlywinks for Experts", "Fun and Games",
                                  PublicationType.Book);

```

그러나 `Book` 클래스에 대한 소스 코드가 나타내는 것처럼 해당 인스턴스 생성자를 파생 클래스 생성자에서 직접 호출할 수 있습니다.

- 출판물과 관련된 두 가지 속성

`Title`은 `Publication` 생성자를 호출하여 해당 값이 제공되는 읽기 전용 `String` 속성입니다.

`Pages`는 출판물에 포함된 총 페이지 수를 나타내는 읽기/쓰기 `Int32` 속성입니다. 값은 `totalPages`라는 private 필드에 저장됩니다. 값은 양수여야 하며 양수가 아니면 `ArgumentOutOfRangeException`이 throw됩니다.

- 출판사 관련 멤버

두 개의 읽기 전용 속성 `Publisher` 및 `Type`입니다. 해당 값은 원래 `Publication` 클래스 생성자를 호출하여 제공됩니다.

- 출판 관련 멤버

두 가지 메서드 `Publish` 및 `GetPublicationDate`가 출판일을 설정하고 반환합니다. 메서드는 `Publish` 호출될 때 프라이빗 `published` 플래그를 `true`로 설정하고 전달된 날짜를 프라이빗 `datePublished` 필드에 인수로 할당합니다. `GetPublicationDate` 메서드는 `published` 플래그가 `false`이면 문자열 "NYP"를 반환하고, `true`이면 `datePublished` 필드 값을 반환합니다.

- 저작권 관련 멤버

`Copyright` 메서드는 저작권 소유자의 이름과 저작권 연도를 인수로 사용한 후 `CopyrightName` 및 `CopyrightDate` 속성에 할당합니다.

- `ToString` 메서드 재정의

형식이 `Object.ToString` 메서드를 재정의하지 않으면 한 인스턴스를 다른 인스턴스와 구분하는 데 별로 도움이 되지 않는 형식의 정규화된 이름을 반환합니다.

`Publication` 클래스는 `Object.ToString`을 재정의하여 `Title` 속성의 값을 반환합니다.

다음 그림에서는 기본 `Publication` 클래스와 암시적으로 상속된 해당 `Object` 클래스 간의 관계를 보여 줍니다.

Object	Publication
Equals(Object)	Equals(Object)
Equals(Object, Object)	Equals(Object, Object)
Finalize()	Finalize()
GetHashCode()	GetHashCode()
GetType()	GetType()
MemberwiseClone()	MemberwiseClone()
ReferenceEquals()	ReferenceEquals()
ToString()	ToString()
#ctor()	#ctor(String, String, PublicationType)
<b>Key</b>	
Unique member	
Inherited member	
Overridden member	

## Book 클래스

Book 클래스는 책을 특수한 출판문 형식으로 나타냅니다. 다음 예제에서는 Book 클래스에 대한 소스 코드를 보여 줍니다.

C#

```
using System;

public sealed class Book : Publication
{
    public Book(string title, string author, string publisher) :
        this(title, string.Empty, author, publisher)
    { }

    public Book(string title, string isbn, string author, string publisher)
        : base(title, publisher, PublicationType.Book)
    {
        // isbn argument must be a 10- or 13-character numeric string
        // without "-" characters.
        // We could also determine whether the ISBN is valid by comparing
        // its checksum digit
        // with a computed checksum.
```

```
//  
if (!string.IsNullOrEmpty(isbn))  
{  
    // Determine if ISBN length is correct.  
    if (!(isbn.Length == 10 || isbn.Length == 13))  
        throw new ArgumentException("The ISBN must be a 10- or 13-  
character numeric string.");  
    if (!ulong.TryParse(isbn, out _))  
        throw new ArgumentException("The ISBN can consist of numeric  
characters only.");  
}  
ISBN = isbn;  
  
Author = author;  
}  
  
public string ISBN { get; }  
  
public string Author { get; }  
  
public decimal Price { get; private set; }  
  
// A three-digit ISO currency symbol.  
public string? Currency { get; private set; }  
  
// Returns the old price, and sets a new price.  
public decimal SetPrice(decimal price, string currency)  
{  
    if (price < 0)  
        throw new ArgumentOutOfRangeException(nameof(price), "The price  
cannot be negative.");  
    decimal oldValue = Price;  
    Price = price;  
  
    if (currency.Length != 3)  
        throw new ArgumentException("The ISO currency symbol is a 3-  
character string.");  
    Currency = currency;  
  
    return oldValue;  
}  
  
public override bool Equals(object? obj)  
{  
    if (obj is not Book book)  
        return false;  
    else  
        return ISBN == book.ISBN;  
}  
  
public override int GetHashCode() => ISBN.GetHashCode();  
  
public override string ToString() => $"{{(string.IsNullOrEmpty(Author) ?  
"" : Author + ", "){Title}}};  
}
```

`Publication`에서 상속하는 멤버 외에 `Book` 클래스는 다음과 같은 고유한 멤버 및 멤버 재정의를 정의합니다.

- 2개의 생성자

두 `Book` 생성자는 3가지 공용 매개 변수를 공유합니다. 두 `title` 및 `publisher`는 `Publication` 생성자의 매개 변수에 해당합니다. 세 번째는 변경할 수 없는 공용 `Author` 속성에 저장되는 `author`입니다. 한 생성자에는 `ISBN` `auto` 속성에 저장되는 `isbn` 매개 변수가 포함됩니다.

첫 번째 생성자는 `this` 키워드를 사용하여 다른 생성자를 호출합니다. 생성자 연결 (chaining)은 생성자를 정의하는 일반적인 패턴입니다. 가장 많은 수의 매개 변수를 사용하여 생성자를 호출하면 더 적은 수의 매개 변수를 사용하는 생성자가 기본값을 제공합니다.

두 번째 생성자는 `base` 키워드를 사용하여 기본 클래스 생성자에 제목 및 출판사 이름을 전달합니다. 소스 코드에서 기본 클래스 생성자를 명시적으로 호출하지 않으면 C# 컴파일러는 기본 클래스의 기본 생성자 또는 매개 변수 없는 생성자에 대한 호출을 자동으로 제공합니다.

- 읽기 전용 `ISBN` 속성: 고유한 10 또는 13자리 숫자인 `Book` 개체의 국제 표준 도서 번호를 반환합니다. `ISBN`은 `Book` 생성자 중 하나에 인수로 제공됩니다. `ISBN`은 컴파일러에서 자동 생성되는 `private` 지원 필드에 저장됩니다.
- 읽기 전용 `Author` 속성. 저자 이름은 두 `Book` 생성자의 인수로 제공되고 속성에 저장됩니다.
- 두 개의 읽기 전용 가격 관련 속성 `Price` 및 `Currency`. 해당 값은 `SetPrice` 메서드 호출에 인수로 제공됩니다. `Currency` 속성은 세 자리 ISO 통화 기호입니다(예: 미국 달러의 경우 USD). ISO 통화 기호는 `ISOCurrencySymbol` 속성에서 검색할 수 있습니다. 이러한 두 속성은 모두 외부적으로 읽기 전용이지만 둘 다 `Book` 클래스의 코드로 설정할 수 있습니다.
- `SetPrice` 메서드는 `Price` 및 `Currency` 속성의 값을 설정합니다. 이러한 값은 동일한 해당 속성으로 반환됩니다.
- `ToString` 메서드(`Publication`에서 상속), `Object.Equals(Object)` 및 `GetHashCode` 메서드(`Object`에서 상속)에 대해 재정의합니다.

재정의되지 않으면 `Object.Equals(Object)` 메서드는 참조 같음 여부를 테스트합니다. 즉, 두 개체 변수는 같은 개체를 참조하는 경우 동일한 것으로 간주됩니다. 반면에

`Book` 클래스에서 두 개의 `Book` 개체에 동일한 ISBN이 있는 경우 이 두 개체는 동일해야 합니다.

`Object.Equals(Object)` 메서드를 재정의할 경우 런타임이 효율적인 검색을 위해 해시된 컬렉션에 항목을 저장하는 데 사용하는 값을 반환하는 `GetHashCode` 메서드도 재정의해야 합니다. 해시 코드는 다음 테스트와 일치하는 값을 반환해야 합니다. 두 `Book` 개체의 ISBN 속성이 같으면 `true`를 반환하도록 `Object.Equals(Object)`를 재정의했으므로 `ISBN` 속성에서 반환된 문자열의 `GetHashCode` 메서드를 호출하여 계산된 해시 코드를 반환합니다.

다음 그림에서는 `Book` 클래스와 해당 기본 클래스인 `Publication` 클래스 간 관계를 보여줍니다.

## Publication

Equals(Object)
Equals(Object, Object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals(Object, Object)
ToString()
#ctor(String, String, PublicationType)
PublicationType
Publisher
Title
CopyrightDate
CopyrightName
Pages
Copyright()
GetPublicationDate()
Publish()

## Book

Equals(Object)
Equals(Object, Object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals(Object, Object)
ToString()
#ctor(String, String, String)
#ctor(String, String, String, String)
PublicationType
Publisher
Author
Title
CopyrightDate
CopyrightName
ISBN
Pages
Price
Currency
Copyright()
GetPublicationDate()
Publish()
SetPrice()

### Key

Unique member	
Inherited member	
Overridden member	

이제 다음 예제와 같이 `Book` 개체를 인스턴스화하고, 고유 멤버 및 상속된 멤버를 모두 호출하고, `Publication` 형식 또는 `Book` 형식의 매개 변수가 필요한 메서드에 인수로 전달할 수 있습니다.

C#

```
public class ClassExample
{
    public static void Main()
    {
        var book = new Book("The Tempest", "0971655819", "Shakespeare,
William",
```

```

        "Public Domain Press");
ShowPublicationInfo(book);
book.Publish(new DateTime(2016, 8, 18));
ShowPublicationInfo(book);

    var book2 = new Book("The Tempest", "Classic Works Press",
"Shakespeare, William");
Console.WriteLine($"{book.Title} and {book2.Title} are the same
publication: " +
    $"{{((Publication)book).Equals(book2)}");
}

public static void ShowPublicationInfo(Publication pub)
{
    string pubDate = pub.GetPublicationDate();
    Console.WriteLine($"{pub.Title}, " +
        $"{(pubDate == "NYP" ? "Not Yet Published" : "published on
" + pubDate):d} by {pub.Publisher}");
}
// The example displays the following output:
//      The Tempest, Not Yet Published by Public Domain Press
//      The Tempest, published on 8/18/2016 by Public Domain Press
//      The Tempest and The Tempest are the same publication: False

```

## 추상 기본 클래스 및 파생 클래스 디자인

앞의 예제에서는 파생 클래스에서 코드를 공유할 수 있도록 여러 메서드 구현을 제공하는 기본 클래스를 정의했습니다. 그러나 대부분의 경우 기본 클래스는 구현을 제공할 것으로 예상되지 않습니다. 대신, 기본 클래스는 추상 메서드를 선언하는 추상 클래스이며, 각 파생 클래스에서 구현해야 하는 멤버를 정의하는 템플릿으로 사용됩니다. 일반적으로 추상 기본 클래스에서 각 파생 형식의 구현은 해당 형식에 고유합니다. 클래스에서 출판물에 공통된 기능의 구현을 제공했지만, `Publication` 개체를 인스턴스화하는 것은 의미가 없으므로 클래스를 `abstract` 키워드로 표시했습니다.

예를 들어 닫힌 2차원 기하 도형 각각에 2개의 속성, 즉 도형의 내부 크기를 나타내는 `area` 속성과 도형 가장자리의 거리를 나타내는 `perimeter` 속성이 포함되어 있습니다. 그러나 이러한 속성이 계산되는 방식은 전적으로 도형에 따라 결정됩니다. 예를 들어 원의 경계(또는 둘레)를 계산하는 수식은 정사각형의 수식과 다릅니다. `Shape` 클래스는 `abstract` 메서드가 있는 `abstract` 클래스입니다. 이는 파생 클래스에서 동일한 기능을 공유한다고 나타내지만, 이러한 파생 클래스는 해당 기능을 다르게 구현합니다.

다음 예제에서는 두 속성 `Area` 및 `Perimeter`를 정의하는 `Shape`라는 추상 기본 클래스를 정의합니다. 클래스를 `abstract` 키워드로 표시하는 것 외에도, 각 인스턴스 멤버도 `abstract` 키워드로 표시됩니다. 이 경우 `Shape` 도 정규화된 이름은 아닌 형식의 이름을 반환하도록 `Object.ToString` 메서드를 재정의합니다. 아울러 두 정적 멤버 `GetArea` 및

`GetPerimeter`를 정의합니다. 이러한 정적 멤버는 호출자가 파생 클래스 인스턴스의 면적 및 둘레를 쉽게 검색할 수 있도록 합니다. 파생 클래스의 인스턴스를 이러한 메서드 중 하나에 전달하면 런타임에서 파생 클래스의 메서드 재정의를 호출합니다.

C#

```
public abstract class Shape
{
    public abstract double Area { get; }

    public abstract double Perimeter { get; }

    public override string ToString() => GetType().Name;

    public static double GetArea(Shape shape) => shape.Area;

    public static double GetPerimeter(Shape shape) => shape.Perimeter;
}
```

그러면 `Shape`에서 특정 도형을 나타내는 일부 클래스를 파생시킬 수 있습니다. 다음 예제에서는 3개의 클래스인 `Square`, `Rectangle` 및 `Circle`을 정의합니다. 각각은 해당 특정 도형에 고유한 수식을 사용하여 면적 및 둘레를 컴퓨팅합니다. 일부 파생 클래스는 나타내는 도형마다 고유한 `Rectangle.Diagonal` 및 `Circle.Diameter`와 같은 속성도 정의합니다.

C#

```
using System;

public class Square : Shape
{
    public Square(double length)
    {
        Side = length;
    }

    public double Side { get; }

    public override double Area => Math.Pow(Side, 2);

    public override double Perimeter => Side * 4;

    public double Diagonal => Math.Round(Math.Sqrt(2) * Side, 2);
}

public class Rectangle : Shape
{
    public Rectangle(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public double Length { get; }

    public double Width { get; }

    public double Area => Length * Width;

    public double Perimeter => 2 * (Length + Width);

    public double Diagonal => Math.Round(Math.Sqrt(Length * Length + Width * Width), 2);
}
```

```

    }

    public double Length { get; }

    public double Width { get; }

    public override double Area => Length * Width;

    public override double Perimeter => 2 * Length + 2 * Width;

    public bool IsSquare() => Length == Width;

    public double Diagonal => Math.Round(Math.Sqrt(Math.Pow(Length, 2) +
Math.Pow(Width, 2)), 2);
}

public class Circle : Shape
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public override double Area => Math.Round(Math.PI * Math.Pow(Radius, 2),
2);

    public override double Perimeter => Math.Round(Math.PI * 2 * Radius, 2);

    // Define a circumference, since it's the more familiar term.
    public double Circumference => Perimeter;

    public double Radius { get; }

    public double Diameter => Radius * 2;
}

```

다음 예제에서는 `Shape`에서 파생된 개체를 사용합니다. 또한 `Shape`에서 파생된 개체의 배열을 인스턴스화하고 반환 `Shape` 속성 값을 래핑하는 `Shape` 클래스의 정적 메서드를 호출합니다. 런타임에서는 파생 형식의 재정의된 속성에서 값을 검색합니다. 또한 이 예제에서는 배열의 각 `Shape` 개체를 해상 파생 형식으로 캐스팅하고, 캐스팅이 성공하면 `Shape`의 해당 특정 하위 클래스 속성을 검색합니다.

C#

```

using System;

public class Example
{
    public static void Main()
    {
        Shape[] shapes = { new Rectangle(10, 12), new Square(5),

```

```
        new Circle(3) };
foreach (Shape shape in shapes)
{
    Console.WriteLine($"{shape}: area, {Shape.GetArea(shape)}; " +
                      $"perimeter, {Shape.GetPerimeter(shape)}");
    if (shape is Rectangle rect)
    {
        Console.WriteLine($"    Is Square: {rect.IsSquare()}, 
Diagonal: {rect.Diagonal}");
        continue;
    }
    if (shape is Square sq)
    {
        Console.WriteLine($"    Diagonal: {sq.Diagonal}");
        continue;
    }
}
// The example displays the following output:
//      Rectangle: area, 120; perimeter, 44
//      Is Square: False, Diagonal: 15.62
//      Square: area, 25; perimeter, 20
//      Diagonal: 7.07
//      Circle: area, 28.27; perimeter, 18.85
```

# 패턴 일치와 is 및 as 연산자를 사용하여 안전하게 캐스트하는 방법

아티클 • 2024. 02. 23.

개체는 다형성이기 때문에 기본 클래스 형식의 변수에 파생 형식이 포함될 수 있습니다. 파생 형식의 인스턴스 멤버에 액세스하려면 값을 파생 형식으로 다시 **캐스팅**해야 합니다. 그러나 캐스트는 `InvalidCastException`이 throw될 위험을 생성합니다. C#은 성공하는 경우에만 조건부로 캐스트를 수행하는 **패턴 일치** 문을 제공합니다. C#은 또한 값이 특정 형식인지 테스트하기 위해 `is` 및 `as` 연산자를 제공합니다.

다음 예제에서는 패턴 일치 `is` 문을 사용하는 방법을 보여 줍니다.

C#

```
var g = new Giraffe();
var a = new Animal();
FeedMammals(g);
FeedMammals(a);
// Output:
// Eating.
// Animal is not a Mammal

SuperNova sn = new SuperNova();
TestForMammals(g);
TestForMammals(sn);

static void FeedMammals(Animal a)
{
    if (a is Mammal m)
    {
        m.Eat();
    }
    else
    {
        // variable 'm' is not in scope here, and can't be used.
        Console.WriteLine($"{a.GetType().Name} is not a Mammal");
    }
}

static void TestForMammals(object o)
{
    // You also can use the as operator and test for null
    // before referencing the variable.
    var m = o as Mammal;
    if (m != null)
    {
        Console.WriteLine(m.ToString());
    }
    else
```

```

    {
        Console.WriteLine($"{o.GetType().Name} is not a Mammal");
    }
}

// Output:
// I am an animal.
// SuperNova is not a Mammal

class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }

```

위의 샘플은 패턴 일치 구문의 몇 가지 기능을 보여 줍니다. `if (a is Mammal m)` 문은 테스트를 초기화 할당과 결합합니다. 할당은 테스트가 성공한 경우에만 발생합니다. 변수 `m`은 할당된 `if` 문의 포함되어 있는 범위에만 있습니다. 나중에 동일한 방법으로 `m`에 액세스할 수 없습니다. 앞의 예제에서는 `as 연산자`를 사용하여 개체를 지정된 형식으로 변환하는 방법도 보여 줍니다.

다음 예제에 표시된 대로 `null` 허용 값 형식에 값이 있는 경우 테스트할 때도 동일한 구문을 사용할 수 있습니다.

C#

```

int i = 5;
PatternMatchingNullable(i);

int? j = null;
PatternMatchingNullable(j);

double d = 9.78654;
PatternMatchingNullable(d);

PatternMatchingSwitch(i);
PatternMatchingSwitch(j);
PatternMatchingSwitch(d);

static void PatternMatchingNullable(ValueType? val)
{
    if (val is int j) // Nullable types are not allowed in patterns
    {
        Console.WriteLine(j);
    }
}

```

```

        else if (val is null) // If val is a nullable type with no value, this
expression is true
    {
        Console.WriteLine("val is a nullable type with the null value");
    }
else
{
    Console.WriteLine("Could not convert " + val.ToString());
}
}

static void PatternMatchingSwitch(ValueType? val)
{
    switch (val)
    {
        case int number:
            Console.WriteLine(number);
            break;
        case long number:
            Console.WriteLine(number);
            break;
        case decimal number:
            Console.WriteLine(number);
            break;
        case float number:
            Console.WriteLine(number);
            break;
        case double number:
            Console.WriteLine(number);
            break;
        case null:
            Console.WriteLine("val is a nullable type with the null value");
            break;
        default:
            Console.WriteLine("Could not convert " + val.ToString());
            break;
    }
}

```

위의 샘플은 변환에 사용하기 위한 패턴 일치의 다른 기능을 보여 줍니다. 구체적으로 `null` 값을 확인하여 `null` 패턴에 대한 변수를 테스트할 수 있습니다. 변수의 런타임 값이 `null`일 때 형식을 확인하는 `is` 문은 항상 `false`를 반환합니다. 패턴 일치 `is` 문은 `int?` 또는 `Nullable<int>`과 같은 nullable 값 형식을 허용하지 않지만 다른 값 형식을 테스트할 수 있습니다. 이전 예의 `is` 패턴은 `null` 허용 값 형식으로 제한되지 않습니다. 이러한 패턴을 사용하여 참조 형식의 변수에 값이 있는지 또는 값이 `null`인지 테스트할 수도 있습니다.

위의 샘플도 변수가 여러 형식 중 하나일 수 있는 `switch` 문에서 형식 패턴을 사용하는 방법을 보여 줍니다.

변수가 지정된 형식인지 테스트하지만 새 변수에 할당하지 않으려는 경우, 참조 형식 및 null 허용 값 형식에 대해 `is` 및 `as` 연산자를 사용할 수 있습니다. 다음 코드는 변수가 지정된 형식인지 테스트하기 위해 패턴 일치가 도입되기 전에 C# 언어의 일부인 `is` 및 `as` 문을 사용하는 방법을 보여 줍니다.

C#

```
// Use the is operator to verify the type.  
// before performing a cast.  
Giraffe g = new();  
UseIsOperator(g);  
  
// Use the as operator and test for null  
// before referencing the variable.  
UseAsOperator(g);  
  
// Use pattern matching to test for null  
// before referencing the variable  
UsePatternMatchingIs(g);  
  
// Use the as operator to test  
// an incompatible type.  
SuperNova sn = new();  
UseAsOperator(sn);  
  
// Use the as operator with a value type.  
// Note the implicit conversion to int? in  
// the method body.  
int i = 5;  
UseAsWithNullable(i);  
  
double d = 9.78654;  
UseAsWithNullable(d);  
  
static void UseIsOperator(Animal a)  
{  
    if (a is Mammal)  
    {  
        Mammal m = (Mammal)a;  
        m.Eat();  
    }  
}  
  
static void UsePatternMatchingIs(Animal a)  
{  
    if (a is Mammal m)  
    {  
        m.Eat();  
    }  
}  
  
static void UseAsOperator(object o)  
{
```

```

Mammal? m = o as Mammal;
if (m is not null)
{
    Console.WriteLine(m.ToString());
}
else
{
    Console.WriteLine($"{o.GetType().Name} is not a Mammal");
}
}

static void UseAsWithNullable(System.ValueType val)
{
    int? j = val as int?;
    if (j is not null)
    {
        Console.WriteLine(j);
    }
    else
    {
        Console.WriteLine("Could not convert " + val.ToString());
    }
}
class Animal
{
    public void Eat() => Console.WriteLine("Eating.");
    public override string ToString() => "I am an animal.";
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }

```

이 코드를 패턴 일치 코드와 비교하여 볼 수 있듯이, 패턴 일치 구문은 테스트와 할당을 단일 명령문으로 결합함으로써 더욱 강력한 기능을 제공합니다. 가능할 때마다 패턴 일치 구문을 사용합니다.

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.



## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

[설명서 문제 열기](#)

[제품 사용자 의견 제공](#)

# 자습서: 패턴 일치를 사용하여 형식 기반 및 데이터 기반 알고리즘 빌드

아티클 • 2023. 04. 08.

다른 라이브러리에 있을 수 있는 형식을 확장한 것처럼 동작하는 기능을 작성할 수 있습니다. 패턴의 또 다른 용도는 확장되는 형식의 기초 기능이 아닌 애플리케이션에 필요한 기능을 만드는 것입니다.

이 자습서에서는 다음과 같은 작업을 수행하는 방법을 알아봅니다.

- ✓ 패턴 일치를 사용해야 하는 상황을 인식합니다.
- ✓ 패턴 일치 식을 사용하여 형식 및 속성 값에 따라 동작을 구현합니다.
- ✓ 패턴 일치를 다른 기술과 결합하여 완전한 알고리즘을 만듭니다.

## 사전 요구 사항

- Windows 또는 Mac용 [Visual Studio](#) 를 사용하는 것이 좋습니다. [Visual Studio 다운로드 페이지에서 무료 버전을 다운로드할 수 있습니다](#). Visual Studio에는 .NET SDK가 포함되어 있습니다.
- [Visual Studio Code](#) 편집기를 사용할 수도 있습니다. 최신 [.NET SDK](#) 를 별도로 설치해야 합니다.
- 다른 편집기를 선호하는 경우 최신 [.NET SDK](#) 를 설치해야 합니다.

이 자습서에서는 여러분이 Visual Studio 또는 .NET CLI를 비롯한 C# 및 .NET에 익숙하다고 가정합니다.

## 패턴 일치 시나리오

최신 개발에는 종종 여러 소스의 데이터를 통합하고 해당 데이터의 정보와 인사이트를 단일 결합 애플리케이션에서 제공하는 작업이 포함됩니다. 여러분과 팀에는 들어오는 데이터를 나타내는 모든 형식에 대한 제어 또는 액세스 권한이 없습니다.

클래식 개체 지향 디자인의 경우 여러 데이터 소스의 각 데이터 형식을 나타내는 형식을 애플리케이션에서 만들어야 합니다. 그런 다음, 애플리케이션은 이 새로운 형식을 사용하고, 상속 계층 구조를 빌드하고, 가상 메서드를 만들고, 추상화를 구현합니다. 이러한 기술은 작동하며 때로는 최고의 도구입니다. 경우에 따라 더 적은 코드를 작성할 수 있습니다. 데이터를 조작하는 작업에서 데이터를 분리하는 기술을 사용하여 보다 명확한 코드를 작성할 수 있습니다.

이 자습서에서는 단일 시나리오에 대해 여러 외부 소스에서 들어오는 데이터를 사용하는 애플리케이션을 만들고 살펴봅니다. **패턴 일치**가 원래 시스템에 포함되지 않은 데이터를 사용하고 처리하는 효율적인 방법을 어떻게 제공하는지 확인합니다.

교통량을 관리하는 데 통행료 및 최대 사용 시간 가격을 사용하는 주요 도시 지역을 살펴보겠습니다. 형식에 따라 차량의 통행료를 계산하는 애플리케이션을 작성합니다. 나중에 차량 탑승자 수에 따른 가격이 개선 사항에 통합됩니다. 추가로 시간 및 요일에 따른 가격이 개선 사항에 추가됩니다.

이 간단한 설명을 통해 이 시스템을 모델링하기 위한 개체 계층 구조를 빠르게 설명했을 수 있습니다. 그러나 데이터는 다른 차량 등록 관리 시스템 같은 다양한 출처에서 수집됩니다. 이 시스템은 해당 데이터를 모델링하는 여러 가지 클래스를 제공하지만 사용자가 사용할 수 있는 단일 개체 모델이 없습니다. 이 자습서에서는 다음 코드와 같이 해당 외부 시스템의 차량 데이터를 모델링하는 데 이 간단한 클래스를 사용합니다.

C#

```
namespace ConsumerVehicleRegistration
{
    public class Car
    {
        public int Passengers { get; set; }
    }
}

namespace CommercialRegistration
{
    public class DeliveryTruck
    {
        public int GrossWeightClass { get; set; }
    }
}

namespace LiveryRegistration
{
    public class Taxi
    {
        public int Fares { get; set; }
    }

    public class Bus
    {
        public int Capacity { get; set; }
        public int Riders { get; set; }
    }
}
```

시작 코드는 [dotnet/samples](#) GitHub 리포지토리에서 다운로드할 수 있습니다. 차량 클래스는 여러 가지 시스템에서 가져오고 서로 다른 네임스페이스에 포함됨을 알 수 있습

니다. 를 제외한 `System.Object` 일반 기본 클래스는 사용할 수 없습니다.

## 패턴 일치 디자인

이 자습서에서 사용된 시나리오는 패턴 일치를 통해 해결하기에 적합한 종류의 문제를 중점적으로 다룹니다.

- 사용해야 하는 개체는 목표와 일치하는 개체 계층 구조에 없습니다. 관련 없는 시스템에 포함된 클래스를 사용 중일 수 있습니다.
- 추가할 기능은 이 클래스에 대한 핵심 추상화에 포함되지 않습니다. 차량에 따른 통행료는 차량 형식에 따라 변경되지만 통행료는 차량의 핵심 기능이 아닙니다.

데이터의 모양과 해당 데이터에 대한 작업이 함께 설명되지 않은 경우 C#의 패턴 일치 기능을 사용하면 더 쉽게 작업할 수 있습니다.

## 기본 통행료 계산 구현

가장 기본적인 통행료 계산에는 차량 형식만 사용됩니다.

- `Car`는 2.00 USD입니다.
- `Taxi`는 3.50 USD입니다.
- `Bus`는 5.00 USD입니다.
- `DeliveryTruck`은 10.00 USD입니다.

새 `TollCalculator` 클래스를 만들고 차량 형식에 대한 패턴 일치를 구현하여 통행료 액수를 얻습니다. 다음 코드에서는 `TollCalculator`의 초기 구현을 보여줍니다.

C#

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace Calculators;

public class TollCalculator
{
    public decimal CalculateToll(object vehicle) =>
        vehicle switch
    {
        Car c          => 2.00m,
        Taxi t         => 3.50m,
        Bus b          => 5.00m,
        DeliveryTruck t => 10.00m,
        { }             => throw new ArgumentException(message: "Not a known vehicle type")
    }
}
```

```
        vehicle type", paramName: nameof(vehicle)),
        null           => throw new ArgumentNullException(nameof(vehicle))
    );
}
```

앞의 코드는 [선언 패턴을](#) 테스트하는 식(문과 동일하지 않음switch)을 사용합니다 switch . switch 식은 앞의 코드에 있는 `vehicle` 변수로 시작하고 그 뒤에 `switch` 키워드가 옵니다. 다음은 중괄호로 묶인 모든 [스위치 암\(arm\)](#)을 제공합니다. `switch` 식은 `switch` 문을 둘러싸는 구문을 다르게 구체화합니다. `case` 키워드가 생략되고 각 암(arm)의 결과는 식입니다. 마지막 두 개의 암(arm)은 새 언어 기능을 보여 줍니다. `{ }` 사례는 이전 암(arm)과 일치하지 않는 `null`이 아닌 개체와 일치합니다. 이 암(arm)은 이 메서드에 전달된 잘못된 형식을 `catch`합니다. `{ }` 사례는 각 차량 형식에 대한 사례를 따라야 합니다. 순서가 반대로 된 경우 `{ }` 사례가 우선적으로 적용됩니다. 마지막으로 [상수 패턴은](#) `null` 가 이 메서드에 전달되는 시기를 `null` 감지합니다. 다른 패턴은 올바른 형식의 `null`이 아닌 개체와만 일치하므로 `null` 패턴이 마지막일 수 있습니다.

`Program.cs`에서 다음 코드를 사용하여 이 코드를 테스트할 수 있습니다.

C#

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

using toll_calculator;

var tollCalc = new TollCalculator();

var car = new Car();
var taxi = new Taxi();
var bus = new Bus();
var truck = new DeliveryTruck();

Console.WriteLine($"The toll for a car is {tollCalc.CalculateToll(car)}");
Console.WriteLine($"The toll for a taxi is {tollCalc.CalculateToll(taxi)}");
Console.WriteLine($"The toll for a bus is {tollCalc.CalculateToll(bus)}");
Console.WriteLine($"The toll for a truck is
{tollCalc.CalculateToll(truck)}");

try
{
    tollCalc.CalculateToll("this will fail");
}
catch (ArgumentException e)
{
    Console.WriteLine("Caught an argument exception when using the wrong
type");
}
```

```

try
{
    tollCalc.CalculateToll(null!);
}
catch (ArgumentNullException e)
{
    Console.WriteLine("Caught an argument exception when using null");
}

```

해당 코드는 시작 프로젝트에 포함되지만 주석으로 처리됩니다. 주석을 제거하면 작성한 내용을 테스트할 수 있습니다.

먼저 패턴을 사용하여 코드 및 데이터가 구분되는 알고리즘을 만드는 방법을 확인하겠습니다. `switch` 식은 형식을 테스트하고 결과에 따라 다른 값을 생성합니다. 이는 시작에 불과합니다.

## 점유 가격 추가

통행료 징수 기관은 차량이 최대 탑승자 수로 이동하도록 권장하려고 합니다. 차량에 더 적은 승객이 있는 경우 추가 요금을 청구하기로 했으며, 더 낮은 가격을 제공하여 최대 탑승자 차량을 장려합니다.

- 승객이 없는 승용차와 택시에는 0.50 USD의 추가 통행료가 부과됩니다.
- 승객이 2명인 승용차와 택시는 \$0.50의 할인을 받습니다.
- 승객이 3명 이상인 승용차와 택시는 1.00 USD의 할인을 받습니다.
- 최대 탑승자 수의 50% 미만이 탑승한 버스는 2.00 USD의 추가 요금이 부과됩니다.
- 탑승자 수가 90%를 초과하는 버스는 1.00 USD의 할인을 받습니다.

이 규칙은 동일한 `switch` 식에서 [속성 패턴](#)을 사용하여 구현할 수 있습니다. 속성 패턴은 속성 값을 상수 값과 비교합니다. 속성 패턴은 형식이 결정된 후 개체의 속성을 검사합니다. `Car` 사례 1개는 다른 사례 4개로 확장됩니다.

C#

```

vehicle switch
{
    Car {Passengers: 0} => 2.00m + 0.50m,
    Car {Passengers: 1} => 2.0m,
    Car {Passengers: 2} => 2.0m - 0.50m,
    Car                  => 2.00m - 1.0m,

    // ...
};

```

처음 3개 사례는 형식을 `Car`로 테스트한 다음, `Passengers` 속성 값을 확인합니다. 둘 다 일치하는 경우 해당 식이 계산되고 반환됩니다.

또한 비슷한 방식으로 택시에 대한 사례를 확장합니다.

C#

```
vehicle switch
{
    // ...

    Taxi {Fares: 0} => 3.50m + 1.00m,
    Taxi {Fares: 1} => 3.50m,
    Taxi {Fares: 2} => 3.50m - 0.50m,
    Taxi             => 3.50m - 1.00m,

    // ...
}
```

다음으로, 다음 예제와 같이 버스 사례를 확장하여 점유 규칙을 구현합니다.

C#

```
vehicle switch
{
    // ...

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
    2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
    1.00m,
    Bus => 5.00m,

    // ...
}
```

통행료 징수 기관은 배달 트럭의 승객 수에 관심이 없습니다. 대신 다음과 같이 트럭의 중량 등급을 기준으로 요금을 조정합니다.

- 5000lbs를 초과하는 트럭에는 5.00 USD가 추가로 부과됩니다.
- 3000lbs 미만의 경량 트럭에는 \$2.00 할인이 제공됩니다.

해당 규칙은 다음 코드로 구현됩니다.

C#

```
vehicle switch
{
    // ...
```

```

DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
DeliveryTruck => 10.00m,
};

```

앞의 코드는 스위치 암(arm)의 `when` 절을 표시합니다. `when` 절을 사용하여 속성에서 같음 이외의 조건을 테스트합니다. 작업을 마치면 다음 코드와 같은 메서드가 생성됩니다.

C#

```

vehicle switch
{
    Car {Passengers: 0}      => 2.00m + 0.50m,
    Car {Passengers: 1}      => 2.0m,
    Car {Passengers: 2}      => 2.0m - 0.50m,
    Car                      => 2.00m - 1.0m,

    Taxi {Fares: 0}   => 3.50m + 1.00m,
    Taxi {Fares: 1}   => 3.50m,
    Taxi {Fares: 2}   => 3.50m - 0.50m,
    Taxi                  => 3.50m - 1.00m,

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
1.00m,
    Bus => 5.00m,

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck => 10.00m,

    { }      => throw new ArgumentException(message: "Not a known vehicle
type", paramName: nameof(vehicle)),
    null     => throw new ArgumentNullException(nameof(vehicle))
};

```

대부분의 이 스위치 암(arm)은 **재귀 패턴**의 예입니다. 예를 들어 `Car { Passengers: 1 }`은 속성 패턴 내의 상수 패턴을 표시합니다.

중첩된 스위치를 사용하여 이 코드의 반복 횟수를 줄일 수 있습니다. `Car` 및 `Taxi`에는 둘다 앞의 예제에 있는 서로 다른 암(arm) 4개가 포함됩니다. 두 경우 모두 상수 패턴으로 피드되는 선언 패턴을 만들 수 있습니다. 이 기술이 다음 코드에 나옵니다.

C#

```

public decimal CalculateToll(object vehicle) =>
    vehicle switch
    {

```

```

        Car c => c.Passengers switch
        {
            0 => 2.00m + 0.5m,
            1 => 2.0m,
            2 => 2.0m - 0.5m,
            _ => 2.00m - 1.0m
        },
        Taxi t => t.Fares switch
        {
            0 => 3.50m + 1.00m,
            1 => 3.50m,
            2 => 3.50m - 0.50m,
            _ => 3.50m - 1.00m
        },
        Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
2.00m,
        Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
1.00m,
        Bus b => 5.00m,
        DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
        DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
        DeliveryTruck t => 10.00m,
        { } => throw new ArgumentException(message: "Not a known vehicle type",
paramName: nameof(vehicle)),
        null => throw new ArgumentNullException(nameof(vehicle))
    };

```

앞의 샘플에서 재귀 식을 사용하면 속성 값을 테스트하는 자식 암(arm)이 포함된 `Car` 및 `Taxi` 암(arm)을 반복하지 않습니다. 해당 암(arm)은 불연속 값이 아닌 속성 범위를 테스트하므로 이 기술은 `Bus` 및 `DeliveryTruck` 암(arm)에 사용되지 않습니다.

## 최대 가격 추가

마지막 기능을 위해 통행료 징수 기관은 시간에 따른 최대 가격을 추가하려고 합니다. 아침 및 저녁 교통 체증 시간 중에 통행료는 2배로 부과됩니다. 해당 규칙은 아침에는 도시로 들어오고 저녁 교통 체증 시간에는 나가는 한 방향의 교통량에만 영향을 줍니다. 평일 중 다른 시간에 통행료는 50% 증가합니다. 늦은 밤과 이른 아침에는 통행료가 25% 감소합니다. 주말에는 시간과 관계없이 정상 요금입니다. 일련의 `if` 및 `else` 문을 사용하여 다음 코드를 사용하여 이를 표현할 수 있습니다.

C#

```

public decimal PeakTimePremiumIfElse(DateTime timeOfToll, bool inbound)
{

```

```

if ((timeOfToll.DayOfWeek == DayOfWeek.Saturday) ||
    (timeOfToll.DayOfWeek == DayOfWeek.Sunday))
{
    return 1.0m;
}
else
{
    int hour = timeOfToll.Hour;
    if (hour < 6)
    {
        return 0.75m;
    }
    else if (hour < 10)
    {
        if (inbound)
        {
            return 2.0m;
        }
        else
        {
            return 1.0m;
        }
    }
    else if (hour < 16)
    {
        return 1.5m;
    }
    else if (hour < 20)
    {
        if (inbound)
        {
            return 1.0m;
        }
        else
        {
            return 2.0m;
        }
    }
    else // Overnight
    {
        return 0.75m;
    }
}
}

```

위 코드는 정상적으로 실행되지만 읽을 수 없습니다. 모든 입력 사례와 중첩된 `if` 문을 연결하여 코드에 대해 추론해야 합니다. 이 기능을 위해 패턴 일치를 대신 사용하되, 다른 기술과 통합합니다. 방향, 요일 및 시간의 모든 조합을 설명하는 단일 패턴 일치 식을 빌드할 수 있습니다. 복잡한 식이 생성됩니다. 읽기 힘들고 이해하기 어려운 식입니다. 따라서 식의 정확성을 보장하기 어렵습니다. 대신, 해당 메서드를 결합하여 모든 상태를 간결하

게 설명하는 값 튜플을 빌드합니다. 그런 다음, 패턴 일치를 사용하여 통행료의 승수를 계산합니다. 튜플에는 다음 세 가지 불연속 조건이 포함됩니다.

- 요일은 주중 또는 주말입니다.
- 통행료가 징수되는 시간대.
- 방향은 도시 진입 또는 도시 진출입니다.

다음 표는 입력 값과 최대 가격 승수의 조합을 보여 줍니다.

일	시간	Direction	Premium
요일	아침 교통 체증	인바운드	x 2.00
요일	아침 교통 체증	아웃바운드	x 1.00
요일	주간	인바운드	x 1.50
요일	주간	아웃바운드	x 1.50
요일	저녁 교통 체증	인바운드	x 1.00
요일	저녁 교통 체증	아웃바운드	x 2.00
요일	야간	인바운드	x 0.75
요일	야간	아웃바운드	x 0.75
주말	아침 교통 체증	인바운드	x 1.00
주말	아침 교통 체증	아웃바운드	x 1.00
주말	주간	인바운드	x 1.00
주말	주간	아웃바운드	x 1.00
주말	저녁 교통 체증	인바운드	x 1.00
주말	저녁 교통 체증	아웃바운드	x 1.00
주말	야간	인바운드	x 1.00
주말	야간	아웃바운드	x 1.00

세 가지 변수의 조합은 16가지입니다. 일부 조건을 결합하여 마지막 switch 식을 단순화 합니다.

통행료를 징수하는 시스템은 통행료가 징수된 시간에 [DateTime](#) 구조체를 사용합니다. 앞의 표에서 변수를 만드는 멤버 메서드를 작성합니다. 다음 함수는 패턴 일치 switch 식을 사용하여 [DateTime](#)이 주말 또는 주중을 나타내는지 여부를 표시합니다.

C#

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Monday     => true,
        DayOfWeek.Tuesday    => true,
        DayOfWeek.Wednesday  => true,
        DayOfWeek.Thursday   => true,
        DayOfWeek.Friday     => true,
        DayOfWeek.Saturday   => false,
        DayOfWeek.Sunday     => false
    };
```

해당 메서드는 올바른 것이지만 반복적입니다. 다음 코드와 같이 단순화할 수 있습니다.

C#

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Saturday => false,
        DayOfWeek.Sunday   => false,
        _                  => true
    };
```

다음으로, 시간을 블록으로 분류하는 비슷한 함수를 추가합니다.

C#

```
private enum TimeBand
{
    MorningRush,
    Daytime,
    EveningRush,
    Overnight
}

private static TimeBand GetTimeBand(DateTime timeOfToll) =>
    timeOfToll.Hour switch
    {
        < 6 or > 19 => TimeBand.Overnight,
        < 10 => TimeBand.MorningRush,
        < 16 => TimeBand.Daytime,
        _          => TimeBand.EveningRush,
    };
```

프라이빗 `enum`을 추가하여 각 시간 범위를 불연속 값으로 변환합니다. 그러면 `GetTimeBand` 메서드는 [관계형 패턴](#) 및 [결합 or 패턴](#)을 사용합니다. 두 패턴은 모두 C# 9.0

에서 추가된 것입니다. 관계형 패턴을 사용하면 `<`, `>`, `<=` 또는 `>=`로 숫자 값을 테스트할 수 있습니다. `or` 패턴은 식이 하나 이상의 패턴과 일치하는지 테스트합니다. `and` 패턴을 사용하여 식이 두 개의 고유한 패턴과 일치하는지 확인하고, `not` 패턴을 사용하여 식이 패턴과 일치하지 않는지 테스트할 수도 있습니다.

이 메서드를 만든 후 **튜플 패턴**과 함께 다른 `switch` 식을 사용하여 할증 가격을 계산할 수 있습니다. 모든 16개 암(arm)을 사용하여 `switch` 식을 작성할 수 있습니다.

C#

```
public decimal PeakTimePremiumFull(DateTime timeOfToll, bool inbound) =>
    IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
{
    (true, TimeBand.MorningRush, true) => 2.00m,
    (true, TimeBand.MorningRush, false) => 1.00m,
    (true, TimeBand.Daytime, true) => 1.50m,
    (true, TimeBand.Daytime, false) => 1.50m,
    (true, TimeBand.EveningRush, true) => 1.00m,
    (true, TimeBand.EveningRush, false) => 2.00m,
    (true, TimeBand.OVERNIGHT, true) => 0.75m,
    (true, TimeBand.OVERNIGHT, false) => 0.75m,
    (false, TimeBand.MorningRush, true) => 1.00m,
    (false, TimeBand.MorningRush, false) => 1.00m,
    (false, TimeBand.Daytime, true) => 1.00m,
    (false, TimeBand.Daytime, false) => 1.00m,
    (false, TimeBand.EveningRush, true) => 1.00m,
    (false, TimeBand.EveningRush, false) => 1.00m,
    (false, TimeBand.OVERNIGHT, true) => 1.00m,
    (false, TimeBand.OVERNIGHT, false) => 1.00m,
};
```

위 코드는 작동하지만 단순화할 수 있습니다. 주말에 대한 8개 조합에는 모두 동일한 통행료가 포함됩니다. 8개 모두를 다음 줄로 바꿀 수 있습니다.

C#

```
(false, _, _) => 1.0m,
```

인바운드 및 아웃바운드 교통량은 둘 다 주중 주간 및 야간 시간 동안 동일한 승수를 포함합니다. 해당하는 4개의 스위치 암(arm)은 다음 두 줄로 바꿀 수 있습니다.

C#

```
(true, TimeBand.OVERNIGHT, _) => 0.75m,
(true, TimeBand.DAYTIME, _)   => 1.5m,
```

해당하는 두 줄이 변경된 후 코드는 다음과 같이 표시됩니다.

C#

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
{
    (true, TimeBand.MorningRush, true) => 2.00m,
    (true, TimeBand.MorningRush, false) => 1.00m,
    (true, TimeBand.Daytime, _) => 1.50m,
    (true, TimeBand.EveningRush, true) => 1.00m,
    (true, TimeBand.EveningRush, false) => 2.00m,
    (true, TimeBand.Overnight, _) => 0.75m,
    (false, _, _) => 1.00m,
};
```

마지막으로 정규 가격을 납부하는 두 번의 교통 체증 시간을 제거할 수 있습니다. 해당 암(arm)을 제거하면 마지막 스위치 암(arm)에서 `false`를 삭제(`_`)로 바꿀 수 있습니다. 완료된 메서드는 다음과 같습니다.

C#

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
{
    (true, TimeBand.Overnight, _) => 0.75m,
    (true, TimeBand.Daytime, _) => 1.5m,
    (true, TimeBand.MorningRush, true) => 2.0m,
    (true, TimeBand.EveningRush, false) => 2.0m,
    _ => 1.0m,
};
```

이 예제는 패턴 일치의 장점 중 하나를 강조 표시합니다. 패턴 분기는 순서대로 평가됩니다. 이전 분기가 이후 사례 중 하나를 처리하도록 분기를 재배열하는 경우 컴파일러는 접근할 수 없는 코드에 대해 경고합니다. 이 언어 규칙을 사용하면 코드가 변경되지 않는다는 확신과 함께 앞의 단순화 작업을 더 쉽게 수행할 수 있습니다.

패턴 일치는 일부 유형의 코드를 더 쉽게 읽을 수 있도록 해주며 클래스에 코드를 추가할 수 없을 때 개체 지향 기술에 대한 대안을 제공합니다. 클라우드에서는 데이터와 기능이 별도로 구분되지 않습니다. 데이터의 모양과 데이터에 대한 작업을 반드시 함께 설명할 필요는 없습니다. 이 자습서에서는 원래 함수와 완전히 다른 방식으로 기존 데이터를 사용했습니다. 해당 형식을 확장할 수 없더라도 패턴 일치를 통해 해당 형식을 재정의하는 기능을 작성할 수 있었습니다.

## 다음 단계

완료된 코드는 [dotnet/samples](#) GitHub 리포지토리에서 다운로드할 수 있습니다. 혼자 패턴을 살펴보고 이 기술을 일반적인 코딩 활동에 추가하세요. 이 기술을 학습하면 다른 방법으로 문제에 접근하고 새 기능을 만들 수 있습니다.

## 참조

- [패턴](#)
- [switch 식](#)

# try/catch를 사용하여 예외를 처리하는 방법

아티클 • 2024. 02. 23.

try-catch 블록은 작업 코드에서 생성된 예외를 catch하고 처리하기 위한 것입니다. 일부 예외는 catch 블록에서 처리될 수 있으며, 예외가 다시 throw되지 않고 문제가 해결됩니다. 그러나 대체로 수행할 수 있는 작업은 적절한 예외가 throw되었는지 확인하는 것뿐입니다.

## 예시

이 예제에서 `IndexOutOfRangeException`은 가장 적합한 예외가 아닙니다. 호출자가 전달한 `index` 인수로 인해 오류가 발생하기 때문에 `ArgumentOutOfRangeException`이 메서드에 더 적합합니다.

```
C#  
  
static int GetInt(int[] array, int index)  
{  
    try  
    {  
        return array[index];  
    }  
    catch (IndexOutOfRangeException e) // CS0168  
    {  
        Console.WriteLine(e.Message);  
        // Set IndexOutOfRangeException to the new exception's  
        InnerException.  
        throw new ArgumentException("index parameter is out of  
        range.", e);  
    }  
}
```

## 설명

예외가 발생하는 코드는 `try` 블록으로 묶여 있습니다. `catch` 문이 발생하는 경우 `IndexOutOfRangeException`을 처리하기 위해 바로 뒤에 추가됩니다. `catch` 블록은 `IndexOutOfRangeException`을 처리하고 대신 더 적절한 `ArgumentException`을 throw합니다. 호출자에게 최대한 많은 정보를 제공하기 위해 원래 예외를 새 예외의 `InnerException`으로 지정하는 것이 좋습니다. `InnerException` 속성은 `읽기 전용`이기 때문에 새 예외의 생성자에 할당해야 합니다.

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# finally를 사용하여 정리 코드를 실행하는 방법

아티클 • 2023. 04. 22.

`finally` 문은 예외가 `throw`된 경우에도 개체, 일반적으로 외부 리소스를 포함하는 개체의 필요한 정리가 즉시 수행되도록 합니다. 이러한 정리 작업의 한 가지 예로 다음과 같이 개체가 공용 언어 런타임에 의해 수집될 때까지 기다리지 않고 사용 후 즉시 `FileStream`에서 `Close`를 호출하는 것을 들 수 있습니다.

C#

```
static void CodeWithoutCleanup()
{
    FileStream? file = null;
    FileInfo fileInfo = new FileInfo("./file.txt");

    file = fileInfo.OpenWrite();
    file.WriteByte(0xF);

    file.Close();
}
```

## 예제

이전 코드를 `try-catch-finally` 문으로 바꾸려면 다음과 같이 정리 코드와 작업 코드를 구분합니다.

C#

```
static void CodeWithCleanup()
{
    FileStream? file = null;
    FileInfo? fileInfo = null;

    try
    {
        fileInfo = new FileInfo("./file.txt");

        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    catch (UnauthorizedAccessException e)
    {
        Console.WriteLine(e.Message);
    }
}
```

```
finally
{
    file?.Close();
}
}
```

`OpenWrite()` 호출 또는 `OpenWrite()` 호출 자체가 실패하기 전에 `try` 블록 내에서 언제든지 예외가 발생할 수 있으므로 파일을 닫으려고 할 때 열려 있다는 보장은 없습니다. `finally` 블록은 검사를 추가하여 `Close` 메서드를 호출하기 전에 `FileStream` 객체가 `null`이 아닌지 확인합니다. `null` 검사가 없으면 `finally` 블록에서 고유한 `NullReferenceException`을 `throw`할 수 있지만 가능하면 `finally` 블록에서 예외를 `throw`하지 않도록 해야 합니다.

데이터베이스 연결은 `finally` 블록에서 닫기에 적합한 또 다른 후보입니다. 데이터베이스 서버에 허용되는 연결 수가 제한된 경우도 있으므로 최대한 빨리 데이터베이스 연결을 닫아야 합니다. 연결을 닫기 전에 예외가 `throw`되는 경우에도 `finally` 블록 사용이 가비지 수집을 기다리는 것보다 더 낫습니다.

## 참고 항목

- [using 문](#)
- [예외 처리 문](#)

# C# 13의 새로운 기능

아티클 • 2024. 03. 20.

C# 13에는 다음과 같은 새로운 기능이 포함되어 있습니다. 최신 [Visual Studio 2022](#) 버전 또는 .NET 9 미리 보기 SDK를 사용하여 이러한 기능을 사용해 볼 수 있습니다.

- 새 이스케이프 시퀀스 - \e.
- 메서드 그룹 자연 형식 개선
- 개체 이니셜라이저의 암시적 인덱서 액세스

C# 13은 .NET 9에서 지원됩니다. 자세한 내용은 [C# 언어 버전 관리](#)를 참조하세요.

.NET 다운로드 페이지에서 [최신 .NET 9 미리 보기 SDK를 다운로드할](#) 수 있습니다. .NET 9 미리 보기 SDK를 포함하는 Visual Studio 2022 - 미리 보기 [다운로드](#) 할 수도 있습니다.

새 기능은 공개 미리 보기 릴리스에서 사용할 수 있는 경우 "C#의 새로운 기능" 페이지에 추가됩니다. roslyn 기능 상태 [페이지의](#) 작업 집합 섹션은 예정된 기능이 기본 분기에 병합될 때 추적됩니다.

## ① 참고

이러한 기능에 대한 사용자의 피드백을 환영합니다. 이러한 새로운 기능과 관련된 문제를 발견하면 [dotnet/roslyn](#) 리포지토리에서 새 문제를 만듭니다.

## 새 이스케이프 시퀀스

문자 유니코드에 대한 문자 리터럴 [이스케이프 시퀀스로 사용할 \e](#) 수 있습니다. ESCAPE `U+001B` 이전에는 사용 `\u001b` 했거나 `\x1b`. 다음 `1b` 문자가 유효한 16진수 숫자인 경우 해당 문자가 이스케이프 시퀀스의 일부가 되었기 때문에 사용하지 `\x1b` 않는 것이 좋습니다.

## 메서드 그룹 자연 형식

이 기능은 메서드 그룹과 관련된 오버로드 해상도를 약간 최적화합니다. 이전 동작은 컴파일러가 메서드 그룹에 대한 후보 메서드의 전체 집합을 생성하는 것이었습니다. 자연 형식이 필요한 경우 자연 형식은 후보 메서드의 전체 집합에서 결정되었습니다.

새 동작은 각 범위에서 후보 메서드 집합을 정리하여 적용할 수 없는 후보 메서드를 제거하는 것입니다. 일반적으로 이러한 메서드는 잘못된 심각도가 있는 제네릭 메서드이거나

충족되지 않는 제약 조건입니다. 후보 메서드를 찾을 수 없는 경우에만 프로세스가 다음 외부 범위로 계속 진행됩니다. 이 프로세스는 오버로드 확인에 대한 일반 알고리즘을 더 밀접하게 따릅니다. 지정된 범위에서 찾은 모든 후보 메서드가 일치하지 않으면 메서드 그룹에 자연 형식이 없습니다.

제안 사양의 변경 내용에 대한 세부 정보를 읽을 수 있습니다.

## 암시적 인덱스 액세스

이제 개체 이니셜라이저 식에서 암시적 "from the end" 인덱스 연산 `^`자가 허용됩니다. 예를 들어 이제 다음 코드와 같이 개체 이니셜라이저에서 배열을 초기화할 수 있습니다.

C#

```
var v = new S()
{
    buffer =
    {
        [^1] = 0,
        [^2] = 1,
        [^3] = 2,
        [^4] = 3,
        [^5] = 4,
        [^6] = 5,
        [^7] = 6,
        [^8] = 7,
        [^9] = 8,
        [^10] = 9
    }
};
```

C# 13 `^` 이전 버전에서는 개체 이니셜라이저에서 연산자를 사용할 수 없습니다. 앞에서 요소를 인덱싱해야 합니다.

## 참고 항목

- .NET 9의 새로운 기능

 GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

수도 있습니다. 자세한 내용은  
참여자 가이드를 참조하세요.

☞ 설명서 문제 열기

☞ 제품 사용자 의견 제공

# C# 12의 새로운 기능

아티클 • 2024. 03. 21.

C# 12에는 다음과 같은 새로운 기능이 포함되어 있습니다. 최신 [Visual Studio 2022](#) 버전 또는 .NET 8 SDK를 사용하여 이러한 기능을 사용해 볼 수 있습니다.

- [기본 생성자](#) - Visual Studio 2022 버전 17.6 미리 보기 2에 도입되었습니다.
- [컬렉션 식](#) - Visual Studio 2022 버전 17.7 미리 보기 5에 도입되었습니다.
- [인라인 배열](#) - Visual Studio 2022 버전 17.7 미리 보기 3에 도입되었습니다.
- [람다 식](#) 의 선택적 매개 변수 - Visual Studio 2022 버전 17.5 미리 보기 2에 도입되었습니다.
- [ref readonly parameters](#) - Visual Studio 2022 버전 17.8 미리 보기 2에 도입되었습니다.
- [모든 형식](#) 의 별칭 - Visual Studio 2022 버전 17.6 미리 보기 3에 도입되었습니다.
- [실험적 특성](#) - Visual Studio 2022 버전 17.7 미리 보기 3에 도입되었습니다.
- [인터셉터](#) - [미리 보기 기능](#)은 Visual Studio 2022 버전 17.7 미리 보기 3에 도입되었습니다.

C# 12는 .NET 8에서 지원됩니다. 자세한 내용은 [C# 언어 버전 관리](#)를 참조하세요.

.NET 다운로드 페이지에서 [최신 .NET 8 SDK를 다운로드할](#) 수 있습니다. .NET 8 SDK를 포함하는 Visual Studio 2022를 [다운로드](#) 할 수도 있습니다.

## ① 참고

이러한 기능에 대한 사용자의 피드백을 환영합니다. 이러한 새로운 기능과 관련된 문제를 발견하면 [dotnet/roslyn](#) 리포지토리에서 [새 문제](#)를 만듭니다.

## 기본 생성자

이제 모든 `class` 항목에서 기본 생성자를 만들 수 있습니다 `struct`. 기본 생성자는 더 이상 형식으로 `record` 제한되지 않습니다. 기본 생성자 매개 변수는 클래스의 전체 본문에 대한 범위에 있습니다. 모든 기본 생성자 매개 변수가 확실히 할당되도록 하려면 명시적으로 선언된 모든 생성자는 구문을 사용하여 `this()` 기본 생성자를 호출해야 합니다. 기본 생성자를 추가하면 `class` 컴파일러가 암시적 매개 변수 없는 생성자를 선언하지 못하

게 됩니다. `struct` 암시적 매개 변수 없는 생성자는 기본 생성자 매개 변수를 포함하여 모든 필드를 0비트 패턴으로 초기화합니다.

컴파일러는 형식 또는 `record struct` 형식 `record class`에서만 기본 생성자 매개 변수에 `record` 대한 공용 속성을 생성합니다. 기록되지 않은 클래스 및 구조체는 항상 기본 생성자 매개 변수에 대해 이 동작을 원하지 않을 수 있습니다.

기본 생성자를 탐색하기 위한 자습서 및 인스턴스 생성자에 대한 [문서에서 기본 생성자에 대해 자세히 알아볼 수 있습니다.](#)

## 컬렉션 식

컬렉션 식은 공통 컬렉션 값을 만들기 위한 새 terse 구문을 도입합니다. 스프레드 연산 `..` 자를 사용하여 다른 컬렉션을 이러한 값에 인라인 처리할 수 있습니다.

외부 BCL 지원을 요구하지 않고 여러 컬렉션과 유사한 형식을 만들 수 있습니다. 그 유형은 다음과 같습니다.

- 배열 형식(예: `int[]`).
- `System.Span<T>`와 `System.ReadOnlySpan<T>`을 참조하세요.
- 컬렉션 이니셜라이저를 지원하는 형식(예: `System.Collections.Generic.List<T>`).

다음 예제에서는 컬렉션 식의 사용을 보여 줍니다.

C#

```
// Create an array:  
int[] a = [1, 2, 3, 4, 5, 6, 7, 8];  
  
// Create a list:  
List<string> b = ["one", "two", "three"];  
  
// Create a span  
Span<char> c = ['a', 'b', 'c', 'd', 'e', 'f', 'h', 'i'];  
  
// Create a jagged 2D array:  
int[][] twoD = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];  
  
// Create a jagged 2D array from variables:  
int[] row0 = [1, 2, 3];  
int[] row1 = [4, 5, 6];  
int[] row2 = [7, 8, 9];  
int[][] twoDFromVariables = [row0, row1, row2];
```

컬렉션 식의 `spread` 연산 `..` 자는 해당 인수를 해당 컬렉션의 요소로 바꿉니다. 인수는 컬렉션 형식이어야 합니다. 다음 예제에서는 스프레드 연산자의 작동 방식을 보여 줍니다.

C#

```
int[] row0 = [1, 2, 3];
int[] row1 = [4, 5, 6];
int[] row2 = [7, 8, 9];
int[] single = [.. row0, .. row1, .. row2];
foreach (var element in single)
{
    Console.Write($"{element}, ");
}
// output:
// 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

스프레드 연산자의 피연산자는 열거할 수 있는 식입니다. spread 연산자는 열거형 식의 각 요소를 평가합니다.

요소 컬렉션이 필요한 모든 위치에서 컬렉션 식을 사용할 수 있습니다. 컬렉션의 초기 값을 지정하거나 컬렉션 형식을 사용하는 메서드에 인수로 전달될 수 있습니다. 컬렉션 식 또는 기능 사양에 대한 언어 참조 문서의 컬렉션 식에 대해 자세히 알아볼 수 있습니다.

## ref readonly 매개 변수

C#은 읽기 전용 참조를 전달하는 방법으로 매개 변수를 추가 `in` 했습니다. `in` 매개 변수는 변수와 값을 모두 허용하며 인수에 대한 주석 없이 사용할 수 있습니다.

매개 변수를 `ref readonly` 추가하면 매개 변수 또는 `in` 매개 변수를 사용할 `ref` 수 있는 API의 명확성을 높일 수 있습니다.

- 도입되기 전에 `in` 만든 API는 인수가 수정되지 않은 경우에도 사용할 `ref` 수 있습니다. 이러한 API는 .로 `ref readonly` 업데이트할 수 있습니다. 매개 변수가 .로 변경 `in` 된 경우 `ref` 와 마찬가지로 호출자에 대한 호환성이 손상되는 변경은 아닙니다. 예제는 [System.Runtime.InteropServices.Marshal.QueryInterface](#)입니다.
- 매개 변수를 사용하지만 논리적으로 `in` 변수가 필요한 API입니다. 값 식이 작동하지 않습니다. 예제는 [System.ReadOnlySpan<T>.ReadOnlySpan<T>\(T\)](#)입니다.
- 변수가 필요하지만 해당 변수를 변경하지 않기 때문에 사용하는 `ref` API입니다. 예제는 [System.Runtime.CompilerServices.Unsafe.IsNotNullRef](#)입니다.

매개 변수에 대한 `ref readonly` 자세한 내용은 언어 참조의 매개 변수 한정자 또는 참조 읽기 전용 매개 변수 기능 사양에 대한 문서를 참조하세요.

## 기본 람다 매개 변수

이제 람다 식의 매개 변수에 대한 기본값을 정의할 수 있습니다. 구문 및 규칙은 모든 메서드 또는 로컬 함수에 인수에 대한 기본값을 추가하는 것과 같습니다.

람다 식에 대한 아티클에서 람다 식의 기본 매개 변수에 대해 [자세히 알아볼 수 있습니다](#).

## 모든 형식의 별칭

별칭 지시문을 사용하여 명명된 `using` 형식뿐만 아니라 모든 형식의 별칭을 지정할 수 있습니다. 즉, 튜플 형식, 배열 형식, 포인터 형식 또는 기타 안전하지 않은 형식에 대한 의미체계 별칭을 만들 수 있습니다. 자세한 내용은 기능 사양을 [참조하세요](#).

## 인라인 배열

인라인 배열은 런타임 팀과 다른 라이브러리 작성자가 앱의 성능을 향상시키는 데 사용됩니다. 인라인 배열을 사용하면 개발자가 형식에서 고정 크기의 배열을 `struct` 만들 수 있습니다. 인라인 버퍼가 있는 구조체는 안전하지 않은 고정 크기 버퍼와 유사한 성능 특성을 제공해야 합니다. 고유한 인라인 배열을 선언하지는 않지만 런타임 API에서 개체로 `System.Span<T>`/`System.ReadOnlySpan<T>` 노출될 때 투명하게 사용합니다.

인라인 배열은 다음과 `struct` 유사하게 선언됩니다.

C#

```
[System.Runtime.CompilerServices.InlineArray(10)]
public struct Buffer
{
    private int _element0;
}
```

다른 배열처럼 사용합니다.

C#

```
var buffer = new Buffer();
for (int i = 0; i < 10; i++)
{
    buffer[i] = i;
}

foreach (var i in buffer)
{
    Console.WriteLine(i);
}
```

차이점은 컴파일러가 인라인 배열에 대한 알려진 정보를 활용할 수 있다는 것입니다. 다른 배열과 마찬가지로 인라인 배열을 사용할 수 있습니다. 인라인 배열을 선언하는 방법에 대한 자세한 내용은 형식에 대한 언어 참조를 [struct 참조하세요](#).

## 실험적 특성

형식, 메서드 또는 어셈블리를 실험적 기능을 나타내기 위해 표시 `System.Diagnostics.CodeAnalysis.ExperimentalAttribute` 할 수 있습니다. 메서드에 `ExperimentalAttribute` 주석이 추가된 형식에 액세스하면 컴파일러에서 경고가 발생합니다. 특성으로 `Experimental` 표시된 어셈블리에 포함된 모든 형식은 실험적입니다. 컴파일러에서 읽은 일반 특성 또는 기능 사양에 대한 문서에서 자세히 읽을 수 있습니다.

## 인터셉터

### ⚠ 경고

인터셉터는 C# 12를 사용하여 미리 보기 모드에서 사용할 수 있는 실험적 기능입니다. 이 기능은 향후 릴리스에서 호환성이 손상되거나 제거될 수 있습니다. 따라서 프로덕션 또는 릴리스된 애플리케이션에는 권장되지 않습니다.

인터셉터를 사용하려면 사용자 프로젝트에서 속성을

`<InterceptorsPreviewNamespaces>` 지정해야 합니다. 인터셉터를 포함할 수 있는 네임스페이스 목록입니다.

예:

```
<InterceptorsPreviewNamespaces>$(<InterceptorsPreviewNamespaces>);Microsoft.AspNetCore.Http.Generated;MyLibrary.Generated</InterceptorsPreviewNamespaces>
```

인터셉터(*interceptor*)는 컴파일 시간에 자체 호출을 사용하여 인터셉트 가능한 메서드에 대한 호출을 선언적으로 대체할 수 있는 메서드입니다. 이 대체는 인터셉터에서 가로채는 호출의 원본 위치를 선언하게 함으로써 발생합니다. 인터셉터는 소스 생성기와 같이 컴파일에 새 코드를 추가하여 기존 코드의 의미 체계를 변경하는 제한된 기능을 제공합니다.

기존 소스 컴파일에 코드를 추가하는 대신 소스 생성기의 일부로 인터셉터를 사용하여 수정합니다. 원본 생성기는 인터셉터 메서드를 호출하여 절편 가능 메서드에 대한 호출을 대체합니다.

인터셉터를 실험하려는 경우 기능 사양을 읽어 자세히 알아볼 수 있습니다. 이 기능을 사용하는 경우 이 실험적 기능에 대한 기능 사양의 변경 내용을 최신 상태로 유지해야 합니다. 기능이 완료된 경우 이 사이트에 대한 추가 지침을 추가합니다.

# 참고 항목

- .NET 8의 새로운 기능

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

ଓ 설명서 문제 열기

▣ 제품 사용자 의견 제공

# C# 11의 새로운 기능

아티클 • 2024. 03. 15.

C# 11에서 추가된 기능은 다음과 같습니다.

- 원시 문자열 리터럴
- 일반 수학 지원
- 제네릭 특성
- UTF-8 문자열 리터럴
- 문자열 보간 식의 줄
- 목록 패턴
- 파일-로컬 형식
- 필수 멤버
- 자동 기본 구조체
- 상수의 패턴 일치 `Span<char> string`
- 확장된 `nameof` 범위
- Numeric IntPtr
- ref 필드 및 scoped ref
- 대리자로의 메서드 그룹 변환 개선
- 경고 웨이브 7

C# 11은 .NET 7에서 지원됩니다. 자세한 내용은 [C# 언어 버전 관리](#)를 참조하세요.

.NET 다운로드 페이지에서 [최신 .NET 7 SDK를 다운로드할](#) 수 있습니다. .NET 7 SDK를 포함하는 Visual Studio 2022를 [다운로드](#) 할 수도 있습니다.

## ① 참고

이러한 기능에 대한 피드백에 관심이 있습니다. 이러한 새로운 기능에 문제가 있는 경우 `dotnet/roslyn` [리포지토리에서](#) 새 문제를 [만듭니다](#).

## 일반 특성

기본 클래스가 `System.Attribute`인 [제네릭 클래스](#)를 선언할 수 있습니다. 이 기능은 매개 변수가 필요한 `System.Type` 특성에 대한 보다 편리한 구문을 제공합니다. 이전에는 `Type`을 생성자 매개 변수로 사용하는 특성을 만들어야 했습니다.

C#

```
// Before C# 11:  
public class TypeAttribute : Attribute
```

```
{  
    public TypeAttribute(Type t) => ParamType = t;  
  
    public Type ParamType { get; }  
}
```

그리고 특성을 적용하려면 `typeof` 연산자를 사용합니다.

C#

```
[TypeAttribute(typeof(string))]  
public string Method() => default;
```

이 새로운 기능을 사용하여 제네릭 특성을 대신 만들 수 있습니다.

C#

```
// C# 11 feature:  
public class GenericAttribute<T> : Attribute { }
```

그런 다음, 특성을 사용할 형식 매개 변수를 지정합니다.

C#

```
[GenericAttribute<string>()]  
public string Method() => default;
```

특성을 적용할 때 모든 형식 매개 변수를 제공해야 합니다. 즉, 제네릭 형식은 [완전히 생성](#)되어야 합니다. 위의 예제에서는 특성에 인수가 없으므로 빈 괄호( ( 및 ))를 생략할 수 있습니다.

C#

```
public class GenericType<T>  
{  
    [GenericAttribute<T>()] // Not allowed! generic attributes must be fully  
    // constructed types.  
    public string Method() => default;  
}
```

형식 인수는 `typeof` 연산자와 동일한 제한을 충족해야 합니다. 메타데이터 주석이 필요한 형식은 허용되지 않습니다. 예를 들어 다음 형식은 형식 매개 변수로 허용되지 않습니다.

- `dynamic`
- `string?` (또는 nullable 참조 형식)

- `(int X, int Y)` (또는 C# 튜플 구문을 사용하는 다른 튜플 형식).

이러한 형식은 메타데이터에서 직접 표시되지 않습니다. 여기에는 형식을 설명하는 주석이 포함됩니다. 모든 경우에 기본 형식을 대신 사용할 수 있습니다.

- `dynamic`에 대한 `object`.
- `string` 다음을 사용하지 않습니다. `string?`
- `ValueTuple<int, int>` 다음을 사용하지 않습니다. `(int X, int Y)`

## 일반 수학 지원

일반 수학 지원을 사용하도록 설정하는 몇 가지 언어 기능이 있습니다.

- `static virtual` 인터페이스의 멤버
- 검사 사용자 정의 연산자
- 완화된 시프트 연산자
- 부호 없는 오른쪽 시프트 연산자

인터페이스에 멤버를 추가하거나 `static virtual` 멤버를 추가하여 `static abstract` 오버로드 가능한 연산자, 기타 정적 멤버 및 정적 속성을 포함하는 인터페이스를 정의할 수 있습니다. 이 기능의 기본 시나리오는 제네릭 형식에서 수학 연산자를 사용하는 것입니다. 예를 들어 구현하는 `System.IAdditionOperators<TSelf, TOther, TResult>` 형식에서 인터페이스를 구현할 수 있습니다 `operator +`. 다른 인터페이스는 다른 수학 연산 또는 잘 정의된 값을 정의합니다. 인터페이스에 대한 문서의 새 구문에 [대해 알아볼 수 있습니다](#). 메서드를 포함하는 `static virtual` 인터페이스는 일반적으로 [제네릭 인터페이스입니다](#). 또한 대부분은 형식 매개 변수가 선언된 인터페이스를 구현하는 제약 조건을 선언합니다.

자습서 [정적 추상 인터페이스 멤버 탐색](#) 또는 [.NET 6의 미리 보기 기능 – 일반 수학 ↗](#) 블로그 게시물에서 이 기능에 대해 자세히 알아보고 직접 사용해 볼 수 있습니다.

제네릭 수학은 언어에 대한 다른 요구 사항을 만들었습니다.

- **부호 없는 오른쪽 시프트 연산자:** C# 11 이전에는 부호 없는 오른쪽 시프트를 적용하려면 부호 없는 정수 형식을 부호 없는 형식으로 캐스팅하고, 시프트를 수행한 다음, 결과를 다시 부호 있는 형식으로 캐스팅해야 합니다. C# 11부터 서명되지 않은 Shift 연산자를 사용할 `>>>` 수 있습니다.
- **완화된 교대 근무 연산자 요구 사항:** C# 11은 두 번째 피연산자가 암시적으로 변환할 수 `int` 있어야 `int` 한다는 요구 사항을 제거합니다. 이 변경을 통해 제네릭 수학 인터페이스를 구현하는 형식을 이러한 위치에서 사용할 수 있습니다.
- **검사 및 검사 없는 사용자 정의 연산자:** 개발자는 이제 산술 연산자를 정의하고 `unchecked` 산술 연산자를 정의 `checked` 할 수 있습니다. 컴파일러는 현재 컨텍스트

에 따라 올바른 변형에 대한 호출을 생성합니다. 연산자에 대한 `checked` 자세한 내용은 산술 연산자 문서에서 확인할 수 있습니다.

## IntPtr 숫자 및 UIntPtr

`nint` 이제 별칭과 `nuint` `System.UIntPtr` 형식을 `System.IntPtr` 각각 지정합니다.

## 문자열 보간의 줄 바꿈

이제 문자열 보간을 위한 `{` 및 `}` 문자 안의 텍스트가 여러 줄에 걸쳐 있을 수 있습니다. `{`과 `}` 마커 사이의 텍스트는 C#으로 구문 분석됩니다. 줄 바꿈을 포함한 모든 합법적인 C#은 허용됩니다. 이 기능을 사용하면 패턴 일치 `switch` 식 또는 LINQ 쿼리와 같이 더 긴 C# 식을 사용하는 문자열 보간을 더 쉽게 읽을 수 있습니다.

언어 참조의 [문자열 보간](#) 문서에서 줄 바꿈 기능에 대해 자세히 알아볼 수 있습니다.

## 목록 패턴

목록 패턴은 목록 또는 배열의 요소 시퀀스에 맞게 패턴 일치를 확장합니다. 예를 들어 `sequence` 가 배열 또는 3개의 정수(1, 2, 3)의 목록인 경우 `sequence is [1, 2, 3]` 은 `true`입니다. 상수, 형식, 속성, 관계형 패턴을 비롯한 모든 패턴을 사용하여 요소를 일치시킬 수 있습니다. 디스크드 패턴(`_`)은 단일 요소와 일치하며 새 범위 패턴(`...`)은 0개 이상의 요소 시퀀스와 일치합니다.

언어 참조의 [패턴 일치](#) 문서에서 목록 패턴에 대한 자세한 내용을 알아볼 수 있습니다.

## 대리자로의 메서드 그룹 변환 개선

이제 [메서드 그룹 변환](#)의 C# 표준에 다음 항목이 포함됩니다.

- 변환은 이러한 참조가 이미 포함된 기존 대리자 인스턴스를 사용할 수 있도록 허용됩니다(필수는 아님).

이전 버전의 표준에서는 컴파일러가 메서드 그룹 변환을 위해 만든 대리자 객체를 다시 사용할 수 없습니다. C# 11 컴파일러는 메서드 그룹 변환에서 만든 대리자 객체를 캐시하고 해당 단일 대리자 객체를 다시 사용합니다. 이 기능은 Visual Studio 2022 버전 17.2에서 미리 보기 기능으로, .NET 7 미리 보기 2에서 처음 사용할 수 있었습니다.

## 원시 문자열 리터럴

원시 문자열 리터럴은 문자열 리터럴의 새 형식입니다. 원시 문자열 리터럴은 이스케이프 시퀀스를 요구하지 않고 공백, 새 줄, 포함된 따옴표, 기타 특수 문자를 비롯한 임의의 텍스트를 포함할 수 있습니다. 원시 문자열 리터럴은 세 개 이상의 큰따옴표(""""") 문자로 시작합니다. 이는 동일한 수의 큰따옴표로 끝납니다. 일반적으로 원시 문자열 리터럴은 한 줄에 세 개의 큰따옴표를 사용하여 문자열을 시작하고 별도의 줄에 세 개의 큰따옴표를 사용하여 문자열을 종료합니다. 여는 따옴표와 닫는 따옴표 앞의 줄 바꿈은 최종 콘텐츠에 포함되지 않습니다.

C#

```
string longMessage = """
    This is a long message.
    It has several lines.
        Some are indented
            more than others.
    Some should start at the first column.
    Some have "quoted text" in them.
""";
```

닫는 큰따옴표 왼쪽의 공백은 문자열 리터럴에서 제거됩니다. 원시 문자열 리터럴을 문자열 보간과 결합하여 출력 텍스트에 중괄호를 포함할 수 있습니다. 여러 \$ 문자는 보간을 시작하고 종료하는 연속 중괄호 수를 나타냅니다.

C#

```
var location = $$"""
    You are at {{Longitude}}, {{Latitude}}
""";
```

앞의 예제에서는 보간을 시작하고 종료하는 두 개의 중괄호를 지정합니다. 세 번째 반복된 여는 중괄호와 닫는 중괄호가 출력 문자열에 포함됩니다.

[프로그래밍 가이드의 문자열](#)에 대한 문서의 원시 문자열 리터럴과 [문자열 리터럴 및 보간된 문자열](#)에 대한 언어 참조 문서에 대해 자세히 알아볼 수 있습니다.

## 자동 기본 구조체

C# 11 컴파일러는 형식의 `struct` 모든 필드가 생성자 실행의 일부로 기본값으로 초기화되도록 합니다. 이 변경은 생성자에 의해 초기화되지 않은 모든 필드 또는 자동 속성이 컴파일러에 의해 자동으로 초기화됨을 의미합니다. 생성자가 모든 필드를 확실히 할당하지 않는 구조체는 이제 컴파일되고 명시적으로 초기화되지 않은 필드는 기본값으로 설정됩니다. 이 변경 내용이 구조체에 대한 문서의 구조체 초기화에 [미치는 영향에 대해 자세히 확인할 수 있습니다.](#)

## 패턴 일치 `Span<char>` 또는 `ReadOnlySpan<char>` 상수 `string`

여러 릴리스에 패턴 일치를 사용하여 특정 상수 값이 있는지 `string` 테스트할 수 있었습니다. 이제 동일한 패턴 일치 논리를 변수 또는 변수와 함께 사용할 수 있습니다

`Span<char>` `ReadOnlySpan<char>`.

## 확장 `nameof` 범위

형식 매개 변수 이름과 매개 변수 이름은 이제 해당 메서드의 특성 선언에 `nameof` 있는 식에서 [사용될 때 범위에 포함](#) 됩니다. 이 기능은 연산자를 `nameof` 사용하여 메서드 또는 매개 변수 선언의 특성에서 메서드 매개 변수의 이름을 지정할 수 있습니다. 이 기능은 nullable 분석을 [위한](#) 특성을 추가하는 데 가장 유용합니다.

## UTF-8 문자열 리터럴

문자열 리터럴에 `u8` 접미사를 지정하여 UTF-8 문자 인코딩을 지정할 수 있습니다. 애플리케이션에 UTF-8 문자열이 필요한 경우 HTTP 문자열 상수 또는 유사한 텍스트 프로토콜의 경우 이 기능을 사용하여 UTF-8 문자열 만들기를 간소화할 수 있습니다.

기본 제공 참조 형식에 대한 문서의 문자열 리터럴 섹션에서 UTF-8 문자열 리터럴에 대해 [자세히 알아볼 수 있습니다](#).

## 필수 멤버

속성 및 필드에 한정자를 [추가하여](#) `required` 생성자와 호출자를 적용하여 해당 값을 초기화할 수 있습니다. `System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute` 생성자에 추가하여 생성자가 필요한 모든 멤버를 초기화한다는 것을 컴파일러에 알릴 수 있습니다.

필요한 멤버에 대한 자세한 내용은 속성 문서의 init 전용 [섹션을 참조하세요](#).

## `ref` 필드 및 `ref scoped` 변수

에서 필드를 선언 `ref` 할 `ref struct` 수 있습니다. 특수 특성이나 숨겨진 내부 형식이 없는 형식 `System.Span<T>` 을 지원합니다.

선언 `ref` 에 `scoped` 한정자를 추가할 수 있습니다. 이렇게 하면 참조가 [이스케이프할 수 있는 범위](#) 가 제한됩니다.

# 파일 로컬 형식

C# 11부터 액세스 한정자를 사용하여 `file` 표시 범위가 선언된 원본 파일로 범위가 지정된 형식을 만들 수 있습니다. 이 기능은 원본 생성기 작성자가 명명 충돌을 방지하는 데 도움이 됩니다. 언어 참조의 파일 범위 형식에 대한 문서에서 이 기능에 대해 자세히 알아볼 수 있습니다.

## 참고 항목

- [.NET 7의 새로운 기능](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

☞ [설명서 문제 열기](#)

☞ [제품 사용자 의견 제공](#)

# C# 10의 새로운 기능

아티클 • 2023. 04. 08.

C# 10에는 다음과 같은 기능과 C# 언어 개선 사항이 추가되었습니다.

- 레코드 구조체
- 구조체 형식 개선
- 보간된 문자열 처리기
- global using 지시문
- 파일 범위 네임스페이스 선언
- 확장 속성 패턴
- 람다 식 개선
- const 보간된 문자열 허용
- 레코드 형식은 `ToString()`을 봉인할 수 있음
- 한정된 할당 개선
- 동일한 분해에서 할당과 선언을 모두 허용
- 메서드의 `AsyncMethodBuilder` 특성을 허용
- `CallerArgumentExpression` 특성
- 향상된 `#line pragma`
- 경고 웨이브 6

C# 10은 .NET 6에서 지원됩니다. 자세한 내용은 [C# 언어 버전 관리](#)를 참조하세요.

.NET 다운로드 페이지 [\[링크\]](#)에서 최신 .NET 6 SDK를 다운로드할 수 있습니다. .NET 6 SDK를 포함하는 [Visual Studio 2022](#) [\[링크\]](#)를 다운로드할 수도 있습니다.

## ① 참고

이러한 기능에 대한 피드백에 관심이 있습니다. 이러한 새로운 기능에 문제가 있는 경우 [dotnet/roslyn](#) [\[링크\]](#) 리포지토리에서 새 문제를 [\[링크\]](#) 만듭니다.

## 레코드 구조체

`record struct` 또는 `readonly record struct` 선언을 사용해 값 형식 레코드를 선언할 수 있습니다. 이제 `record`가 `record class` 선언이 있는 참조 형식임을 명시할 수 있습니다.

## 구조체 형식 개선

C# 10에서는 다음과 같은 구조체 형식 관련 개선이 도입되었습니다.

- 구조체 형식에서 매개 변수가 없는 인스턴스 생성자를 선언하고 해당 선언에서 인스턴스 필드 또는 속성을 초기화할 수 있습니다. 자세한 내용은 [구조체 형식](#) 문서의 [구조체 초기화 및 기본값](#) 섹션을 참조하세요.
- `with` 식의 왼쪽 피연산자는 구조체 형식 또는 무명(참조) 형식일 수 있습니다.

## 보간된 문자열 처리기

[보간된 문자열 식](#)에서 결과 문자열을 빌드하는 형식을 만들 수 있습니다. .NET 라이브러리는 많은 API에서 이 기능을 사용합니다. [이 자습서에 따라](#) 빌드할 수 있습니다.

## 전역 using 지시문

지시문이 컴파일의 모든 소스 파일에 적용되도록 컴파일러에 지시하려면 [using 지시문](#)에 `global` 한정자를 추가할 수 있습니다. 이는 일반적으로 프로젝트의 모든 소스 파일입니다.

## 파일 범위 네임스페이스 선언

모든 후속 선언이 선언된 네임스페이스의 멤버임을 선언하려면 새로운 형식의 [namespace 선언](#)을 사용할 수 있습니다.

```
C#
```

```
namespace MyNamespace;
```

이 새 구문을 사용하면 `namespace` 선언에서 가로와 세로 공간이 모두 절약됩니다.

## 확장 속성 패턴

C# 10부터 속성 패턴 내에서 중첩된 속성 또는 필드를 참조할 수 있습니다. 예를 들어 다음의 형식 패턴은

```
C#
```

```
{ Prop1.Prop2: pattern }
```

C# 10 이상에서 유효하며 다음과 동일합니다.

```
C#
```

```
{ Prop1: { Prop2: pattern } }
```

C# 8.0 이상

자세한 내용은 [확장 속성 패턴](#) 기능 제안 노트를 참조하세요. 속성 패턴에 대한 자세한 내용은 [패턴](#) 문서의 [속성 패턴 섹션](#)을 참조하세요.

## 람다 식 개선

C# 10에서는 람다 식이 처리되는 방식에 몇 가지 개선이 적용되었습니다.

- 람다 식이 컴파일러가 람다 식 또는 메서드 그룹으로부터 대리자 형식을 유추할 수 있는 [자연 형식](#)을 가질 수 있습니다.
- 컴파일러가 반환 형식을 유추할 수 없는 경우 람다 식이 [반환 형식](#)을 선언할 수 있습니다.
- 람다 식에 [특성을](#) 적용할 수 있습니다.

이러한 기능은 람다 식을 메서드 및 로컬 함수와 더 비슷하게 만들어 줍니다. 대리자 형식의 변수를 선언하지 않고도 람다 식을 더 쉽게 사용할 수 있으며 새로운 ASP.NET Core 최소 API와 더 원활하게 작동합니다.

## 보간된 상수 문자열

C# 10에서는 모든 자리 표시자가 그 자체로 상수 문자열인 경우 [문자열 보간](#)을 사용하여 `const` 문자열을 초기화할 수 있습니다. 문자열 보간을 사용하면 애플리케이션에서 사용되는 상수 문자열을 작성할 때 보다 읽기 쉬운 상수 문자열을 만들 수 있습니다. 해당 상수는 런타임에 문자열로 변환되기 때문에 자리 표시자 식은 숫자 상수일 수 없습니다. 현재 문화권이 해당 문자열 표현에 영향을 줄 수 있습니다. [const 식](#)에 대한 언어 참조에서 자세히 알아보세요.

## 레코드 형식은 `ToString`을 봉인할 수 있음

C# 10에서는 레코드 형식에서 `ToString`을 재정의할 때 `sealed` 한정자를 추가할 수 있습니다. `ToString` 메서드를 봉인하면 컴파일러가 파생된 레코드 형식에 대해 `ToString` 메서드를 합성할 수 없습니다. `sealed ToString` 파생된 모든 레코드 형식이 `ToString` 공통 기본 레코드 형식에 정의된 메서드를 사용하도록 합니다. [레코드](#)에 대한 문서에서 이 기능에 대해 자세히 알아볼 수 있습니다.

## 동일한 분해의 할당 및 선언

이 변경을 통해 이전 버전 C#의 제한이 제거됩니다. 이전에는 분해에서 모든 값을 기준 변수에 할당하거나 새로 선언된 변수를 초기화할 수 있었습니다.

```
C#  
  
// Initialization:  
(int x, int y) = point;  
  
// assignment:  
int x1 = 0;  
int y1 = 0;  
(x1, y1) = point;
```

C# 10에서는 이 제한이 제거됩니다.

```
C#  
  
int x = 0;  
(x, int y) = point;
```

## 한정된 할당 개선

C# 10 이전에는 한정된 할당과 null 상태 분석이 가상성이 경고를 생성하는 시나리오가 많았습니다. 대표적인 시나리오로 부울 상수와 비교하는 경우, `if` 문에서 `true` 또는 `false` 문에만 있는 변수에 액세스하는 경우, null 병합 식을 들 수 있습니다. 이러한 시나리오는 이전 버전의 C#에서는 경고를 생성했지만 C# 10에서는 경고를 생성하지 않습니다.

```
C#  
  
string representation = "N/A";  
if ((c != null && c.GetDependentValue(out object obj)) == true)  
{  
    representation = obj.ToString(); // undesired error  
}  
  
// Or, using ?.  
if (c?.GetDependentValue(out object obj) == true)  
{  
    representation = obj.ToString(); // undesired error  
}  
  
// Or, using ??  
if (c?.GetDependentValue(out object obj) ?? false)  
{  
    representation = obj.ToString(); // undesired error  
}
```

이 개선으로 인해 한정된 할당 및 null 상태 분석에 대한 경고가 더 정확해졌습니다.

## 메서드의 AsyncMethodBuilder 특성 허용

C# 10 이상에서는 지정된 작업과 유사한 형식을 반환하는 모든 메서드에 대해 메서드 작성기 형식을 지정하는 것에 더해, 단일 메서드에 대해 여러 비동기 메서드 작성기를 지정할 수 있습니다. 사용자 지정 비동기 메서드 작성기를 이용하면, 지정된 메서드가 사용자 지정 작성기를 활용할 수 있는 고급 성능 조정 시나리오를 사용할 수 있습니다.

자세한 내용은 컴파일러에서 읽은 특성에 대한 문서에서 [AsyncMethodBuilder](#)에 대한 섹션을 참조하세요.

## CallerArgumentExpression 특성 진단

[System.Runtime.CompilerServices.CallerArgumentExpressionAttribute](#)를 사용하여, 컴파일러가 다른 인수의 텍스트 표현으로 바꿀 매개 변수를 지정할 수 있습니다. 이 기능은 라이브러리가 보다 구체적인 진단을 만들 수 있도록 지원합니다. 다음 코드는 조건을 테스트합니다. 조건이 false이면 예외 메시지가 `condition`으로 전달된 인수의 텍스트 표현을 포함합니다.

C#

```
public static void Validate(bool condition,
[CallerArgumentExpression("condition")] string? message=null)
{
    if (!condition)
    {
        throw new InvalidOperationException($"Argument failed validation:
<{message}>");
    }
}
```

이 기능은 언어 참조 섹션의 [호출자 정보 특성](#)에 관한 문서에서 자세히 알아볼 수 있습니다.

## 향상된 #line pragma

C# 10은 `#line` pragma에 대한 새 형식을 지원합니다. 새 형식을 직접 사용할 일은 없겠지만, 그 영향은 확인할 수 있습니다. 이로 인해 Razor와 같은 DSL(Domain-Specific Language)에서 더욱 정교한 출력을 얻을 수 있습니다. Razor 엔진은 이러한 개선 사항을 사용하여 디버깅 경험을 향상합니다. 디버거가 Razor 소스를 보다 정확하게 강조 표시하

는 것을 볼 수 있습니다. 새로운 구문에 대해 자세히 알아보려면 언어 참조의 [전처리기 지시문](#)에 관한 문서를 참조하세요. Razor 기반 예제는 [기능 사양](#)을 참조하세요.

# C# 컴파일러의 호환성이 손상되는 변경에 대해 알아봅니다

아티클 • 2023. 05. 10.

C# 10 릴리스 이후 호환성이 손상되는 변경 내용은 여기에서 확인할 수 [있습니다](#).

Roslyn [\[↗\]](#) 팀은 C# 및 Visual Basic 컴파일러의 호환성이 손상되는 변경 목록을 유지 관리 합니다. 변경 내용에 대한 정보는 GitHub 리포지토리의 다음 링크에서 확인할 수 있습니다.

- C# 10.0/.NET 6에서 Roslyn의 호환성이 손상되는 변경 [\[↗\]](#)
- .NET 5 이후 Roslyn의 호환성이 손상되는 변경 [\[↗\]](#)
- .NET 5 및 C# 9.0부터 도입된 VS2019 버전 16.8의 호환성이 손상되는 변경 [\[↗\]](#)
- VS2019 대비 VS2019 업데이트 1 이상의 호환성이 손상되는 변경 [\[↗\]](#)
- VS2017 이후 호환성이 손상되는 변경(C# 7) [\[↗\]](#)
- Roslyn 2.\* (VS2017)과 달라진 Roslyn 3.0 (VS2019)의 호환성이 손상되는 변경 [\[↗\]](#)
- Roslyn 1.\* (VS2015) 및 네이티브 C# 컴파일러 (VS2013 및 이전)와 달라진 Roslyn 2.0 (VS2017)의 호환성이 손상되는 변경 [\[↗\]](#)
- 네이티브 C# 컴파일러 (VS2013 및 이전)와 달라진 Roslyn 1.0 (VS2015)의 호환성이 손상되는 변경 [\[↗\]](#)
- C# 6의 유니코드 버전 변경 [\[↗\]](#)

# C#의 역사

아티클 • 2024. 04. 17.

이 문서에서는 C# 언어의 각 주요 릴리스에 대한 기록을 제공합니다. C# 팀은 계속해서 새로운 기능을 혁신하고 추가하고 있습니다. 예정된 릴리스에서 고려되는 기능을 비롯한 자세한 언어 기능 상태는 GitHub의 [dotnet/roslyn 리포지토리](#)에서 확인할 수 있습니다.

## ① 중요

C# 언어는 C# 사양이 일부 기능에 대해 표준 라이브러리로 정의하는 형식 및 메서드를 사용합니다. .NET 플랫폼은 다양한 패키지에서 이러한 유형과 메서드를 제공합니다. 한 가지 예는 예외 처리입니다. 모든 `throw` 문 또는 식은 `throw`된 개체가 [Exception](#)에서 파생되는지 확인합니다. 마찬가지로 모든 `catch`는 발견되는 형식이 [Exception](#)에서 파생되는지 확인합니다. 각 버전은 새 요구 사항을 추가할 수 있습니다. 이전 환경에서 최신 언어 기능을 사용하려면 특정 라이브러리를 설치해야 합니다. 이러한 종속성은 각 특정 버전에 대한 페이지에서 설명합니다. 이 종속성의 배경은 [언어 및 라이브러리 간 관계](#)에서 자세히 알아볼 수 있습니다.

## C# 버전 12

릴리스 날짜: 2023년 11월

C# 12에서 추가된 기능은 다음과 같습니다.

- **기본 생성자** - 모든 `class` 또는 `struct` 형식으로 기본 생성자를 만들 수 있습니다.
- **컬렉션 식** - 모든 컬렉션을 확장하기 위해 스프레드 연산자()..를 포함하여 컬렉션 식을 지정하는 새 구문입니다.
- **인라인 배열** - 인라인 배열을 사용하면 형식에서 고정 크기의 배열을 `struct` 만들 수 있습니다.
- **람다 식**의 선택적 매개 변수 - 람다 식의 매개 변수에 대한 기본값을 정의할 수 있습니다.
- **ref readonly 매개 변수** - `ref readonly` 변수 또는 `in` 매개 변수를 사용할 `ref` 수 있는 API에 대한 명확성을 높일 수 있습니다.
- **모든 형식**의 별칭 - 별칭 지시문을 사용하여 `using` 명명된 형식뿐만 아니라 모든 형식의 별칭을 지정할 수 있습니다.
- **실험적 특성** - 실험적 기능을 나타냅니다.

또한 [인터셉터](#)는 미리 보기 기능으로 릴리스되었습니다.

전반적으로 C# 12는 C# 코드를 보다 생산적으로 작성할 수 있는 새로운 기능을 제공합니다. 이미 알고 있는 구문은 더 많은 위치에서 사용할 수 있습니다. 다른 구문을 사용하면 관련 개념에 대한 일관성을 사용할 수 있습니다.

## C# 버전 11

릴리스 날짜: 2022년 11월

C# 11에서 추가된 기능은 다음과 같습니다.

- 원시 문자열 리터럴
- 일반 수학 지원
- 제네릭 특성
- UTF-8 문자열 리터럴
- 문자열 보간 식의 줄
- 목록 패턴
- 파일-로컬 형식
- 필수 멤버
- 자동 기본 구조체
- 상수의 패턴 일치 `Span<char>string`
- 확장된 `nameof` 범위
- Numeric IntPtr
- ref 필드 및 scoped ref
- 대리자로의 메서드 그룹 변환 개선
- 경고 웨이브 7

C# 11에서는 제네릭 수학 및 해당 목표를 지원하는 몇 가지 기능을 소개합니다. 모든 숫자 형식에 대해 숫자 알고리즘을 한 번 작성할 수 있습니다. 필수 멤버 및 자동 기본 구조체와 같이 형식을 `struct` 더 쉽게 사용할 수 있는 기능이 더 많이 있습니다. 원시 문자열 리터럴, 문자열 보간의 줄 바꿈 및 UTF-8 문자열 리터럴을 사용하면 문자열 작업을 더 쉽게 할 수 있습니다. 파일 로컬 형식과 같은 기능을 사용하면 원본 생성기가 더 간단해질 수 있습니다. 마지막으로 목록 패턴은 패턴 일치에 대한 지원을 더 추가합니다.

## C# 버전 10

릴리스 날짜: 2021년 11월

C# 10에는 다음과 같은 기능과 C# 언어 개선 사항이 추가되었습니다.

- 레코드 구조체
- 구조체 형식 개선
- 보간된 문자열 처리기

- global using 지시문
- 파일 범위 네임스페이스 선언
- 확장 속성 패턴
- 람다 식 개선
- const 보간된 문자열 허용
- 레코드 형식은 `ToString()`을 봉인할 수 있음
- 한정된 할당 개선
- 동일한 분해에서 할당과 선언을 모두 허용
- 메서드의 `AsyncMethodBuilder` 특성을 허용
- `CallerArgumentExpression` 특성
- 향상된 `#line pragma`

미리 보기 모드에서 더 많은 기능을 사용할 수 있었습니다. 이러한 기능을 사용하려면 프로젝트에서 다음으로 `Preview` 설정 `<LangVersion>` 해야 합니다.

- 이 문서의 뒷부분에 있는 제네릭 특성입니다.
- 인터페이스의 정적 추상 멤버.

C# 10에서는 의식 제거, 알고리즘에서 데이터 분리, .NET 런타임 성능 향상 등의 테마에 대한 작업을 계속합니다.

대부분의 기능은 동일한 개념을 표현하기 위해 코드를 더 적게 입력한다는 것을 의미합니다. 레코드 구조체는 레코드 클래스와 동일한 많은 메서드를 합성합니다. 구조체 및 무명 형식은 식을 지원 합니다. 전역 `using` 지시문 및 파일 범위 네임스페이스 선언은 종속성 및 네임스페이스 조직을 보다 명확하게 표현한다는 것을 의미합니다. 람다를 개선하면 람다 식이 사용되는 위치를 더 쉽게 선언할 수 있습니다. 새로운 속성 패턴 및 분해 개선은 보다 간결한 코드를 만듭니다.

보간된 새 문자열 처리기 및 `AsyncMethodBuilder` 동작은 성능을 향상시킬 수 있습니다. 이러한 언어 기능은 .NET 런타임에 적용되어 .NET 6의 성능 향상을 달성했습니다.

또한 C# 10은 .NET 릴리스의 연간 주기로의 전환을 더 많이 표시합니다. 모든 기능을 연간 기간에 완료할 수 있는 것은 아니므로 C# 10에서 몇 가지 "미리 보기" 기능을 사용해 볼 수 있습니다. 인터페이스에서 제네릭 특성과 정적 추상 멤버를 모두 사용할 수 있지만 이러한 미리 보기 기능은 최종 릴리스 전에 변경될 수 있습니다.

## C# 버전 9

릴리스 날짜: 2020년 11월

C# 9는 .NET 5와 함께 릴리스되었습니다. .NET 5 릴리스를 대상으로 하는 모든 어셈블리의 기본 언어 버전입니다. 여기에는 다음과 같은 새롭고 향상된 기능이 포함되어 있습니다.

- 레코드
- setter만 초기화
- 최상위 문
- 패턴 일치 향상된 기능: 관계형 패턴 및 논리 패턴
- 성능 및 interop
  - 네이티브 크기의 정수
  - 함수 포인터
  - localsinit 플래그 내보내기 표시 안 함
  - 모듈 이니셜라이저
  - 부분 메서드에 대한 새로운 기능
- 기능 맞춤 및 완료
  - 대상으로 형식화된 new 식
  - static 익명 함수
  - 대상 형식 조건식
  - 공변 반환 형식
  - foreach 루프에 대한 확장 GetEnumerator 지원
  - 람다 무시 항목 매개 변수
  - 로컬 함수의 특성

C# 9는 이전 릴리스의 세 가지 테마인 공식 제거, 알고리즘에서 데이터 분리, 더 많은 위치에서 더 많은 패턴 제공을 계속합니다.

**최상위 문**은 기본 프로그램을 읽기가 더 간단하다는 것을 의미합니다. 공식 업무에는 네 임스페이스, `Program` 클래스 및 `static void Main()`이 모두 필요하지 않습니다.

이 소개 `records`에서는 같음을 위해 값 의미 체계를 따르는 참조 형식에 대한 간결한 구문을 제공합니다. 이러한 형식을 사용하여 일반적으로 최소 동작을 정의하는 데이터 컨테이너를 정의합니다. **Init 전용 setter**는 레코드에서 비파괴적 변형(`with` 식)에 대한 기능을 제공합니다. 또한 C# 9는 파생 레코드가 가상 메서드를 재정의하고 기본 메서드의 반환 형식에서 파생된 형식을 반환할 수 있도록 **공변 반환 형식**을 추가합니다.

**패턴 일치** 기능은 여러 가지 방법으로 확장되었습니다. 숫자 형식은 이제 범위 패턴을 지원합니다. 패턴은 `and`, `or` 및 `not` 패턴을 사용하여 결합할 수 있습니다. 더 복잡한 패턴을 명확히 하기 위해 괄호를 추가할 수 있습니다.

C# 9에는 새로운 패턴 일치 개선 사항이 포함되어 있습니다.

- **개체가 특정 형식과 일치하는 형식 패턴**
- **괄호로 묶인 패턴**은 패턴 조합의 우선 순위를 적용하거나 강조합니다.
- **결합 and 패턴**은 두 패턴이 모두 일치해야 합니다.
- **분리 or 패턴**은 패턴 중 하나가 일치해야 합니다.
- **부정 not 패턴**에서는 패턴이 일치하지 않아야 합니다.

- **관계형 패턴**은 입력이 지정된 상수보다 작거나, 크거나, 작거나 같거나, 크거나 같아야 합니다.

새로운 패턴은 패턴 구문을 보강합니다. 이 예제를 참조하십시오.

C#

```
public static bool IsLetter(this char c) =>
    c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';
```

선택적 괄호를 사용하여 `and`에 `or`보다 높은 우선 순위가 있음을 명확하게 지정합니다.

C#

```
public static bool IsLetterOrSeparator(this char c) =>
    c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z') or '.' or ',';
```

가장 일반적인 용도 중 하나는 새로운 null 검사 구문입니다.

C#

```
if (e is not null)
{
    // ...
}
```

`is` 패턴 식, `switch` 식, 중첩 패턴, `switch` 문의 `case` 레이블 패턴 등 패턴이 허용되는 모든 컨텍스트에서 새로운 패턴을 사용할 수 있습니다.

다른 기능 집합은 C#에서 고성능 컴퓨팅을 지원합니다.

- `nint` 및 `nuint` 형식은 대상 CPU에서 네이티브 크기 정수 형식을 모델링합니다.
- **함수 포인터**는 대리자 개체를 만드는 데 필요한 할당을 방지하면서 대리자 같은 기능을 제공합니다.
- `localsinit` 명령을 생략하여 명령을 저장할 수 있습니다.

## 성능 및 interop

다른 개선 사항 집합은 코드 생성기에서 기능을 추가하는 시나리오를 지원합니다.

- **모듈 아나要学会**는 어셈블리가 로드할 때 런타임에서 호출하는 메서드입니다.
- **부분 메서드**는 액세스 가능한 새 한정자 및 `void`가 아닌 반환 형식을 지원합니다. 이러한 경우 구현을 제공해야 합니다.

## 기능 마무리

C# 9는 코드 작성과 읽기 모두를 포함하여 개발자 생산성을 향상시키는 다른 많은 작은 기능을 추가합니다.

- 대상으로 형식화된 `new` 식
- `static` 익명 함수
- 대상으로 형식화된 조건식
- `foreach` 루프에 대한 확장 `GetEnumerator()` 지원
- 람다 식은 무시 항목 매개 변수를 선언할 수 있습니다.
- 로컬 함수에 특성을 적용할 수 있습니다.

C# 9 릴리스는 C#을 최신 범용 프로그래밍 언어로 유지하기 위한 작업을 계속합니다. 기능은 최신 워크로드 및 애플리케이션 유형을 계속 지원합니다.

## C# 버전 8.0

릴리스 날짜: 2019년 9월

C# 8.0은 특히 .NET Core C#을 대상으로 하는 첫 번째 주 릴리스입니다. 일부 기능은 새로운 CLR(공용 언어 런타임) 기능을 사용하며, 다른 기능은 .NET Core에만 추가된 라이브러리 형식을 사용합니다. C# 8.0은 다음 기능 및 향상된 기능을 C# 언어에 추가합니다.

- 읽기 전용 멤버
- 기본 인터페이스 메서드
- 패턴 일치 향상된 기능:
  - `Switch` 식
  - 속성 패턴
  - 튜플 패턴
  - 위치 패턴
- `using` 선언
- 정적 로컬 함수
- 삭제 가능한 `ref struct`
- `nullble` 참조 형식
- 비동기 스트림
- 인덱스 및 범위
- `null` 병합 할당
- 관리되지 않는 생성 형식
- 중첩 식의 `stackalloc`
- 보간된 약어 문자열의 향상된 기능

기본 인터페이스 멤버에는 CLR의 향상된 기능이 필요합니다. 해당 기능은 .NET Core 3.0 용 CLR에 추가되었습니다. 범위 및 인덱스와 비동기 스트림에는 .NET Core 3.0 라이브러리의 새 형식이 필요합니다. 인수 및 반환 값의 null 상태에 관한 의미 체계 정보를 제공하도록 라이브러리에 주석이 달린 경우 nullable 참조 형식은 컴파일러에서 구현되는 동안 훨씬 더 유용합니다. 해당 주석은 .NET Core 라이브러리에 추가됩니다.

## C# 버전 7.3

릴리스 날짜: 2018년 5월

C# 7.3 릴리스에는 두 개의 기본 테마가 있습니다. 하나의 테마는 안전한 코드의 성능을 안전하지 않은 코드만큼 향상할 수 있는 기능을 제공합니다. 두 번째 테마는 기존 기능에 대한 점진적인 개선을 제공합니다. 또한 새 컴파일러 옵션이 이 릴리스에 추가되었습니다.

다음 새로운 기능은 안전한 코드에 대해 향상된 성능의 테마를 지원합니다.

- 고정하지 않고 고정 필드에 액세스할 수 있습니다.
- `ref` 지역 변수를 다시 할당할 수 있습니다.
- `stackalloc` 배열에서 이니셜라이저를 사용할 수 있습니다.
- 패턴을 지원하는 모든 형식과 함께 `fixed` 문을 사용할 수 있습니다.
- 더 많은 제네릭 제약 조건을 사용할 수 있습니다.

기존 기능이 다음과 같이 개선되었습니다.

- 튜플 형식으로 `==` 및 `!=`를 테스트할 수 있습니다.
- 더 많은 위치에서 식 변수를 사용할 수 있습니다.
- 자동 구현 속성의 지원 필드에 특성을 연결할 수 있습니다.
- 인수가 다를 `in` 때의 메서드 확인이 향상되었습니다.
- 이제 오버로드 해결에 모호한 사례가 감소했습니다.

새 컴파일러 옵션은 다음과 같습니다.

- `-publicsign` - OSS(오픈 소스 소프트웨어)가 어셈블리에 서명할 수 있도록 설정합니다.
- `-pathmap` - 소스 디렉터리에 대한 매팅을 제공합니다.

## C# 버전 7.2

릴리스 날짜: 2017년 11월

C# 7.2는 몇 가지 작은 언어 기능을 추가했습니다.

- `stackalloc` 배열의 이니셜라이저
- 패턴을 지원하는 모든 형식과 함께 `fixed` 문을 사용합니다.
- 고정하지 않고 고정 필드에 액세스합니다.
- `ref` 지역 변수를 다시 할당합니다.
- `readonly struct` 형식을 선언하여 구조체가 변경이 불가능하고 `in` 매개 변수로 멤버 메서드에 전달되어야 함을 나타냅니다.
- 매개 변수에 `in` 한정자를 추가하여 인수가 참조로 전달되지만, 호출된 메서드에 의해 수정되지 않도록 지정합니다.
- 메서드 반환에 `ref readonly` 한정자를 사용하여 메서드가 참조로 값을 반환하지만, 해당 개체에 대한 쓰기를 허용하지 않음을 나타냅니다.
- `ref struct` 형식을 선언하여 구조체 형식이 관리되는 메모리에 직접 액세스하고 항상 스택에 할당되어야 함을 나타냅니다.
- 추가 제네릭 제약 조건을 사용합니다.
- **후행이 아닌 명명된 인수:**
  - 위치 인수는 명명된 인수를 따를 수 있습니다.
- 숫자 리터럴의 선행 밑줄:
  - 숫자 리터럴은 이제 인쇄된 숫자 앞에 선행 밑줄이 있을 수 있습니다.
- **private protected 액세스 한정자:**
  - `private protected` 액세스 한정자는 동일한 어셈블리의 파생된 클래스에 대해 액세스를 사용합니다.
- 조건 `ref` 식:
  - 이제 조건식(`? :`)의 결과가 참조일 수 있습니다.

## C# 버전 7.1

릴리스 날짜: 2017년 8월

C#은 C# 7.1과 함께 '포인트 릴리스'를 제공하기 시작했습니다. 이 버전은 언어 선택 구성 요소, 세 개의 새로운 언어 기능 및 새로운 컴파일러 동작을 추가했습니다.

이 릴리스의 새로운 언어 기능은 다음과 같습니다.

- **asyncMain 메서드**
  - 애플리케이션에 대한 진입점은 `async` 한정자를 가질 수 있습니다.
- **default 리터럴 식**
  - 대상 형식을 유추할 수 있는 경우 기본 값 식에서 기본 리터럴 식을 사용할 수 있습니다.
- **유추된 튜플 요소 이름**
  - 튜플 요소의 이름은 대부분의 경우에 튜플 초기화에서 유추할 수 있습니다.
- **제네릭 형식 매개 변수에서 패턴 일치**

- 형식이 제네릭 형식 매개 변수인 변수에서 패턴 일치 식을 사용할 수 있습니다.

마지막으로 컴파일러에는 두 가지 옵션 `-refout` 과 `-refonly` 해당 컨트롤 참조 어셈블리 생성이 있습니다.

## C# 버전 7.0

릴리스 날짜: 2017년 3월

C# 버전 7.0은 Visual Studio 2017과 함께 릴리스되었습니다. 이 버전은 C# 6.0의 맥락에서 몇 가지 진화적이고 멋진 물건을 가지고 있습니다. 다음은 새 기능 중 일부입니다.

- 외부 변수
- 튜플 및 분해
- 패턴 일치
- 로컬 함수
- 확장된 식 본문 멤버
- Ref locals
- Ref가 반환됩니다.

이러한 기능에는 다음이 포함됩니다.

- 무시 항목
- 이진 리터럴 및 자릿수 구분 기호
- Throw 식

이러한 모든 기능은 개발자를 위한 새로운 기능과 그 어느 때보다 클린 코드를 작성할 수 있는 기회를 제공합니다. 하이라이트는 `out` 키워드와 함께 사용할 변수의 선언을 압축하고 튜플을 통해 여러 개의 반환 값을 허용하는 것입니다. .NET Core는 이제 모든 운영 체제를 대상으로 하며 클라우드와 휴대성에 확실히 집중하고 있습니다. 이는 새로운 기능을 제공하는 것 외에도 언어 디자이너가 많이 생각하고 시간을 투자하게 만듭니다.

## C# 버전 6.0

릴리스 날짜: 2015년 7월

Visual Studio 2015와 함께 릴리스된 버전 6.0은 C# 프로그래밍의 생산성을 높이는 더 작은 기능을 많이 릴리스했습니다. 다음은 몇 가지 예입니다.

- 정적 가져오기
- 예외 필터
- Auto 속성 이니셜라이저
- 식 본문 멤버

- Null 전파자
- 문자열 보간
- nameof 연산자

기타 새로운 기능은 다음과 같습니다.

- 인덱스 이니셜라이저
- Catch/Finally 블록의 Await
- Getter 전용 속성의 기본값

이러한 기능을 함께 살펴보면 흥미로운 패턴이 표시됩니다. 이 버전에서 C#은 코드를 더 간결하고 읽을 수 있도록 언어 상용구 제거를 시작했습니다. 따라서 깔끔하고 간단한 코드를 좋아하는 사람들에게 이 언어 버전은 큰 선물이었습니다.

이 버전에는 또 다른 변화가 있지만 본질적으로 기존 언어 기능은 아닙니다. [Roslyn 서비스형 컴파일러](#) 가 릴리스되었습니다. C# 컴파일러는 이제 C#으로 작성되며, 프로그래밍 작업의 일부로 컴파일러를 사용할 수 있습니다.

## C# 버전 5.0

릴리스 날짜: 2012년 8월

Visual Studio 2012과 함께 릴리스된 C# 버전 5.0은 언어에 중점을 둔 버전이었습니다. 해당 버전에 대한 거의 모든 노력은 다른 획기적인 언어 개념인 비동기 프로그래밍을 위한 `async` 및 `await` 모델로 옮겨 갔습니다. 다음은 주요 기능 목록입니다.

- 비동기 멤버
- 호출자 정보 특성
- 코드 프로젝트: [C# 5.0에서 호출자 정보 특성](#)

호출자 정보 특성을 사용하면 엄청난 양의 상용구 리플렉션 코드를 사용하지 않고도 실행 중인 컨텍스트에 대한 정보를 쉽게 검색할 수 있습니다. 진단 및 로깅 작업의 용도는 매우 다양합니다.

하지만 이 릴리스의 진정한 스타는 `async`과 `await`입니다. 이러한 기능이 2012년에 출시되었을 때 C#은 언어에 첫 번째 클래스 참여자로 비동기를 적용하여 다시 업계의 판도를 바꾸었습니다.

## C# 버전 4.0

릴리스 날짜: 2010년 4월

Visual Studio 2010과 함께 릴리스된 C# 버전 4.0에는 몇 가지 흥미로운 새로운 기능이 도입되었습니다.

- 동적 바인딩
- 명명된/선택적 인수
- 제네릭 공변(covariant) 및 반공변(contravariant)
- 포함된 interop 형식

포함 interop 형식은 애플리케이션에 대한 COM interop 어셈블리를 만들 때의 배포 불만을 줄어들게 합니다. 제네릭 공변성(Covariance)과 반공변성(Contravariance)은 제네릭을 사용하는 기능을 더 많이 제공하지만, 약간 학문적이며, 프레임워크와 라이브러리 작성자에게 가장 높은 평가를 받을 것입니다. 명명되고 선택적인 매개 변수를 사용하면 많은 메서드 오버로드를 제거하고 편리성을 제공할 수 있습니다. 그러나 이러한 기능 중 어느 것도 정확히 패러다임의 변화는 아닙니다.

주요 기능은 `dynamic` 키워드였습니다. `dynamic` 키워드는 C# 버전 4.0에 컴파일 시간에 컴파일러를 재정의하는 기능을 도입했습니다. 동적 키워드를 사용하면 JavaScript와 같이 동적으로 형식화된 언어와 유사한 구조를 만들 수 있습니다. `dynamic x = "a string"` 을 만든 다음, 6을 추가하여 다음에 수행해야 할 작업을 런타임에 맡길 수 있습니다.

동적 바인딩은 오류를 유발할 수 있지만 언어 내에서 훌륭한 기능도 제공합니다.

## C# 버전 3.0

릴리스 날짜: 2007년 11월

C# 버전 3.0은 Visual Studio 2008과 함께 2007년말에 출시되었지만 언어 기능을 완전히 갖춘 버전은 .NET Framework 버전 3.5와 함께 제공됩니다. 이 버전은 C#의 성장에 큰 변화를 가져왔습니다. C#은 진정으로 강력한 프로그래밍 언어로 자리매김했습니다. 이 버전의 몇 가지 주요 특징을 살펴보겠습니다.

- 자동 구현 속성
- 무명 형식
- 쿼리 식
- 람다 식
- 식 트리
- 확장 메서드
- 암시적 형식 지역 변수
- 부분 메서드
- 개체 및 컬렉션 이니셜라이저

되돌아보면, 이러한 특징은 대부분 필연적이고 불가분한 것입니다. 이러한 모든 특징은 전략적으로 잘 맞습니다. 이 C# 버전의 퀄리 기능은 LINQ(언어 통합 쿼리)라고도 하는 쿼리

리 식이었습니다.

더 미묘한 뷰는 LINQ가 생성되는 기반인 식 트리, 람다 식 및 익명 형식을 검사합니다. 하지만 두 경우 모두 C# 3.0은 혁신적인 개념을 제공합니다. C# 3.0은 C#을 하이브리드 개체 지향/기능 언어로 전환하기 위한 토대를 마련하기 시작했습니다.

특히, 이제 무엇보다도 컬렉션에 대한 작업을 수행할 수 있는 SQL 스타일의 선언적 쿼리를 작성할 수 있습니다. 정수 목록의 평균을 계산하는 `for` 루프를 작성하는 대신 이제 간단하게 `list.Average()`로 처리할 수 있습니다. 쿼리 식과 확장 메서드의 조합으로 정수 목록이 훨씬 더 스마트해졌습니다.

## C# 버전 2.0

릴리스 날짜: 2005년 11월

Visual Studio 2005와 함께 2005년에 릴리스된 C# 2.0의 몇 가지 주요 기능을 살펴보겠습니다.

- 제네릭
- 부분 형식(Partial Type)
- 무명 메서드
- Nullable 값 형식
- 반복기
- 공변성(Covariance) 및 반공변성(Contravariance)

기존 기능에 추가된 기타 C# 2.0 기능은 다음과 같습니다.

- getter/setter 별도의 액세스 가능
- 메서드 그룹 변환(대리자)
- 정적 클래스
- 대리자 유추

C#은 제네릭 OO(개체 지향) 언어로 시작하지만 C# 버전 2.0은 서둘러 변경했습니다. 제네릭을 사용하면 형식을 안전하게 유지하면서 임의의 형식에서 형식 및 메서드를 작동할 수 있습니다. 예를 들어 `List<T>`를 사용하면 `List<string>` 또는 `List<int>`를 사용하고 이를 반복하는 동안 해당 문자열이나 정수에 형식이 안전한 작업을 수행할 수 있습니다. 제네릭을 사용하는 것이 모든 작업에서 `ArrayList` `Object` 파생되거나 캐스팅되는 형식을 만드는 `ListInt` 것보다 낫습니다.

C# 버전 2.0에서는 반복기라는 기능이 도입되었습니다. 간단히 말해서, 반복기를 사용하면 `List`(또는 다른 열거 가능 형식)의 모든 항목을 `foreach` 루프로 검사할 수 있습니다. 반복기를 언어의 첫 번째 클래스 부분에 사용하면 언어의 가독성과 사용자의 코드 추론 능력이 크게 향상됩니다.

## C# 버전 1.2

릴리스 날짜: 2003년 4월

C# 버전 1.2는 Visual Studio .NET 2003과 함께 제공됩니다. 여기에는 언어에 대한 몇 가지 작은 개선이 포함되어 있습니다. 가장 주목할 만한 점은 이 버전부터 [IEnumerator](#)가 [IDisposable](#)를 구현할 때 [IEnumerator](#)의 [Dispose](#)라는 `foreach` 루트에서 생성된 코드입니다.

## C# 버전 1.0

릴리스 날짜: 2002년 1월

돌이켜보면 Visual Studio.NET 2002와 함께 릴리스된 C# 버전 1.0은 Java와 매우 비슷했습니다. ECMA에 대한 명시된 디자인 목표의 일환으로 ↗ "단순하고 현대적인 범용 개체 지향 언어"가 되고자 했습니다. 당시 Java처럼 보이는 것은 초기 디자인 목표를 달성했음을 의미했습니다.

그러나 지금 다시 C# 1.0을 돌이켜보면 조금 어지러워질 것입니다. 기본 제공 비동기 기능과 당연한 것으로 여겨지는 제네릭과 관련된 멋진 기능 중 일부가 부족했습니다. 사실, 제네릭이 아예 없었습니다. 그리고 [LINQ](#)는 아직 사용할 수 없습니다. 그러한 추가 사항은 나올 때까지 몇 년이 걸릴 것입니다.

C# 버전 1.0은 오늘날보다 기능이 없는 편이었습니다. 좀 더 자세한 코드를 작성해야 했습니다. 하지만 출발점이 필요했습니다. C# 버전 1.0은 Windows 플랫폼에서 Java를 대체하는 실용적인 방법이었습니다.

C# 1.0의 주요 기능에는 다음이 포함되어 있습니다.

- [클래스](#)
- [구조체](#)
- [인터페이스](#)
- [이벤트](#)
- [속성](#)
- [대리자](#)
- [연산자 및 식](#)
- [문](#)
- [특성](#)

아티클[NDepend 블로그에 최초로 게시됨](#), Erik Dietrich 및 Patrick Smacchia 제공

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 언어 기능 및 라이브러리 형식 간의 관계

아티클 • 2024. 03. 08.

C# 언어 정의는 특정 형식 및 이러한 형식에서 액세스할 수 있는 특정 멤버를 갖는 표준 라이브러리를 필요로 합니다. 컴파일러는 다양한 많은 언어 기능에 이러한 필요한 이러한 형식과 멤버를 사용하는 코드를 생성합니다. 이러한 이유로 C# 버전은 해당 .NET 버전 이상에서만 지원됩니다. 이를 통해 올바른 런타임 동작과 필요한 모든 형식 및 멤버의 가용성이 보장됩니다.

표준 라이브러리 기능에 대한 이 종속성은 첫 번째 버전부터 C# 언어의 일부였습니다. 해당 버전에서 예제가 포함되어 있습니다.

- [Exception](#) - 모든 컴파일러 생성 예외에 사용됩니다.
- [String](#) - `string`의 동의어입니다.
- [Int32](#) - `int`의 동의어입니다.

첫 번째 버전은 간단했습니다. 컴파일러 및 표준 라이브러리는 함께 제공되었고 각각 하나의 버전만 있었습니다.

후속 버전의 C#은 종속성에 가끔 새 형식 또는 멤버를 추가했습니다. 예를 들면 [INotifyCompletion](#), [CallerFilePathAttribute](#) 및 [CallerMemberNameAttribute](#)입니다. C# 7.0에서는 [튜플](#) 언어 기능을 구현하기 위해 [ValueTuple](#)에 대한 종속성을 추가했습니다. C# 8에는 다른 기능 중에서도 [범위](#) 및 [인덱스](#)에 대해 [System.Index](#) 및 [System.Range](#)가 필요합니다. 새로운 버전마다 추가 요구 사항이 추가될 수 있습니다.

언어 디자인 팀은 호환 표준 라이브러리에 필요한 형식 및 멤버의 노출 영역을 최소화하려고 합니다. 해당 목표는 새 라이브러리 기능이 해당 언어로 원활하게 통합되는 단순한 디자인에 따라 조정됩니다. 표준 라이브러리에 새 형식 및 멤버를 필요로 하는 이후 버전의 C#에는 새 기능이 있을 예정입니다. C# 컴파일러 도구는 이제 지원되는 플랫폼의 .NET 라이브러리의 릴리스 주기에서 분리됩니다.

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.



## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

[설명서 문제 열기](#)

[제품 사용자 의견 제공](#)

# C# 개발자를 위한 버전 및 업데이트 고려 사항

아티클 • 2024. 03. 08.

C# 언어에 새로운 기능이 추가됨에 따라 호환성은 중요한 목표입니다. 대부분의 경우 문제 없이 새 컴파일러 버전으로 기존 코드를 다시 컴파일할 수 있습니다. .NET 런타임 팀은 또한 업데이트된 라이브러리에 대한 호환성을 보장한다는 목표를 가지고 있습니다. 거의 모든 경우에 업데이트된 라이브러리가 포함된 업데이트된 런타임에서 앱이 시작되면 동작은 이전 버전과 정확히 동일합니다.

앱을 컴파일하는 데 사용되는 언어 버전은 일반적으로 프로젝트에서 참조되는 런타임 대상 프레임워크 모니터(TFM)와 일치합니다. 기본 언어 버전 변경에 대한 자세한 내용은 [언어 버전 구성](#)이라는 제목의 문서를 참조하세요. 이 기본 동작은 최대 호환성을 보장합니다.

호환성이 손상되는 변경이 도입되면 다음과 같이 분류됩니다.

- 이진 파일 호환성이 손상되는 변경:** 이진 파일 호환성이 손상되는 변경은 새 런타임을 사용하여 시작될 때 애플리케이션이나 라이브러리에서 크래시 가능성을 포함하여 다양한 동작을 유발합니다. 이러한 변경 내용을 통합하려면 앱을 다시 컴파일해야 합니다. 기존 이진 파일이 올바르게 작동하지 않습니다.
- 원본 호환성이 손상되는 변경:** 원본 호환성이 손상되는 변경은 소스 코드의 의미를 변경합니다. 최신 언어 버전으로 애플리케이션을 컴파일하기 전에 소스 코드를 편집해야 합니다. 기존 이진 파일은 최신 호스트 및 런타임에서 올바르게 실행됩니다. 언어 구문의 경우 원본 호환성이 손상되는 변경은 [런타임 호환성이 손상되는 변경](#)에 정의된대로 동작 변경이기도 합니다.

이진 파일 호환성이 손상되는 변경이 앱에 영향을 미치는 경우 앱을 다시 컴파일해야 하지만 소스 코드를 편집할 필요는 없습니다. 원본 호환성이 손상되는 변경이 앱에 영향을 미치는 경우 기존 이진 파일은 업데이트된 런타임 및 라이브러리가 있는 환경에서 계속 올바르게 실행됩니다. 그러나 새 언어 버전 및 런타임으로 다시 컴파일하려면 원본을 변경해야 합니다. 변경 내용이 원본 중단 및 이진 파일 중단 모두인 경우 최신 버전으로 애플리케이션을 다시 컴파일하고 원본을 업데이트해야 합니다.

C# 언어 팀과 런타임 팀의 호환성이 손상되는 변경을 방지하려는 목표 때문에 애플리케이션 업데이트는 일반적으로 TFM을 업데이트하고 앱을 다시 빌드하는 문제입니다. 그러나 공개적으로 배포되는 라이브러리의 경우 지원되는 TFM 및 지원되는 언어 버전에 대한 정책을 신중하게 평가해야 합니다. 최신 버전에 있는 기능을 사용하여 새 라이브러리를 만드는 경우 이전 버전의 컴파일러로 빌드된 앱이 새 라이브러리를 사용할 수 있는지 확인해야 합니다. 또는 기존 라이브러리를 업그레이드하는 경우 아직 업그레이드된 버전이 없는 사용자가 많을 수 있습니다.

# 라이브러리의 호환성이 손상되는 변경 소개

라이브러리의 공용 API에 새로운 언어 기능을 채택할 때 해당 기능을 채택하면 라이브러리 사용자에게 이진 파일 또는 원본 호환성이 손상되는 변경이 도입되는지 평가해야 합니다. `public` 또는 `protected` 인터페이스에 표시되지 않는 내부 구현에 대한 모든 변경 내용은 호환됩니다.

## ① 참고

`System.Runtime.CompilerServices.InternalVisibleToAttribute`를 사용하여 형식이 내부 멤버를 볼 수 있도록 설정하면 내부 멤버가 호환성이 손상되는 변경을 도입할 수 있습니다.

이진 파일 호환성이 손상되는 변경을 사용하려면 사용자가 새 버전을 사용하기 위해 코드를 다시 컴파일해야 합니다. 예를 들어, 다음 공용 메서드를 고려해보세요.

C#

```
public double CalculateSquare(double value) => value * value;
```

메서드에 `in` 한정자를 추가하면 이는 이진 파일 호환성이 손상되는 변경입니다.

C#

```
public double CalculateSquare(in double value) => value * value;
```

새 라이브러리가 올바르게 작동하려면 사용자가 `CalculateSquare` 메서드를 사용하는 모든 애플리케이션을 다시 컴파일해야 합니다.

원본 호환성이 손상되는 변경에서는 사용자가 다시 컴파일하기 전에 코드를 변경해야 합니다. 예를 들어, 다음 형식을 고려해보세요.

C#

```
public class Person
{
    public string FirstName { get; }
    public string LastName { get; }

    public Person(string firstName, string lastName) => (FirstName,
    LastName) = (firstName, lastName);
```

```
// other details omitted  
}
```

최신 버전에서는 `record` 형식에 대해 생성된 합성 멤버를 활용하려고 합니다. 다음과 같이 변경합니다.

C#

```
public record class Person(string FirstName, string LastName);
```

이전 변경 내용을 적용하려면 `Person`에서 파생된 모든 형식을 변경해야 합니다. 이러한 모든 선언은 해당 선언에 `record` 한정자를 추가해야 합니다.

## 호환성이 손상되는 변경의 영향

라이브러리에 이진 파일 호환성이 손상되는 변경을 추가하면 라이브러리를 사용하는 모든 프로젝트가 강제로 다시 컴파일됩니다. 그러나 해당 프로젝트의 소스 코드는 변경할 필요가 없습니다. 결과적으로 각 프로젝트에 대한 호환성이 손상되는 변경의 영향은 비교적 작습니다.

라이브러리에 원본 호환성이 손상되는 변경을 수행하는 경우 새 라이브러리를 사용하려면 모든 프로젝트에서 원본을 변경해야 합니다. 필요한 변경에 새로운 언어 기능이 필요한 경우 해당 프로젝트를 현재 사용 중인 것과 동일한 언어 버전 및 TFM으로 강제로 업그레이드합니다. 사용자에게 더 많은 작업이 필요했으며 강제로 업그레이드해야 할 수도 있습니다.

호환성이 손상되는 변경의 영향은 라이브러리에 종속된 프로젝트 수에 따라 달라집니다. 라이브러리가 몇몇 애플리케이션에서 내부적으로 사용되는 경우 영향을 받는 모든 프로젝트의 호환성이 손상되는 변경에 대응할 수 있습니다. 그러나 라이브러리를 공개적으로 다운로드하는 경우 잠재적인 영향을 평가하고 대안을 고려해야 합니다.

- 기존 API와 유사한 새 API를 추가할 수 있습니다.
- 다양한 TFM에 대한 별별 빌드를 고려할 수 있습니다.
- 멀티 대상 지정을 고려해 볼 수도 있습니다.

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

수도 있습니다. 자세한 내용은  
참여자 가이드를 참조하세요.

☞ 설명서 문제 열기

☞ 제품 사용자 의견 제공

# 자습서: 기본 생성자 탐색

아티클 • 2024. 03. 08.

C# 12에는 형식 본문 어디에서나 매개 변수를 사용할 수 있는 생성자를 선언하는 간결한 구문인 [기본 생성자가](#) 도입되었습니다.

이 자습서에서는 다음에 대해 알아봅니다.

- ✓ 형식에 기본 생성자를 선언해야 하는 경우
- ✓ 다른 생성자에서 기본 생성자를 호출하는 방법
- ✓ 형식의 멤버에서 기본 생성자 매개 변수를 사용하는 방법
- ✓ 기본 생성자 매개 변수가 저장되는 위치

## 필수 조건

C# 12 이상 컴파일러를 포함하여 .NET 8 이상을 실행하도록 머신을 설정해야 합니다. C# 12 컴파일러는 [Visual Studio 2022 버전 17.7](#) 또는 [.NET 8 SDK](#) 부터 사용할 수 있습니다.

## 기본 생성자

`struct` 또는 `class` 선언에 매개 변수를 추가하여 기본 생성자를 만들 수 있습니다. 기본 생성자 매개 변수는 클래스 정의 전체의 범위에 포함됩니다. 기본 생성자 매개 변수가 클래스 정의 전체의 범위 내에 있더라도 매개 변수로 보는 것이 중요합니다. 여러 규칙에서는 매개 변수임을 명확히 합니다.

- 기본 생성자 매개 변수는 필요하지 않은 경우 저장되지 않을 수 있습니다.
- 기본 생성자 매개 변수는 클래스의 멤버가 아닙니다. 예를 들어, `param`이라는 기본 생성자 매개 변수는 `this.param`으로 액세스할 수 없습니다.
- 기본 생성자 매개 변수를 할당할 수 있습니다.
- 기본 생성자 매개 변수는 `record` 형식을 제외하고 속성이 되지 않습니다.

이러한 규칙은 다른 생성자 선언을 포함하여 모든 메서드에 대한 매개 변수와 동일합니다.

기본 생성자 매개 변수의 가장 일반적인 용도는 다음과 같습니다.

- `base()` 생성자 호출에 대한 인수로 사용됩니다.
- 멤버 필드 또는 속성을 초기화합니다.
- 인스턴스 멤버에서 생성자 매개 변수를 참조하세요.

클래스의 다른 모든 생성자는 반드시 `this()` 생성자 호출을 통해 직접 또는 간접적으로 기본 생성자를 호출해야 합니다. 이 규칙은 기본 생성자 매개 변수가 형식 본문의 어느 위치에나 할당되도록 보장합니다.

## 속성 초기화

다음 코드는 기본 생성자 매개 변수에서 계산되는 두 개의 읽기 전용 속성을 초기화합니다.

C#

```
public readonly struct Distance(double dx, double dy)
{
    public readonly double Magnitude { get; } = Math.Sqrt(dx * dx + dy * dy);
    public readonly double Direction { get; } = Math.Atan2(dy, dx);
}
```

앞의 코드는 계산된 읽기 전용 속성을 초기화하는 데 사용되는 기본 생성자를 보여 줍니다. `Magnitude` 및 `Direction`의 필드 이니셜라이저는 기본 생성자 매개 변수를 사용합니다. 기본 생성자 매개 변수는 구조체의 다른 곳에서는 사용되지 않습니다. 이전 구조체는 다음 코드를 작성한 것과 같습니다.

C#

```
public readonly struct Distance
{
    public readonly double Magnitude { get; }

    public readonly double Direction { get; }

    public Distance(double dx, double dy)
    {
        Magnitude = Math.Sqrt(dx * dx + dy * dy);
        Direction = Math.Atan2(dy, dx);
    }
}
```

새로운 기능을 사용하면 필드나 속성을 초기화하기 위해 인수가 필요할 때 필드 이니셜라이저를 더 쉽게 사용할 수 있습니다.

## 변경 가능한 상태 만들기

앞의 예에서는 기본 생성자 매개 변수를 사용하여 읽기 전용 속성을 초기화합니다. 속성이 읽기 전용이 아닌 경우 기본 생성자를 사용할 수도 있습니다. 다음 코드를 생각해 봅시

다.

C#

```
public struct Distance(double dx, double dy)
{
    public readonly double Magnitude => Math.Sqrt(dx * dx + dy * dy);
    public readonly double Direction => Math.Atan2(dy, dx);

    public void Translate(double deltaX, double deltaY)
    {
        dx += deltaX;
        dy += deltaY;
    }

    public Distance() : this(0,0) { }
}
```

앞의 예에서 `Translate` 메서드는 `dx` 및 `dy` 구성 요소를 변경합니다. 이를 위해서는 액세스 시 `Magnitude` 및 `Direction` 속성을 계산해야 합니다. `=>` 연산자는 식 본문 `get` 접근자를 지정하는 반면, `=` 연산자는 이니셜라이저를 지정합니다. 이 버전은 매개 변수 없는 생성자를 구조체에 추가합니다. 매개 변수가 없는 생성자는 기본 생성자를 호출해야 모든 기본 생성자 매개 변수가 초기화됩니다.

이전 예에서 기본 생성자 속성은 메서드에서 액세스됩니다. 따라서 컴파일러는 각 매개 변수를 나타내기 위해 숨겨진 필드를 만듭니다. 다음 코드는 컴파일러가 생성하는 내용을 대략적으로 보여 줍니다. 실제 필드 이름은 유효한 CIL 식별자이지만 유효한 C# 식별자는 아닙니다.

C#

```
public struct Distance
{
    private double __unspeakable_dx;
    private double __unspeakable_dy;

    public readonly double Magnitude => Math.Sqrt(__unspeakable_dx *
__unspeakable_dx + __unspeakable_dy * __unspeakable_dy);
    public readonly double Direction => Math.Atan2(__unspeakable_dy,
__unspeakable_dx);

    public void Translate(double deltaX, double deltaY)
    {
        __unspeakable_dx += deltaX;
        __unspeakable_dy += deltaY;
    }

    public Distance(double dx, double dy)
    {
```

```
    __unspeakable_dx = dx;
    __unspeakable_dy = dy;
}
public Distance() : this(0, 0) { }
}
```

첫 번째 예에서는 컴파일러가 기본 생성자 매개 변수의 값을 저장하기 위한 필드를 만들 필요가 없다는 점을 이해해야 합니다. 두 번째 예에서는 메서드 내에서 기본 생성자 매개 변수를 사용했기 때문에 컴파일러에서 해당 매개 변수에 대한 스토리지를 만들어야 합니다. 컴파일러는 해당 형식의 멤버 본문에서 해당 매개 변수에 액세스할 때만 기본 생성자에 대한 스토리지를 만듭니다. 그렇지 않으면 기본 생성자 매개 변수가 개체에 저장되지 않습니다.

## 종속성 주입

기본 생성자의 또 다른 일반적인 용도는 종속성 주입을 위한 매개 변수를 지정하는 것입니다. 다음 코드는 사용을 위해 서비스 인터페이스가 필요한 간단한 컨트롤러를 만듭니다.

C#

```
public interface IService
{
    Distance GetDistance();
}

public class ExampleController(IService service) : ControllerBase
{
    [HttpGet]
    public ActionResult<Distance> Get()
    {
        return service.GetDistance();
    }
}
```

기본 생성자는 클래스에 필요한 매개 변수를 명확하게 나타냅니다. 클래스의 다른 변수와 마찬가지로 기본 생성자 매개 변수를 사용합니다.

## 기본 클래스 초기화

파생 클래스의 기본 생성자에서 기본 클래스의 기본 생성자를 호출할 수 있습니다. 이는 기본 클래스에서 기본 생성자를 호출해야 하는 파생 클래스를 작성하는 가장 쉬운 방법입니다. 예를 들어, 다양한 계좌 형식을 은행으로 나타내는 클래스 계층을 생각해 보세요. 기본 클래스는 다음 코드와 유사합니다.

C#

```
public class BankAccount(string accountID, string owner)
{
    public string AccountID { get; } = accountID;
    public string Owner { get; } = owner;

    public override string ToString() => $"Account ID: {AccountID}, Owner: {Owner}";
}
```

형식에 관계없이 모든 은행 계좌에는 계좌 번호와 소유자에 대한 속성이 있습니다. 완료된 애플리케이션에서는 다른 공통 기능이 기본 클래스에 추가됩니다.

많은 형식에는 생성자 매개 변수에 대한 보다 구체적인 유효성 검사가 필요합니다. 예를 들어, `BankAccount`에는 `owner` 및 `accountID` 매개 변수에 대한 특정 요구 사항이 있습니다. `owner`는 `null` 또는 공백이 아니어야 하고, `accountID`는 10자리를 포함하는 문자열어야 합니다. 해당 속성을 할당할 때 이 유효성 검사를 추가할 수 있습니다.

C#

```
public class BankAccount(string accountID, string owner)
{
    public string AccountID { get; } = ValidAccountNumber(accountID)
        ? accountID
        : throw new ArgumentException("Invalid account number",
nameof(accountID));

    public string Owner { get; } = string.IsNullOrWhiteSpace(owner)
        ? throw new ArgumentException("Owner name cannot be empty",
nameof(owner))
        : owner;

    public override string ToString() => $"Account ID: {AccountID}, Owner: {Owner}";

    public static bool ValidAccountNumber(string accountID) =>
accountID?.Length == 10 && accountID.All(c => char.IsDigit(c));
}
```

이전 예에서는 생성자 매개 변수를 속성에 할당하기 전에 유효성을 검사하는 방법을 보여 줍니다. `String.IsNullOrWhiteSpace(String)`과 같은 기본 제공 메서드나

`ValidAccountNumber`와 같은 자체 유효성 검사 메서드를 사용할 수 있습니다. 이전 예에서는 이니셜라이저를 호출할 때 생성자에서 모든 예외가 `throw`됩니다. 필드를 할당하는 데 생성자 매개 변수를 사용하지 않는 경우 생성자 매개 변수에 처음 액세스할 때 모든 예외가 `throw`됩니다.

하나의 파생 클래스는 당좌 예금 계좌를 제공합니다.

C#

```
public class CheckingAccount(string accountID, string owner, decimal
overdraftLimit = 0) : BankAccount(accountID, owner)
{
    public decimal CurrentBalance { get; private set; } = 0;

    public void Deposit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Deposit
amount must be positive");
        }
        CurrentBalance += amount;
    }

    public void Withdrawal(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount),
"Withdrawal amount must be positive");
        }
        if (CurrentBalance - amount < -overdraftLimit)
        {
            throw new InvalidOperationException("Insufficient funds for
withdrawal");
        }
        CurrentBalance -= amount;
    }

    public override string ToString() => $"Account ID: {AccountID}, Owner:
{Owner}, Balance: {CurrentBalance}";
}
```

파생된 `CheckingAccount` 클래스에는 기본 클래스에 필요한 모든 매개 변수를 사용하는 기본 생성자와 기본값이 있는 또 다른 매개 변수가 있습니다. 기본 생성자는 `: BankAccount(accountID, owner)` 구문을 사용하여 기본 생성자를 호출합니다. 이 식은 기본 클래스의 형식과 기본 생성자의 인수를 모두 지정합니다.

파생 클래스는 기본 생성자를 사용할 필요가 없습니다. 다음 예와 같이 기본 클래스의 기본 생성자를 호출하는 파생 클래스에서 생성자를 만들 수 있습니다.

C#

```
public class LineOfCreditAccount : BankAccount
{
    private readonly decimal _creditLimit;
    public LineOfCreditAccount(string accountID, string owner, decimal
creditLimit) : base(accountID, owner)
```

```

    {
        _creditLimit = creditLimit;
    }
    public decimal CurrentBalance { get; private set; } = 0;

    public void Deposit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Deposit
amount must be positive");
        }
        CurrentBalance += amount;
    }

    public void Withdrawal(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount),
"Withdrawal amount must be positive");
        }
        if (CurrentBalance - amount < -_creditLimit)
        {
            throw new InvalidOperationException("Insufficient funds for
withdrawal");
        }
        CurrentBalance -= amount;
    }

    public override string ToString() => $"{base.ToString()}, Balance:
{CurrentBalance}";
}

```

클래스 계층 구문과 기본 생성자에는 한 가지 잠재적인 우려 사항이 있습니다. 즉, 파생 클래스와 기본 클래스 모두에서 사용되는 기본 생성자 매개 변수의 복사본을 여러 개 만들 수 있다는 것입니다. 다음 코드 예에서는 `owner` 및 `accountID` 필드 각각에 두 개의 복사본을 만듭니다.

C#

```

public class SavingsAccount(string accountID, string owner, decimal
interestRate) : BankAccount(accountID, owner)
{
    public SavingsAccount() : this("default", "default", 0.01m) { }
    public decimal CurrentBalance { get; private set; } = 0;

    public void Deposit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Deposit
amount must be positive");
        }
        CurrentBalance += amount;
    }

    public override string ToString() => $"{base.ToString()}, Balance:
{CurrentBalance}";
}

```

```

        amount must be positive");
    }
    CurrentBalance += amount;
}

public void Withdrawal(decimal amount)
{
    if (amount < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount),
"Withdrawal amount must be positive");
    }
    if (CurrentBalance - amount < 0)
    {
        throw new InvalidOperationException("Insufficient funds for
withdrawal");
    }
    CurrentBalance -= amount;
}

public void ApplyInterest()
{
    CurrentBalance *= 1 + interestRate;
}

public override string ToString() => $"Account ID: {accountID}, Owner:
{owner}, Balance: {CurrentBalance}";
}

```

강조 표시된 줄은 `ToString` 메서드가 기본 클래스 속성(`Owner` 및 `AccountID`) 대신 기본 생성자 매개 변수(`owner` 및 `accountID`)를 사용함을 보여 줍니다. 결과적으로 파생 클래스인 `SavingsAccount`는 해당 복사본에 대한 스토리지를 만듭니다. 파생 클래스의 복사본은 기본 클래스의 속성과 다릅니다. 기본 클래스 속성을 수정할 수 있는 경우 파생 클래스의 인스턴스에는 해당 수정 사항이 표시되지 않습니다. 컴파일러는 파생 클래스에서 사용되고 기본 클래스 생성자에 전달되는 기본 생성자 매개 변수에 대해 경고를 표시합니다. 이 경우 수정 방법은 기본 클래스의 속성을 사용하는 것입니다.

## 요약

디자인에 가장 적합한 기본 생성자를 사용할 수 있습니다. 클래스 및 구조체의 경우 기본 생성자 매개 변수는 호출해야 하는 생성자에 대한 매개 변수입니다. 이를 사용하여 속성을 초기화할 수 있습니다. 필드를 초기화할 수 있습니다. 해당 속성이나 필드는 변경할 수 없거나 변경할 수 있습니다. 메서드에서 사용할 수 있습니다. 이는 매개 변수이며 디자인에 가장 적합한 방식으로 사용합니다. [인스턴스 생성자에 대한 C# 프로그래밍 가이드 문서](#) 및 [제안된 기본 생성자 사양](#)에서 기본 생성자에 대해 자세히 알아볼 수 있습니다.

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 자습서: C# 11 기능 살펴보기 - 인터페이스의 정적 가상 멤버

아티클 • 2024. 03. 08.

C# 11 및 .NET 7에는 인터페이스의 정적 가상 멤버가 포함되어 있습니다. 이 기능을 사용하면 [오버로드된 연산자](#) 또는 기타 정적 멤버를 포함하는 인터페이스를 정의할 수 있습니다. 정적 멤버로 인터페이스를 정의한 후에는 해당 인터페이스를 [제약 조건](#)으로 사용하여 연산자나 기타 정적 메서드를 사용하는 제네릭 형식을 만들 수 있습니다. 오버로드된 연산자가 포함된 인터페이스를 만들지 않더라도 이 기능과 언어 업데이트로 사용하도록 설정된 제네릭 수학 클래스의 이점을 활용할 수 있습니다.

이 자습서에서는 다음 작업을 수행하는 방법을 알아봅니다.

- ✓ 정적 멤버로 인터페이스를 정의합니다.
- ✓ 인터페이스를 사용하여 연산자가 정의된 인터페이스를 구현하는 클래스를 정의합니다.
- ✓ 정적 인터페이스 메서드에 의존하는 제네릭 알고리즘을 만듭니다.

## 필수 조건

C# 11을 지원하는 .NET 7을 실행하려면 컴퓨터를 설정해야 합니다. C# 11 컴파일러는 [Visual Studio 2022 버전 17.3](#) 또는 [.NET 7 SDK](#) 부터 사용할 수 있습니다.

## 정적 추상 인터페이스 메서드

예제를 확인해보겠습니다. 다음 메서드는 두 `double` 숫자의 중간점을 반환합니다.

C#

```
public static double MidPoint(double left, double right) =>
    (left + right) / (2.0);
```

`int`, `short`, `long`, `float`, `decimal` 또는 숫자를 나타내는 모든 형식 등 모든 숫자 형식에 대해 동일한 논리가 작동합니다. `+` 및 `/` 연산자를 사용하고 `2`의 값을 정의할 수 있는 방법이 필요합니다. [System.Numerics.INumber<TSelf>](#) 인터페이스를 사용하여 이전 메서드를 다음 제네릭 메서드로 작성할 수 있습니다.

C#

```
public static T MidPoint<T>(T left, T right)
    where T : INumber<T> => (left + right) / T.CreateChecked(2); // note:
the addition of left and right may overflow here; it's just for
demonstration purposes
```

`INumber<TSelf>` 인터페이스를 구현하는 모든 형식은 `operator +` 및 `operator /`에 대한 정의를 포함해야 합니다. 분모는 숫자 형식에 대해 `2` 값을 만들기 위해 `T.CreateChecked(2)`에 의해 정의되며, 이는 분모가 두 매개 변수와 동일한 형식이 되도록 강제합니다. `INumberBase<TSelf>.CreateChecked<TOther>(TOther)`는 지정된 값에서 해당 형식의 인스턴스를 만들고 값이 표현 가능한 범위를 벗어나는 경우 `OverflowException`을 throw합니다. (이 구현은 `left`와 `right`가 모두 충분히 큰 값인 경우 오버플로가 발생할 가능성이 있습니다. 이러한 잠재적인 문제를 피할 수 있는 대체 알고리즘이 있습니다.)

익숙한 구문을 사용하여 인터페이스에서 정적 추상 멤버를 정의합니다. 구현을 제공하지 않는 모든 정적 멤버에 `static` 및 `abstract` 한정자를 추가합니다. 다음 예에서는 `operator ++`를 재정의하는 모든 형식에 적용할 수 있는 `IGetNext<T>` 인터페이스를 정의합니다.

C#

```
public interface IGetNext<T> where T : IGetNext<T>
{
    static abstract T operator ++(T other);
}
```

형식 인수 `T` 가 `IGetNext<T>` 를 구현하는 제약 조건은 연산자의 서명에 포함 형식 또는 해당 형식 인수가 포함되도록 보장합니다. 많은 연산자는 해당 매개 변수가 형식과 일치해야 하거나 포함 형식을 구현하도록 제한되는 형식 매개 변수가 되도록 강제합니다. 이 제약 조건이 없으면 `IGetNext<T>` 인터페이스에서 `++` 연산자를 정의할 수 없습니다.

다음 코드를 사용하면 각 증분이 문자열에 다른 문자를 추가하는 'A' 문자의 문자열을 만드는 구조를 만들 수 있습니다.

C#

```
public struct RepeatSequence : IGetNext<RepeatSequence>
{
    private const char Ch = 'A';
    public string Text = new string(Ch, 1);

    public RepeatSequence() {}

    public static RepeatSequence operator ++(RepeatSequence other)
        => other with { Text = other.Text + Ch };
```

```
    public override string ToString() => Text;
}
```

보다 일반적으로, "이 형식의 다음 값을 생성한다"는 의미로 `++`을 정의하려는 알고리즘을 빌드할 수 있습니다. 이 인터페이스를 사용하면 명확한 코드와 결과가 생성됩니다.

C#

```
var str = new RepeatSequence();

for (int i = 0; i < 10; i++)
    Console.WriteLine(str++);
```

앞의 예에서는 다음 출력을 생성합니다.

PowerShell

```
A
AA
AAA
AAAA
AAAAA
AAAAAA
AAAAAAA
AAAAAAA
AAAAAA
AAAAAA
```

이 작은 예는 이 기능의 동기를 보여 줍니다. 연산자, 상수 값 및 기타 정적 연산에 자연 구문을 사용할 수 있습니다. 오버로드된 연산자를 포함하여 정적 멤버에 의존하는 여러 형식을 만들 때 이러한 기술을 탐색할 수 있습니다. 형식의 기능과 일치하는 인터페이스를 정의한 다음 새 인터페이스에 대한 해당 형식의 지원을 선언합니다.

## 제네릭 수학

인터페이스에서 연산자를 포함한 정적 메서드를 허용하는 동기 부여 시나리오는 [제네릭 수학](#) 알고리즘을 지원하는 것입니다. .NET 7 기본 클래스 라이브러리에는 많은 산술 연산자에 대한 인터페이스 정의와 `INumber<T>` 인터페이스에서 많은 산술 연산자를 결합하는 파생 인터페이스가 포함되어 있습니다. 이러한 형식을 적용하여 `T`에 대해 모든 숫자 형식을 사용할 수 있는 `Point<T>` 레코드를 빌드해 보겠습니다. 점은 `+` 연산자를 사용하여 일부 `xOffset` 및 `yOffset`로 이동할 수 있습니다.

`dotnet new` 또는 Visual Studio를 사용하여 새 콘솔 애플리케이션을 만드는 것부터 시작합니다.

`Translation<T>` 및 `Point<T>`의 공용 인터페이스는 다음 코드와 유사해야 합니다.

C#

```
// Note: Not complete. This won't compile yet.  
public record Translation<T>(T XOffset, T YOffset);  
  
public record Point<T>(T X, T Y)  
{  
    public static Point<T> operator +(Point<T> left, Translation<T> right);  
}
```

`Translation<T>` 및 `Point<T>` 형식 모두에 대해 `record` 형식을 사용합니다. 둘 다 두 개의 값을 저장하며 정교한 동작보다는 데이터 스토리지를 나타냅니다. `operator +`의 구현은 다음 코드와 같습니다.

C#

```
public static Point<T> operator +(Point<T> left, Translation<T> right) =>  
    left with { X = left.X + right.XOffset, Y = left.Y + right.YOffset };
```

이전 코드를 컴파일하려면 `T` 가 `IAdditionOperators<TSelf, TOther, TResult>` 인터페이스를 지원한다고 선언해야 합니다. 해당 인터페이스에는 `operator +` 정적 메서드가 포함되어 있습니다. 이는 세 가지 형식 매개 변수(왼쪽 피연산자용, 오른쪽 피연산자용, 결과용)를 선언합니다. 일부 형식은 다양한 피연산자 및 결과 형식에 대해 `+`을 구현합니다. 형식 인수 `T` 가 `IAdditionOperators<T, T, T>`를 구현한다는 선언을 추가합니다.

C#

```
public record Point<T>(T X, T Y) where T : IAdditionOperators<T, T, T>
```

해당 제약 조건을 추가한 후 `Point<T>` 클래스는 더하기 연산자로 `+`를 사용할 수 있습니다. `Translation<T>` 선언에 동일한 제약 조건을 추가합니다.

C#

```
public record Translation<T>(T XOffset, T YOffset) where T :  
IAdditionOperators<T, T, T>;
```

`IAdditionOperators<T, T, T>` 제약 조건은 클래스를 사용하는 개발자가 점 추가 제약 조건을 충족하지 않는 형식을 사용하여 `Translation`을 만드는 것을 방지합니다. 이 코드가 작동하도록 `Translation<T>` 및 `Point<T>`의 형식 매개 변수에 필요한 제약 조건을 추가했

습니다. `Program.cs` 파일의 `Translation` 및 `Point` 선언 위에 다음과 같은 코드를 추가하여 테스트할 수 있습니다.

```
C#  
  
var pt = new Point<int>(3, 4);  
  
var translate = new Translation<int>(5, 10);  
  
var final = pt + translate;  
  
Console.WriteLine(pt);  
Console.WriteLine(translate);  
Console.WriteLine(final);
```

이러한 형식이 적절한 산술 인터페이스를 구현한다고 선언하면 이 코드를 더 쉽게 재사용할 수 있습니다. 첫 번째 변경 내용은 `Point<T, T>` 가 `IAdditionOperators<Point<T>, Translation<T>, Point<T>>` 인터페이스를 구현한다고 선언하는 것입니다. `Point` 형식은 피연산자 및 결과에 대해 다양한 형식을 사용합니다. `Point` 형식은 이미 해당 서명을 사용하여 `operator +` 를 구현하므로 선언에 인터페이스만 추가하면 됩니다.

```
C#  
  
public record Point<T>(T X, T Y) : IAdditionOperators<Point<T>, Translation<T>, Point<T>>  
    where T : IAdditionOperators<T, T, T>
```

마지막으로 추가를 수행할 때 해당 형식에 대한 추가 ID 값을 정의하는 속성을 갖는 것이 유용합니다. 해당 기능에 대한 새 인터페이스가 있습니다.

`IAdditiveIdentity<TSelf, TResult>`. `{0, 0}` 의 변환은 덧셈 항등식입니다. 결과 점은 왼쪽 피연산자와 동일합니다. `IAdditiveIdentity<TSelf, TResult>` 인터페이스는 ID 값을 반환하는 하나의 읽기 전용 속성인 `AdditiveIdentity` 를 정의합니다. 이 인터페이스를 구현하려면 `Translation<T>` 에 몇 가지 변경이 필요합니다.

```
C#  
  
using System.Numerics;  
  
public record Translation<T>(T XOffset, T YOffset) :  
IAdditiveIdentity<Translation<T>, Translation<T>>  
    where T : IAdditionOperators<T, T, T>, IAdditiveIdentity<T, T>  
{  
    public static Translation<T> AdditiveIdentity =>  
        new Translation<T>(XOffset: T.AdditiveIdentity, YOffset:
```

```
T.AdditiveIdentity);  
}
```

여기에는 몇 가지 변경 내용이 있으므로 하나씩 살펴보겠습니다. 먼저, `Translation` 형식이 `IAdditiveIdentity` 인터페이스를 구현한다고 선언합니다.

C#

```
public record Translation<T>(T XOffset, T YOffset) :  
IAdditiveIdentity<Translation<T>, Translation<T>>
```

다음에는 다음 코드에 표시된 대로 인터페이스 멤버를 구현해 볼 수 있습니다.

C#

```
public static Translation<T> AdditiveIdentity =>  
    new Translation<T>(XOffset: 0, YOffset: 0);
```

`0`은 형식에 따라 다르므로 앞의 코드는 컴파일되지 않습니다. 답: `0`에는 `IAdditiveIdentity<T>.AdditiveIdentity`를 사용합니다. 이러한 변경은 이제 제약 조건에 `T`가 `IAdditiveIdentity<T>`를 구현한다는 것을 포함해야 함을 의미합니다. 그 결과 다음과 같은 구현이 이루어집니다.

C#

```
public static Translation<T> AdditiveIdentity =>  
    new Translation<T>(XOffset: T.AdditiveIdentity, YOffset:  
T.AdditiveIdentity);
```

이제 `Translation<T>`에 해당 제약 조건을 추가했으므로 `Point<T>`에도 동일한 제약 조건을 추가해야 합니다.

C#

```
using System.Numerics;  
  
public record Point<T>(T X, T Y) : IAdditionOperators<Point<T>,  
Translation<T>, Point<T>>  
    where T : IAdditionOperators<T, T, T>, IAdditiveIdentity<T, T>  
{  
    public static Point<T> operator +(Point<T> left, Translation<T> right)  
=>  
        left with { X = left.X + right.XOffset, Y = left.Y + right.YOffset  
};  
}
```

이 샘플에서는 제네릭 수학용 인터페이스가 어떻게 구성되는지 살펴보았습니다. 다음 방법에 대해 알아보았습니다.

- ✓ 메서드가 모든 숫자 형식과 함께 사용될 수 있도록 `INumber<T>` 인터페이스에 의존하는 메서드를 작성합니다.
- ✓ 하나의 수학 연산만 지원하는 형식을 구현하기 위해 추가 인터페이스에 의존하는 형식을 빌드합니다. 해당 형식은 동일한 인터페이스에 대한 지원을 선언하므로 다른 방식으로 구성될 수 있습니다. 알고리즘은 수학 연산자의 가장 자연스러운 구문을 사용하여 작성되었습니다.

이러한 기능을 실험하고 피드백을 등록합니다. Visual Studio의 **피드백 보내기** 메뉴 항목을 사용하거나 GitHub의 roslyn 리포지토리에서 새 문제 [문제](#)를 만들 수 있습니다. 모든 숫자 형식에서 작동하는 제네릭 알고리즘을 빌드합니다. 형식 인수가 숫자와 유사한 기능의 하위 집합만 구현할 수 있는 이러한 인터페이스를 사용하여 알고리즘을 빌드합니다. 이러한 기능을 사용하는 새로운 인터페이스를 빌드하지 않더라도 알고리즘에서 이를 사용해 실험할 수 있습니다.

## 참고 항목

- [제네릭 수학](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

[설명서 문제 열기](#)

[제품 사용자 의견 제공](#)

# 패턴 일치를 사용하여 코드를 개선하는 클래스 동작 빌드

아티클 • 2023. 11. 15.

C#의 패턴 일치 기능은 알고리즘을 표현하는 구문을 제공합니다. 이러한 방법을 사용하여 클래스에서 동작을 구현할 수 있습니다. 개체 지향 클래스 디자인을 데이터 지향 구현과 결합하면 실제 개체를 모델링하면서 간결한 코드를 제공할 수 있습니다.

이 자습서에서는 다음 작업을 수행하는 방법을 알아봅니다.

- ✓ 데이터 패턴을 사용하여 개체 지향 클래스를 표현합니다.
- ✓ C#의 패턴 일치 기능을 사용하여 이러한 패턴을 구현합니다.
- ✓ 컴파일러 진단을 활용하여 구현의 유효성을 검사합니다.

## 필수 조건

.NET을 실행하도록 컴퓨터를 설정해야 합니다. Visual Studio 2022 또는 [.NET SDK](#)를 다운로드합니다.

## 운하 갑문 시뮬레이션 빌드

이 자습서에서는 [운하 갑문](#)을 시뮬레이트하는 C# 클래스를 빌드합니다. 간단히 말해 운하 갑문은 서로 수위가 다른 두 수역 간을 이동하는 배들을 올리고 내리는 장치입니다. 갑문에는 두 개의 수문과 수위를 변경하는 몇 가지 메커니즘이 있습니다.

정상 작동 시 배는 갑문 안의 수위와 배가 진입하는 쪽의 수위가 일치할 때 두 개의 수문 중 하나로 들어옵니다. 갑문 안에 들어오면 배가 갑문에서 나가는 쪽의 수위와 일치하도록 갑문 안 수위가 변경됩니다. 수위가 이쪽 수위와 일치하면 출구 쪽 수문이 열립니다. 조작자가 운하에서 위험한 상황을 초래하지 않도록 안전 조치가 마련되어 있습니다. 두 수문을 모두 닫은 경우에만 수위를 변경할 수 있습니다. 하나의 수문만 열려 있을 수 있습니다. 수문을 열려면 갑문 내 수위가 열려는 수문 밖의 수위와 일치해야 합니다.

이 동작을 C# 클래스를 빌드하여 모델링할 수 있습니다. `CanalLock` 클래스는 두 수문을 열거나 닫는 명령을 지원합니다. 수위를 높이거나 낮추는 다른 명령도 클래스에 포함됩니다. 클래스는 양쪽 수문의 현재 상태와 수위를 읽는 속성도 지원해야 합니다. 메서드는 안전 조치를 구현합니다.

## 클래스 정의

`CanalLock` 클래스를 테스트하는 콘솔 애플리케이션을 빌드합니다. Visual Studio 또는 .NET CLI를 사용하여 .NET 5용 콘솔 프로젝트를 새로 만듭니다. 그런 다음 새 클래스를 추가하고 이름을 `CanalLock`으로 지정합니다. 다음으로 공용 API를 디자인하되 메서드는 구현하지 않은 상태로 둡니다.

C#

```
public enum WaterLevel
{
    Low,
    High
}
public class CanalLock
{
    // Query canal lock state:
    public WaterLevel CanalLockWaterLevel { get; private set; } =
        WaterLevel.Low;
    public bool HighWaterGateOpen { get; private set; } = false;
    public bool LowWaterGateOpen { get; private set; } = false;

    // Change the upper gate.
    public void SetHighGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change the lower gate.
    public void SetLowGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change water level.
    public void SetWaterLevel(WaterLevel newLevel)
    {
        throw new NotImplementedException();
    }

    public override string ToString() =>
        $"The lower gate is {(LowWaterGateOpen ? "Open" : "Closed")}. " +
        $"The upper gate is {(HighWaterGateOpen ? "Open" : "Closed")}. " +
        $"The water level is {CanalLockWaterLevel}.";
```

앞의 코드는 개체를 초기화하므로 두 수문이 모두 닫혀 있고 수위는 낮습니다. 그런 다음 클래스의 첫 번째 구현을 만들 때 지침이 될 다음 테스트 코드를 `Main` 메서드에 작성합니다.

C#

```

// Create a new canal lock:
var canalGate = new CanalLock();

// State should be doors closed, water level low:
Console.WriteLine(canalGate);

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate: {canalGate}");

Console.WriteLine("Boat enters lock from lower gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate: {canalGate}");

canalGate.SetWaterLevel(WaterLevel.High);
Console.WriteLine($"Raise the water level: {canalGate}");

canalGate.SetHighGate(open: true);
Console.WriteLine($"Open the higher gate: {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");
Console.WriteLine("Boat enters lock from upper gate");

canalGate.SetHighGate(open: false);
Console.WriteLine($"Close the higher gate: {canalGate}");

canalGate.SetWaterLevel(WaterLevel.Low);
Console.WriteLine($"Lower the water level: {canalGate}");

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate: {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate: {canalGate}");

```

다음으로 `CanalLock` 클래스에서 각 메서드의 첫 번째 구현을 추가합니다. 다음 코드는 안전 규칙을 고려하지 않고 클래스의 메서드를 구현합니다. 안전 테스트는 나중에 추가합니다.

C#

```

// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = open;
}

// Change the lower gate.
public void SetLowGate(bool open)

```

```

{
    LowWaterGateOpen = open;
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = newLevel;
}

```

지금까지 작성한 테스트는 통과됩니다. 기초를 구현했습니다. 이제 첫 번째 실패 조건에 대한 테스트를 작성합니다. 이전 테스트의 끝에서는 두 수문이 모두 닫혀 있고 수위가 낮음으로 설정되었습니다. 상류 수문을 열려고 시도하는 테스트를 추가합니다.

C#

```

Console.WriteLine("=====");
Console.WriteLine("      Test invalid commands");
// Open "wrong" gate (2 tests)
try
{
    canalGate = new CanalLock();
    canalGate.SetHighGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("Invalid operation: Can't open the high gate. Water is
low.");
}
Console.WriteLine($"Try to open upper gate: {canalGate}");

```

수문이 열리기 때문에 이 테스트는 실패합니다. 첫 번째 구현으로 다음 코드를 사용하여 이를 해결할 수 있습니다.

C#

```

// Change the upper gate.
public void SetHighGate(bool open)
{
    if (open && (CanalLockWaterLevel == WaterLevel.High))
        HighWaterGateOpen = true;
    else if (open && (CanalLockWaterLevel == WaterLevel.Low))
        throw new InvalidOperationException("Cannot open high gate when the
water is low");
}

```

테스트에 통과됩니다. 하지만 더 많은 테스트를 추가함에 따라 점점 더 많은 `if` 절을 추가하고 여러 속성을 테스트하게 됩니다. 추가하는 조건이 많아지면 이러한 메서드는 금방 너무 복잡해집니다.

# 패턴을 사용하여 명령 구현

더 나은 방법은 패턴을 사용하여 *개체*가 유효한 상태인지 확인하여 명령을 실행하는 것입니다. 명령이 세 변수(수문 상태, 수위, 새 설정)의 함수로 허용되는지 여부를 표현할 수 있습니다.

새 설정	수문 상태	수위	결과
닫힘	닫힘	높음	닫힘
닫힘	닫힘	낮음	닫힘
닫힘	시작	높음	닫힘
닫힘	시작	낮음	닫힘
시작	닫힘	높음	시작
시작	닫힘	낮음	닫힘(오류)
시작	시작	높음	시작
시작	시작	낮음	닫힘(오류)

테이블의 네 번째 및 마지막 행은 잘못되었기 때문에 텍스트에 취소선이 사용됩니다. 이제 추가할 코드는 수위가 낮을 때 상류 수문이 절대 열리지 않도록 해야 합니다. 이러한 상태는 단일 switch 식으로 코딩할 수 있습니다(`false` 가 "닫힘"을 나타냅을 명심하세요).

C#

```
HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel) switch
{
    (false, false, WaterLevel.High) => false,
    (false, false, WaterLevel.Low) => false,
    (false, true, WaterLevel.High) => false,
    (false, true, WaterLevel.Low) => false, // should never happen
    (true, false, WaterLevel.High) => true,
    (true, false, WaterLevel.Low) => throw new
        InvalidOperationException("Cannot open high gate when the water is low"),
    (true, true, WaterLevel.High) => true,
    (true, true, WaterLevel.Low) => false, // should never happen
};
```

이 버전을 시험해 보세요. 테스트가 통과되면 코드의 유효성이 검사됩니다. 전체 테이블은 입력과 결과의 가능한 조합을 보여 줍니다. 즉, 개발자는 테이블을 빠르게 살펴보고 가능한 모든 입력이 포함된 것을 확인할 수 있습니다. 컴파일러를 사용하면 훨씬 더 쉬워집니다. 이전 코드를 추가한 후 컴파일러에서 경고를 생성하는 것을 볼 수 있습니다. CS8524 는 스위치 식이 가능한 모든 입력을 포함하지 않음을 나타냅니다. 이 경고가 발생

하는 이유는 입력 중 하나가 `enum` 형식이기 때문입니다. 컴파일러는 "가능한 모든 입력"을 기본 형식(일반적으로 `int`)의 모든 입력으로 해석합니다. 이 `switch` 식은 `enum`에서 선언된 값만 검사합니다. 경고를 제거하려면 식의 마지막 암(arm)에 대해 catch-all 무시 패턴을 추가하면 됩니다. 이 조건은 잘못된 입력을 나타내므로 예외를 `throw`합니다.

C#

```
_ => throw new InvalidOperationException("Invalid internal state"),
```

위의 `switch arm`은 모든 입력과 일치하므로 `switch` 식에서 마지막에 와야 합니다. 앞 순서로 옮겨 실험해 보세요. 그러면 패턴의 연결할 수 없는 코드를 나타내는 컴파일러 오류 CS8510이 발생합니다. `switch` 식의 자연적 구조를 사용하면 컴파일러가 가능한 실수에 대한 오류 및 경고를 생성할 수 있습니다. "safety net" 컴파일러를 사용하면 더 적은 반복으로 올바른 코드를 보다 쉽게 만들 수 있으며, `switch arm`을 와일드카드와 조합할 수 있습니다. 컴파일러는 조합으로 인해 예상하지 못한 연결 불가능한 arm이 발생하는 경우 오류를 생성하고, 필요하지 않은 arm을 제거하는 경우 경고를 생성합니다.

첫 번째 변경은 명령이 수문 닫기인 모든 arm을 결합하는 것입니다. 이는 항상 허용됩니다. `switch` 식의 첫 번째 arm으로 다음 코드를 추가합니다.

C#

```
(false, _, _) => false,
```

이전 `switch arm`을 추가한 후 명령이 `false`인 각 arm에 하나씩 4개의 컴파일러 오류가 발생합니다. 이러한 arm은 새로 추가된 arm에 이미 포함되어 있습니다. 이 네 줄은 안전하게 제거할 수 있습니다. 이 새로운 `switch arm`은 이 조건을 대체하기 위한 것입니다.

다음으로 명령이 수문 열기인 네 개의 arm을 단순화할 수 있습니다. 수위가 높은 두 경우 모두 수문을 열 수 있습니다. (하나는 이미 열려 있습니다.) 수위가 낮은 한 가지 경우는 예외를 `throw`하고 다른 사례는 발생하지 않아야 합니다. 갑문이 이미 잘못된 상태인 경우 동일한 예외를 안전하게 `throw`해야 합니다. 이러한 arm에 대해 다음과 같은 단순화를 만들 수 있습니다.

C#

```
(true, _, WaterLevel.High) => true,
(true, false, WaterLevel.Low) => throw new InvalidOperationException("Cannot
open high gate when the water is low"),
_ => throw new InvalidOperationException("Invalid internal state"),
```

테스트를 다시 실행하면 통과됩니다. `SetHighGate` 메서드의 최종 버전은 다음과 같습니다.

C#

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel)
    switch
    {
        (false, _, _)          => false,
        (true, _, WaterLevel.High) => true,
        (true, false, WaterLevel.Low) => throw new
        InvalidOperationException("Cannot open high gate when the water is low"),
                                         => throw new
        InvalidOperationException("Invalid internal state"),
    };
}
```

## 직접 패턴 구현

방법을 살펴보았으므로 이제 `SetLowGate` 및 `SetWaterLevel` 메서드를 직접 입력합니다. 먼저 다음 코드를 추가하여 이러한 메서드에 대한 잘못된 작업을 테스트합니다.

C#

```
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetLowGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't open the lower gate. Water
is high.");
}
Console.WriteLine($"Try to open lower gate: {canalGate}");
// change water level with gate open (2 tests)
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetLowGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.High);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't raise water when the lower
```

```

        gate is open.");
    }
    Console.WriteLine($"Try to raise water with lower gate open: {canalGate}");
    Console.WriteLine();
    Console.WriteLine();
    try
    {
        canalGate = new CanalLock();
        canalGate.SetWaterLevel(WaterLevel.High);
        canalGate.SetHighGate(open: true);
        canalGate.SetWaterLevel(WaterLevel.Low);
    }
    catch (InvalidOperationException)
    {
        Console.WriteLine("invalid operation: Can't lower water when the high
gate is open.");
    }
    Console.WriteLine($"Try to lower water with high gate open: {canalGate}");

```

애플리케이션을 다시 실행합니다. 새 테스트가 실패하는 것을 볼 수 있고, 문하 갑문이 잘못된 상태가 됩니다. 나머지 메서드를 직접 구현해 보세요. 하류 수문을 설정하는 메서드는 상류 수문을 설정하는 메서드와 비슷해야 합니다. 수위를 변경하는 메서드에 포함된 검사는 서로 다르지만 비슷한 구조를 따라야 합니다. 수위를 설정하는 메서드에 동일한 프로세스를 사용하는 것이 유용할 수 있습니다. 두 게이트의 상태, 수위의 현재 상태 및 요청된 새 수위의 네 가지 입력으로 시작합니다. switch 식은 다음으로 시작해야 합니다.

C#

```

CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen,
HighWaterGateOpen) switch
{
    // elided
};

```

입력할 switch arm은 총 16개입니다. 그런 다음 테스트하고 단순화합니다.

만든 메서드가 다음과 비슷합니까?

C#

```

// Change the lower gate.
public void SetLowGate(bool open)
{
    LowWaterGateOpen = (open, LowWaterGateOpen, CanalLockWaterLevel) switch
    {
        (false, _, _) => false,
        (true, _, WaterLevel.Low) => true,
        (true, false, WaterLevel.High) => throw new
InvalidOperationException("Cannot open high gate when the water is low"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}

```

```
};

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen,
HighWaterGateOpen) switch
    {
        (WaterLevel.Low, WaterLevel.Low, true, false) => WaterLevel.Low,
        (WaterLevel.High, WaterLevel.High, false, true) => WaterLevel.High,
        (WaterLevel.Low, _, false, false) => WaterLevel.Low,
        (WaterLevel.High, _, false, false) => WaterLevel.High,
        (WaterLevel.Low, WaterLevel.High, false, true) => throw new
InvalidOperationException("Cannot lower water when the high gate is open"),
        (WaterLevel.High, WaterLevel.Low, true, false) => throw new
InvalidOperationException("Cannot raise water when the low gate is open"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}
```

테스트에 통과되어야 하며, 운하 갑문이 안전하게 작동해야 합니다.

## 요약

이 자습서에서는 개체의 내부 상태에 변경 내용을 적용하기 전에 패턴 일치를 사용하여 해당 상태를 확인하는 방법을 배웠습니다. 속성의 조합을 검사할 수 있습니다. 이러한 전환을 위한 테이블을 빌드한 후 코드를 테스트한 다음 가독성 및 유지 관리를 위해 단순화하세요. 이러한 초기 리팩터링은 내부 상태의 유효성을 검사하거나 다른 API 변경을 관리하는 추가 리팩터링을 시사할 수 있습니다. 이 자습서에서는 클래스와 개체를 더 많은 데 이터 지향 및 패턴 기반 접근 방식과 결합하여 이러한 클래스를 구현했습니다.

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# 자습서: 사용자 지정 문자열 보간 처리기 작성

아티클 • 2023. 04. 06.

이 자습서에서 학습할 방법은 다음과 같습니다.

- ✓ 문자열 보간 처리기 패턴 구현
- ✓ 문자열 보간 작업에서 수신기와 상호 작용
- ✓ 문자열 보간 처리기에 인수 추가
- ✓ 문자열 보간을 위한 새로운 라이브러리 기능 이해

## 사전 요구 사항

C# 10 컴파일러를 포함하여 .NET 6을 실행하도록 컴퓨터를 설정해야 합니다. C# 10 컴파일러는 [Visual Studio 2022](#) 또는 [.NET 6 SDK](#) 부터 사용할 수 있습니다.

이 자습서에서는 여러분이 Visual Studio 또는 .NET CLI를 비롯한 C# 및 .NET에 익숙하다고 가정합니다.

## 새로운 개요

C# 10은 사용자 지정 '보간된 문자열 처리기'에 대한 지원을 추가합니다. 보간된 문자열 처리기는 보간된 문자열에서 자리 표시자 식을 처리하는 형식입니다. 사용자 지정 처리기가 없으면 자리 표시자는 `String.Format`과 비슷하게 처리됩니다. 각 자리 표시자는 텍스트 형식으로 지정되고 구성 요소는 결합되어 결과 문자열을 구성합니다.

결과 문자열에 관한 정보를 사용하는 모든 시나리오를 위한 처리기를 작성할 수 있습니다. 사용되나요? 형식에 대한 제약 조건은 무엇인가요? 일부 사례:

- 결과 문자열이 80자 등의 일부 한도보다 크지 않아야 할 수 있습니다. 보간된 문자열을 처리하여 고정 길이 버퍼를 채우고 해당 버퍼 길이에 도달하면 처리를 중지할 수 있습니다.
- 테이블 형식이 있을 수 있으며 각 자리 표시자에는 고정된 길이가 있어야 합니다. 사용자 지정 처리기는 모든 클라이언트 코드가 준수하도록 강제하는 대신 이를 적용 할 수 있습니다.

이 자습서에서는 핵심 성능 시나리오 중 하나인 로깅 라이브러리에 대한 문자열 보간 처리기를 만듭니다. 구성된 로그 수준에 따라 로그 메시지를 생성하는 작업이 필요하지 않습니다. 로깅이 꺼져 있으면 보간된 문자열 식에서 문자열을 생성하는 작업이 필요하지

않습니다. 메시지는 인쇄되지 않으므로 문자열 연결을 건너뛸 수 있습니다. 또한 스택 추적 생성을 포함하여 자리 표시자에서 사용되는 식은 수행할 필요가 없습니다.

보간된 문자열 처리기는 형식이 지정된 문자열이 사용되는지 확인할 수 있으며 필요한 경우에만 필요한 작업을 수행합니다.

## 초기 구현

다양한 수준을 지원하는 기본 `Logger` 클래스에서 시작하겠습니다.

C#

```
public enum LogLevel
{
    Off,
    Critical,
    Error,
    Warning,
    Information,
    Trace
}

public class Logger
{
    public LogLevel EnabledLevel { get; init; } = LogLevel.Error;

    public void LogMessage(LogLevel level, string msg)
    {
        if (EnabledLevel < level) return;
        Console.WriteLine(msg);
    }
}
```

이 `Logger`는 6가지 수준을 지원합니다. 메시지가 로그 수준 필터를 전달하지 않는 경우 출력이 없습니다. 로거의 퍼블릭 API는 (완전히 형식이 지정된) 문자열을 메시지로 허용합니다. 문자열을 만드는 모든 작업이 이미 수행되었습니다.

## 처리기 패턴 구현

이 단계는 현재 동작을 다시 만드는 '보간된 문자열 처리기'를 빌드하는 것입니다. 보간된 문자열 처리기는 다음 특징이 있어야 하는 형식입니다.

- 형식에 적용된 `System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute`.
- 두 개의 `int` 매개 변수인 `literalLength` 및 `formatCount`가 있는 생성자. (추가 매개 변수가 허용됨).

- `public void AppendLiteral(string s)` 시그니처가 있는 `public AppendLiteral` 메서드.
- `public void AppendFormattted<T>(T t)` 시그니처가 있는 제네릭 `public AppendFormattted` 메서드.

내부적으로 작성기는 형식이 지정된 문자열을 만들고 클라이언트가 해당 문자열을 검색할 수 있도록 멤버를 제공합니다. 다음 코드는 다음 요구 사항을 충족하는 `LogInterpolatedStringHandler` 형식을 보여 줍니다.

C#

```
[InterpolatedStringHandler]
public ref struct LogInterpolatedStringHandler
{
    // Storage for the built-up string
    StringBuilder builder;

    public LogInterpolatedStringHandler(int literalLength, int
formattedCount)
    {
        builder = new StringBuilder(literalLength);
        Console.WriteLine($"\\tliteral length: {literalLength},\n
formattedCount: {formattedCount}");
    }

    public void AppendLiteral(string s)
    {
        Console.WriteLine($"\\tAppendLiteral called: {{s}}");

        builder.Append(s);
        Console.WriteLine($"\\tAppended the literal string");
    }

    public void AppendFormattted<T>(T t)
    {
        Console.WriteLine($"\\tAppendFormattted called: {{t}} is of type\n
{typeof(T)}");

        builder.Append(t?.ToString());
        Console.WriteLine($"\\tAppended the formatted object");
    }

    internal string GetFormattedText() => builder.ToString();
}
```

이제 `Logger` 클래스의 `LogMessage`에 오버로드를 추가하여 새 보간된 문자열 처리기를 시도할 수 있습니다.

C#

```
public void LogMessage(LogLevel level, LogInterpolatedStringHandler builder)
{
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.GetFormattedText());
}
```

원래 `LogMessage` 메서드를 제거할 필요가 없습니다. 컴파일러는 인수가 보간된 문자열 식인 경우 `string` 매개 변수가 있는 메서드보다 보간된 처리기 매개 변수가 있는 메서드를 선호합니다.

다음 코드를 주 프로그램으로 사용하여 새 처리기가 호출되는지 확인할 수 있습니다.

C#

```
var logger = new Logger() { EnabledLevel = LogLevel.Warning };
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}. This
is an error. It will be printed.");
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time}. This
won't be printed.");
logger.LogMessage(LogLevel.Warning, "Warning Level. This warning is a
string, not an interpolated string expression.");
```

애플리케이션을 실행하면 다음 텍스트와 유사한 출력이 생성됩니다.

PowerShell

```
literal length: 65, formattedCount: 1
AppendLiteral called: {Error Level. CurrentTime: }
Appended the literal string
AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
    Appended the formatted object
    AppendLiteral called: {. This is an error. It will be printed.}
    Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error. It will
be printed.
    literal length: 50, formattedCount: 1
    AppendLiteral called: {Trace Level. CurrentTime: }
    Appended the literal string
    AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
    Appended the formatted object
    AppendLiteral called: {. This won't be printed.}
    Appended the literal string
Warning Level. This warning is a string, not an interpolated string
expression.
```

출력을 통해 추적하면 컴파일러가 처리기를 호출하고 문자열을 빌드하는 코드를 추가하는 방법을 확인할 수 있습니다.

- 컴파일러는 처리기를 생성하는 호출을 추가하여 형식 문자열의 리터럴 텍스트 총 길이와 자리 표시자 수를 전달합니다.
- 컴파일러는 리터럴 문자열의 각 섹션과 각 자리 표시자에 대해 `AppendLiteral` 및 `AppendFormattted`에 대한 호출을 추가합니다.
- 컴파일러는 `CoreInterpolatedStringHandler`를 인수로 사용하여 `LogMessage` 메서드를 호출합니다.

마지막으로, 마지막 경고는 보간된 문자열 처리기를 호출하지 않습니다. 인수는 `string` 이므로 호출은 문자열 매개 변수가 있는 다른 오버로드를 호출합니다.

## 처리기에 더 많은 기능 추가

이전 버전의 보간된 문자열 처리기는 패턴을 구현합니다. 모든 자리 표시자 식을 처리하지 않으려면 처리기에 더 많은 정보가 필요합니다. 이 섹션에서는 생성된 문자열이 로그에 기록되지 않을 때 더 적은 작업을 수행하도록 처리기를 개선합니다.

`System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute`를 사용하여 퍼블릭 API에 대한 매개 변수와 처리기의 생성자에 대한 매개 변수 간에 매핑을 지정합니다. 이렇게 하면 보간된 문자열을 평가해야 하는지 결정하는 데 필요한 정보가 처리기에 제공됩니다.

처리기 변경을 시작하겠습니다. 먼저 처리기가 사용되는지 추적할 필드를 추가합니다. 생성자에 두 개의 매개 변수를 추가합니다. 하나는 이 메시지의 로그 수준을 지정하고 다른 하나는 로그 개체에 대한 참조입니다.

C#

```
private readonly bool enabled;

public LogInterpolatedStringHandler(int literalLength, int formattedCount,
Logger logger, LogLevel logLevel)
{
    enabled = logger.EnabledLevel >= logLevel;
    builder = new StringBuilder(literalLength);
    Console.WriteLine($"\\tliteral length: {literalLength}, formattedCount:
{formattedCount}");
}
```

다음으로, 마지막 문자열이 사용될 때 처리기가 리터럴 또는 형식이 지정된 개체만 추가하도록 필드를 사용합니다.

C#

```

public void AppendLiteral(string s)
{
    Console.WriteLine($"\\tAppendLiteral called: {{s}}");
    if (!enabled) return;

    builder.Append(s);
    Console.WriteLine($"\\tAppended the literal string");
}

public void AppendFormatted<T>(T t)
{
    Console.WriteLine($"\\tAppendFormatted called: {{t}} is of type
{typeof(T)}");
    if (!enabled) return;

    builder.Append(t?.ToString());
    Console.WriteLine($"\\tAppended the formatted object");
}

```

그런 다음, 컴파일러가 추가 매개 변수를 처리기의 생성자에 전달하도록 `LogMessage` 선언을 업데이트해야 합니다. 이는 처리기 인수에서 `System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute`를 사용하여 처리됩니다.

C#

```

public void LogMessage(LogLevel level,
[InterpolatedStringHandlerArgument("", "level")]
LogInterpolatedStringHandler builder)
{
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.GetFormattedMessage());
}

```

이 특성은 필수 `literalLength` 및 `formattedCount` 매개 변수 뒤에 오는 매개 변수에 매핑되는 `LogMessage`에 대한 인수 목록을 지정합니다. 빈 문자열("")은 수신기를 지정합니다. 컴파일러는 `this`가 나타내는 `Logger` 개체의 값을 처리기의 생성자에 대한 다음 인수로 대체합니다. 컴파일러는 `level` 값을 다음 인수로 대체합니다. 작성하는 모든 처리기에 대해 원하는 개수의 인수를 제공할 수 있습니다. 추가하는 인수는 문자열 인수입니다.

동일한 테스트 코드를 사용하여 이 버전을 실행할 수 있습니다. 이번에는 다음 결과가 표시됩니다.

PowerShell

```

literal length: 65, formattedCount: 1
AppendLiteral called: {Error Level. CurrentTime: }
Appended the literal string

```

```

AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
    Appended the formatted object
    AppendLiteral called: {. This is an error. It will be printed.}
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error. It will
be printed.
    literal length: 50, formattedCount: 1
    AppendLiteral called: {Trace Level. CurrentTime: }
    AppendFormatted called: {10/20/2021 12:19:10 PM} is of type
System.DateTime
    AppendLiteral called: {. This won't be printed.}
Warning Level. This warning is a string, not an interpolated string
expression.

```

`AppendLiteral` 및 `AppendFormat` 메서드가 호출되고 있지만 작업을 수행하지 않는 것을 알 수 있습니다. 처리기는 마지막 문자열이 필요하지 않다고 결정했으므로 해당 문자열을 빌드하지 않습니다. 그래도 몇 가지 개선해야 할 사항이 있습니다.

먼저 `System.IFormattable`를 구현하는 형식으로 인수를 제한하는 `AppendFormatted`의 오버로드를 추가할 수 있습니다. 이 오버로드를 사용하면 호출자가 자리 표시자에 형식 문자열을 추가할 수 있습니다. 이렇게 변경하는 동안 다른 `AppendFormatted` 및 `AppendLiteral` 메서드의 반환 형식도 `void`에서 `bool`로 변경해 보겠습니다(이러한 메서드에 다른 반환 형식이 있는 경우 컴파일 오류가 발생함). 이 변경은 '단락'을 가능하게 합니다. 메서드는 `false`를 반환하여 보간된 문자열 식의 처리를 중지해야 함을 나타냅니다. `true`를 반환하면 처리를 계속해야 함을 나타냅니다. 이 예제에서는 결과 문자열이 필요하지 않을 때 처리를 중지하는 데 해당 메서드를 사용하고 있습니다. 단락은 보다 세분화된 작업을 지원합니다. 고정 길이 버퍼를 지원하기 위해 특정 길이에 도달하면 식 처리를 중지할 수 있습니다. 또는 일부 조건은 나머지 요소가 필요하지 않음을 나타낼 수 있습니다.

C#

```

public void AppendFormatted<T>(T t, string format) where T : IFormattable
{
    Console.WriteLine($"\\tAppendFormatted (IFormattable version) called: {t}
with format {{format}} is of type {typeof(T)},");

    builder.Append(t?.ToString(format, null));
    Console.WriteLine($"\\tAppended the formatted object");
}

```

이 추가를 통해 보간된 문자열 식에서 형식 문자열을 지정할 수 있습니다.

C#

```
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}. The
time doesn't use formatting.");
logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time:t}. This
is an error. It will be printed.");
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time:t}. This
won't be printed.");
```

첫 번째 메시지의 `:t`는 현재 시간의 “짧은 시간 형식”을 지정합니다. 이전 예제에서는 처리기에 대해 만들 수 있는 `AppendFormatted` 메서드에 대한 오버로드 중 하나를 보여 주었습니다. 형식이 지정되는 개체의 제네릭 인수를 지정할 필요는 없습니다. 만드는 형식을 문자열로 변환하는 더 효율적인 방법이 있을 수 있습니다. 제네릭 인수 대신 해당 형식을 사용하는 `AppendFormatted`의 오버로드를 작성할 수 있습니다. 컴파일러는 최적 오버로드를 선택합니다. 런타임은 이 기술을 사용하여 `System.Span<T>`을 문자열 출력으로 변환합니다. 정수 매개 변수를 추가하여 `IFormattable`의 유무에 관계없이 출력의 ‘맞춤’을 지정할 수 있습니다. .NET 6과 함께 제공되는

`System.Runtime.CompilerServices.DefaultInterpolatedStringHandler`에는 다른 용도를 위한 `AppendFormatted`의 오버로드 9개가 포함됩니다. 용도에 맞게 처리기를 빌드하는 동안 참조로 이를 사용할 수 있습니다.

이제 샘플을 실행하면 `Trace` 메시지에 대한 첫 번째 `AppendLiteral`만 호출되는 것을 알 수 있습니다.

PowerShell

```
literal length: 60, formattedCount: 1
AppendLiteral called: Error Level. CurrentTime:
Appended the literal string
AppendFormatted called: 10/20/2021 12:18:29 PM is of type
System.DateTime
Appended the formatted object
AppendLiteral called: . The time doesn't use formatting.
Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:18:29 PM. The time doesn't use
formatting.
literal length: 65, formattedCount: 1
AppendLiteral called: Error Level. CurrentTime:
Appended the literal string
AppendFormatted (IFormattable version) called: 10/20/2021 12:18:29
PM with format {t} is of type System.DateTime,
Appended the formatted object
AppendLiteral called: . This is an error. It will be printed.
Appended the literal string
Error Level. CurrentTime: 12:18 PM. This is an error. It will be printed.
literal length: 50, formattedCount: 1
AppendLiteral called: Trace Level. CurrentTime:
```

Warning Level. This warning is a string, not an interpolated string expression.

효율성을 개선하는 처리기의 생성자에 대한 하나의 마지막 업데이트를 수행할 수 있습니다. 처리기는 마지막 `out bool` 매개 변수를 추가할 수 있습니다. 해당 매개 변수를 `false`로 설정하면 보간된 문자열 식을 처리하기 위해 처리기를 호출하지 않아야 함을 나타냅니다.

C#

```
public LogInterpolatedStringHandler(int literalLength, int formattedCount,
Logger logger, LogLevel level, out bool isEnabled)
{
    isEnabled = logger.EnabledLevel >= level;
    Console.WriteLine($"\\tliteral length: {literalLength}, formattedCount:
{formattedCount}");
    builder = isEnabled ? new StringBuilder(literalLength) : default!;
}
```

이러한 변경은 이제 `enabled` 필드를 제거할 수 있음을 의미합니다. 필드를 제거하면 `AppendLiteral` 및 `AppendFormatted` 반환 형식을 `void`로 변경할 수 있습니다. 이제 샘플을 실행하면 다음 출력이 표시됩니다.

PowerShell

```
literal length: 60, formattedCount: 1
AppendLiteral called: Error Level. CurrentTime:
Appended the literal string
AppendFormatted called: 10/20/2021 12:19:10 PM is of type
System.DateTime
    Appended the formatted object
    AppendLiteral called: . The time doesn't use formatting.
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. The time doesn't use
formatting.
    literal length: 65, formattedCount: 1
    AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted (IFormattable version) called: 10/20/2021 12:19:10
PM with format {t} is of type System.DateTime,
        Appended the formatted object
        AppendLiteral called: . This is an error. It will be printed.
            Appended the literal string
Error Level. CurrentTime: 12:19 PM. This is an error. It will be printed.
    literal length: 50, formattedCount: 1
Warning Level. This warning is a string, not an interpolated string
expression.
```

`LogLevel.Trace` 가 지정된 경우 유일한 출력은 생성자의 출력입니다. 처리기가 해당 항목이 사용되지 않음을 나타냈으므로 `Append` 메서드가 호출되지 않았습니다.

이 예제에서는 특히 로깅 라이브러리가 사용되는 경우 보간된 문자열 처리기의 중요한 내용을 보여 줍니다. 자리 표시자의 부작용이 발생하지 않을 수 있습니다. 주 프로그램에 다음 코드를 추가하고 이 동작이 작동하는지 확인합니다.

C#

```
int index = 0;
int numberOfIncrements = 0;
for (var level = LogLevel.Critical; level <= LogLevel.Trace; level++)
{
    Console.WriteLine(level);
    logger.LogMessage(level, $"{level}: Increment index a few times
{index++}, {index++}, {index++}, {index++}, {index++}");
    numberOfIncrements += 5;
}
Console.WriteLine($"Value of index {index}, value of numberOfIncrements:
{numberOfIncrements}");
```

루프가 반복될 때마다 `index` 변수가 5배 증분됨을 알 수 있습니다. 자리 표시자는 `Critical`, `Error`, `Warning`에 대해서만 평가되며 `Information`, `Trace`에 대해서는 평가되지 않으므로 `index`의 마지막 값이 예상과 일치하지 않습니다.

PowerShell

```
Critical
Critical: Increment index a few times 0, 1, 2, 3, 4
Error
Error: Increment index a few times 5, 6, 7, 8, 9
Warning
Warning: Increment index a few times 10, 11, 12, 13, 14
Information
Trace
Value of index 15, value of numberOfIncrements: 25
```

보간된 문자열 처리기를 사용하면 보간된 문자열 식이 문자열로 변환되는 방식을 더 효과적으로 제어할 수 있습니다. .NET 런타임 팀은 여러 영역에서 성능을 향상하는 데 이미 이 기능을 사용했습니다. 고유한 라이브러리에서 동일한 기능을 사용할 수 있습니다. 자세히 살펴보려면 [System.Runtime.CompilerServices.DefaultInterpolatedStringHandler](#)를 참조하세요. 여기서 빌드한 것보다 더 완벽한 구현을 제공합니다. `Append` 메서드에 가능한 더 많은 오버로드가 표시됩니다.

# 레코드 종류 만들기

아티클 • 2023. 11. 14.

레코드는 값 기반 같음을 사용하는 형식입니다. C# 10은 레코드를 값 형식으로 정의할 수 있도록 '레코드 구조체'를 추가합니다. 레코드 종류의 두 변수는 레코드 종류 정의가 동일한 경우와 모든 필드에 대해 두 레코드의 값이 같은 경우에 같습니다. 클래스 형식의 두 변수는 참조되는 개체가 동일한 클래스 형식이고 변수가 동일한 개체를 참조하는 경우에 같습니다. 값 기반 같음은 레코드 종류에 필요할 수 있는 다른 기능을 의미합니다. 컴파일러는 `class` 대신 `record`를 선언할 때 이러한 멤버 대부분을 생성합니다. 컴파일러는 `record struct` 형식에 대해 동일한 메서드를 생성합니다.

이 자습서에서는 다음 작업을 수행하는 방법을 알아봅니다.

- ✓ 형식에 `record` 한정자를 `class` 추가할지 여부를 결정합니다.
- ✓ 레코드 종류 및 위치 레코드 종류를 선언합니다.
- ✓ 레코드에서 컴파일러 생성 메서드를 자신의 메서드로 대체합니다.

## 필수 조건

C# 10 이상 컴파일러를 포함하여 .NET 6 이상을 실행하도록 컴퓨터를 설정해야 합니다. C# 10 컴파일러는 [Visual Studio 2022](#) 또는 [.NET 6 SDK](#) 부터 사용할 수 있습니다.

## 레코드의 특징

키워드(keyword) 사용하여 형식 `record` 을 선언하거나 `struct` 선언을 수정하여 레코드를 `class` 정의합니다. 필요에 따라 키워드(keyword) 생략 `class` 하여 를 만들 수 있습니다 `record class`. 레코드는 값 기반 같음 의미 체계를 따릅니다. 값 의미 체계를 적용하기 위해 컴파일러는 레코드 형식(`record class` 및 `record struct` 형식 모두)에 대해 여러 가지 메서드를 생성합니다.

- `Object.Equals(Object)`의 재정의.
- 매개 변수가 레코드 종류인 가상 `Equals` 메서드.
- `Object.GetHashCode()`의 재정의.
- `operator ==` 및 `operator !=`에 대한 메서드.
- 레코드 종류는 `System.IEquatable<T>`를 구현.

또한 레코드는 `Object.ToString()`의 재정의를 제공합니다. 컴파일러는 `Object.ToString()`을 사용하여 레코드를 표시하기 위해 메서드를 합성합니다. 이 자습서에 대한 코드를 작성할 때 이러한 멤버를 살펴봅니다. 레코드는 레코드의 비파괴적 변경을 사용하도록 설정하는 `with` 식을 지원합니다.

더욱 간결한 구문을 사용하여 '위치 레코드'를 선언할 수도 있습니다. 컴파일러는 위치 레코드를 선언할 때 더 많은 메서드를 합성합니다.

- 매개 변수가 레코드 선언의 위치 매개 변수와 일치하는 기본 생성자.
- 기본 생성자의 각 매개 변수에 대한 퍼블릭 속성. 이러한 속성은 `record class` 및 `readonly record struct` 형식에 대해 '초기화 전용'입니다. `record struct` 형식의 경우 '읽기-쓰기'입니다.
- 레코드에서 속성을 추출하는 `Deconstruct` 메서드.

## 온도 데이터 빌드

데이터 및 통계는 레코드를 사용하려는 시나리오 중 하나입니다. 이 자습서에서는 서로 다른 용도로 '도일'(degree days)을 계산하는 애플리케이션을 빌드합니다. '도일'은 일정 기간(일, 주, 월) 동안 열(또는 열 부족)을 측정한 값입니다. 도일은 에너지 사용량을 추적하고 예측합니다. 더운 날이 많을수록 에어컨 사용량이 많아지고, 추운 날이 많을수록 난로 사용량이 많아집니다. 도일은 식물 개체 수를 관리하고 계절 변화에 따라 식물 성장과 상관 관계를 얻는 데 도움이 됩니다. 도일은 기후에 따라 이동하는 동물 종의 이동을 추적하는 데 도움이 됩니다.

수식은 지정된 날짜의 평균 온도와 기준 온도를 기반으로 합니다. 시간에 따른 도일을 계산하려면 일정 기간 동안 각 날짜의 높은 온도와 낮은 온도가 필요합니다. 먼저 새 애플리케이션을 만들어 보겠습니다. 새 콘솔 애플리케이션을 만듭니다. "DailyTemperature.cs"라는 새 파일에 새 레코드 종류를 만듭니다.

C#

```
public readonly record struct DailyTemperature(double HighTemp, double LowTemp);
```

위의 코드는 '위치 레코드'를 정의합니다. `DailyTemperature` 레코드는 `readonly record struct` 인데, 그 이유는 이 레코드에서 상속할 의도가 없고 변경이 불가능해야 하기 때문입니다. `HighTemp` 및 `LowTemp` 속성은 '초기화 전용 속성'이므로 생성자에서 설정하거나 속성 이니셜라이저를 사용하여 설정할 수 있습니다. 위치 매개 변수를 읽기-쓰기로 만들려면 `readonly record struct` 대신 `record struct`를 선언합니다. `DailyTemperature` 형식에는 두 개의 속성과 일치하는 두 개의 매개 변수가 있는 '기본 생성자'도 있습니다. 기본 생성자를 사용하여 `DailyTemperature` 레코드를 초기화합니다. 다음 코드는 여러 `DailyTemperature` 레코드를 만들고 초기화합니다. 첫 번째 레코드는 명명된 매개 변수를 사용하여 `HighTemp`와 `LowTemp`를 명료화합니다. 나머지 이니셜라이저는 위치 매개 변수를 사용하여 `HighTemp`와 `LowTemp`를 초기화합니다.

C#

```
private static DailyTemperature[] data = [
    new DailyTemperature(HighTemp: 57, LowTemp: 30),
    new DailyTemperature(60, 35),
    new DailyTemperature(63, 33),
    new DailyTemperature(68, 29),
    new DailyTemperature(72, 47),
    new DailyTemperature(75, 55),
    new DailyTemperature(77, 55),
    new DailyTemperature(72, 58),
    new DailyTemperature(70, 47),
    new DailyTemperature(77, 59),
    new DailyTemperature(85, 65),
    new DailyTemperature(87, 65),
    new DailyTemperature(85, 72),
    new DailyTemperature(83, 68),
    new DailyTemperature(77, 65),
    new DailyTemperature(72, 58),
    new DailyTemperature(77, 55),
    new DailyTemperature(76, 53),
    new DailyTemperature(80, 60),
    new DailyTemperature(85, 66)
];
```

위치 레코드를 포함하여 고유한 속성 또는 메서드를 레코드에 추가할 수 있습니다. 각 날짜의 평균 온도를 계산해야 합니다. `DailyTemperature` 레코드에 해당 속성을 추가할 수 있습니다.

C#

```
public readonly record struct DailyTemperature(double HighTemp, double
LowTemp)
{
    public double Mean => (HighTemp + LowTemp) / 2.0;
}
```

이 데이터를 사용할 수 있는지 확인해 보겠습니다. `Main` 메서드에 다음 코드를 추가합니다.

C#

```
foreach (var item in data)
    Console.WriteLine(item);
```

애플리케이션을 실행하면 다음과 같은 출력이 표시됩니다(공간상의 이유로 여러 행을 제거함).

.NET CLI

```
DailyTemperature { HighTemp = 57, LowTemp = 30, Mean = 43.5 }
DailyTemperature { HighTemp = 60, LowTemp = 35, Mean = 47.5 }
```

```
DailyTemperature { HighTemp = 80, LowTemp = 60, Mean = 70 }
DailyTemperature { HighTemp = 85, LowTemp = 66, Mean = 75.5 }
```

위의 코드는 컴파일러에 의해 합성된 `ToString` 재정의의 출력을 보여 줍니다. 다른 텍스트를 선호하는 경우 컴파일러에서 버전을 합성하지 못하도록 하는 `ToString`의 고유한 버전을 작성할 수 있습니다.

## 도일 계산

도일을 계산하려면 지정된 날짜의 기준 온도와 평균 온도의 차이를 계산합니다. 시간에 따른 더위를 측정하려면 평균 온도가 기준보다 낮은 날짜를 모두 삭제합니다. 시간에 따른 추위를 측정하려면 평균 온도가 기준보다 높은 날짜를 모두 삭제합니다. 예를 들어 미국은 난방 및 냉방 도일의 기준으로 65F를 사용합니다. 이 온도는 난방 또는 냉방이 필요하지 않은 온도입니다. 하루 평균 온도가 70F인 경우 그날 냉방 도일은 5이고 난방 도일은 0입니다. 반대로 평균 온도가 55F인 경우 난방 도일은 10이고 냉방 도일은 0입니다.

이러한 수식을 레코드 형식의 작은 계층 구조, 즉 하나의 추상적 도일 형식과 그 아래에 난방 도일 및 냉방 도일이라는 구체적인 두 가지 형식으로 표현할 수 있습니다. 이러한 형식은 위치 레코드일 수도 있습니다. 기준 온도와 일련의 일일 온도를 기본 생성자의 인수로 사용합니다.

C#

```
public abstract record DegreeDays(double BaseTemperature,
IEnumerable<DailyTemperature> TempRecords);

public sealed record HeatingDegreeDays(double BaseTemperature,
IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean <
BaseTemperature).Sum(s => BaseTemperature - s.Mean);
}

public sealed record CoolingDegreeDays(double BaseTemperature,
IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean >
BaseTemperature).Sum(s => s.Mean - BaseTemperature);
}
```

추상 `DegreeDays` 레코드는 `HeatingDegreeDays` 및 `CoolingDegreeDays` 레코드의 공유 기본 클래스입니다. 파생 레코드의 기본 생성자 선언에서는 기본 레코드 초기화를 관리하는 방법을 보여 줍니다. 파생 레코드는 기본 레코드 기본 생성자의 모든 매개 변수에 대한 매개 변수를 선언합니다. 기본 레코드는 이러한 속성을 선언하고 초기화합니다. 파생 레코드는 이러한 속성을 숨기지는 않지만 기본 레코드에 선언되지 않은 매개 변수의 속성만 만들고 초기화합니다. 이 예제에서 파생 레코드는 새로운 기본 생성자 매개 변수를 추가하지 않습니다. `Main` 메서드에 다음 코드를 추가하여 코드를 테스트합니다.

C#

```
var heatingDegreeDays = new HeatingDegreeDays(65, data);
Console.WriteLine(heatingDegreeDays);

var coolingDegreeDays = new CoolingDegreeDays(65, data);
Console.WriteLine(coolingDegreeDays);
```

다음과 같은 출력이 표시됩니다.

.NET CLI

```
HeatingDegreeDays { BaseTemperature = 65, TempRecords =
record_types.DailyTemperature[], DegreeDays = 85 }
CoolingDegreeDays { BaseTemperature = 65, TempRecords =
record_types.DailyTemperature[], DegreeDays = 71.5 }
```

## 컴파일러 합성 메서드 정의

코드는 해당 기간 동안의 정확한 난방 및 냉방 도일 수를 계산합니다. 그러나 이 예제에서는 레코드에 대한 합성 메서드 중 일부를 바꿔야 하는 이유를 보여 줍니다. `clone` 메서드를 제외하고 고유한 버전의 컴파일러 합성 메서드를 레코드 종류에서 선언할 수 있습니다. `clone` 메서드는 컴파일러 생성 이름을 사용하므로 다른 구현을 제공할 수 없습니다. 이러한 합성 메서드로는 복사 생성자, `System.IEquatable<T>` 인터페이스의 멤버, 같음 및 같지 않음 테스트, `GetHashCode()`가 있습니다. 이러한 용도로 `PrintMembers`를 합성합니다. 고유한 `ToString`을 선언할 수도 있지만 `PrintMembers`는 상속 시나리오에 더 나은 옵션을 제공합니다. 고유한 버전의 합성 메서드를 제공하려면 서명이 합성 메서드와 일치해야 합니다.

콘솔 출력의 `TempRecords` 요소는 유용하지 않습니다. 이 요소는 형식만 표시하고 다른 항목은 전혀 표시하지 않습니다. 합성 `PrintMembers` 메서드의 고유한 구현을 제공하여 이 동작을 변경할 수 있습니다. 서명은 `record` 선언에 적용된 한정자에 따라 달라집니다.

- 레코드 형식이 `sealed` 또는 `record struct` 이면 시그니처는 `private bool PrintMembers(StringBuilder builder);` 입니다.
- 레코드 종류가 `sealed`가 아니고 `object`에서 파생되면(즉, 기본 레코드를 선언하지 않음) 서명은 `protected virtual bool PrintMembers(StringBuilder builder);` 입니다.
- 레코드 종류가 `sealed`가 아니고 다른 레코드에서 파생되면 서명은 `protected override bool PrintMembers(StringBuilder builder);` 입니다.

이러한 규칙은 `PrintMembers`의 용도에 대한 이해를 통해 가장 쉽게 이해할 수 있습니다. `PrintMembers`는 레코드 종류의 각 속성에 대한 정보를 문자열에 추가합니다. 계약에서는 표시할 멤버를 추가하려면 기본 레코드가 필요하며 파생 멤버는 해당 멤버를 추가하는 것으로 가정합니다. 각 레코드 종류는 `HeatingDegreeDays`에 대한 다음 예제와 유사하게 표시되는 `ToString` 재정의를 합성합니다.

C#

```
public override string ToString()
{
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append("HeatingDegreeDays");
    stringBuilder.Append(" { ");
    if (PrintMembers(stringBuilder))
    {
        stringBuilder.Append(" ");
    }
    stringBuilder.Append("}");
    return stringBuilder.ToString();
}
```

컬렉션의 형식을 출력하지 않는 `DegreeDays` 레코드에 `PrintMembers` 메서드를 선언합니다.

C#

```
protected virtual bool PrintMembers(StringBuilder stringBuilder)
{
    stringBuilder.Append($"BaseTemperature = {BaseTemperature}");
    return true;
}
```

서명은 컴파일러의 버전과 일치하도록 `virtual protected` 메서드를 선언합니다. 잘못된 접근자를 가져와도 걱정할 필요가 없습니다. 언어에서 올바른 서명을 적용합니다. 합성 메서드에 대한 올바른 한정자를 잊은 경우 컴파일러는 올바른 서명을 가져오는 데 도움이 되는 경고 또는 오류를 발생시킵니다.

C# 10 이상에서는 `ToString` 메서드를 레코드 형식에서 `sealed`로 선언할 수 있습니다. 그러면 파생된 레코드에서 새로운 구현을 제공하지 못합니다. 파생된 레코드에는 `PrintMembers` 재정의가 계속 포함됩니다. 레코드의 런타임 형식을 표시하지 않으려면 봉인 `ToString` 합니다. 위의 예제에서는 레코드가 난방 또는 냉방 도일을 측정하는 위치에 대한 정보가 손실됩니다.

## 비파괴적 변경

위치 레코드 클래스의 합성 멤버는 레코드의 상태를 수정하지 않습니다. 목표는 변경이 불가능한 레코드를 더욱 쉽게 만들 수 있도록 하는 것입니다. `readonly record struct`를 선언하여 변경이 불가능한 레코드 구조체를 만들어야 합니다. `HeatingDegreeDays` 및 `CoolingDegreeDays`에 대한 위의 선언을 다시 살펴보세요. 추가된 멤버는 레코드의 값에 대해 계산을 수행하지만 상태를 변경하지는 않습니다. 위치 레코드를 사용하면 변경이 불가능한 참조 형식을 더욱 쉽게 만들 수 있습니다.

변경이 불가능한 참조 형식을 만드는 것은 비파괴적 변경을 사용한다는 의미입니다. `with` 식을 사용하여 기존 레코드 인스턴스와 유사한 새 레코드 인스턴스를 만듭니다. 이러한 식은 복사본을 수정하는 추가 할당이 있는 복사본 생성입니다. 그러면 각 속성이 기존 레코드에서 복사되고 선택적으로 수정된 새 레코드 인스턴스가 생성됩니다. 원래 레코드는 변경되지 않습니다.

`with` 식을 보여 주는 몇 가지 기능을 프로그램에 추가해 보겠습니다. 먼저 동일한 데이터를 사용하여 증가하는 도일을 계산하는 새 레코드를 만들어 보겠습니다. 일반적으로 '증가하는 도일'은 41F를 기준으로 사용하고 기준보다 높은 온도를 측정합니다. 동일한 데이터를 사용하려면 `coolingDegreeDays` 와 유사하지만 기본 온도는 다른 새 레코드를 만들 수 있습니다.

C#

```
// Growing degree days measure warming to determine plant growing rates
var growingDegreeDays = coolingDegreeDays with { BaseTemperature = 41 };
Console.WriteLine(growingDegreeDays);
```

계산된 도수를 더 높은 기준 온도로 생성된 수와 비교할 수 있습니다. 레코드는 '참조 형식'이며 이러한 복사본은 단순 복사본입니다. 데이터의 배열은 복사되지 않지만 두 레코드 모두 동일한 데이터를 참조합니다. 이러한 사실은 또 다른 시나리오에서 장점이 됩니다. 증가하는 도일의 경우 이전 5일간의 합계를 추적하는 것이 유용합니다. `with` 식을 사용하여 다른 소스 데이터로 새 레코드를 만들 수 있습니다. 다음 코드에서는 이러한 누적의 컬렉션을 빌드하고 값을 표시합니다.

C#

```
// showing moving accumulation of 5 days using range syntax
List<CoolingDegreeDays> movingAccumulation = new();
int rangeSize = (data.Length > 5) ? 5 : data.Length;
for (int start = 0; start < data.Length - rangeSize; start++)
{
    var fiveDayTotal = growingDegreeDays with { TempRecords = data[start..
(start + rangeSize)] };
    movingAccumulation.Add(fiveDayTotal);
}
Console.WriteLine();
Console.WriteLine("Total degree days in the last five days");
foreach(var item in movingAccumulation)
{
    Console.WriteLine(item);
}
```

`with` 식을 사용하여 레코드 복사본을 만들 수도 있습니다. `with` 식의 중괄호 사이에 속성을 지정하지 마세요. 그러면 복사본을 만들고 속성은 변경하지 않습니다.

C#

```
var growingDegreeDaysCopy = growingDegreeDays with { };
```

완성된 애플리케이션을 실행하여 결과를 확인합니다.

## 요약

이 자습서에서는 레코드의 여러 측면을 보여 주었습니다. 레코드는 기본 용도가 데이터 저장인 형식에 대해 간결한 구문을 제공합니다. 개체 지향 클래스의 기본 용도는 책임을 정의하는 것입니다. 이 자습서에서는 간결한 구문을 사용하여 레코드에 대한 속성을 선언할 수 있는 ‘위치 레코드’를 집중적으로 살펴보았습니다. 컴파일러는 레코드를 복사하고 비교하기 위해 레코드의 여러 멤버를 합성합니다. 레코드 종류에 필요한 다른 멤버를 추가할 수 있습니다. 컴파일러 생성 멤버는 상태가 변경되지 않는다는 점을 이해하고 변경 불가능한 레코드 형식을 만들 수 있습니다. 그리고 `with` 식을 사용하여 비파괴적 변경을 쉽게 지원할 수 있습니다.

레코드는 형식을 정의하는 또 다른 방법을 추가합니다. `class` 정의를 사용하여 개체의 책임이나 동작에 중점을 둔 개체 지향 계층 구조를 만듭니다. 데이터를 저장할 뿐만 아니라 아주 작아서 효율적으로 복사할 수 있는 데이터 구조에 대해 `struct` 형식을 만듭니다. 값 기반 같음과 비교를 원하지만 값을 복사하지 않고 참조 변수를 사용하려는 경우 `record` 형식을 만듭니다. 효율적으로 복사할 수 있을 만큼 작은 형식의 레코드 기능을 원하는 경우 `record struct` 형식을 만듭니다.

레코드 형식에 대한 C# 언어 참조 문서와 제안된 레코드 형식 사양 및 레코드 구조체 사양에서 레코드에 대해 자세히 알아볼 수 있습니다.

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

ଓ 설명서 문제 열기

☒ 제품 사용자 의견 제공

# 자습서: 학습할 때 최상위 문을 사용하여 코드를 빌드하는 아이디어 탐색

아티클 • 2023. 11. 15.

이 자습서에서는 다음 작업을 수행하는 방법을 알아봅니다.

- ✓ 최상위 문 사용을 제어하는 규칙을 알아봅니다.
- ✓ 최상위 문을 사용하여 알고리즘을 탐색합니다.
- ✓ 탐색을 재사용 가능한 구성 요소로 리팩터링합니다.

## 필수 조건

C# 10 컴파일러를 포함하여 .NET 6을 실행하도록 컴퓨터를 설정해야 합니다. C# 10 컴파일러는 [Visual Studio 2022](#) 또는 [.NET 6 SDK](#) 부터 사용할 수 있습니다.

이 자습서에서는 여러분이 Visual Studio 또는 .NET CLI를 비롯한 C# 및 .NET에 익숙하다고 가정합니다.

## 살펴보기 시작하기

최상위 문을 사용하면 프로그램의 진입점을 클래스의 정적 메서드에 배치하여 필요한 추가 공식 절차를 방지할 수 있습니다. 새 콘솔 애플리케이션의 일반적인 시작점은 다음 코드와 같습니다.

```
C#  
  
using System;  
  
namespace Application  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.WriteLine("Hello World!");  
        }  
    }  
}
```

위 코드는 `dotnet new console` 명령을 실행하고 새 콘솔 애플리케이션을 만든 결과입니다. 11줄의 코드에 단 한 줄의 실행 코드가 포함됩니다. 새 최상위 문 기능을 사용하여 해

당 프로그램을 단순화할 수 있습니다. 이렇게 하면 해당 프로그램에서 두 개 줄을 제외한 모든 줄을 제거할 수 있습니다.

C#

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

## ① 중요

.NET 6 용 C# 템플릿은 '최상위 문'을 사용합니다. .NET 6으로 이미 업그레이드한 경우 애플리케이션이 이 문서의 코드와 일치하지 않을 수 있습니다. 자세한 내용은 [최상위 문을 생성하는 새 C# 템플릿을 참조하세요](#).

.NET 6 SDK는 다음 SDK를 사용하는 프로젝트에 대한 암시적 *global using* 지시문 집합도 추가합니다.

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker

이러한 암시적 *global using* 지시문에는 해당 프로젝트 형식의 가장 일반적인 네임 스페이스가 포함됩니다.

자세한 내용은 암시적 *using* 지시문에 대한 [문서를 참조하세요](#).

이 기능은 새로운 아이디어 탐색을 시작하는 데 필요한 작업을 간소화합니다. 스크립팅 시나리오에서 또는 탐색하는 데 최상위 문을 사용할 수 있습니다. 적용되는 기본 사항을 확인한 후 코드의 리팩터링을 시작하고 빌드한 재사용 가능 구성 요소의 메서드, 클래스 또는 기타 어셈블리를 만들 수 있습니다. 최상위 문은 빠른 실험 및 초보자 자습서를 가능하게 합니다. 또한 실험에서 전체 프로그램까지 원활한 경로를 제공합니다.

최상위 문은 파일에 표시된 순서대로 실행됩니다. 최상위 문은 애플리케이션의 한 소스 파일에만 사용할 수 있습니다. 둘 이상의 파일에서 사용하는 경우 컴파일러에서 오류를 생성합니다.

## 매직 .NET 응답 머신 빌드

이 자습서에서는 임의 응답을 사용하여 "예" 또는 "아니요" 질문에 대답하는 콘솔 애플리케이션을 빌드해 보겠습니다. 기능을 단계별로 빌드합니다. 일반적인 프로그램의 구조에 필요한 공식 절차가 아닌 작업에 집중할 수 있습니다. 그런 다음, 기능에 만족하면 필요한 경우 애플리케이션을 리팩터링할 수 있습니다.

좋은 시작점은 질문을 다시 콘솔에 쓰는 것입니다. 먼저 다음 코드를 작성할 수 있습니다.

C#

```
Console.WriteLine(args);
```

`args` 변수를 선언하지 않습니다. 최상위 문이 포함된 단일 소스 파일의 경우 컴파일러는 명령줄 인수를 의미하는 `args`를 인식합니다. 인수 형식은 모든 C# 프로그램과 같이 `string[]`입니다.

다음 `dotnet run` 명령을 실행하여 코드를 테스트할 수 있습니다.

.NET CLI

```
dotnet run -- Should I use top level statements in all my programs?
```

명령줄에 있는 `--` 뒤의 인수는 프로그램에 전달됩니다. 콘솔에 인쇄된 항목인 `args` 변수 형식을 확인할 수 있습니다.

콘솔

```
System.String[]
```

콘솔에 질문을 쓰려면 인수를 열거하고 공백으로 구분해야 합니다. `WriteLine` 호출을 다음과 코드로 바꿉니다.

C#

```
Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();
```

이제 프로그램을 실행하면 질문을 인수 문자열로 올바르게 표시합니다.

## 임의 응답으로 응답

질문을 에코한 후에는 임의 응답을 생성하는 코드를 추가할 수 있습니다. 먼저 가능한 응답의 배열을 추가합니다.

C#

```
string[] answers =
[
    "It is certain.",           "Reply hazy, try again.",      "Don't count on
it.",
    "It is decidedly so.",     "Ask again later.",            "My reply is no.",
    "Without a doubt.",        "Better not tell you now.",   "My sources say
no.",
    "Yes - definitely.",      "Cannot predict now.",         "Outlook not so
good.",
    "You may rely on it.",    "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
];
```

이 배열에는 긍정 응답 10개, 커밋이 아닌 응답 5개, 부정 응답 5개가 포함되어 있습니다. 다음으로, 다음 코드를 추가하여 배열에서 임의 응답을 생성하고 표시합니다.

C#

```
var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

애플리케이션을 다시 실행하여 결과를 볼 수 있습니다. 다음 출력과 같은 정보가 표시됩니다.

.NET CLI

```
dotnet run -- Should I use top level statements in all my programs?

Should I use top level statements in all my programs?
Better not tell you now.
```

이 코드는 질문에 응답하지만 기능을 하나 더 추가하겠습니다. 질문 앱에서 응답에 관한 생각을 시뮬레이트하려고 합니다. 이렇게 하려면 약간의 ASCII 애니메이션을 추가하고 작동 중에 일시 중지합니다. 질문을 에코하는 줄 뒤에 다음 코드를 추가합니다.

C#

```
for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b");
```

```
Console.WriteLine("/");
await Task.Delay(50);
Console.Write("\b\b\b");
Console.Write("- | ");
await Task.Delay(50);
Console.Write("\b\b\b");
Console.Write("\\" "/");
await Task.Delay(50);
Console.Write("\b\b\b");
}
Console.WriteLine();
```

또한 소스 파일의 맨 위에 `using` 문을 추가해야 합니다.

C#

```
using System.Threading.Tasks;
```

`using` 문은 파일의 다른 문 앞에 있어야 합니다. 그렇지 않으면 컴파일러 오류가 발생합니다. 프로그램을 다시 실행하여 애니메이션을 확인할 수 있습니다. 이를 통해 더 나은 환경을 제공할 수 있습니다. 원하는 대로 자연 길이를 실험합니다.

앞의 코드는 공백으로 구분된 회전하는 선 세트를 만듭니다. `await` 키워드를 추가하면 컴파일러가 프로그램 진입점을 `async` 한정자가 있는 메서드로 생성하고 `System.Threading.Tasks.Task`를 반환하도록 지시합니다. 이 프로그램은 값을 반환하지 않으므로 프로그램 진입점이 `Task`를 반환합니다. 프로그램이 정수 값을 반환하는 경우 최상위 문의 끝에 `return` 문을 추가합니다. `return` 문은 반환할 정수 값을 지정합니다. 최상위 문에 `await` 식이 포함된 경우 반환 형식은 `System.Threading.Tasks.Task<TResult>`이 됩니다.

## 미래를 위한 리팩터링

프로그램은 다음 코드와 같이 표시됩니다.

C#

```
Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

for (int i = 0; i < 20; i++)
{
```

```

        Console.Write("| -");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("/ \\");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("- |");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("\\ /");
        await Task.Delay(50);
        Console.Write("\b\b\b");
    }
    Console.WriteLine();

string[] answers =
[
    "It is certain.",      "Reply hazy, try again.",      "Don't count on
it.",
    "It is decidedly so.", "Ask again later.",            "My reply is no.",
    "Without a doubt.",   "Better not tell you now.",    "My sources say
no.",
    "Yes - definitely.",  "Cannot predict now.",         "Outlook not so
good.",
    "You may rely on it.", "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
];
var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);

```

앞의 코드는 적절합니다. 예상대로 작동합니다. 하지만 재사용할 수 없습니다. 이제 애플리케이션이 작동하고 있으므로 재사용 가능한 부분을 가져오겠습니다.

한 후보는 대기 중인 애니메이션을 표시하는 코드입니다. 해당 코드 조각은 메서드가 될 수 있습니다.

먼저 파일에서 로컬 함수를 만들 수 있습니다. 현재 애니메이션을 다음 코드로 바꿉니다.

```

C#

await ShowConsoleAnimation();

static async Task ShowConsoleAnimation()
{
    for (int i = 0; i < 20; i++)
    {
        Console.Write("| -");
    }
}

```

```

        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("/ \\");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("- |");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("\\ /");
        await Task.Delay(50);
        Console.Write("\b\b\b");
    }
    Console.WriteLine();
}

```

앞의 코드는 main 메서드 내에 로컬 함수를 만듭니다. 그래도 재사용할 수 없습니다. 따라서 해당 코드를 클래스로 추출합니다. *utilities.cs*라는 새 파일을 만들고 다음 코드를 추가합니다.

C#

```

namespace MyNamespace
{
    public static class Utilities
    {
        public static async Task ShowConsoleAnimation()
        {
            for (int i = 0; i < 20; i++)
            {
                Console.Write(" | -");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("/ \\");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("- |");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("\\ /");
                await Task.Delay(50);
                Console.Write("\b\b\b");
            }
            Console.WriteLine();
        }
    }
}

```

최상위 문이 있는 파일은 최상위 문 뒤에 파일 끝 부분에 네임스페이스 및 형식을 포함할 수도 있습니다. 그러나 이 자습서에서는 애니메이션 메서드를 별도의 파일에 배치하여 쉽게 다시 사용할 수 있도록 합니다.

마지막으로, 애니메이션 코드를 정리하여 일부 중복을 제거할 수 있습니다.

C#

```
foreach (string s in animations)
{
    Console.Write(s);
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
```

이제 전체 애플리케이션이 있으며 나중에 사용할 수 있도록 재사용 가능한 부분을 리팩터링했습니다. 아래와 같이 주 프로그램의 최종 버전에 표시된 대로 최상위 문에서 새 유틸리티 메서드를 호출할 수 있습니다.

C#

```
using MyNamespace;

Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

await Utilities.ShowConsoleAnimation();

string[] answers =
[
    "It is certain.",           "Reply hazy, try again.",      "Don't count on
it.",
    "It is decidedly so.",     "Ask again later.",            "My reply is no.",
    "Without a doubt.",        "Better not tell you now.",   "My sources say
no.",
    "Yes - definitely.",       "Cannot predict now.",         "Outlook not so
good.",
    "You may rely on it.",     "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
];

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

앞의 예제는 `Utilities.ShowConsoleAnimation`에 대한 호출을 추가하고 `using` 문을 추가합니다.

## 요약

최상위 문을 사용하면 새 알고리즘을 탐색하는 데 사용할 간단한 프로그램을 더 쉽게 만들 수 있습니다. 코드의 다른 조각을 시도하여 알고리즘을 실험할 수 있습니다. 작동 기능을 알아본 후에는 코드를 더 쉽게 유지 관리할 수 있도록 리팩터링할 수 있습니다.

최상위 문은 콘솔 애플리케이션을 기반으로 하는 프로그램을 단순화합니다. 여기에는 Azure Functions, GitHub Actions, 기타 작은 유ти리티가 포함됩니다. 자세한 내용은 [최상위 문\(C# 프로그래밍 가이드\)](#)을 참조하세요.

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 설명서 문제 열기

 제품 사용자 의견 제공

# 인덱스 및 범위

아티클 • 2023. 11. 14.

범위와 인덱스는 시퀀스의 단일 요소 또는 범위에 액세스하기 위한 간결한 구문을 제공합니다.

이 자습서에서는 다음과 같은 작업을 수행하는 방법을 알아봅니다.

- ✓ 시퀀스에서 범위에 구문을 사용합니다.
- ✓ 암시적으로 Range를 정의합니다.
- ✓ 각 시퀀스의 시작 및 끝에 대한 설계 의사 결정을 이해합니다.
- ✓ Index 및 Range 형식에 대한 시나리오를 살펴봅니다.

## 인덱스 및 범위에 대한 언어 지원

인덱스와 범위는 시퀀스에서 단일 요소 또는 범위에 액세스하기 위한 간결한 구문을 제공합니다.

이 언어 지원은 다음과 같은 두 가지 새 형식 및 두 가지 새 연산자를 사용합니다.

- System.Index는 인덱스를 시퀀스로 표현합니다.
- 시퀀스의 끝을 기준으로 인덱스를 지정하는 끝 연산<sup>^</sup>자의 인덱스입니다.
- System.Range는 시퀀스의 하위 범위를 나타냅니다.
- 범위 연산<sub>.<sub>자</sub></sub>입니다. 범위의 시작과 끝을 피연산자로 지정합니다.

인덱스에 대한 규칙을 사용하여 시작하겠습니다. sequence 배열을 고려합니다. 0 인덱스는 sequence[0]과 동일합니다. ^0 인덱스는 sequence[sequence.Length]와 동일합니다. sequence[^0] 식은 sequence[sequence.Length]처럼 예외를 throw합니다. n이 어떤 숫자 이든, 인덱스 ^n은 sequence.Length - n과 동일합니다.

C#

```
string[] words = [
    // index from start      index from end
    "The",        // 0            ^9
    "quick",      // 1            ^8
    "brown",      // 2            ^7
    "fox",        // 3            ^6
    "jumps",      // 4            ^5
    "over",       // 5            ^4
    "the",        // 6            ^3
    "lazy",       // 7            ^2
    "dog"         // 8            ^1
];                // 9 (or words.Length) ^0
```

다음과 같이 `^1` 인덱스를 사용하여 마지막 단어를 가져올 수 있습니다. 초기화 아래에 다음 코드를 추가합니다.

C#

```
Console.WriteLine($"The last word is {words[^1]}");
```

한 범위는 어떤 범위의 시작 및 끝을 지정합니다. 범위의 시작은 포함되지만, 범위의 끝은 포함되지 않으므로, 시작은 범위에 포함되고 끝은 범위에 포함되지 않습니다.

`[0..sequence.Length]` 가 전체 범위를 나타내는 것처럼 `[0..^0]` 범위는 전체 범위를 나타냅니다.

다음 코드는 "quick", "brown", "fox"라는 단어를 포함하는 하위 범위를 만듭니다. 이 하위 범위에는 `words[1]` 부터 `words[3]` 까지 포함되며, `words[4]` 요소가 범위에 없습니다.

C#

```
string[] quickBrownFox = words[1..4];
foreach (var word in quickBrownFox)
    Console.Write("< {word} >");
Console.WriteLine();
```

다음 코드는 "lazy"와 "dog"을 포함하는 범위를 반환합니다. 이 하위 범위에는 `words[^2]` 과 `words[^1]`이 포함되며, 끝 인덱스 `words[^0]`는 포함되지 않습니다. 다음 코드도 추가합니다.

C#

```
string[] lazyDog = words[^2..^0];
foreach (var word in lazyDog)
    Console.Write("< {word} >");
Console.WriteLine();
```

다음 예제는 시작만, 끝만, 그리고 시작과 끝이 모두 열린 범위를 만듭니다.

C#

```
string[] allWords = words[..]; // contains "The" through "dog".
string[] firstPhrase = words[..4]; // contains "The" through "fox"
string[] lastPhrase = words[6..]; // contains "the", "lazy" and "dog"
foreach (var word in allWords)
    Console.Write("< {word} >");
Console.WriteLine();
foreach (var word in firstPhrase)
    Console.Write("< {word} >");
Console.WriteLine();
```

```
foreach (var word in lastPhrase)
    Console.WriteLine($"< {word} >");
Console.WriteLine();
```

범위나 인덱스를 변수로 선언할 수도 있습니다. 그러면 이 변수를 [ 및 ] 문자 사이에 사용할 수 있습니다.

C#

```
Index the = ^3;
Console.WriteLine(words[the]);
Range phrase = 1..4;
string[] text = words[phrase];
foreach (var word in text)
    Console.WriteLine($"< {word} >");
Console.WriteLine();
```

다음 샘플에서는 이러한 선택에 대한 여러 이유를 보여 줍니다. x, y 및 z를 수정하여 다양한 조합을 시도해 봅니다. 실험할 때는 올바른 조합을 위해 x가 y보다 작고 y가 z보다 작은 값을 사용합니다. 새 메서드에 다음 코드를 추가합니다. 다양한 조합을 시도해 봅니다.

C#

```
int[] numbers = ..Enumerable.Range(0, 100);
int x = 12;
int y = 25;
int z = 36;

Console.WriteLine($"{numbers[^x]} is the same as {numbers[numbers.Length - x]}");
Console.WriteLine($"{numbers[x..y].Length} is the same as {y - x}");

Console.WriteLine("numbers[x..y] and numbers[y..z] are consecutive and disjoint:");
Span<int> x_y = numbers[x..y];
Span<int> y_z = numbers[y..z];
Console.WriteLine($"\\tnumbers[x..y] is {x_y[0]} through {x_y[^1]},\nnumbers[y..z] is {y_z[0]} through {y_z[^1]}");

Console.WriteLine("numbers[x..^x] removes x elements at each end:");
Span<int> x_x = numbers[x..^x];
Console.WriteLine($"\\tnumbers[x..^x] starts with {x_x[0]} and ends with\n{x_x[^1]}");

Console.WriteLine("numbers[..x] means numbers[0..x] and numbers[x..] means\nnumbers[x..^0]");
Span<int> start_x = numbers[..x];
Span<int> zero_x = numbers[0..x];
Console.WriteLine($"\\t{start_x[0]}..{start_x[^1]} is the same as
```

```

{zero_x[0]}..{zero_x[^1]}");
Span<int> z_end = numbers[z..];
Span<int> z_zero = numbers[z..^0];
Console.WriteLine($"\\t{z_end[0]}..{z_end[^1]} is the same as {z_zero[0]}..
{z_zero[^1]}");

```

배열만 인덱스와 범위를 지원합니다. [문자열](#), [Span<T>](#) 또는 [ReadOnlySpan<T>](#)에서 인덱스 및 범위를 사용할 수도 있습니다.

## 암시적 범위 연산자 식 변환

범위 연산자 식 구문을 사용하면 컴파일러는 암시적으로 시작 및 끝 값을 [Index](#)로 변환하고 이 값에서 새 [Range](#) 인스턴스를 만듭니다. 다음 코드에서는 범위 연산자 식 구문에서의 암시적 변환 예제 및 해당 명시적 대안을 보여 줍니다.

C#

```

Range implicitRange = 3..^5;

Range explicitRange = new(
    start: new Index(value: 3, fromEnd: false),
    end: new Index(value: 5, fromEnd: true));

if (implicitRange.Equals(explicitRange))
{
    Console.WriteLine(
        $"The implicit range '{implicitRange}' equals the explicit range
'{explicitRange}'");
}
// Sample output:
//      The implicit range '3..^5' equals the explicit range '3..^5'

```

### ① 중요

[Int32](#)에서 [Index](#)로의 암시적 변환은 값이 음수일 경우

[ArgumentOutOfRangeException](#)을 throw합니다. 마찬가지로 [Index](#) 생성자는 [value](#) 매개 변수가 음수일 경우 [ArgumentOutOfRangeException](#)을 throw합니다.

## 인덱스 및 범위에 대한 형식 지원

인덱스 및 범위는 시퀀스에서 단일 요소 또는 요소의 범위에 액세스하기 위한 명확하고 간결한 구문을 제공합니다. 인덱스 식은 일반적으로 시퀀스의 요소 형식을 반환합니다. 범위 식은 일반적으로 소스 시퀀스와 동일한 시퀀스 형식을 반환합니다.

[Index](#) 또는 [Range](#) 매개 변수를 사용하여 [인덱서](#)를 제공하는 형식은 각각 인덱스 또는 범위를 명시적으로 지원합니다. 단일 [Range](#) 매개 변수를 사용하는 인덱서는 다양한 시퀀스 형식(예: [System.Span<T>](#))을 반환할 수 있습니다.

### ① 중요

범위 연산자를 사용하는 코드의 성능은 시퀀스 피연산자의 형식에 따라 다릅니다.

범위 연산자의 시간 복잡성은 시퀀스 형식에 따라 다릅니다. 예를 들어 시퀀스가 [string](#) 또는 배열인 경우 결과는 지정된 입력 섹션의 복사본이므로, 시간 복잡성은  $O(N)$ 입니다(여기서  $N$ 은 범위의 길이입니다.). 반면, 시퀀스가 [System.Span<T>](#) 또는 [System.Memory<T>](#)인 경우 결과는 동일한 백업 저장소를 참조합니다. 다시 말해서, 복사본이 없고 작업은  $O(1)$  이 됩니다.

이로 인해 시간 복잡성 외에도 추가 할당과 복사본이 발생하여 성능에 영향을 미칩니다. 성능에 중요한 코드에서는 시퀀스 형식으로 [Span<T>](#) 또는 [Memory<T>](#)를 사용하는 것이 좋습니다. 이에 대해 범위 연산자가 할당되지 않기 때문입니다.

이름이 [Length](#) 또는 [Count](#)이고 액세스 가능한 getter 및 반환 형식 [int](#)를 갖는 속성이 있는 경우 형식은 [countable](#)입니다. 인덱스 또는 범위를 명시적으로 지원하지 않는 [countable](#) 형식은 해당 형식에 대한 암시적 지원을 제공할 수 있습니다. 자세한 내용은 [기능 제한 참고의 암시적 인덱스 지원 및 암시적 범위 지원](#) 섹션을 참조하세요. 암시적 범위 지원을 사용하는 범위는 소스 시퀀스와 동일한 시퀀스 형식을 반환합니다.

예를 들어, .NET 형식 [String](#), [Span<T>](#) 및 [ReadOnlySpan<T>](#)은 인덱스와 범위를 모두 지원합니다. [List<T>](#)는 인덱스는 지원하고 범위는 지원하지 않습니다.

[Array](#)에는 좀 더 미묘한 동작이 더 있습니다. 1차원 배열은 인덱스와 범위를 모두 지원합니다. 2차원 배열은 인덱서 또는 범위를 지원하지 않습니다. 2차원 배열에 대한 인덱서에는 단일 매개 변수가 아닌 여러 개의 매개 변수가 있습니다. 배열의 배열이라고도 하는 가변 배열은 범위와 인덱서를 모두 지원합니다. 다음 예에서는 가변 배열의 사각형 하위 섹션을 반복하는 방법을 보여 줍니다. 첫 행과 마지막 3개 행을 제외하고, 선택된 각 행에서 첫 열과 마지막 2개 열을 제외하고 중앙의 섹션을 반복합니다.

C#

```
int[][] jagged =
[
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
    [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
    [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
    [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
    [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
    [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
```

```

[60,61,62,63,64,65,66,67,68,69],
[70,71,72,73,74,75,76,77,78,79],
[80,81,82,83,84,85,86,87,88,89],
[90,91,92,93,94,95,96,97,98,99],
];

var selectedRows = jagged[3..^3];

foreach (var row in selectedRows)
{
    var selectedColumns = row[2..^2];
    foreach (var cell in selectedColumns)
    {
        Console.Write($"{cell}, ");
    }
    Console.WriteLine();
}

```

모든 경우에 [Array](#)의 범위 연산자는 반환된 요소를 저장할 배열을 할당합니다.

## 인덱스 및 범위에 대한 시나리오

더 큰 시퀀스의 부분을 분석할 때 자주 범위와 인덱스를 사용하게 됩니다. 새 구문에서는 시퀀스의 어떤 부분이 관련되었는지 더 명확히 이해할 수 있습니다. 로컬 함수 `MovingAverage`는 [Range](#)를 인수로 사용합니다. 그러면 메서드가 최솟값, 최댓값, 평균을 계산할 때 이 범위만 열거합니다. 프로젝트에 다음 코드를 시도해 봅니다.

C#

```

int[] sequence = Sequence(1000);

for(int start = 0; start < sequence.Length; start += 100)
{
    Range r = start..(start+10);
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}: \tMin: {min}, \tMax: {max}, \tAverage: {average}");
}

for (int start = 0; start < sequence.Length; start += 100)
{
    Range r = ^start..^start;
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}: \tMin: {min}, \tMax: {max}, \tAverage: {average}");
}

(int min, int max, double average) MovingAverage(int[] subSequence, Range range) =>
(

```

```
        subSequence[range].Min(),
        subSequence[range].Max(),
        subSequence[range].Average()
    );

int[] Sequence(int count) => [..Enumerable.Range(0, count).Select(x => (int)
(Math.Sqrt(x) * 100))];
```

## 범위 인덱스 및 배열 참고 사항

배열에서 범위를 가져오는 경우 결과는 참조되지 않고 초기 배열에서 복사되는 배열입니다. 결과 배열의 값을 수정해도 초기 배열의 값은 변경되지 않습니다.

다음은 그 예입니다.

C#

```
var arrayOfFiveItems = new[] { 1, 2, 3, 4, 5 };

var firstThreeItems = arrayOfFiveItems[..3]; // contains 1,2,3
firstThreeItems[0] = 11; // now contains 11,2,3

Console.WriteLine(string.Join(", ", firstThreeItems));
Console.WriteLine(string.Join(", ", arrayOfFiveItems));

// output:
// 11,2,3
// 1,2,3,4,5
```

## 참고 항목

- 멤버 액세스 연산자 및 식

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback here.

#### 설명서 문제 열기

#### 제품 사용자 의견 제공

# 자습서: nullable 참조 형식 및 nullable이 아닌 참조 형식을 사용하여 디자인 의도를 보다 명확하게 표현

아티클 • 2023. 04. 08.

Nullable 값 형식이 값 형식을 보완하는 것과 동일한 방식으로 참조 형식을 보완하는 [nullable 참조 형식](#)이 도입되었습니다. 형식에 `?` 를 추가하여 변수를 nullable 참조 형식으로 선언합니다. 예를 들어 `string?` 는 nullable `string` 을 나타냅니다. 이러한 새 형식을 사용하여 디자인 의도를 보다 명확하게 표현할 수 있습니다. '항상 값이 있어야 하는' 변수도 있고, '값이 누락될 수 있는' 변수도 있습니다.

이 자습서에서는 다음과 같은 작업을 수행하는 방법을 알아봅니다.

- ✓ 디자인에 nullable 참조 형식 및 nullable이 아닌 참조 형식 통합
- ✓ 코드 전체에서 nullable 참조 형식 확인을 사용하도록 설정합니다.
- ✓ 컴파일러가 이러한 디자인 결정을 적용하는 코드를 작성합니다.
- ✓ 고유한 디자인에 nullable 참조 기능 사용

## 사전 요구 사항

C# 컴파일러를 포함하여 .NET을 실행하도록 머신을 설정해야 합니다. C# 컴파일러는 [Visual Studio 2022](#) 또는 [.NET SDK](#)에서 사용할 수 있습니다.

이 자습서에서는 여러분이 Visual Studio 또는 .NET CLI를 비롯한 C# 및 .NET에 익숙하다고 가정합니다.

## 디자인에 nullable 참조 형식 통합

이 자습서에서는 설문 조사 실행을 모델링하는 라이브러리를 빌드합니다. 코드는 nullable 참조 형식 및 nullable이 아닌 참조 형식을 둘 다 사용하여 실제 세계의 개념을 표현합니다. 설문 조사 질문은 null일 수 없습니다. 응답자는 질문에 응답하지 않을 수 있습니다. 이 경우 응답이 `null` 일 수 있습니다.

이 샘플에 대해 작성하는 코드는 해당 의도를 표현하고 컴파일러가 의도를 적용합니다.

## 애플리케이션을 만들고 nullable 참조 형식을 사용하도록 설정

Visual Studio 또는 `dotnet new console`을 사용하여 명령줄에서 새로운 콘솔 애플리케이션을 만듭니다. 애플리케이션 이름을 `NullableIntroduction`으로 지정합니다. 애플리케이션을 만든 후에는 전체 프로젝트가 활성화된 nullable 주석 컨텍스트에서 컴파일하도록 지정해야 합니다. `.csproj` 파일을 열고 `Nullable` 요소에 `PropertyGroup` 요소를 추가합니다. 해당 값을 `enable`로 설정합니다. C# 11 이전 프로젝트에서는 nullable 참조 형식 기능을 옵트인해야 합니다. 기능이 켜지고 나면 기존 참조 변수 선언이 nullable이 아닌 참조 형식으로 바뀌기 때문입니다. 이러한 의사 결정은 기존 코드에 적절한 null 확인이 없다는 문제를 찾는 데는 도움이 되지만, 원래 설계 의도를 정확하게 반영하지 않을 수 있습니다.

#### XML

```
<Nullable>enable</Nullable>
```

.NET 6 이전 버전에서는 새 프로젝트에 `Nullable` 요소가 포함되지 않습니다. .NET 6부터 새 프로젝트에는 프로젝트 파일에 `<Nullable>enable</Nullable>` 요소가 포함됩니다.

## 애플리케이션의 형식 디자인

이 설문 조사 애플리케이션에서는 다음과 같은 많은 클래스를 만들어야 합니다.

- 질문 목록을 모델링하는 클래스
- 설문 조사를 위해 연락한 사람 목록을 모델링하는 클래스
- 설문 조사를 수행한 사람의 응답을 모델링하는 클래스

이러한 형식은 nullable 참조 형식 및 nullable이 아닌 참조 형식을 둘 다 사용하여 필수 멤버가 선택적 멤버를 표현합니다. nullable 참조 형식은 해당 디자인 의도를 명확하게 전달합니다.

- 설문 조사의 일부인 질문은 null일 수 없습니다. 빈 질문을 하는 것은 타당하지 않습니다.
- 응답자는 null일 수 없습니다. 참여를 거부한 응답자를 포함하여 연락한 사람을 추적하는 것이 좋습니다.
- 질문에 대한 응답은 null일 수 있습니다. 응답자는 일부 또는 모든 질문에 대한 응답을 거부할 수 있습니다.

C#으로 프로그래밍한 경우 `null` 값을 허용하는 참조 형식에 익숙해서 null 허용이 아닌 인스턴스를 선언할 다른 기회를 놓칠 수도 있습니다.

- 질문 컬렉션은 nullable이 아니어야 합니다.
- 응답자 컬렉션은 nullable이 아니어야 합니다.

코드를 작성할 때 null 허용이 아닌 참조 형식을 참조의 기본값으로 사용하면 [NullReferenceException](#)을 발생시킬 수 있는 일반적인 실수를 방지할 수 있습니다. 이 자습서에서 배운 한 가지 교훈은 null 일 수 있는 변수와 그렇지 않은 변수를 결정하는 것입니다. 이러한 결정을 표현하는 구문은 언어에서 제공하지 않았지만 이제 제공합니다.

빌드할 앱은 다음 단계를 수행합니다.

- 설문 조사를 만들고 질문을 추가합니다.
- 설문 조사 응답자의 의사 임의 세트를 만듭니다.
- 완료된 설문 조사 크기가 목표 수치에 도달할 때까지 응답자에게 연락합니다.
- 설문 조사 응답에 대한 중요한 통계를 작성합니다.

## null 허용 참조 형식과 null을 허용하지 않는 참조 형식을 사용하여 설문 조사 작성

작성하는 첫 번째 코드에서 설문 조사를 만듭니다. 설문 조사 질문과 설문 조사 실행을 모델링하는 클래스를 작성합니다. 설문 조사에는 다음 응답 형식으로 구분되는 세 가지 질문 유형이 있습니다. 예/아니요 응답, 숫자 응답, 텍스트 응답. `public SurveyQuestion` 클래스를 만듭니다.

```
C#  
  
namespace NullableIntroduction  
{  
    public class SurveyQuestion  
    {  
    }  
}
```

컴파일러가 모든 참조 형식 변수 선언을 활성화된 nullable 주석 컨텍스트에 있는 코드의 nullable이 아닌 참조 형식으로 해석합니다. 다음 코드에 표시된 대로 질문 텍스트 및 질문 유형의 속성을 추가하여 첫 번째 경고를 표시할 수 있습니다.

```
C#  
  
namespace NullableIntroduction  
{  
    public enum QuestionType  
    {  
        YesNo,  
        Number,  
        Text  
    }  
  
    public class SurveyQuestion
```

```
{  
    public string QuestionText { get; }  
    public QuestionType TypeOfQuestion { get; }  
}  
}
```

`QuestionText` 를 초기화하지 않았기 때문에 컴파일러에서 nullable이 아닌 속성이 초기화 되지 않았다는 경고를 실행합니다. 디자인에 따라 질문 텍스트는 null이 아니어야 하므로 초기화할 생성자와 `QuestionType` 값도 추가합니다. 완성된 클래스 정의는 다음 코드와 같습니다.

C#

```
namespace NullableIntroduction;  
  
public enum QuestionType  
{  
    YesNo,  
    Number,  
    Text  
}  
  
public class SurveyQuestion  
{  
    public string QuestionText { get; }  
    public QuestionType TypeOfQuestion { get; }  
  
    public SurveyQuestion(QuestionType typeOfQuestion, string text) =>  
        (TypeOfQuestion, QuestionText) = (typeOfQuestion, text);  
}
```

생성자를 추가하면 경고가 제거됩니다. 생성자 인수도 nullable이 아닌 참조 형식이므로 컴파일러에서 경고를 실행하지 않습니다.

`SurveyRun` 이라는 `public` 클래스를 만듭니다. 이 클래스에는 다음 코드에 표시된 것처럼 설문 조사에 질문을 추가하는 메서드 및 `SurveyQuestion` 개체 목록이 포함되어 있습니다.

C#

```
using System.Collections.Generic;  
  
namespace NullableIntroduction  
{  
    public class SurveyRun  
    {  
        private List<SurveyQuestion> surveyQuestions = new  
List<SurveyQuestion>();  
  
        public void AddQuestion(QuestionType type, string question) =>
```

```
        AddQuestion(new SurveyQuestion(type, question));
    public void AddQuestion(SurveyQuestion surveyQuestion) =>
    surveyQuestions.Add(surveyQuestion);
}
}
```

이전과 마찬가지로, 목록 개체를 null이 아닌 값으로 초기화해야 합니다. 초기화하지 않으면 컴파일러에서 경고를 실행합니다. `AddQuestion`의 두 번째 오버로드에는 null 확인이 없습니다. 왜냐하면 해당 변수를 nullable이 아닌 것으로 선언했기 때문입니다. 해당 값은 null일 수 있습니다.

편집기에서 `Program.cs`로 전환하고 `Main`의 내용을 다음 코드 줄로 바꿉니다.

C#

```
var surveyRun = new SurveyRun();
surveyRun.AddQuestion(QuestionType.YesNo, "Has your code ever thrown a
NullReferenceException?");
surveyRun.AddQuestion(new SurveyQuestion(QuestionType.Number, "How many
times (to the nearest 100) has that happened?"));
surveyRun.AddQuestion(QuestionType.Text, "What is your favorite color?");
```

프로젝트 전체가 활성화된 nullable 주석 컨텍스트이므로 nullable이 아닌 참조 형식을 기대하고 메서드로 null을 전달하면 경고가 발생합니다. 다음 줄을 `Main`에 추가하여 시도해 보세요.

C#

```
surveyRun.AddQuestion(QuestionType.Text, default);
```

## 응답자를 만들고 설문 조사에 대한 응답 받기

다음으로, 설문 조사에 대한 응답을 생성하는 코드를 작성합니다. 이 프로세스에는 몇 개의 작은 작업이 포함됩니다.

1. 응답자 개체를 생성하는 메서드를 빌드합니다. 이러한 개체는 설문 조사를 작성하도록 요청된 사람을 나타냅니다.
2. 응답자에게 질문하고 응답을 수집하거나 응답자가 응답하지 않았음을 확인하는 과정을 시뮬레이션하는 논리를 빌드합니다.
3. 충분한 응답자가 설문 조사에 응답할 때까지 반복합니다.

설문 조사 응답을 나타낼 클래스가 필요하므로 지금 추가합니다. nullable 지원을 사용하도록 설정합니다. 다음 코드에 표시된 대로 `Id` 속성 및 이 속성을 초기화하는 생성자를 추가합니다.

C#

```
namespace NullableIntroduction
{
    public class SurveyResponse
    {
        public int Id { get; }

        public SurveyResponse(int id) => Id = id;
    }
}
```

다음으로, `static` 메서드를 추가해서 임의 ID를 생성하여 새 참가자를 만듭니다.

C#

```
private static readonly Random randomGenerator = new Random();
public static SurveyResponse GetRandomId() => new
SurveyResponse(randomGenerator.Next());
```

이 클래스의 주요 책임은 설문 조사의 질문에 대한 참가자의 응답을 생성하는 것입니다. 이 책임에는 몇 가지 단계가 있습니다.

1. 설문 조사에 참여하도록 요청합니다. 개인이 동의하지 않을 경우 `missing`(또는 `null`) 응답을 반환합니다.
2. 각 질문을 하고 응답을 기록합니다. 각 응답도 `missing`(또는 `null`)일 수 있습니다.

`SurveyResponse` 클래스에 다음 코드를 추가합니다.

C#

```
private Dictionary<int, string>? surveyResponses;
public bool AnswerSurvey(IEnumerable<SurveyQuestion> questions)
{
    if (ConsentToSurvey())
    {
        surveyResponses = new Dictionary<int, string>();
        int index = 0;
        foreach (var question in questions)
        {
            var answer = GenerateAnswer(question);
            if (answer != null)
            {
                surveyResponses.Add(index, answer);
            }
            index++;
        }
    }
    return surveyResponses != null;
```

```

    }

    private bool ConsentToSurvey() => randomGenerator.Next(0, 2) == 1;

    private string? GenerateAnswer(SurveyQuestion question)
    {
        switch (question.TypeOfQuestion)
        {
            case QuestionType.YesNo:
                int n = randomGenerator.Next(-1, 2);
                return (n == -1) ? default : (n == 0) ? "No" : "Yes";
            case QuestionType.Number:
                n = randomGenerator.Next(-30, 101);
                return (n < 0) ? default : n.ToString();
            case QuestionType.Text:
            default:
                switch (randomGenerator.Next(0, 5))
                {
                    case 0:
                        return default;
                    case 1:
                        return "Red";
                    case 2:
                        return "Green";
                    case 3:
                        return "Blue";
                }
                return "Red. No, Green. Wait.. Blue... AAARGGGGGHHH!";
        }
    }
}

```

설문 조사 응답의 스토리지는 `Dictionary<int, string>?`로, `null`일 수 있음을 나타냅니다. 새 언어 기능을 사용하여 컴파일러 및 나중에 코드를 읽는 모든 사람에게 디자인 의도를 선언하고 있습니다. 먼저 `null` 값을 확인하지 않고 `surveyResponses`를 역참조하는 경우 컴파일러 경고가 표시됩니다. 위에서 `surveyResponses` 변수가 `null`이 아닌 값으로 설정되었음을 컴파일러가 판별할 수 있으므로 `AnswerSurvey` 메서드에 경고가 표시되지 않습니다.

누락된 응답을 확인하기 위해 `null`을 사용하는 것에서 nullable 참조 형식을 사용할 때 유의해야 할 중요한 점을 확인할 수 있습니다. 즉, 프로그램에서 `null` 값을 모두 제거하는 것이 목표가 되어서는 안 됩니다. 내가 작성하는 코드가 내 설계 의도를 그대로 표현하고 있는지 확인하는 것이 목표가 되어야 합니다. 누락된 값은 코드에서 반드시 표현해야 하는 개념입니다. `null` 값은 누락된 값을 분명하게 표현할 수 있는 방법입니다. `null` 값을 모두 제거하는 것을 목표로 하면 `null`을 사용하지 않고 누락된 값을 표현할 다른 방법을 정의해야 할 뿐입니다.

다음으로, `SurveyRun` 클래스에 `PerformSurvey` 메서드를 작성해야 합니다. `SurveyRun` 클래스에 다음 코드를 추가합니다.

C#

```
private List<SurveyResponse>? respondents;
public void PerformSurvey(int numberofRespondents)
{
    int respondentsConsenting = 0;
    respondents = new List<SurveyResponse>();
    while (respondentsConsenting < numberofRespondents)
    {
        var respondent = SurveyResponse.GetRandomId();
        if (respondent.AnswerSurvey(surveyQuestions))
            respondentsConsenting++;
        respondents.Add(respondent);
    }
}
```

여기서 다시, nullable `List<SurveyResponse>?` 선택은 응답이 null일 수 있음을 나타냅니다. 이는 설문 조사가 아직 어떠한 응답자에게도 제공되지 않았음을 나타냅니다. 충분한 응답자가 동의할 때까지 응답자가 추가됩니다.

설문 조사를 실행하는 마지막 단계는 `Main` 메서드의 끝에 설문 조사를 수행할 호출을 추가하는 것입니다.

C#

```
surveyRun.PerformSurvey(50);
```

## 설문 조사 응답 조사

마지막 단계는 설문 조사 결과를 표시하는 것입니다. 작성한 여러 클래스에 코드를 추가합니다. 이 코드는 nullable 참조 형식과 nullable이 아닌 참조 형식 구분의 가치를 보여 줍니다. 먼저 다음 두 개의 식 본문 멤버를 `SurveyResponse` 클래스에 추가합니다.

C#

```
public bool AnsweredSurvey => surveyResponses != null;
public string Answer(int index) => surveyResponses?.GetValueOrDefault(index)
?? "No answer";
```

`surveyResponses`는 null이 가능한 참조 형식이므로 역참조하기 전에 null 확인이 필요합니다. `Answer` 메서드는 null 허용이 아닌 문자열을 반환하므로 null 병합 연산자를 사용하여 누락된 답변의 사례를 포함해야 합니다.

다음 세 개의 식 본문 멤버를 `SurveyRun` 클래스에 추가합니다.

C#

```
public IEnumerable<SurveyResponse> AllParticipants => (respondents ??  
Enumerable.Empty<SurveyResponse>());  
public ICollection<SurveyQuestion> Questions => surveyQuestions;  
public SurveyQuestion GetQuestion(int index) => surveyQuestions[index];
```

`respondents` 변수는 null일 수 있지만 반환 값은 null일 수 없음을 `AllParticipants` 멤버가 고려해야 합니다. ?? 및 뒤에 오는 빈 시퀀스를 제거하여 해당 식을 변경하면 컴파일러에서 메서드가 `null`을 반환할 수 있고 해당 반환 시그니처가 nullable이 아닌 형식을 반환한다고 경고합니다.

마지막으로, `Main` 메서드의 맨 아래에 다음 루프를 추가합니다.

C#

```
foreach (var participant in surveyRun.AllParticipants)  
{  
    Console.WriteLine($"Participant: {participant.Id}");  
    if (participant.AnsweredSurvey)  
    {  
        for (int i = 0; i < surveyRun.Questions.Count; i++)  
        {  
            var answer = participant.Answer(i);  
            Console.WriteLine($"{surveyRun.GetQuestion(i).QuestionText} :  
{answer}");  
        }  
    }  
    else  
    {  
        Console.WriteLine("\tNo responses");  
    }  
}
```

모두 nullable이 아닌 참조 형식을 반환하도록 기본 인터페이스를 디자인했기 때문에 이 코드에는 `null` 확인이 필요하지 않습니다.

## 코드 가져오기

[csharp/NullableIntroduction](#) 폴더의 [samples](#) 리포지토리에서 완료된 자습서의 코드를 가져올 수 있습니다.

nullable 참조 형식과 nullable이 아닌 참조 형식 간에 형식 선언을 변경하여 실험합니다. 실수로 `null`을 역참조하지 않도록 어떻게 다양한 경고를 생성하는지 확인합니다.

# 다음 단계

Entity Framework를 사용할 때 nullable 참조 형식을 사용하는 방법을 알아봅니다.

## Entity Framework Core 기본 사항: nullable 참조 형식 사용

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# 문자열 보간을 사용하여 형식이 지정된 문자열 생성

아티클 • 2023. 04. 08.

이 자습서에서는 C# [문자열 보간](#)을 사용하여 단일 결과 문자열에 값을 삽입하는 방법을 설명합니다. C# 코드를 작성하고 컴파일 및 실행 결과를 확인합니다. 이 자습서는 문자열에 값을 삽입하고, 이러한 값을 다양한 방식으로 형식화하는 방법을 보여 주는 일련의 단원으로 구성됩니다.

이 자습서에서는 개발에 사용할 수 있는 머신이 있다고 예상합니다. .NET 자습서 [Hello World 10분 완성](#)에는 Windows, Linux 또는 macOS의 로컬 개발 환경 설정에 대한 지침이 포함되어 있습니다. 브라우저에서 이 자습서의 [대화형 버전](#)을 완료할 수도 있습니다.

## 보간된 문자열 만들기

*interpolated*라는 디렉터리를 만듭니다. 현재 디렉터리로 만들고 콘솔 창에서 다음 명령을 실행합니다.

.NET CLI

```
dotnet new console
```

이 명령은 현재 디렉터리에 새 .NET Core 콘솔 애플리케이션을 만듭니다.

원하는 편집기에서 *Program.cs*를 열고 `Console.WriteLine("Hello World!");` 줄을 다음 코드로 바꿉니다. 여기서 `<name>`을 사용자 이름으로 바꿉니다.

C#

```
var name = "<name>";
Console.WriteLine($"Hello, {name}. It's a pleasure to meet you!");
```

콘솔 창에 `dotnet run`을 입력하여 이 코드를 사용해 봅니다. 프로그램을 실행하면 인사말에 사용자 이름이 포함된 단일 문자열이 표시됩니다. `WriteLine` 메서드 호출에 포함된 문자열은 [보간된 문자열](#) 식입니다. 이는 포함 코드가 들어있는 문자열에서 단일 문자열(결과 문자열이라고 함)을 생성할 수 있게 해주는 일종의 템플릿입니다. 보간된 문자열은 문자열에 값을 삽입하거나 문자열을 연결(함께 조인)하는 데 특히 유용합니다.

다음 간단한 예제에서는 모든 보간된 문자열이 포함해야 하는 두 가지 요소를 보여 줍니다.

- \$ 문자로 시작한 후에는 따옴표 문자가 다음에 나오는 문자열 리터럴. \$ 기호와 따옴표 문자 사이에는 공백이 없어야 합니다. (포함할 경우 어떤 일이 발생하는지 확인하려면 문자 뒤에 공백을 삽입하고 파일을 저장한 다음 \$ 콘솔 창에 을 입력하여 dotnet run 프로그램을 다시 실행합니다. C# 컴파일러에 "오류 CS1056: 예기치 않은 문자 '\$'"라는 오류 메시지가 표시됩니다.
- 하나 이상의 '보간 식'. 보간 식은 열기 및 닫기 중괄호({ 및 })로 표시됩니다. 중괄호 안에 값을 반환(null 포함)하는 C# 식을 배치할 수 있습니다.

몇 가지 다른 데이터 형식을 포함하는 문자열 보간 예제를 더 살펴보겠습니다.

## 다양한 데이터 형식 포함

이전 섹션에서는 한 문자열을 다른 문자열 내에 삽입하는 데 문자열 보간을 사용했습니다. 보간 식의 결과는 모든 데이터 형식일 수 있습니다. 보간된 문자열에 다양한 데이터 형식의 값을 포함시켜 보겠습니다.

다음 예제에서는 먼저 [메서드](#)의 동작을 재정의하는 속성과 [ToString](#) 메서드가 있는 [클래스](#) 데이터 형식 Vegetable Name 을 [Object.ToString\(\)](#) 정의합니다. public 액세스 한정자를 지정하면 해당 메서드를 모든 클라이언트 코드에 사용하여 Vegetable 인스턴스의 문자열 표현을 가져올 수 있습니다. 예제에서 메서드는 Vegetable.ToString 생성자에서 Vegetable 초기화된 속성의 Name 값을 반환합니다.

C#

```
public Vegetable(string name) => Name = name;
```

그런 다음, [new 연산자](#)를 사용하고 생성자 Vegetable의 이름을 제공하여 item이라는 Vegetable 클래스의 인스턴스를 만듭니다.

C#

```
var item = new Vegetable("eggplant");
```

마지막으로 값, 값 및 [열거형](#) 값도 포함하는 DateTime 보간된 문자열에 Decimal 변수를 Unit 포함합니다 item. 편집기에서 모든 C# 코드를 다음 코드로 바꾼 후 dotnet run 명령을 사용하여 실행합니다.

C#

```
using System;
```

```

public class Vegetable
{
    public Vegetable(string name) => Name = name;

    public string Name { get; }

    public override string ToString() => Name;
}

public class Program
{
    public enum Unit { item, kilogram, gram, dozen };

    public static void Main()
    {
        var item = new Vegetable("eggplant");
        var date = DateTime.Now;
        var price = 1.99m;
        var unit = Unit.item;
        Console.WriteLine($"On {date}, the price of {item} was {price} per
{unit}.");
    }
}

```

보간된 문자열의 보간 식 `item`은 결과 문자열에 "eggplant"라는 텍스트로 확인됩니다. 이것은 식 결과의 형식이 문자열이 아닌 경우 다음과 같은 방식으로 결과가 문자열로 확인되기 때문입니다.

- 보간 식이 `null`로 평가되면 빈 문자열("") 또는 `String.Empty`이 사용됩니다.
- 보간 식이 `null`로 계산되지 않고 결과 형식의 `ToString` 메서드가 호출됩니다. 이것은 `Vegetable.ToString` 메서드의 구현을 업데이트하여 테스트할 수 있습니다. 모든 형식에는 `ToString` 메서드가 구현되어 있으므로 이 메서드를 구현할 필요가 없을 수도 있습니다. 이것을 테스트하려면 예제에서 `Vegetable.ToString` 메서드 정의를 주석으로 처리합니다. (이렇게 하려면 그 앞에 주석 기호 즉, `//`를 추가합니다.) 출력에서 "eggplant" 문자열은 정규화된 형식 이름(이 예제의 경우 "Vegetable")으로 바뀌며 이것이 `Object.ToString()` 메서드의 기본 동작입니다. 열거형 값에 대한 `ToString` 메서드의 기본 동작은 값의 문자열 표현을 반환하는 것입니다.

이 예제의 출력에서 날짜는 매우 정확하며(`eggplant` 가격은 초마다 변경되지 않음), 가격 값은 통화 단위를 나타내지 않습니다. 다음 섹션에서는 식 결과에 대한 문자열 표현의 형식을 제어하여 해당 문제를 해결하는 방법을 알아봅니다.

## 보간 식의 서식 제어

이전 섹션에서는 형식이 잘못 지정된 두 개의 문자열을 결과 문자열에 삽입했습니다. 하 나는 날짜만 적절한 날짜 및 시간 값이었습니다. 두 번째는 통화 단위를 나타내지 않는 가 격이었습니다. 두 가지 문제는 쉽게 해결할 수 있습니다. 문자열 보간을 통해 특정 유형의 형식을 제어하는 형식 문자열을 지정할 수 있습니다. 다음 줄에 표시된 것처럼 이전 예제의 `Console.WriteLine`에 대한 호출을 수정하여 날짜 및 가격 식의 형식 문자열을 포함시 킵니다.

C#

```
Console.WriteLine($"On {date:d}, the price of {item} was {price:C2} per
{unit}.");
```

콜론(":")과 형식 문자열을 사용하여 보간 식에 따라 형식 문자열을 지정합니다. "d"는 간 단한 날짜 형식을 나타내는 표준 날짜 및 시간 형식 문자열입니다. "C2"는 소수점 뒤 두 자릿수를 포함하는 통화 값으로 숫자를 나타내는 표준 숫자 형식 문자열입니다.

.NET 라이브러리의 많은 형식은 미리 정의된 형식 문자열 집합을 지원합니다. 여기에는 모든 숫자 형식과 날짜 및 시간 형식이 포함됩니다. 형식 문자열을 지원하는 형식의 전체 목록을 보려면 [.NET의 서식 지정 형식](#) 문서의 [형식 문자열 및 .NET 클래스 라이브러리 형식](#)을 참조하세요.

텍스트 편집기에서 형식 문자열을 수정하고, 변경할 때마다 프로그램을 다시 실행하여 날짜 및 시간의 서식과 숫자 값에 미치는 영향을 확인해 보세요. `{date:d}`의 "d"를 "t"(짧은 시간 형식 표시), "y"(연도 및 월 표시) 및 "yyyy"(연도를 4자리 숫자로 표시)로 변경합니다. `{price:C2}`의 "C2"를 "e"(지수 표기) 및 "F3"(소수점 뒤 세 자릿수의 숫자 값)으로 변경합니다.

형식을 제어하는 것 외에도, 결과 문자열에 포함된 형식이 지정된 문자열의 필드 너비와 맞춤을 제어할 수 있습니다. 다음 섹션에서 이 작업을 수행하는 방법을 알아봅니다.

## 필드 너비와 보간 식의 맞춤을 제어합니다.

일반적으로 보간 식의 결과가 문자열로 형식이 지정되면 해당 문자열은 결과 문자열에 선행 또는 후행 공백 없이 포함됩니다. 특히 데이터 집합을 가지고 작업하는 경우 필드 너비와 텍스트 맞춤을 제어할 수 있으면 보다 읽기 쉬운 출력을 생성하는 데 도움이 됩니다. 이것을 확인하려면 텍스트 편집기에서 모든 코드를 다음 코드로 바꾼 후 `dotnet run`을 입력하여 프로그램을 실행합니다.

C#

```
using System;
using System.Collections.Generic;
```

```

public class Example
{
    public static void Main()
    {
        var titles = new Dictionary<string, string>()
        {
            ["Doyle, Arthur Conan"] = "Hound of the Baskervilles, The",
            ["London, Jack"] = "Call of the Wild, The",
            ["Shakespeare, William"] = "Tempest, The"
        };

        Console.WriteLine("Author and Title List");
        Console.WriteLine();
        Console.WriteLine($"|{\"Author\", -25}|{\"Title\", 30}|");
        foreach (var title in titles)
            Console.WriteLine($"|{title.Key, -25}|{title.Value, 30}|");
    }
}

```

작성자의 이름은 왼쪽 맞춤되며 이들이 작성한 제목은 오른쪽 맞춤됩니다. 보간 식 뒤에 쉼표(",")를 추가하고 '최소' 필드 너비를 지정하여 맞춤을 지정합니다. 지정한 값이 양수이면 필드는 오른쪽 맞춤입니다. 음수이면 필드는 왼쪽 맞춤입니다.

다음 코드처럼 `{"Author", -25}` 및 `{title.Key, -25}` 코드에서 음수 기호를 제거하고 예제를 다시 실행해 보세요.

C#

```

Console.WriteLine($"|{\"Author\", 25}|{\"Title\", 30}|");
foreach (var title in titles)
    Console.WriteLine($"|{title.Key, 25}|{title.Value, 30}|");

```

이때 작성자 정보는 오른쪽 맞춤입니다.

하나의 보간 식에 대해 맞춤 지정자 및 형식 문자열을 결합할 수 있습니다. 이렇게 하려면 먼저 맞춤을 지정하고 콜론과 형식 문자열을 지정합니다. `Main` 메서드 내에 있는 모든 코드를 다음 코드로 바꾸면, 필드 너비가 정의된 세 가지 형식 지정된 문자열이 표시됩니다. 그런 다음 `dotnet run` 명령을 입력하여 프로그램을 실행합니다.

C#

```

Console.WriteLine($"[{DateTime.Now, -20:d}] Hour [{DateTime.Now, -10:HH}]
[{1063.342, 15:N2}] feet");

```

다음과 같은 출력을 얻을 수 있습니다.

콘솔

[04/14/2018

] Hour [16

] [

1,063.34] feet

문자열 보간 자습서를 완료했습니다.

자세한 내용은 [문자열 보간 항목](#) 및 [C#에서 문자열 보간](#) 자습서를 참조하세요.

# C#의 문자열 보간

아티클 • 2023. 08. 31.

이 자습서에서는 [문자열 보간](#)을 사용하여 결과 문자열에서 식 결과의 서식을 지정하고 포함하는 방법을 보여줍니다. 예제에서는 사용자가 기본 C# 개념 및 .NET 형식 서식 지정에 익숙하다고 가정합니다. 문자열 보간 또는 .NET 형식 서식 지정을 처음 접하는 경우 [대화형 문자열 보간 자습서](#)를 먼저 체크 아웃합니다. .NET의 형식 서식 지정에 대한 자세한 내용은 [.NET의 서식 지정 형식을 참조하세요](#).

## 소개

문자열 리터럴을 보간된 문자열로 식별하려면 `$` 기호를 사용하여 추가합니다. 보간된 문자열에서 값을 반환하는 유효한 C# 식을 포함할 수 있습니다. 다음 예제에서는 식이 계산되는 즉시 결과가 문자열로 변환되고 결과 문자열에 포함됩니다.

C#

```
double a = 3;
double b = 4;
Console.WriteLine($"Area of the right triangle with legs of {a} and {b} is
{0.5 * a * b}");
Console.WriteLine($"Length of the hypotenuse of the right triangle with legs
of {a} and {b} is {CalculateHypotenuse(a, b)}");
double CalculateHypotenuse(double leg1, double leg2) => Math.Sqrt(leg1 *
leg1 + leg2 * leg2);
// Output:
// Area of the right triangle with legs of 3 and 4 is 6
// Length of the hypotenuse of the right triangle with legs of 3 and 4 is 5
```

이 예제에서 볼 수 있듯이 중괄호를 포함하여 보간된 문자열에 식을 포함합니다.

C#

```
{<interpolationExpression>}
```

보간된 문자열은 [문자열 복합 서식 지정](#) 기능의 모든 기능을 지원합니다. 따라서 [String.Format](#) 메서드를 사용할 때 보다 읽기 쉬운 대안이 됩니다.

## 보간 식에 대한 서식 문자열을 지정하는 방법

식 결과의 형식에서 지원되는 형식 문자열을 지정하려면 콜론(:) 및 형식 문자열을 사용하여 보간 식을 따릅니다.

C#

```
{<interpolationExpression>:<formatString>}
```

다음 예제에서는 날짜 및 시간 또는 숫자 결과를 생성하는 식의 표준 및 사용자 지정 서식 지정 문자열을 지정하는 방법을 보여줍니다.

C#

```
var date = new DateTime(1731, 11, 25);
Console.WriteLine($"On {date:dddd, MMMM dd, yyyy} L. Euler introduced the
letter e to denote {Math.E:F5}.");
// Output:
// On Sunday, November 25, 1731 L. Euler introduced the letter e to denote
2.71828.
```

자세한 내용은 복합 서식 문서의 [문자열 구성 요소 서식 섹션](#)을 참조하세요.

## 필드 너비와 서식이 지정된 보간 식의 맞춤을 제어하는 방법

최소 필드 너비와 서식이 지정된 식 결과의 맞춤을 지정하려면 쉼표(",") 및 상수 식을 사용하여 보간 식을 따릅니다.

C#

```
{<interpolationExpression>,<alignment>}
```

맞춤 값이 양수이면 서식이 지정된 식 결과는 오른쪽 맞춤입니다. 값이 음수이면 왼쪽 맞춤입니다.

맞춤 및 서식 문자열을 모두 지정해야 할 경우 맞춤 구성 요소를 시작합니다.

C#

```
{<interpolationExpression>,<alignment>:<formatString>}
```

다음 예제에서는 맞춤을 지정하고 파이프 문자("|")를 사용하여 텍스트 필드를 구분하는 방법을 보여줍니다.

C#

```
const int NameAlignment = -9;
const int ValueAlignment = 7;
```

```

double a = 3;
double b = 4;
Console.WriteLine($"Three classical Pythagorean means of {a} and {b}:");
Console.WriteLine($"|{"Arithmetic",NameAlignment}|{0.5 * (a +
b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Geometric",NameAlignment}|{Math.Sqrt(a *
b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Harmonic",NameAlignment}|{2 / (1 / a + 1 /
b),ValueAlignment:F3}|");
// Output:
// Three classical Pythagorean means of 3 and 4:
// |Arithmetc| 3.500|
// |Geometric| 3.464|
// |Harmonic | 3.429|

```

출력 표시 예제에서 볼 수 있듯이 서식이 지정된 식 결과의 길이가 지정된 필드 너비를 초과하는 경우 맞춤 값은 무시됩니다.

자세한 내용은 복합 서식 문서의 맞춤 구성 요소 [섹션을](#) 참조하세요.

## 보간된 문자열에서 이스케이프 시퀀스를 사용하는 방법

보간된 문자열에서는 일반 문자열 리터럴을 사용할 수 있는 모든 이스케이프 시퀀스를 지원합니다. 자세한 내용은 [문자열 이스케이프 시퀀스](#)를 참조하세요.

이스케이프 시퀀스를 문자 그대로 해석하려면 [약어](#) 리터럴 문자열을 사용합니다. 보간된 축자 문자열은 둘 다 `$` 와 `@` 문자로 시작합니다. 어떤 순서로든 사용할 `$ @` 수 있습니다 `@$"..."`. 둘 다 `$@"..."` 유효한 보간된 축자 문자열입니다.

중괄호("{" 또는 "}")를 포함하려면 결과 문자열에서 2개의 중괄호("{{" 또는 "}}")를 사용합니다. 자세한 내용은 복합 서식 문서의 이스케이프 중괄호 [섹션을](#) 참조하세요.

다음 예제에서는 결과 문자열에 중괄호를 포함하고 약어 보간된 문자열을 만드는 방법을 보여줍니다.

C#

```

var xs = new int[] { 1, 2, 7, 9 };
var ys = new int[] { 7, 9, 12 };
Console.WriteLine($"Find the intersection of the {{{{string.Join(", ",xs)}}}}
and {{{string.Join(", ",ys)}}} sets.");
// Output:
// Find the intersection of the {1, 2, 7, 9} and {7, 9, 12} sets.

var userName = "Jane";
var stringWithEscapes = $"C:\\\\Users\\\\{userName}\\\\Documents";

```

```
var verbatimInterpolated = $@"C:\Users\{userName}\Documents";
Console.WriteLine(stringWithEscapes);
Console.WriteLine(verbatimInterpolated);
// Output:
// C:\Users\Jane\Documents
// C:\Users\Jane\Documents
```

C# 11부터 보간된 원시 문자열 리터럴을 사용할 수 있습니다.

## 보간 식에서 3개로 구성된 ?: 조건부 연산자를 사용하는 방법

보간 식에서 콜론(":")에 특별한 의미가 있으므로 식에서 조건부 연산자를 사용하기 위해 다음 예제에서 볼 수 있듯이 해당 식을 괄호로 묶습니다.

C#

```
var rand = new Random();
for (int i = 0; i < 7; i++)
{
    Console.WriteLine($"Coin flip: { (rand.NextDouble() < 0.5 ? "heads" :
    "tails") }");
```

## 문자열 보간을 사용하여 문화권별 결과 문자열을 만드는 방법

기본적으로는 보간된 문자열은 모든 서식 지정 작업에 대해 `CultureInfo.CurrentCulture` 속성에서 정의한 현재 문화권을 사용합니다.

.NET 6부터 다음 예제와 같이 보간된 문자열을 문화권별 결과 문자열로 확인하는 데 이 메서드를 사용할 `String.Create(IFormatProvider, DefaultInterpolatedStringHandler)` 수 있습니다.

C#

```
var cultures = new System.Globalization.CultureInfo[]
{
    System.Globalization.CultureInfo.GetCultureInfo("en-US"),
    System.Globalization.CultureInfo.GetCultureInfo("en-GB"),
    System.Globalization.CultureInfo.GetCultureInfo("nl-NL"),
    System.Globalization.CultureInfo.InvariantCulture
};
var date = DateTime.Now;
var number = 31_415_926.536;
```

```

foreach (var culture in cultures)
{
    var cultureSpecificMessage = string.Create(culture, $"{date,23}
{number,20:N3}");
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
}
// Output is similar to:
// en-US      8/27/2023 12:35:31 PM      31,415,926.536
// en-GB      27/08/2023 12:35:31      31,415,926.536
// nl-NL      27-08-2023 12:35:31      31.415.926,536
//          08/27/2023 12:35:31      31,415,926.536

```

이전 버전의 .NET에서는 보간된 문자열을 인스턴스로 [System.FormattableString](#) 암시적으로 변환하고 해당 메서드를 [ToString\(IFormatProvider\)](#) 호출하여 문화권별 결과 문자열을 만듭니다. 다음 예제에서는 해당 작업을 수행하는 방법을 보여줍니다.

C#

```

var cultures = new System.Globalization.CultureInfo[]
{
    System.Globalization.CultureInfo.GetCultureInfo("en-US"),
    System.Globalization.CultureInfo.GetCultureInfo("en-GB"),
    System.Globalization.CultureInfo.GetCultureInfo("nl-NL"),
    System.Globalization.CultureInfo.InvariantCulture
};
var date = DateTime.Now;
var number = 31_415_926.536;
FormattableString message = $"{date,23}{number,20:N3}";
foreach (var culture in cultures)
{
    var cultureSpecificMessage = message.ToString(culture);
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
}
// Output is similar to:
// en-US      8/27/2023 12:35:31 PM      31,415,926.536
// en-GB      27/08/2023 12:35:31      31,415,926.536
// nl-NL      27-08-2023 12:35:31      31.415.926,536
//          08/27/2023 12:35:31      31,415,926.536

```

이 예제에서 볼 수 있듯이 하나의 [FormattableString](#) 인스턴스를 사용하여 다양한 문화권에 여러 결과 문자열을 생성할 수 있습니다.

## 고정 문화권을 사용하여 결과 문자열을 만드는 방법

.NET 6부터 다음 예제와 같이 보간된 문자열을 결과 문자열 [InvariantCulture](#)로 확인하려면 이 메서드를 사용합니다 [String.Create\(IFormatProvider,](#)

`DefaultInterpolatedStringHandler).`

C#

```
string message = string.Create(CultureInfo.InvariantCulture, $"Date and time  
in invariant culture: {DateTime.Now}");  
Console.WriteLine(message);  
// Output is similar to:  
// Date and time in invariant culture: 05/17/2018 15:46:24
```

이전 버전의 .NET에서는 메서드와 함께 다음 예제와 `FormattableString.ToString(IFormatProvider)` 같이 정적 `FormattableString.Invariant` 메서드를 사용할 수 있습니다.

C#

```
string message = FormattableString.Invariant($"Date and time in invariant  
culture: {DateTime.Now}");  
Console.WriteLine(message);  
// Output is similar to:  
// Date and time in invariant culture: 05/17/2018 15:46:24
```

## 결론

이 자습서에서는 문자열 보간 사용법의 일반적인 시나리오를 설명합니다. 문자열 보간에 대한 자세한 내용은 문자열 보간을 참조 [하세요](#). .NET의 형식 서식 지정에 대한 자세한 내용은 .NET 및 복합 서식 문서의 서식 지정 형식을 참조하세요.

## 참고 항목

- [String.Format](#)
- [System.FormattableString](#)
- [System.IFormattable](#)
- [문자열](#)

# 콘솔 앱

아티클 • 2023. 03. 14.

이 자습서에서는 .NET 및 C# 언어의 다양한 기능에 대해 설명합니다. 다음 내용을 배웁니다.

- .NET CLI의 기본 사항
- C# 콘솔 애플리케이션의 구조
- 콘솔 I/O
- .NET에 포함된 파일 I/O API의 기본 사항
- .NET에 포함된 작업 비동기 프로그래밍의 기본 사항

텍스트 파일을 읽고 콘솔에 해당 텍스트 파일의 내용을 에코하는 애플리케이션을 빌드해 보겠습니다. 콘솔의 출력은 소리 내어 읽는 속도에 맞춰집니다. "(보다 작음) 또는 '<'(보다 큼) 키를 눌러 속도를 향상하거나> 속도를 늦출 수 있습니다. Windows, Linux, macOS 또는 Docker 컨테이너에서 이 애플리케이션을 실행할 수 있습니다.

이 자습서에는 많은 기능이 있습니다. 하나씩 빌드해 보겠습니다.

## 사전 요구 사항

- [.NET 6 SDK](#).
- 코드 편집기.

## 앱 만들기

첫 번째 단계에서는 새 애플리케이션을 만듭니다. 명령 프롬프트를 열고 애플리케이션에 대한 새 디렉터리를 만듭니다. 해당 디렉터리를 현재 디렉터리로 지정합니다. 명령 프롬프트에 명령 `dotnet new console`을 입력합니다. 이렇게 하면 기본 "Hello World" 애플리케이션에 대한 시작 파일이 만들어집니다.

수정을 시작하기 전에 간단한 Hello World 애플리케이션을 실행해 보겠습니다. 애플리케이션을 만든 후에 명령 프롬프트에서 `dotnet run`를 입력합니다. 이 명령은 NuGet 패키지 복원 프로세스를 실행하고 애플리케이션 실행 파일을 만든 다음 실행 파일을 실행합니다.

간단한 Hello World 애플리케이션 코드는 `Program.cs`에 들어 있습니다. 원하는 텍스트 편집기를 사용하여 해당 파일을 엽니다. `Program.cs`의 코드를 다음 코드로 바꿉니다.

C#

```
namespace TeleprompterConsole;
```

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

파일의 맨 위에 있는 `namespace` 문을 살펴보세요. 사용해본 적이 있을 수 다른 개체 지향 언어와 마찬가지로 C#에서도 네임스페이스를 사용하여 형식을 구성합니다. 이 Hello World 프로그램도 다르지 않습니다. 프로그램이 이름이 `TeleprompterConsole`인 네임스페이스에 있는 것을 알 수 있습니다.

## 파일 읽기 및 에코

추가할 첫 번째 기능은 텍스트 파일을 읽고 콘솔에 해당 텍스트를 모두 표시하는 기능입니다. 먼저 텍스트 파일을 추가해 보겠습니다. 이 [샘플](#)에 대한 GitHub 리포지토리의 [sampleQuotes.txt](#) 파일을 프로젝트 디렉터리로 복사합니다. 이 파일은 애플리케이션에 대한 스크립트로 작동합니다. 이 자습서의 샘플 앱을 다운로드하는 방법을 알아보려면 [샘플 및 자습서](#)의 지침을 참조하세요.

다음에는 다음 메서드를 `Program` 클래스에 추가합니다(`Main` 메서드 바로 아래).

C#

```
static IEnumerable<string> ReadFrom(string file)
{
    string? line;
    using (var reader = File.OpenText(file))
    {
        while ((line = reader.ReadLine()) != null)
        {
            yield return line;
        }
    }
}
```

이 메서드는 [반복기](#) 메서드라는 특수한 형식의 C# 메서드입니다. 반복기 메서드는 지연 계산되는 시퀀스를 반환합니다. 즉, 시퀀스의 각 항목은 시퀀스를 사용하는 코드에서 요청된 것처럼 생성됩니다. 반복기 메서드는 하나 이상의 `yield return` 문을 포함하는 메서드입니다. `ReadFrom` 메서드에서 반환된 개체는 시퀀스의 각 항목을 생성하는 코드를 포함합니다. 이 예제에서는 소스 파일에서 다음 텍스트 줄을 읽고 해당 문자열을 반환하는 코드에 해당합니다. 호출하는 코드가 시퀀스에서 다음 항목을 요청할 때마다 코드는 파일에서 다음 텍스트 줄을 읽고 반환합니다. 파일이 완전히 읽히면 시퀀스는 더 이상 항목이 없음을 나타냅니다.

여러분에게 생소할 수 있는 두 개의 C# 구문 요소가 있습니다. 이 메서드의 `using` 문은 리소스 정리를 관리합니다. `using` 문(이 예제의 `reader`)에서 초기화되는 변수는 `IDisposable` 인터페이스를 구현해야 합니다. 이 인터페이스는 리소스를 해제해야 할 때 호출되어야 하는 단일 메서드 `Dispose`를 정의합니다. 컴파일러는 실행 중에 `using` 문의 닫는 중괄호에 도달하면 해당 호출을 생성합니다. 컴파일러에서 생성된 코드는 `using` 문으로 정의된 블록의 코드에서 예외가 `throw`되더라도 리소스가 해제되도록 합니다.

`reader` 변수는 `var` 키워드를 사용하여 정의됩니다. `var`은 암시적으로 형식화한 지역 변수를 정의합니다. 즉, 변수의 형식이 변수에 할당된 개체의 컴파일 시간 형식에 의해 결정됩니다. 여기서는 `StreamReader` 개체에 해당하는 `OpenText(String)` 메서드의 반환 값입니다.

이제 `Main` 메서드에서 파일을 읽는 코드를 채웁니다.

C#

```
var lines = ReadFrom("sampleQuotes.txt");
foreach (var line in lines)
{
    Console.WriteLine(line);
}
```

프로그램을 실행(`dotnet run` 사용)하면 모든 줄이 콘솔에 출력되는 것을 볼 수 있습니다.

## 지연 추가 및 출력 서식 지정

결과가 너무 빨리 표시되어 큰 표시로 읽을 수 없습니다. 이제 출력에 지연을 추가해야 합니다. 처음에는 비동기 처리를 가능하게 하는 일부 코어 코드를 빌드합니다. 그러나 이러한 첫 번째 단계는 몇 가지 안티 패턴을 따르게 됩니다. 안티 패턴은 코드를 추가할 때 주석에 표시되며 코드는 이후 단계에서 업데이트됩니다.

이 섹션에는 두 단계가 있습니다. 먼저 전체 줄이 아니라 단일 단어를 반환하는 반복기 메서드를 업데이트하게 됩니다. 이 작업은 다음과 같이 수정하면 수행됩니다. `yield return line;` 문을 다음 코드로 바꿉니다.

C#

```
var words = line.Split(' ');
foreach (var word in words)
{
    yield return word + " ";
}
yield return Environment.NewLine;
```

다음에는 해당 파일 줄을 사용하는 방법을 수정하고 각 단어를 쓴 후에 지연을 추가합니다. `Main` 메서드의 `Console.WriteLine(line)` 문을 다음 블록으로 바꿉니다.

C#

```
Console.WriteLine();
if (!string.IsNullOrWhiteSpace(line))
{
    var pause = Task.Delay(200);
    // Synchronously waiting on a task is an
    // anti-pattern. This will get fixed in later
    // steps.
    pause.Wait();
}
```

샘플을 실행하고 출력을 확인합니다. 이제 개별 단어가 출력되고 200밀리초의 지연이 발생합니다. 그러나 표시된 출력은 소스 텍스트 파일이 줄 바꿈 없는 80자 이상을 포함하는 여러 줄로 구성되므로 몇 가지 문제를 나타냅니다. 스크롤하면서 읽기 어려울 수 있습니다. 이 문제는 쉽게 해결할 수 있습니다. 각 줄의 길이를 추적하고 줄 길이가 특정 임계값에 도달할 때마다 새 줄을 생성하도록 하면 됩니다. 줄 길이를 포함하는 `ReadFrom` 메서드의 `words` 선언 다음에 지역 변수를 선언합니다.

C#

```
var lineLength = 0;
```

그런 후 `yield return word + " ";` 문 뒤에(닫는 중괄호 이전) 다음 코드를 추가합니다.

C#

```
lineLength += word.Length + 1;
if (lineLength > 70)
{
    yield return Environment.NewLine;
    lineLength = 0;
}
```

샘플을 실행하면 미리 구성된 속도로 크게 읽을 수 있습니다.

## 비동기 작업

이 마지막 단계에서는 텍스트 표시 속도를 높이거나 낮추려는 경우 또는 텍스트 표시를 완전히 중지하려는 경우 사용자로부터 입력을 읽는 작업을 실행하면서 다른 작업에서 비동기적으로 출력을 쓰는 코드를 추가합니다. 이 과정은 몇 가지 단계로 진행되며 마지막

에 필요한 모든 업데이트가 수행됩니다. 첫 번째 단계는 지금까지 파일을 읽고 표시하기 위해 만든 코드를 나타내는 비동기 Task 반환 메서드를 만드는 것입니다.

이 메서드를 `Program` 클래스(`Main` 메서드 본문에서 가져옴)에 추가합니다.

C#

```
private static async Task ShowTeleprompter()
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(200);
        }
    }
}
```

두 가지가 변경된 것을 알 수 있습니다. 첫째, 이 버전은 메서드 본문에서 `Wait()`를 호출하여 작업이 완료되기를 동기식으로 대기하지 않고, `await` 키워드를 사용합니다. 이 작업을 수행하기 위해 메서드 시그니처에 `async` 한정자를 추가해야 합니다. 이 메서드는 `Task`를 반환합니다. `Task` 개체를 반환하는 `Return` 문은 없습니다. 대신, `Task` 개체는 `await` 연산자를 사용할 때 컴파일러가 생성하는 코드에 의해 만들어집니다. `await`에 도달하면 이 메서드가 반환되는 것을 상상할 수 있습니다. 반환된 `Task`는 작업이 완료되지 않았음을 나타냅니다. 이 메서드는 대기 중인 작업이 완료되면 다시 시작됩니다. 실행되어 완료되면 반환된 `Task`는 완료되었음을 나타냅니다. 호출하는 코드는 반환된 `Task`를 모니터링하여 완료되었는지 확인합니다.

를 `await` 호출하기 전에 키워드(keyword) 추가합니다 `ShowTeleprompter`.

C#

```
await ShowTeleprompter();
```

이렇게 하려면 메서드 서명을 다음으로 `Main` 변경해야 합니다.

C#

```
static async Task Main(string[] args)
```

기본 사항 섹션에서 메서드에 대해 `async Main` 자세히 알아봅니다.

다음으로 콘솔에서 읽을 두 번째 비동기 메서드를 작성하고 "(보다 작음)", "(보다 큼)" 및 '<X' 또는 'x' 키에 대한 watch 합니다. 해당 작업에 대해 추가하는 메서드는 다음과 같습니다.

C#

```
private static async Task GetInput()
{
    var delay = 200;
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
            {
                delay -= 10;
            }
            else if (key.KeyChar == '<')
            {
                delay += 10;
            }
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
            {
                break;
            }
        } while (true);
    };
    await Task.Run(work);
}
```

콘솔에서 키를 읽는 대리자를 나타내는 `Action` 람다 식을 만들고 사용자가 "(보다 작음)" 또는 '<>' (보다 큼) 키를 누를 때 지연을 나타내는 지역 변수를 수정합니다. 대리자 메서드는 사용자가 언제든지 텍스트 표시를 중지할 수 있는 'X' 또는 'x' 키를 누르면 완료됩니다. 이 메서드는 `ReadKey()` 를 사용하여 차단한 후 사용자가 키를 누를 때까지 기다립니다.

이 기능을 완료하려면 이러한 두 작업(`GetInput` 및 `ShowTeleprompter`)을 시작하고 이러한 두 작업 간에 공유 데이터를 관리하는 새 `async Task` 반환 메서드를 만들어야 합니다.

이러한 두 작업 간에 공유 데이터를 처리할 수 있는 클래스를 만들 차례입니다. 이 클래스에는 두 개의 공용 속성, 즉 지연과 파일이 완전히 읽혔음을 나타내는 `Done` 플래그가 포함됩니다.

C#

```
namespace TeleprompterConsole;

internal class TelePrompterConfig
{
```

```

public int DelayInMilliseconds { get; private set; } = 200;
public void UpdateDelay(int increment) // negative to speed up
{
    var newDelay = Min(DelayInMilliseconds + increment, 1000);
    newDelay = Max(newDelay, 20);
    DelayInMilliseconds = newDelay;
}
public bool Done { get; private set; }
public void SetDone()
{
    Done = true;
}
}

```

이 클래스를 새 파일에 추가하고 위와 같이 `TeleprompterConsole` 네임스페이스에 이 클래스를 포함합니다. 또한 바깥쪽 클래스 또는 네임스페이스 이름 없이 `Min` 및 `Max` 메서드를 참조할 수 있도록 맨 위에 `using static` 문을 추가해야 합니다. `using static` 문은 하나의 클래스에서 메서드를 가져옵니다. 이것은 네임스페이스에서 모든 클래스를 가져오는 `static` 없는 `using` 문과는 반대됩니다.

C#

```
using static System.Math;
```

다음에는 새 `config` 개체를 사용하도록 `ShowTeleprompter` 및 `GetInput` 메서드를 업데이트해야 합니다. 두 작업을 모두 시작한 다음 첫 번째 작업이 완료될 때 종료되는 하나의 최종 `Task` 반환 `async` 메서드를 작성합니다.

C#

```

private static async Task RunTeleprompter()
{
    var config = new TelePrompterConfig();
    var displayTask = ShowTeleprompter(config);

    var speedTask = GetInput(config);
    await Task.WhenAny(displayTask, speedTask);
}

```

여기서 한 가지 새로운 메서드가 `WhenAny(Task[])` 호출입니다. 그러면 인수 목록의 모든 작업이 완료되는 즉시 완료되는 `Task`가 만들어집니다.

다음에는 지연을 위해 `config` 개체를 사용하도록 `ShowTeleprompter` 및 `GetInput` 메서드를 모두 업데이트해야 합니다.

C#

```

private static async Task ShowTeleprompter(TelePrompterConfig config)
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(config.DelayInMilliseconds);
        }
    }
    config.SetDone();
}

private static async Task GetInput(TelePrompterConfig config)
{
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
                config.UpdateDelay(-10);
            else if (key.KeyChar == '<')
                config.UpdateDelay(10);
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
                config.SetDone();
        } while (!config.Done);
    };
    await Task.Run(work);
}

```

이 새 버전의 `ShowTeleprompter`는 `TelePrompterConfig` 클래스에서 새 메서드를 호출합니다. 이제 `ShowTeleprompter` 대신 `RunTeleprompter`를 호출하도록 `Main`을 업데이트해야 합니다.

C#

```
await RunTeleprompter();
```

## 결론

이 자습서에서는 콘솔 애플리케이션 사용과 관련된 다양한 C# 언어 및 .NET Core 라이브러리 기능을 살펴보았습니다. 이 지식을 토대로 해당 언어 및 여기서 소개된 클래스에 대해 좀 더 자세히 알아볼 수 있습니다. 지금까지 파일 및 콘솔 I/O 기본 사항, 작업 기반 비동기 프로그래밍의 차단 및 비차단 사용, C# 언어 둘러보기, C# 프로그램이 구성되는 방식, .NET CLI에 대해 살펴보았습니다.

파일 I/O에 대한 자세한 내용은 [파일 및 스트림 I/O](#)를 참조하세요. 이 자습서에서 사용된 비동기 프로그래밍 모델에 대해 자세히 알아보려면 [작업 기반 비동기 프로그래밍](#) 및 [비동기 프로그래밍](#)을 참조하세요.

# 자습서: C#을 사용하여 .NET 콘솔 앱에서 HTTP 요청 만들기

아티클 • 2023. 04. 08.

이 자습서에서는 GitHub에서 REST 서비스에 대해 HTTP 요청을 발행하는 앱을 빌드합니다. 앱은 정보를 JSON 형식으로 읽고 JSON을 C# 개체로 변환합니다. JSON에서 C# 개체로 변환하는 것을 *deserialization*이라고 합니다.

자습서에서는 다음을 수행하는 방법을 보여 줍니다.

- ✓ HTTP 요청 보내기
- ✓ JSON 응답 역직렬화하기
- ✓ 특성을 사용하여 deserialization 구성하기

이 자습서의 [최종 샘플](#)을 따르려는 경우 해당 샘플을 다운로드할 수 있습니다. 다운로드 지침은 [샘플 및 자습서](#)를 참조하세요.

## 사전 요구 사항

- [.NET SDK 6.0 이상](#)
- [Visual Studio Code] (오픈 소스 플랫폼 간 편집기)와 같은 코드 편집기입니다. 샘플 앱은 Windows, Linux, macOS 또는 Docker 컨테이너에서 실행할 수 있습니다.

## 클라이언트 앱 만들기

1. 명령 프롬프트를 열고 앱을 저장할 새 디렉터리를 만듭니다. 해당 디렉터리를 현재 디렉터리로 지정합니다.
2. 콘솔 창에 다음 명령을 입력합니다.

```
.NET CLI  
dotnet new console --name WebAPIClient
```

이 명령은 기본 “Hello World” 앱을 위한 시작 파일을 만듭니다. 프로젝트 이름은 “WebAPIClient”입니다.

3. “WebAPIClient” 디렉터리로 이동하여 앱을 실행합니다.

```
.NET CLI
```

```
cd WebAPIClient
```

.NET CLI

```
dotnet run
```

`dotnet run`은 자동으로 `dotnet restore`를 실행하여 앱에 필요한 모든 종속성을 복원합니다. 필요한 경우 `dotnet build`도 실행합니다. 앱 출력 "Hello, World!" 이 표시됩니다. 터미널에서 `Ctrl+C` 를 눌러 앱을 중지합니다.

## HTTP 요청 만들기

이 앱은 [GitHub API](#) 를 호출하여 [.NET Foundation](#) 의 프로젝트에 관한 정보를 가져옵니다. 엔드포인트가 <https://api.github.com/orgs/dotnet/repos> 인 경우 정보를 가져오기 위해 HTTP GET 요청을 수행합니다. 브라우저도 HTTP GET 요청을 수행하므로 해당 URL 을 브라우저 주소 표시줄에 붙여넣어 수신되고 처리될 정보를 볼 수 있습니다.

`HttpClient` 클래스를 사용하여 HTTP 요청을 수행합니다. `HttpClient`는 오래 실행되는 API 에 대해 비동기 메서드만 지원합니다. 따라서 다음 단계는 비동기 메서드를 만들고 Main 메서드에서 해당 메서드를 호출합니다.

1. `Program.cs` 프로젝트 디렉터리에서 파일을 열고 해당 내용을 다음으로 바꿉니다.

```
C#  
  
await ProcessRepositoriesAsync();  
  
static async Task ProcessRepositoriesAsync(HttpClient client)  
{  
}
```

이 코드에서는 다음을 수행합니다.

- `Console.WriteLine` 문을 `await` 키워드를 사용하는 `ProcessRepositoriesAsync` 에 대한 호출로 바꿉니다.
- 빈 `ProcessRepositoriesAsync` 메서드를 정의합니다.

2. 클래스에서 `Program` 를 사용하여 `HttpClient` 콘텐츠를 다음 C#으로 바꿔 요청 및 응답을 처리합니다.

```
C#
```

```

using System.Net.Http.Headers;

using HttpClient client = new();
client.DefaultRequestHeaders.Accept.Clear();
client.DefaultRequestHeaders.Accept.Add(
    new
    MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));
client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation
Repository Reporter");

await ProcessRepositoriesAsync(client);

static async Task ProcessRepositoriesAsync(HttpClient client)
{
}

```

이 코드에서는 다음을 수행합니다.

- 모든 요청에 대해 다음과 같은 HTTP 헤더를 설정합니다.
  - JSON 응답을 받는 [Accept ↴](#) 헤더
  - [User-Agent ↴](#) 헤더입니다. 이러한 헤더는 GitHub 서버 코드에서 확인되며 GitHub에서 정보를 가져오는 데 필요합니다.

3. `ProcessRepositoriesAsync` 메서드에서 .NET Foundation 조직에 있는 모든 리포지토리의 목록을 반환하는 GitHub 엔드포인트를 호출합니다.

C#

```

static async Task ProcessRepositoriesAsync(HttpClient client)
{
    var json = await client.GetStringAsync(
        "https://api.github.com/orgs/dotnet/repos");

    Console.WriteLine(json);
}

```

이 코드에서는 다음을 수행합니다.

- 호출 [HttpClient.GetStringAsync\(String\)](#) 메서드에서 반환된 작업을 기다립니다. 이 메서드는 지정된 URI에 HTTP GET 요청을 보냅니다. 응답의 본문은 [String](#)으로 반환되는데, 이것은 작업이 완료되면 사용할 수 있습니다.
- 응답 문자열 `json`이 콘솔에 인쇄됩니다.

4. 앱을 빌드하고 실행합니다.

.NET CLI

```
dotnet run
```

이제 `ProcessRepositoriesAsync`에 `await` 연산자가 있으므로 빌드 경고가 발생하지 않습니다. 출력으로 JSON 텍스트가 길게 표시됩니다.

## JSON 결과 역직렬화

다음 단계는 JSON 응답을 C# 개체로 변환합니다. JSON을 개체로 역직렬화하려면 `System.Text.Json.JsonSerializer` 클래스를 사용합니다.

1. `Repository.cs`라는 파일을 만들고 다음 코드를 추가합니다.

```
C#
```

```
public record class Repository(string name);
```

위 코드는 GitHub API에서 반환된 JSON 개체를 나타내도록 클래스를 정의합니다. 이 클래스는 리포지토리 이름 목록을 표시하는 데 사용합니다.

리포지토리 개체의 JSON은 수십 개의 속성을 포함하지만, 이 중에서 `name` 속성만 역직렬화됩니다. 직렬 변환기는 대상 클래스에 일치하는 항목이 없는 JSON 속성을 자동으로 무시합니다. 따라서 대규모 JSON 패킷의 일부 필드에만 작용하는 형식을 쉽게 만들 수 있습니다.

C# 변환은 [속성 이름의 첫 문자를 대문자로 변환](#)하지만, 여기서 `name` 속성은 JSON에 있는 항목과 정확하게 일치하도록 소문자로 시작합니다. 뒤에서 JSON 속성 이름과 일치하지 않는 C# 속성 이름을 사용하는 방법을 알아봅니다.

2. 직렬 변환기를 사용하여 JSON을 C# 개체로 변환합니다. `ProcessRepositoriesAsync` 메서드의 `GetStringAsync(String)` 호출을 다음 줄로 바꿉니다.

```
C#
```

```
await using Stream stream =
    await
client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories =
    await JsonSerializer.DeserializeAsync<List<Repository>>(stream);
```

업데이트된 코드는 `GetStringAsync(String)`을 `GetStreamAsync(String)`으로 바꿉니다. 이 직렬 변환기 메서드는 문자열이 아닌 스트림을 소스로 사용합니다.

`JsonSerializer.DeserializeAsync< TValue >(Stream, JsonSerializerOptions, CancellationToken)`에 대한 첫 번째 인수는 `await` 식입니다. `await` 식은 지금까지는 대입문의 일부로만 볼 수 있었지만 코드의 거의 모든 위치에 나올 수 있습니다. 다른 두 매개 변수 `JsonSerializerOptions`와 `CancellationToken`은 선택 사항이며 코드 조각에서 생략됩니다.

`DeserializeAsync` 메서드는 '[제네릭](#)'입니다. 즉, JSON 텍스트에서 만들어야 하는 개체 종류에 대한 형식 인수를 제공해야 합니다. 이 예제에서는 다른 제네릭 개체 `System.Collections.Generic.List<T>`인 `List<Repository>`로 역직렬화합니다.

`List<T>` 클래스는 개체의 컬렉션을 저장합니다. 형식 인수는 `List<T>`에 저장된 개체의 형식을 선언합니다. JSON 텍스트는 `Repository` 리포지토리 개체의 컬렉션을 나타내므로 `type` 인수는 레코드입니다.

3. 각 리포지토리의 이름을 표시하도록 코드를 추가합니다. 다음 줄을

```
C#
```

```
Console.WriteLine(json);
```

다음 코드와 바꿉니다.

```
C#
```

```
foreach (var repo in repositories ?? Enumerable.Empty<Repository>())
    Console.WriteLine(repo.name);
```

4. 다음 `using` 지시문은 파일의 맨 위에 있어야 합니다.

```
C#
```

```
using System.Net.Http.Headers;
using System.Text.Json;
```

5. 앱을 실행합니다.

```
.NET CLI
```

```
dotnet run
```

.NET Foundation에 포함된 리포지토리의 이름 목록이 출력됩니다.

## deserialization 구성

1. *Repository.cs*에서 파일 내용을 다음 C#으로 바꿉니다.

```
C#  
  
using System.Text.Json.Serialization;  
  
public record class Repository(  
    [JsonPropertyName("name")] string Name);
```

이 코드에서는 다음을 수행합니다.

- `name` 속성의 이름을 `Name`로 변경합니다.
- 를 `JsonPropertyNameAttribute` 추가하여 이 속성이 JSON에 표시되는 방식을 지정합니다.

2. *Program.cs*에서 `Name` 속성의 새로운 대문자 표시를 사용하도록 코드를 업데이트합니다.

```
C#  
  
foreach (var repo in repositories)  
    Console.WriteLine(repo.Name);
```

3. 앱을 실행합니다.

출력은 동일합니다.

## 코드 리팩터링

`ProcessRepositoriesAsync` 메서드는 비동기 작업을 수행하고 리포지토리 컬렉션을 반환할 수 있습니다. 를 반환 `Task<List<Repository>>`하도록 해당 메서드를 변경하고 호출자 근처에 있는 콘솔에 쓰는 코드를 이동합니다.

1. `ProcessRepositoriesAsync`의 시그니처를 변경하여 `Repository` 개체의 목록을 해당 결과로 표시하는 작업을 반환합니다.

```
C#  
  
static async Task<List<Repository>> ProcessRepositoriesAsync()
```

2. JSON 응답을 처리한 후 리포지토리를 반환합니다.

```
C#
```

```
await using Stream stream =
    await
client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories =
    await JsonSerializer.DeserializeAsync<List<Repository>>(stream);
return repositories ?? new();
```

이 개체를 `async`로 표시했으므로 컴파일러는 반환 값에 대해 `Task<T>` 개체를 생성합니다.

3. `Program.cs` 파일을 수정하고 예 대한 호출을 `ProcessRepositoriesAsync` 다음으로 바꿔 결과를 캡처하고 각 리포지토리 이름을 콘솔에 씁니다.

```
C#
```

```
var repositories = await ProcessRepositoriesAsync(client);

foreach (var repo in repositories)
    Console.WriteLine(repo.Name);
```

4. 앱을 실행합니다.

출력은 동일합니다.

## 역직렬화 추가 속성

다음 단계에서는 받은 JSON 패킷에서 더 많은 속성을 처리하는 코드를 추가합니다. 모든 속성을 추가할 필요는 없겠지만, 몇 개를 추가해 봄으로써 C#의 다른 기능에 대해 알아볼 수 있습니다.

1. 클래스의 `Repository` 내용을 다음 `record` 정의로 바꿉니다.

```
C#
```

```
using System.Text.Json.Serialization;

public record class Repository(
    [property: JsonPropertyName("name")] string Name,
    [property: JsonPropertyName("description")] string Description,
    [property: JsonPropertyName("html_url")] Uri GitHubHomeUrl,
    [property: JsonPropertyName("homepage")] Uri Homepage,
    [property: JsonPropertyName("watchers")] int Watchers);
```

`Uri` 및 `int` 형식은 문자열 표현 간에 변환하는 기능을 기본적으로 제공합니다.

JSON 문자열에서 이러한 대상 유형으로 역직렬화할 때 추가 코드를 사용할 필요가

없습니다. JSON 패킷에 대상 형식으로 변환되지 않는 데이터가 있는 경우 serialization 작업이 예외를 throw합니다.

2. `foreach` *Program.cs* 파일의 루프를 업데이트하여 속성 값을 표시합니다.

C#

```
foreach (var repo in repositories)
{
    Console.WriteLine($"Name: {repo.Name}");
    Console.WriteLine($"Homepage: {repo.Homepage}");
    Console.WriteLine($"GitHub: {repo.GitHubHomeUrl}");
    Console.WriteLine($"Description: {repo.Description}");
    Console.WriteLine($"Watchers: {repo.Watchers:#,0}");
    Console.WriteLine();
}
```

3. 앱을 실행합니다.

이제 목록에 추가 속성이 포함됩니다.

## 데이터 속성 추가

JSON 응답에서 마지막 푸시 작업의 날짜는 다음과 같은 형식을 갖습니다.

JSON

```
2016-02-08T21:27:00Z
```

이 형식은 UTC(협정 세계시) 형식이므로 deserialization의 결과는 `Kind` 속성이 `Utc`인 `DateTime` 값입니다.

사용자의 표준 시간대로 표현된 날짜와 시간을 가져오려면 사용자 지정 변환 메서드를 작성해야 합니다.

1. *Repository.cs*에서 날짜 및 시간의 UTC 표현에 대한 속성과 로컬 시간으로 변환된 날짜를 반환하는 `readonly` `LastPush` 속성을 추가합니다. 파일은 다음과 같습니다.

C#

```
using System.Text.Json.Serialization;

public record class Repository(
    [property: JsonPropertyName("name")] string Name,
    [property: JsonPropertyName("description")] string Description,
    [property: JsonPropertyName("html_url")] Uri GitHubHomeUrl,
    [property: JsonPropertyName("homepage")] Uri Homepage,
```

```
[property: JsonPropertyName("watchers")] int Watchers,  
[property: JsonPropertyName("pushed_at")] DateTime LastPushUtc  
{  
    public DateTime LastPush => LastPushUtc.ToLocalTime();  
}
```

`LastPush` 속성은 `get` 접근자에 대한 식 본문 멤버를 사용하여 정의됩니다. `set` 접근자는 없습니다. 접근자를 `set` 생략하는 것은 C#에서 읽기 전용 속성을 정의하는 한 가지 방법입니다. (C#에서 쓰기 전용 속성을 만들 수 있지만 해당 값은 제한됩니다.)

2. *Program.cs*에서 또 다른 출력 문을 추가합니다.

```
C#  
  
Console.WriteLine($"Last push: {repo.LastPush}");
```

3. 전체 앱은 다음 *Program.cs* 파일과 유사해야 합니다.

```
C#  
  
using System.Net.Http.Headers;  
using System.Text.Json;  
  
using HttpClient client = new();  
client.DefaultRequestHeaders.Accept.Clear();  
client.DefaultRequestHeaders.Accept.Add(  
    new  
    MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));  
client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation  
Repository Reporter");  
  
var repositories = await ProcessRepositoriesAsync(client);  
  
foreach (var repo in repositories)  
{  
    Console.WriteLine($"Name: {repo.Name}");  
    Console.WriteLine($"Homepage: {repo.Homepage}");  
    Console.WriteLine($"GitHub: {repo.GitHubHomeUrl}");  
    Console.WriteLine($"Description: {repo.Description}");  
    Console.WriteLine($"Watchers: {repo.Watchers:#,0}");  
    Console.WriteLine($"{repo.LastPush}");  
    Console.WriteLine();  
}  
  
static async Task<List<Repository>> ProcessRepositoriesAsync(HttpClient  
client)  
{  
    await using Stream stream =  
        await
```

```
client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
    var repositories =
        await JsonSerializer.DeserializeAsync<List<Repository>>
(stream);
    return repositories ?? new();
}
```

#### 4. 앱을 실행합니다.

출력에 각 리포지토리에 대한 마지막 푸시의 날짜 및 시간이 포함됩니다.

## 다음 단계

이 자습서에서는 웹 요청을 수행하고 결과를 구문 분석하는 앱을 만들었습니다. 여러분이 만든 앱의 버전은 이제 [완성된 샘플](#)과 일치할 것입니다.

[.NET에서 JSON을 직렬화 및 역직렬화\(마샬링 및 역 마샬링\)하는 방법](#)에서 JSON serialization을 구성하는 방법을 알아보세요.

# LINQ(Language-Integrated Query) 작업

아티클 • 2023. 04. 08.

## 소개

이 자습서에서는 .NET Core 및 C# 언어의 기능에 대해 설명합니다. 이 문서에서 배울 내용은 다음과 같습니다.

- LINQ를 사용하여 시퀀스를 생성합니다.
- LINQ 쿼리에서 쉽게 사용할 수 있는 메서드를 작성합니다.
- 즉시 계산 및 지연 계산을 구분합니다.

모든 마술사들이 기본적으로 익히는 기술 중 하나인 [파로 셔플](#)을 보여 주는 애플리케이션을 빌드하여 이러한 기술을 살펴봅니다. 간단히 말해서 파로 셔플은 카드 데크를 정확히 절반으로 분할한 다음 각 절반의 각 카드를 교차로 섞어 원래 데크 순서로 다시 빌드하는 기술입니다.

마술사들은 카드를 섞은 후에 모든 카드가 알려진 위치로 들어가고 순서가 반복 패턴을 가지게 되므로 이 기술을 사용합니다.

이 자습서에서는 데이터 시퀀스 조작 과정을 간단하게 살펴봅니다. 빌드할 애플리케이션은 카드 데크를 생성한 다음 섞기 시퀀스를 수행하여 매번 시퀀스를 작성합니다. 또한 업데이트된 순서를 원래 순서와 비교할 것입니다.

이 자습서는 여러 단계로 구성됩니다. 각 단계 후에 애플리케이션을 실행하고 진행 상황을 확인할 수 있습니다. [완료된 샘플](#)은 GitHub의 dotnet/samples 리포지토리에서도 확인할 수 있습니다. 다운로드 지침은 [샘플 및 자습서](#)를 참조하세요.

## 사전 요구 사항

.NET Core를 실행하려면 컴퓨터에 설정해야 합니다. [.NET Core 다운로드](#) 페이지에서 설치 지침을 확인할 수 있습니다. Windows, Ubuntu Linux나 OS X 또는 Docker 컨테이너에서 이 애플리케이션을 실행할 수 있습니다. 선호하는 코드 편집기를 설치해야 합니다. 아래 설명에서는 오픈 소스 플랫폼 간 편집기인 [Visual Studio Code](#)를 사용합니다. 그러나 익숙한 어떤 도구도 사용 가능합니다.

## 애플리케이션 만들기

첫 번째 단계에서는 새 애플리케이션을 만듭니다. 명령 프롬프트를 열고 애플리케이션에 대한 새 디렉터리를 만듭니다. 해당 디렉터리를 현재 디렉터리로 지정합니다. 명령 프롬

프트에 명령 `dotnet new console`을 입력합니다. 이렇게 하면 기본 "Hello World" 애플리케이션에 대한 시작 파일이 만들어집니다.

이전에 C#을 사용해본 적이 없으면 [이 자습서](#)에서 C# 프로그램의 구조를 확인하세요. 해당 부분을 읽고 여기로 돌아와 LINQ에 대해 자세히 알아볼 수 있습니다.

## 데이터 세트 만들기

시작하기 전에 `dotnet new console`에서 생성된 `Program.cs` 파일의 맨 위에 다음 줄이 있는지 확인합니다.

```
C#  
  
// Program.cs  
using System;  
using System.Collections.Generic;  
using System.Linq;
```

이러한 세 줄(`using` 문)이 파일 맨 위에 없으면 프로그램이 컴파일되지 않습니다.

이제 필요한 참조가 모두 있으므로 카드 데크의 구성 요소를 고려합니다. 일반적으로 플레잉 카드 데크에는 네 개의 짹패가 있으며, 각 짹패에 13개의 값이 있습니다. 일반적으로 즉시 `Card` 클래스를 만들고 `Card` 개체 컬렉션을 수동으로 채우는 것을 고려할 수 있습니다. LINQ를 사용하면 일반적인 방법보다 더 간결하게 카드 데크 생성을 처리할 수 있습니다. `Card` 클래스를 만드는 대신, 각각 짹패와 순위를 나타내는 두 개의 시퀀스를 만들 수 있습니다. 순위와 짹패를 `IEnumerable<T>` 문자열로 생성하는 간단한 [반복기 메서드](#) 쌍을 만듭니다.

```
C#  
  
// Program.cs  
// The Main() method  
  
static IEnumerable<string> Suits()  
{  
    yield return "clubs";  
    yield return "diamonds";  
    yield return "hearts";  
    yield return "spades";  
}  
  
static IEnumerable<string> Ranks()  
{  
    yield return "two";  
    yield return "three";  
    yield return "four";  
    yield return "five";
```

```

        yield return "six";
        yield return "seven";
        yield return "eight";
        yield return "nine";
        yield return "ten";
        yield return "jack";
        yield return "queen";
        yield return "king";
        yield return "ace";
    }
}

```

이 코드를 `Program.cs` 파일의 `Main` 메서드 아래에 배치합니다. 이러한 두 메서드는 `yield return` 구문을 활용하여 실행 시 시퀀스를 생성합니다. 컴파일러는 `IEnumerable<T>`을 구현하는 개체를 빌드하고 요청 시 문자열 시퀀스를 생성합니다.

이제 이러한 반복기 메서드를 사용하여 카드 데크를 만듭니다. LINQ 쿼리를 `Main` 메서드에 배치합니다. 다음과 같이 표시됩니다.

C#

```

// Program.cs
static void Main(string[] args)
{
    var startingDeck = from s in Suits()
                        from r in Ranks()
                        select new { Suit = s, Rank = r };

    // Display each card that we've generated and placed in startingDeck in
    // the console
    foreach (var card in startingDeck)
    {
        Console.WriteLine(card);
    }
}

```

여러 `from` 절이 `SelectMany`를 생성합니다. 그러면 첫 번째 시퀀스의 각 요소를 두 번째 시퀀스의 각 요소와 조합하는 단일 시퀀스가 만들어집니다. 이 순서는 현재 목적에 따라 매우 중요합니다. 첫 번째 소스 시퀀스(Suites)의 첫 번째 요소는 두 번째 시퀀스(Ranks)의 모든 요소와 조합됩니다. 그 결과 첫 번째 세트의 13개 카드가 생성됩니다. 이 프로세스는 첫 번째 시퀀스(Suites)의 각 요소에 대해 반복됩니다. 최종 결과는 카드 데크를 세트별로 정렬한 후 다시 값별로 정렬한 상태입니다.

위에서 사용한 쿼리 구문에 LINQ를 작성할지, 아니면 메서드 구문을 대신 사용할지에 따라 항상 하나의 구문 형식에서 다른 구문 형식으로 전환할 수 있다는 것을 명심하세요. 쿼리 구문으로 작성된 위 쿼리는 다음과 같이 메서드 구문으로 작성할 수 있습니다.

C#

```
var startingDeck = Suits().SelectMany(suit => Ranks()).Select(rank => new {  
    Suit = suit, Rank = rank }));
```

컴파일러는 쿼리 구문으로 작성된 LINQ 문을 동등한 메서드 호출 구문으로 변환합니다. 따라서 선택한 구문과 관계없이 두 버전의 쿼리가 동일한 결과를 생성합니다. 사용자의 상황에 가장 적합한 구문을 선택합니다. 예를 들어 일부 멤버가 메서드 구문을 사용하는 데 어려움을 겪고 있는 팀에서는 쿼리 구문을 사용하는 것이 좋습니다.

계속해서 지금 빌드한 샘플을 실행합니다. 데크에 있는 52개의 모든 카드가 표시됩니다. 디버거에서 이 샘플을 실행하면 `Suits()` 및 `Ranks()` 메서드가 실행되는 방식을 확인할 수 있어서 매우 유용하다는 것을 알게 될 것입니다. 각 시퀀스의 각 문자열이 필요할 때만 생성된다는 것도 명확히 알 수 있습니다.

```
C:\>dotnet run
{ Suit = clubs, Rank = two }
{ Suit = clubs, Rank = three }
{ Suit = clubs, Rank = four }
{ Suit = clubs, Rank = five }
{ Suit = clubs, Rank = six }
{ Suit = clubs, Rank = seven }
{ Suit = clubs, Rank = eight }
{ Suit = clubs, Rank = nine }
{ Suit = clubs, Rank = ten }
{ Suit = clubs, Rank = jack }
{ Suit = clubs, Rank = queen }
{ Suit = clubs, Rank = king }
{ Suit = clubs, Rank = ace }
{ Suit = diamonds, Rank = two }
{ Suit = diamonds, Rank = three }
{ Suit = diamonds, Rank = four }
{ Suit = diamonds, Rank = five }
{ Suit = diamonds, Rank = six }
{ Suit = diamonds, Rank = seven }
{ Suit = diamonds, Rank = eight }
{ Suit = diamonds, Rank = nine }
{ Suit = diamonds, Rank = ten }
```

순서 조작

다음으로, 데크의 카드 순서를 섞는 메서드를 중심으로 살펴보겠습니다. 좋은 순서 섞기의 첫 번째 단계는 데크를 두 개로 분할하는 것입니다. LINQ API에 속하는 [Take](#) 및 [Skip](#) 메서드가 이 기능을 제공합니다. `foreach` 루프 아래에 카드를 배치합니다.

C#

```

foreach (var c in startingDeck)
{
    Console.WriteLine(c);
}

// 52 cards in a deck, so 52 / 2 = 26
var top = startingDeck.Take(26);
var bottom = startingDeck.Skip(26);
}

```

하지만 표준 라이브러리에는 이용할 수 있는 순서 섞기 메서드가 없으므로 고유한 메서드를 작성해야 합니다. 만들려는 순서 섞기 메서드는 LINQ 기반 프로그램에서 사용할 여러 기술을 보여 주므로 단계에서 이 프로세스의 각 부분을 설명하겠습니다.

LINQ 쿼리에서 반환되는 `IEnumerable<T>`을 조작하는 방법에 몇 가지 기능을 추가하려면 [확장 메서드](#)라는 특수한 종류의 메서드를 작성해야 합니다. 간단히 말해, 확장 메서드는 기능을 추가하려는 원래 형식을 수정하지 않고 기존 형식에 새로운 기능을 추가하는 특별한 용도의 정적 메서드입니다.

`Extensions.cs`라는 프로그램에 새 '정적' 클래스 파일을 추가하여 확장 메서드에 새로운 험을 제공한 다음, 첫 번째 확장 메서드 빌드를 시작합니다.

C#

```

// Extensions.cs
using System;
using System.Collections.Generic;
using System.Linq;

namespace LinqFaroShuffle
{
    public static class Extensions
    {
        public static IEnumerable<T> InterleaveSequenceWith<T>(this
            IEnumerable<T> first, IEnumerable<T> second)
        {
            // Your implementation will go here soon enough
        }
    }
}

```

메서드 시그니처, 특히 매개 변수를 잠시 살펴봅니다.

C#

```

public static IEnumerable<T> InterleaveSequenceWith<T> (this IEnumerable<T>
    first, IEnumerable<T> second)

```

이 메서드에 첫 번째 인수에 대한 `this` 한정자가 추가되는 것을 볼 수 있습니다. 즉, 마치 첫 번째 인수 형식의 멤버 메서드인 것처럼 이 메서드를 호출합니다. 또한 이 메서드 선언은 입력 및 출력 형식이 `IEnumerable<T>` 인 표준 관용구를 따릅니다. 이러한 방식에서는 LINQ 메서드가 서로 사슬처럼 연결되어 좀 더 복잡한 쿼리를 수행할 수 있도록 합니다.

데크를 절반씩 분할했으므로 이러한 절반을 조인해야 합니다. 코드에서는 이 작업을 위해 `Take` 및 `Skip`을 통해 얻은 두 시퀀스를 동시에 모두 열거하고 요소를 `interLeaving` 한다음, 이제 순서가 섞인 카드 데크인 하나의 시퀀스를 만듭니다. 두 시퀀스에 작동하는 LINQ 메서드를 작성하려면 `IEnumerable<T>` 작동 방식을 이해해야 합니다.

`IEnumerable<T>` 인터페이스는 한 가지 메서드인 `GetEnumerator`을 포함합니다. `GetEnumerator`에서 반환된 개체에는 다음 요소로 이동하기 위한 메서드와 시퀀스의 현재 요소를 검색하는 속성이 있습니다. 이러한 두 멤버를 사용하여 컬렉션을 열거하고 요소를 반환합니다. 이 `Interleave` 메서드는 반복기 메서드이므로, 컬렉션을 빌드하고 반환하는 대신, 위에 표시된 `yield return` 구문을 사용합니다.

해당 메서드의 구현은 다음과 같습니다.

```
C#  
  
public static IEnumerable<T> InterleaveSequenceWith<T>  
    (this IEnumerable<T> first, IEnumerable<T> second)  
{  
    var firstIter = first.GetEnumerator();  
    var secondIter = second.GetEnumerator();  
  
    while (firstIter.MoveNext() && secondIter.MoveNext())  
    {  
        yield return firstIter.Current;  
        yield return secondIter.Current;  
    }  
}
```

이 메서드를 작성했으므로 `Main` 메서드로 돌아가 데크 순서를 한 번 섞습니다.

```
C#  
  
// Program.cs  
public static void Main(string[] args)  
{  
    var startingDeck = from s in Suits()  
                        from r in Ranks()  
                        select new { Suit = s, Rank = r };  
  
    foreach (var c in startingDeck)  
    {  
        Console.WriteLine(c);  
    }  
}
```

```

var top = startingDeck.Take(26);
var bottom = startingDeck.Skip(26);
var shuffle = top.InterleaveSequenceWith(bottom);

foreach (var c in shuffle)
{
    Console.WriteLine(c);
}
}

```

## 비교

데크가 원래 순서로 돌아가는 데 몇 번을 섞어야 할까요? 알아내려면 두 시퀀스가 서로 같은지 확인하는 메서드를 작성해야 합니다. 이 메서드를 만든 후에는 데크 순서를 섞는 코드를 루프에 배치하고 데크가 원래 순서로 돌아갈 때를 확인해야 합니다.

두 시퀀스가 서로 같은지를 확인하는 메서드를 작성하는 작업은 간단합니다. 데크를 섞기 위해 작성한 메서드와 비슷한 구조를 갖습니다. 그렇지만 이번에는 각 요소를 `yield return`하는 대신, 각 시퀀스의 일치하는 요소를 비교합니다. 전체 시퀀스가 열거된 경우 모든 요소가 일치하면 시퀀스도 같습니다.

C#

```

public static bool SequenceEquals<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while ((firstIter?.MoveNext() == true) && secondIter.MoveNext())
    {
        if ((firstIter.Current is not null) &&
!firstIter.Current.Equals(secondIter.Current))
        {
            return false;
        }
    }

    return true;
}

```

다음에서는 두 번째 LINQ 관용구인 터미널 메서드를 보여 줍니다. 여기서는 시퀀스를 입력으로 사용하고(또는 이 경우 두 개의 시퀀스) 단일 스칼라 값을 반환합니다. 터미널 메서드를 사용하는 경우, 항상 LINQ 쿼리의 메서드 체인에서 최종 메서드이므로 이름이 “터미널”입니다.

이 메서드를 사용하여 데크가 원래 순서로 돌아갈 때를 확인하면 작동 방식을 확인할 수 있습니다. 순서 섞기 코드를 루프 내에 포함하고, `SequenceEquals()` 메서드를 적용하여 시퀀스가 원래 순서가 될 때 중지합니다. 이 메서드는 시퀀스 대신 단일 값을 반환하므로 어떤 쿼리에서든지 항상 마지막 메서드로 사용되는 것을 확인할 수 있습니다.

C#

```
// Program.cs
static void Main(string[] args)
{
    // Query for building the deck

    // Shuffling using InterleaveSequenceWith<T>();

    var times = 0;
    // We can re-use the shuffle variable from earlier, or you can make a
new one
    shuffle = startingDeck;
    do
    {
        shuffle = shuffle.Take(26).InterleaveSequenceWith(shuffle.Skip(26));

        foreach (var card in shuffle)
        {
            Console.WriteLine(card);
        }
        Console.WriteLine();
        times++;

    } while (!startingDeck.SequenceEquals(shuffle));

    Console.WriteLine(times);
}
```

지금까지 작성한 코드를 실행하고 순서를 섞을 때마다 데크가 어떻게 다시 배열되는지 확인합니다. 8번 순서 섞기(do-while 루프 반복) 후에 데크는 시작하는 LINQ 쿼리에서 처음 만들었을 때 데크의 원래 구성으로 돌아갑니다.

## 최적화

지금까지 빌드한 샘플은 ‘외부 순서 섞기’를 실행합니다. 즉, 맨 위 및 맨 아래 카드가 실행할 때마다 항상 같은 위치에 있습니다. 한 가지 부분을 변경하겠습니다. 52장 카드의 위치가 모두 변경되는 ‘내부 순서 섞기’를 대신 사용하겠습니다. 내부 순서 섞기의 경우 반으로 나눈 아래쪽 부분의 첫 번째 카드가 데크의 첫 번째 카드가 되도록 데크를 인터리빙합니다. 즉, 반으로 나눈 위쪽 부분의 마지막 카드가 맨 아래 카드가 됩니다. 이는 단일 코드 줄에 대한 간단한 변경입니다. `Take` 및 `Skip`의 위치를 전환하여 현재 순서 섞기 쿼리를 업데이트합니다. 이렇게 하면 데크의 위쪽 절반과 아래쪽 절반의 순서가 바뀝니다.

C#

```
shuffle = shuffle.Skip(26).InterleaveSequenceWith(shuffle.Take(26));
```

프로그램을 다시 실행합니다. 그러면 데크가 자체적으로 순서를 변경하는 데 52회 반복된다는 것을 알 수 있습니다. 또한 프로그램이 계속 실행될 때 몇 가지 심각한 성능 저하를 알 수 있습니다.

그 이유로는 여러 가지가 있습니다. 이 성능 저하의 주요 원인 중 하나인 비효율적인 '지연 계산' 사용을 해결할 수 있습니다.

간단히 말해, 지연 계산은 해당 값이 필요할 때까지 문이 계산되지 않음을 나타냅니다. LINQ 쿼리는 지연 계산되는 문입니다. 요소가 요청될 때만 시퀀스가 생성됩니다. 일반적으로 이것이 LINQ의 큰 장점입니다. 그러나 이러한 프로그램에서 사용하면 실행 시간이 기하급수적으로 늘어납니다.

LINQ 쿼리를 사용하여 원래 데크를 생성했습니다. 각 순서 섞기는 이전 데크에 대해 세 개의 LINQ 쿼리를 수행하여 생성됩니다. 이러한 모든 쿼리는 느리게 수행됩니다. 즉, 시퀀스가 요청될 때마다 다시 수행됩니다. 52번째 반복에 도달할 때까지 원래 데크를 아주 여러 번 다시 생성하게 됩니다. 이 동작을 보여 주기 위해 로그를 작성해 보겠습니다. 그런 후에 문제를 해결해 보겠습니다.

`Extensions.cs` 파일에서 아래 메서드를 입력하거나 복사합니다. 이 확장 메서드는 프로젝트 디렉터리에 `debug.log`라는 새 파일을 만들고 현재 실행 중인 쿼리를 로그 파일에 기록합니다. 이 확장 메서드를 임의 쿼리에 추가하여 해당 쿼리가 실행되었음을 표시할 수 있습니다.

C#

```
public static IEnumerable<T> LogQuery<T>
    (this IEnumerable<T> sequence, string tag)
{
    // File.AppendText creates a new file if the file doesn't exist.
    using (var writer = File.AppendText("debug.log"))
    {
        writer.WriteLine($"Executing Query {tag}");
    }

    return sequence;
}
```

`File` 아래에 빨간색 물결이 나타나면 존재하지 않는다는 것을 의미합니다. 컴파일러가 `File`을 인식하지 못하기 때문에 컴파일되지 않습니다. 이 문제를 해결하려면 `Extensions.cs`의 첫 번째 줄 아래에 다음 코드 줄을 추가해야 합니다.

C#

```
using System.IO;
```

이렇게 하면 문제가 해결되고 빨간색 오류가 사라집니다.

그런 후 로그 메시지를 사용하여 각 쿼리의 정의를 계측합니다.

C#

```
// Program.cs
public static void Main(string[] args)
{
    var startingDeck = (from s in Suits().LogQuery("Suit Generation")
                         from r in Ranks().LogQuery("Rank Generation")
                         select new { Suit = s, Rank = r
}).LogQuery("Starting Deck");

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();
    var times = 0;
    var shuffle = startingDeck;

    do
    {
        // Out shuffle
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26)
            .LogQuery("Bottom Half"))
            .LogQuery("Shuffle");
        */

        // In shuffle
        shuffle = shuffle.Skip(26).LogQuery("Bottom Half")
            .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top
Half"))
            .LogQuery("Shuffle");

        foreach (var c in shuffle)
        {
            Console.WriteLine(c);
        }

        times++;
        Console.WriteLine(times);
    } while (!startingDeck.SequenceEquals(shuffle));
```

```
        Console.WriteLine(times);
    }
```

쿼리에 액세스할 때마다 로깅하지는 않고, 원래 쿼리를 만들 때만 로깅합니다. 프로그램을 실행하는 데 여전히 오래 걸리지만 이제 이유를 확인할 수 있습니다. 로깅을 켜 상태로 내부 순서 섞기를 실행하다가 지치면 외부 순서 섞기로 다시 전환합니다. 여전히 자연 계산 효과가 나타날 것입니다. 한 번 실행에서 모든 값 및 세트 생성을 비롯한 2592개의 쿼리가 실행됩니다.

여기서 코드 성능을 개선하여 수행하는 실행 횟수를 줄일 수 있습니다. 간단한 해결 방법은 카드 데크를 구성하는 원래 LINQ 쿼리의 결과를 '캐시'하는 것입니다. 현재, do-while 루프가 반복될 때마다 쿼리를 계속해서 다시 실행하고 카드 데크를 다시 구성하며 매번 순서를 변경합니다. 카드 데크를 캐시하려면 LINQ 메서드 [ToArray](#) 및 [ToList](#)를 활용합니다. 두 메서드를 쿼리에 추가하면 지정된 대로 동일한 작업을 수행하지만, 이제 호출하도록 선택한 메서드에 따라 배열이나 목록에 결과를 저장합니다. LINQ 메서드 [ToArray](#)를 두 쿼리에 모두 추가하고 프로그램을 다시 실행합니다.

C#

```
public static void Main(string[] args)
{
    IEnumerable<Suit>? suits = Suits();
    IEnumerable<Rank>? ranks = Ranks();

    if ((suits is null) || (ranks is null))
        return;

    var startingDeck = (from s in suits.LogQuery("Suit Generation")
                        from r in ranks.LogQuery("Value Generation")
                        select new { Suit = s, Rank = r })
                        .LogQuery("Starting Deck")
                        .ToArray();

    foreach (var c in startingDeck)
    {
        Console.WriteLine(c);
    }

    Console.WriteLine();

    var times = 0;
    var shuffle = startingDeck;

    do
    {
        /*
        shuffle = shuffle.Take(26)
            .LogQuery("Top Half")
            .InterleaveSequenceWith(shuffle.Skip(26).LogQuery("Bottom
Half")))
    }
```

```

        .LogQuery("Shuffle")
        .ToArray();
    */

    shuffle = shuffle.Skip(26)
        .LogQuery("Bottom Half")
        .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
        .LogQuery("Shuffle")
        .ToArray();

    foreach (var c in shuffle)
    {
        Console.WriteLine(c);
    }

    times++;
    Console.WriteLine(times);
} while (!startingDeck.SequenceEquals(shuffle));

Console.WriteLine(times);
}

```

이제 외부 순서 섞기가 30개 쿼리로 줄었습니다. 내부 순서 섞기로 다시 실행해도 비슷하게 개선된 것을 확인할 수 있습니다. 이제 162개 쿼리가 실행됩니다.

이 예제는 지연 계산이 성능 문제를 일으킬 수 있는 사용 사례를 중점적으로 나타내도록 작성되었습니다. 지연 계산이 코드 성능에 영향을 줄 수 있는 위치를 확인하는 것만큼이나 모든 쿼리를 즉시 실행해야 하는 것은 아님을 이해하는 것도 중요합니다. `ToArray`를 사용하지 않을 경우 발생하는 성능 저하는 카드 데크의 새로운 배열이 각각 이전 배열에서 빌드되기 때문입니다. 지연 계산을 사용한다는 것은 각 새 데크 구성이 원래 데크에서 빌드되며, 심지어 `startingDeck`를 빌드한 코드를 실행하는 것을 의미합니다. 이로 인해 많은 양의 추가 작업이 발생합니다.

실제로 즉시 계산을 사용할 때 잘 실행되는 알고리즘도 있고, 지연 계산을 사용할 때 잘 실행되는 알고리즘도 있습니다. 일상적인 사용에서 지연 계산은 대체로 데이터베이스 엔진과 같이 데이터 소스가 별도 프로세스일 때 사용하는 것이 더 좋습니다. 데이터베이스의 경우 지연 계산을 통해 더 복잡한 쿼리에서 데이터베이스 프로세스로 하나의 왕복만 실행하고 나머지 코드 부분으로 돌아갈 수 있습니다. LINQ에서는 지연 계산을 실행할지, 아니면 즉시 계산을 실행할지를 유연하게 선택할 수 있으므로 프로세스를 측정하여 최상의 성능을 제공하는 계산 종류를 선택합니다.

## 결론

이 프로젝트에서는 다음 내용을 설명했습니다.

- LINQ 쿼리를 사용하여 데이터를 의미 있는 시퀀스로 집계

- 고유한 사용자 지정 기능을 LINQ 쿼리에 추가하는 확장 메서드 작성
- LINQ 쿼리에서 성능 저하와 같은 성능 문제가 발생할 수 있는 코드 영역 찾기
- LINQ 쿼리와 관련된 자연 및 즉시 계산과 쿼리 성능에 미치는 영향

LINQ 외에도 마법사가 카드 속임수에 사용하는 기술에 대해 약간 알아보았습니다. 마술사는 데크에서 모든 카드가 이동하는 위치를 제어할 수 있으므로 파로 순서 섞기 기술을 사용합니다. 이제 기술을 알고 있으니, 다른 모든 사용자를 위해 망치지 마세요.

LINQ에 대한 자세한 내용은 다음을 참조하세요.

- [LINQ\(Language-Integrated Query\)](#)
- [LINQ 소개](#)
- [기본 LINQ 쿼리 작업\(C#\)](#)
- [LINQ를 통한 데이터 변환\(C#\)](#)
- [LINQ의 쿼리 구문 및 메서드 구문\(C#\)](#)
- [LINQ를 지원하는 C# 기능](#)

# LINQ(Language-Integrated Query)

아티클 • 2024. 02. 15.

LINQ(Language-Integrated Query)는 C# 언어에 직접 쿼리 기능을 통합하는 방식을 기반으로 하는 기술 집합 이름입니다. 일반적으로 데이터에 대한 쿼리는 컴파일 시간의 형식 검사나 IntelliSense 지원 없이 간단한 문자열로 표현됩니다. 또한 SQL 데이터베이스, XML 문서, 다양한 웹 서비스 등의 각 데이터 소스 형식에 대해 서로 다른 쿼리 언어를 배워야 합니다. LINQ에서 쿼리는 클래스, 메서드 및 이벤트와 마찬가지로 최고의 언어 구문입니다.

쿼리를 작성할 때 LINQ에서 가장 눈에 띄는 "언어 통합" 부분은 쿼리 식입니다. 쿼리 식은 선언적 쿼리 구문으로 작성됩니다. 쿼리 구문을 사용하면 최소한의 코드로 데이터 원본에 대한 필터링, 정렬 및 그룹화 작업을 수행할 수 있습니다. 동일한 쿼리 식 패턴을 사용하여 모든 형식의 데이터 원본에서 데이터를 쿼리하고 변환할 수 있습니다.

다음 예제에서는 전체 쿼리 작업을 보여줍니다. 전체 작업에는 데이터 소스 만들기, 쿼리 식 정의 및 `foreach` 문의 쿼리 실행이 포함됩니다.

```
C#  
  
// Specify the data source.  
int[] scores = [97, 92, 81, 60];  
  
// Define the query expression.  
IQueryable<int> scoreQuery =  
    from score in scores  
    where score > 80  
    select score;  
  
// Execute the query.  
foreach (var i in scoreQuery)  
{  
    Console.Write(i + " ");  
}  
  
// Output: 97 92 81
```

이전 예제를 컴파일하려면 `using` 지시문 `using System.Linq;`를 추가해야 할 수도 있습니다. 최신 버전의 .NET에서는 암시적 `using`을 사용하여 이 지시문을 전역 `using`으로 추가합니다. 이전 버전에서는 원본에 추가해야 합니다.

## 쿼리 식 개요

- 쿼리 식은 LINQ 사용 데이터 원본에서 데이터를 쿼리하고 변환합니다. 예를 들어 단일 쿼리는 SQL 데이터베이스에서 데이터를 검색하고 XML 스트림을 출력으로 생성할 수 있습니다.
- 쿼리 식은 친숙한 C# 언어 구문을 많이 사용하므로 쉽게 읽을 수 있습니다.
- 쿼리 식의 변수는 모두 강력한 형식입니다.
- 예를 들어 `foreach` 문에서 쿼리 변수를 반복할 때까지 쿼리는 실행되지 않습니다.
- 컴파일 시간에 쿼리 식은 C# 사양에 정의된 규칙에 따라 표준 쿼리 연산자 메서드 호출로 변환됩니다. 쿼리 구문을 사용하여 표현할 수 있는 모든 쿼리는 메서드 구문으로도 표현할 수 있습니다. 경우에 따라 쿼리 구문이 더 읽기 쉽고 간결합니다. 다른 경우에는 메서드 구문이 더 읽기 쉽습니다. 두 가지 형식 간에 의미 체계 또는 성능의 차이가 없습니다. 자세한 내용은 [C# 언어 사양](#) 및 [표준 쿼리 연산자 개요](#)를 참조하세요.
- [Count](#) 또는 [Max](#)와 같은 일부 쿼리 작업은 해당하는 쿼리 식 절이 없으므로 메서드 호출로 표현해야 합니다. 메서드 구문을 다양한 방법으로 쿼리 구문에 조합할 수 있습니다.
- 쿼리 식은 쿼리가 적용되는 형식에 따라 식 트리 또는 대리자로 컴파일될 수 있습니다. [IEnumerable<T>](#) 쿼리는 대리자로 컴파일됩니다. [IQueryable](#) 및 [IQueryable<T>](#) 쿼리는 식 트리로 컴파일됩니다. 자세한 내용은 [식 트리](#)를 참조하세요.

## 데이터 원본의 LINQ 쿼리를 사용하도록 설정하는 방법

### 메모리 내 데이터

메모리 내 데이터의 LINQ 쿼리를 사용하도록 설정하는 방법에는 두 가지가 있습니다. 데이터가 [IEnumerable<T>](#)을 구현하는 형식인 경우 LINQ to Objects를 사용하여 데이터를 쿼리합니다. [IEnumerable<T>](#) 인터페이스를 구현하여 열거형을 사용하도록 설정하는 것이 적절하지 않은 경우 해당 형식 또는 해당 형식에 대한 [확장 메서드](#)로 LINQ 표준 쿼리 연산자 메서드를 정의합니다. 표준 쿼리 연산자의 사용자 지정 구현에서는 지연된 실행을 사용하여 결과를 반환해야 합니다.

### 원격 데이터

원격 데이터 소스의 LINQ 쿼리를 사용 설정하는 가장 좋은 방법은 [IQueryable<T>](#) 인터페이스를 구현하는 것입니다.

## IQueryable LINQ 공급자

[IQueryable<T>](#)을 구현하는 LINQ 공급자의 복잡성은 경우에 따라 크게 다릅니다.

덜 복잡한 `IQueryable` 공급자는 웹 서비스에서 단일 메서드에 액세스할 수 있습니다. 이러한 형식의 공급자는 쿼리에 처리할 특정 정보가 있다고 가정하므로 매우 한정적이며, 대개 단일 결과 형식을 노출하는 폐쇄형 형식 시스템을 갖고 있습니다. 대부분의 쿼리 실행은 로컬에서 수행됩니다. 예를 들어 표준 쿼리 연산자의 `Enumerable` 구현을 사용하여 실행됩니다. 복잡성이 낮은 공급자는 식 트리에서 쿼리를 나타내는 하나의 메서드 호출식만 검사하고 쿼리의 나머지 논리는 다른 곳에서 처리되도록 할 수 있습니다.

복잡성이 보통인 `IQueryable` 공급자는 부분적으로 표현되는 쿼리 언어가 있는 데이터 소스를 대상으로 할 수 있습니다. 웹 서비스를 대상으로 하는 경우 두 개 이상의 웹 서비스 메서드에 액세스하고 쿼리에서 찾는 정보에 따라 호출할 메서드를 선택할 수 있습니다. 복잡성이 보통인 공급자는 복잡성이 낮은 공급자보다 풍부한 형식 시스템을 가질 수 있지만 여전히 고정된 형식 시스템을 유지합니다. 예를 들어 공급자는 순회할 수 있는 일대다 관계가 있는 형식을 노출할 수 있지만 사용자 정의 형식에 대한 매핑 기술은 제공하지 않습니다.

[Entity Framework Core](#) 공급자와 같은 복잡한 `IQueryable` 공급자는 전체 LINQ 쿼리를 SQL과 같은 표현 쿼리 언어로 변환할 수 있습니다. 복잡한 공급자는 쿼리에서 더욱 다양한 질문을 처리할 수 있으므로 더 일반적입니다. 또한 개방형 형식 시스템을 가지므로 사용자 정의 형식을 매핑하는 확장 인프라를 포함해야 합니다. 복잡성이 높은 공급자를 개발하려면 상당한 노력이 필요합니다.

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

💡 설명서 문제 열기

↗️ 제품 사용자 의견 제공

# C#의 LINQ 쿼리 소개

아티클 • 2024. 04. 28.

쿼리는 데이터 소스에서 데이터를 검색하는 식입니다. 서로 다른 데이터 원본에는 관계형 데이터베이스용 SQL 및 XML용 XQuery와 같은 다양한 네이티브 쿼리 언어가 있습니다. 개발자는 지원해야 하는 데이터 원본 또는 데이터 형식의 각 형식에 대한 새 쿼리 언어를 학습해야 합니다. LINQ는 데이터 원본 및 형식의 종류에 일관된 C# 언어 모델을 제공하여 이러한 상황을 간소화합니다. LINQ 쿼리에서는 항상 C# 개체를 사용합니다. 동일한 기본 코딩 패턴을 사용하여 LINQ 공급자를 사용할 수 있는 경우 XML 문서, SQL 데이터베이스, .NET 컬렉션 및 기타 형식의 데이터를 쿼리하고 변환합니다.

## 쿼리 작업의 세 부분

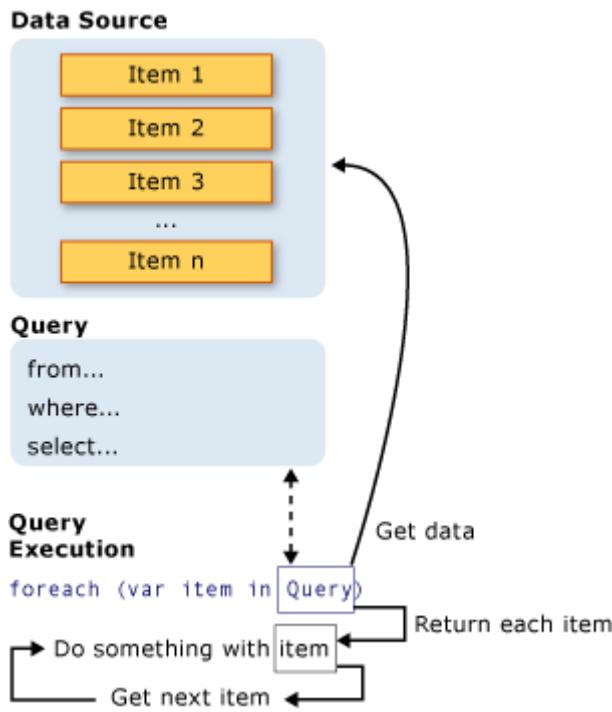
모든 LINQ 쿼리 작업은 다음과 같은 세 가지 고유한 작업으로 구성됩니다.

- 데이터 소스 가져오기.
- 쿼리 만들기.
- 쿼리를 실행합니다.

다음 예제에서는 쿼리 작업의 세 부분이 소스 코드로 표현되는 방식을 보여 줍니다. 예제에서는 편의상 정수 배열을 데이터 소스로 사용하지만 다른 데이터 소스에도 동일한 개념이 적용됩니다. 이 예제는 이 문서의 나머지 부분 전체에서 참조됩니다.

```
C#  
  
// The Three Parts of a LINQ Query:  
// 1. Data source.  
int[] numbers = [ 0, 1, 2, 3, 4, 5, 6 ];  
  
// 2. Query creation.  
// numQuery is an IEnumerable<int>  
var numQuery =  
    from num in numbers  
    where (num % 2) == 0  
    select num;  
  
// 3. Query execution.  
foreach (int num in numQuery)  
{  
    Console.Write("{0,1} ", num);  
}
```

다음 그림에서는 전체 쿼리 작업을 보여 줍니다. LINQ에서 쿼리 실행은 쿼리 자체와 다릅니다. 즉, 쿼리 변수를 만들어 데이터를 검색하지 않습니다.



## 데이터 소스

이전 예제의 데이터 원본은 제네릭 `IEnumerable<T>` 인터페이스를 지원하는 배열입니다. 즉, LINQ로 쿼리할 수 있다는 의미입니다. 쿼리가 `foreach` 문에서 실행되고, `foreach`는 `IEnumerable` 또는 `IEnumerable<T>`이 필요합니다. `IEnumerable<T>` 또는 제네릭 `IQueryable<T>` 같은 파생된 인터페이스를 지원하는 형식을 *쿼리 가능 형식*이라고 합니다.

쿼리 가능 형식은 LINQ 데이터 소스로 사용하기 위해 수정하거나 특별하게 처리할 필요가 없습니다. 원본 데이터가 쿼리 가능 형식으로 메모리에 아직 없는 경우 LINQ 공급자는 이를 나타내야 합니다. 예를 들어 LINQ to XML은 XML 문서를 쿼리 가능 `XElement` 형식으로 로드합니다.

C#

```
// Create a data source from an XML document.
// using System.Xml.Linq;
 XElement contacts = XElement.Load(@"c:\myContactList.xml");
```

`EntityFramework`를 사용하여 C# 클래스와 데이터베이스 스키마 간에 개체 관계형 매핑을 만듭니다. 개체에 대한 쿼리를 작성하고 런타임에 EntityFramework가 데이터베이스와의 통신을 처리합니다. 다음 예에서 `Customers`는 데이터베이스의 특정 테이블을 나타내며, `IQueryable<T>` 쿼리 결과 형식은 `IEnumerable<T>`에서 파생됩니다.

C#

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

// Query for customers in London.
IQueryable<Customer> custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;
```

특정 형식의 데이터 소스를 만드는 방법에 대한 자세한 내용은 다양한 LINQ 공급자에 대한 설명서를 참조하세요. 그러나 기본 규칙은 간단합니다. LINQ 데이터 원본은 제네릭 **IEnumerable<T>** 인터페이스를 지원하는 모든 개체이거나, 거기서 상속되는 인터페이스(일반적으로 **IQueryable<T>**)입니다.

### ① 참고

제네릭이 아닌 **IEnumerable** 인터페이스를 지원하는 **ArrayList** 같은 형식은 LINQ 데이터 소스로도 사용됩니다. 자세한 내용은 [LINQ를 사용하여 ArrayList를 쿼리하는 방법\(C#\)](#)을 참조하세요.

## 쿼리

쿼리는 데이터 소스 또는 소스에서 검색할 정보를 지정합니다. 선택적으로 쿼리는 해당 정보가 반환되기 전에 정렬, 그룹화 및 형성되는 방법도 지정합니다. 쿼리는 쿼리 변수에 저장되고 쿼리 식으로 초기화됩니다. [C# 쿼리 구문](#)을 사용하여 쿼리를 작성합니다.

이전 예제의 쿼리는 정수 배열에서 모든 짝수를 반환합니다. 쿼리 식에는 `from`, `where`, `select`의 세 가지 절이 포함되어 있습니다. (SQL에 익숙한 경우 절의 순서가 SQL의 순서와 반대임을 알고 있을 것입니다.) `from` 절은 데이터 소스를 지정하고 `where` 절은 필터를 적용하며 `select` 절은 반환되는 요소의 형식을 지정합니다. 이 섹션에서는 모든 쿼리 절에 대해 자세히 논의합니다. 여기에서 중요한 점은 LINQ에서 쿼리 변수 자체는 아무 작업도 수행하지 않고 데이터를 반환하지 않는다는 것입니다. 나중에 쿼리가 실행될 때 결과를 생성하는 데 필요한 정보를 저장합니다. 쿼리가 생성되는 방법에 대한 자세한 내용은 [표준 쿼리 연산자 개요\(C#\)](#)를 참조하세요.

### ① 참고

쿼리는 메서드 구문을 사용하여 표현할 수도 있습니다. 자세한 내용은 [LINQ의 쿼리 구문 및 메서드 구문](#)을 참조하세요.

# 실행 방식에 따른 표준 쿼리 연산자 분류

표준 쿼리 연산자 메서드의 LINQ to Objects 구현은 즉시 실행 또는 지연된 실행의 두 가지 기본 방식 중 하나로 실행됩니다. 지연된 실행을 사용하는 쿼리 연산자는 스트리밍 및 비스트리밍의 두 가지 범주로 추가로 구분할 수 있습니다.

## Immediate

즉시 실행은 데이터 소스를 읽고 작업이 한 번 수행됨을 의미합니다. 스칼라 결과를 반환하는 모든 표준 쿼리 연산자는 즉시 실행됩니다. 이러한 쿼리의 예로 `Count`, `Max`, `Average` 및 `First`가 있습니다. 이러한 메서드는 쿼리 자체가 결과를 반환하기 위해 `foreach`(을)를 사용해야 하므로 명시적 `foreach` 문 없이 실행됩니다. 이러한 쿼리는 `IEnumerable` 컬렉션이 아닌 단일 값을 반환합니다. `Enumerable.ToList` 또는 `Enumerable.ToArray` 메서드를 사용하면 모든 쿼리가 즉시 실행되도록 할 수 있습니다. 즉시 실행하면 쿼리 선언이 아닌 쿼리 결과를 다시 사용할 수 있습니다. 결과는 한 번 검색된 다음 나중에 사용할 수 있도록 저장됩니다. 다음 쿼리는 소스 배열에서 짝수의 개수를 반환합니다.

C#

```
var evenNumQuery =
    from num in numbers
    where (num % 2) == 0
    select num;

int evenNumCount = evenNumQuery.Count();
```

모든 쿼리를 즉시 실행하고 그 결과를 캐시하기 위해 `ToList` 또는 `ToArray` 메서드를 호출할 수 있습니다.

C#

```
List<int> numQuery2 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToList();

// or like this:
// numQuery3 is still an int[]

var numQuery3 =
    (from num in numbers
     where (num % 2) == 0
     select num).ToArray();
```

또한 `foreach` 루프를 쿼리 식 바로 다음에 배치하여 강제로 실행할 수 있습니다. 그러나 `ToList` 또는 `ToArray`를 호출하여 단일 컬렉션 개체에서 모든 데이터를 캐시할 수도 있습니다.

## 연기됨

지연된 실행은 코드의 쿼리가 선언되는 지점에서 작업이 수행되지 않음을 의미합니다. 예를 들어 `foreach` 문을 사용하여 쿼리 변수가 열거될 경우에만 작업이 수행됩니다. 쿼리 실행 결과는 쿼리 정의 시점이 아닌 쿼리 실행 시점의 데이터 원본 콘텐츠에 따라 달라집니다. 쿼리 변수가 여러 번 열거될 경우 매번 결과가 다를 수 있습니다. 반환 형식이 `IEnumerable<T>` 또는 `IOrderedEnumerable<TElement>`인 표준 쿼리 연산자는 대부분 지연 방식으로 실행됩니다. 지연 실행은 쿼리 결과가 반복될 때마다 쿼리가 데이터 소스에서 업데이트된 데이터를 가져오므로 쿼리 재사용 기능을 제공합니다. 다음 코드는 지연된 실행의 예를 보여 줍니다.

C#

```
foreach (int num in numQuery)
{
    Console.WriteLine("{0,1} ", num);
}
```

`foreach` 문은 쿼리 결과가 검색되는 위치이기도 합니다. 예를 들어 이전 쿼리에서 반복 변수 `num`은 반환된 시퀀스에서 각 값을 한 번에 하나씩 저장합니다.

쿼리 변수 자체는 쿼리 결과를 보유하지 않으므로 반복적으로 실행하여 업데이트된 데이터를 검색할 수 있습니다. 예를 들어, 별도의 애플리케이션이 데이터베이스를 지속적으로 업데이트할 수 있습니다. 사용 중인 애플리케이션에서 최신 데이터를 검색하는 하나의 쿼리를 만들 수 있으며, 업데이트된 결과를 검색하기 위해 간격을 두고 실행할 수 있습니다.

지연된 실행을 사용하는 쿼리 연산자는 스트리밍 및 비스트리밍으로 추가로 분류할 수 있습니다.

## 스트리밍

스트리밍 운영자는 요소를 생성하기 전에 모든 원본 데이터를 읽을 필요가 없습니다. 실행 시 스트리밍 연산자는 소스 요소를 읽을 때 각 소스 요소에 대해 작업을 수행하고 해당하는 경우 요소를 생성합니다. 스트리밍 연산자는 결과 요소가 생성될 때까지 소스 요소를 계속 읽습니다. 즉, 두 개 이상의 소스 요소를 읽어 하나의 결과 요소를 생성할 수 있습니다.

## 비스트리밍

비스트리밍 연산자는 결과 요소를 생성하기 전에 모든 원본 데이터를 읽어야 합니다. 정렬 또는 그룹화 등의 작업은 이 범주로 분류됩니다. 실행 시 비스트리밍 쿼리 연산자는 모든 원본 데이터를 읽고, 데이터 구조에 넣고, 작업을 수행하고, 결과 요소를 생성합니다.

## 분류 표

다음 표에서는 실행 방법에 따라 각 표준 쿼리 연산자 메서드를 분류합니다.

### ① 참고

한 연산자가 두 개의 열에 표시되어 있으면 두 개의 입력 시퀀스가 작업에 포함되고 각 시퀀스는 다르게 계산됩니다. 이러한 경우에 지연된 스트리밍 방식으로 계산되는 것은 항상 매개 변수 목록의 첫 번째 시퀀스입니다.

### [] 테이블 확장

표준 쿼리 연산자	반환 형식	즉시 실행	지연된 스트리밍 실행	지연된 비스트리밍 실행
Aggregate	TSource	X		
All	Boolean	X		
Any	Boolean	X		
AsEnumerable	IEnumerable<T>		X	
Average	단일 숫자 값		X	
Cast	IEnumerable<T>		X	
Concat	IEnumerable<T>		X	
Contains	Boolean		X	
Count	Int32		X	
DefaultIfEmpty	IEnumerable<T>		X	
Distinct	IEnumerable<T>		X	
ElementAt	TSource	X		
ElementAtOrDefault	TSource?	X		
Empty	IEnumerable<T>	X		

표준 쿼리 연산자	반환 형식	즉시 실행	지연된 스트리밍 실행	지연된 비스트리밍 실행
Except	IEnumerable<T>		X	X
First	TSource	X		
FirstOrDefault	TSource?	X		
GroupBy	IEnumerable<T>			X
GroupJoin	IEnumerable<T>		X	X
Intersect	IEnumerable<T>		X	X
Join	IEnumerable<T>		X	X
Last	TSource	X		
LastOrDefault	TSource?	X		
LongCount	Int64	X		
Max	단일 숫자 값, TSource 또는 TResult?	X		
Min	단일 숫자 값, TSource 또는 TResult?	X		
OfType	IEnumerable<T>	X		
OrderBy	IOrderedEnumerable<TElement>			X
OrderByDescending	IOrderedEnumerable<TElement>			X
Range	IEnumerable<T>	X		
Repeat	IEnumerable<T>	X		
Reverse	IEnumerable<T>			X
Select	IEnumerable<T>		X	
SelectMany	IEnumerable<T>		X	
SequenceEqual	Boolean	X		
Single	TSource	X		
SingleOrDefault	TSource?	X		
Skip	IEnumerable<T>		X	

표준 쿼리 연산자	반환 형식	즉시 실행	지연된 스트리밍 실행	지연된 비스트리밍 실행
SkipWhile	IEnumerable<T>		X	
Sum	단일 숫자 값	X		
Take	IEnumerable<T>		X	
TakeWhile	IEnumerable<T>		X	
ThenBy	IOrderedEnumerable<TElement>			X
ThenByDescending	IOrderedEnumerable<TElement>			X
ToArray	TSource[] 배열	X		
ToDictionary	Dictionary< TKey, TValue >	X		
ToList	IList<T>	X		
ToLookup	ILookup< TKey, TElement >	X		
Union	IEnumerable<T>		X	
Where	IEnumerable<T>		X	

## LINQ to Objects

"LINQ to Objects"는 `IEnumerable` 또는 `IEnumerable<T>` 컬렉션과 함께 LINQ 쿼리를 직접 사용하는 것을 의미합니다. LINQ를 사용하면 `List<T>`, `Array`, `Dictionary< TKey, TValue >` 등의 모든 열거 가능 컬렉션을 쿼리할 수 있습니다. 컬렉션은 사용자 정의일 수도 있고 .NET API에서 반환된 형식일 수도 있습니다. 그러나 LINQ 방식에서는 검색할 항목을 설명하는 선언적 코드를 작성합니다. LINQ to Objects는 LINQ를 사용한 프로그래밍에 대한 유용한 소개를 제공합니다.

LINQ 쿼리는 기존 `foreach` 루프에 비해 세 가지 주요 이점을 제공합니다.

- 보다 간결하며 쉽게 읽을 수 있습니다(특히 여러 조건을 필터링하는 경우).
- 최소한의 애플리케이션 코드로도 강력한 필터링, 순서 지정 및 그룹화 기능을 제공합니다.
- 거의 또는 전혀 수정하지 않고도 다른 데이터 소스에 이식할 수 있습니다.

데이터에 대해 수행하려는 작업이 복잡할수록 기존 반복 기술 대신 LINQ를 사용하여 더 많은 이점을 얻을 수 있습니다.

# 쿼리 결과를 메모리에 저장

쿼리는 기본적으로 데이터를 검색하고 구성하는 방법에 대한 명령 집합입니다. 쿼리는 결과의 각 후속 항목이 요청될 때 지연 실행됩니다. `foreach`를 사용하여 결과를 반복하는 경우 항목이 액세스될 때 반환됩니다. 쿼리를 평가한 후 `foreach` 루프를 실행하지 않고 결과를 저장하려면 쿼리 변수에 대해 다음 메서드 중 하나를 호출합니다.

- [ToList](#)
- [ToArray](#)
- [ToDictionary](#)
- [ToLookup](#)

다음 예와 같이 쿼리 결과를 저장할 때 반환된 컬렉션 개체를 새 변수에 할당해야 합니다.

C#

```
List<int> numbers = [1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20];

IEnumerable<int> queryFactorsOfFour =
    from num in numbers
    where num % 4 == 0
    select num;

// Store the results in a new variable
// without executing a foreach loop.
var factorsofFourList = queryFactorsOfFour.ToList();

// Read and write from the newly created list to demonstrate that it holds
// data.
Console.WriteLine(factorsofFourList[2]);
factorsofFourList[2] = 0;
Console.WriteLine(factorsofFourList[2]);
```

## 참고 항목

- [연습: C#에서 쿼리 작성](#)
- [foreach, in](#)
- [쿼리 키워드\(LINQ\)](#)

 GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

끌어오기 요청을 만들고 검토할  
수도 있습니다. 자세한 내용은  
[참여자 가이드](#)를 참조하세요.

☞ 설명서 문제 열기

☞ 제품 사용자 의견 제공

# 쿼리 식 기본 사항

아티클 • 2024. 03. 06.

이 문서에서는 C#의 쿼리 식과 관련된 기본 개념을 소개합니다.

## 쿼리란 무엇이며 쿼리의 기능은 무엇인가요?

쿼리는 지정된 데이터 소스(또는 소스)에서 검색할 데이터 및 반환된 데이터에 필요한 모양과 구성을 설명하는 지침 집합입니다. 쿼리와 쿼리에서 생성되는 결과는 다릅니다.

일반적으로 소스 데이터는 논리적으로 같은 종류의 요소 시퀀스로서 구성됩니다. 예를 들어 SQL 데이터베이스 테이블에는 행 시퀀스가 포함됩니다. XML 파일에는 XML 요소의 "시퀀스"가 있습니다(XML 요소는 트리 구조에서 계층적으로 구성되지만). 메모리 내 컬렉션은 개체의 시퀀스를 포함합니다.

애플리케이션의 관점에서 원본 데이터의 특정 형식과 구조는 중요하지 않습니다. 애플리케이션에는 소스 데이터가 항상 `IEnumerable<T>` 또는 `IQueryable<T>` 컬렉션으로 표시됩니다. 예를 들어 LINQ to XML에서 원본 데이터는 `IEnumerable< XElement >`(으)로 표시됩니다.

이 소스 시퀀스를 고려할 때 쿼리는 다음 세 가지 중 하나를 수행할 수 있습니다.

- 개별 요소를 수정하지 않고 요소의 하위 집합을 검색하여 새 시퀀스를 생성합니다. 그런 다음 쿼리는 다음 예제와 같이 반환된 시퀀스를 다양한 방식으로 정렬하거나 그룹화할 수 있습니다(`scores`(을)를 `int[]`(이)라고 가정).

C#

```
IEnumerable<int> highScoresQuery =
    from score in scores
    where score > 80
    orderby score descending
    select score;
```

- 이전 예제와 같이 요소의 시퀀스를 검색하지만, 이를 새로운 개체 형식으로 변환합니다. 예를 들어 쿼리는 데이터 원본의 특정 고객 레코드에서 패밀리 이름만 검색할 수 있습니다. 또는 전체 레코드를 검색한 다음 이를 사용하여 최종 결과 시퀀스를 생성하기 전에 다른 메모리 내 개체 형식 또는 XML 데이터를 생성할 수 있습니다. 다음 예제는 `int`에서 `string`으로의 프로젝션을 보여 줍니다. `highScoresQuery`의 새 형식을 참조하세요.

C#

```
IEnumerable<string> highScoresQuery2 =  
    from score in scores  
    where score > 80  
    orderby score descending  
    select $"The score is {score}";
```

- 소스 데이터에 대한 다음과 같은 singleton 값을 검색합니다.
  - 특정 조건과 일치하는 요소의 수.
  - 최대값 또는 최소값을 가지고 있는 요소.
  - 조건과 일치하는 첫 번째 요소 또는 지정된 요소 집합에서 특정 값의 합계. 예를 들어 다음 쿼리는 `scores` 정수 배열에서 80보다 큰 점수를 반환합니다.

C#

```
var highScoreCount = (  
    from score in scores  
    where score > 80  
    select score  
) .Count();
```

이전 예제에서는 `Enumerable.Count` 메서드를 호출하기 전에 쿼리 식 주변에 괄호를 사용했습니다. 새 변수를 사용하여 구체적인 결과를 저장할 수도 있습니다.

C#

```
IEnumerable<int> highScoresQuery3 =  
    from score in scores  
    where score > 80  
    select score;  
  
var scoreCount = highScoresQuery3.Count();
```

이전 예제에서는 `highScoresQuery`에 의해 반환된 요소 수를 확인하려면 `Count`가 결과를 반복해야 하기 때문에 `Count`에 대한 호출에서 쿼리가 실행됩니다.

## 쿼리 식이란 무엇입니까?

쿼리 식은 쿼리 구문으로 표현되는 쿼리입니다. 쿼리 식은 고급 언어 구문으로서, 이는 다른 식과 같으며 C# 식이 유효한 모든 컨텍스트에서 사용할 수 있습니다. 쿼리 식은 SQL 또는 XQuery와 유사한 선언적 구문으로 작성된 절 집합으로 구성됩니다. 각 절에는 하나

이상의 C# 식이 포함되며, 이러한 식 자체는 쿼리 식이거나 쿼리 식을 포함할 수 있습니다.

쿼리 식은 `from` 절로 시작하고 `select` 또는 `group` 절로 끝나야 합니다. 첫 번째 `from` 절과 마지막 `select` 또는 `group` 절 사이에는 `where`, `orderby`, `join`, `let` 및 심지어 다른 `from` 절과 같은 선택적 절 중 하나 이상을 포함할 수 있습니다. `into` 키워드로 사용하여 `join` 또는 `group` 절의 결과를 동일한 쿼리 식에서 더 많은 쿼리 절의 소스로 사용할 수 있습니다.

## 쿼리 변수

LINQ에서 쿼리 변수는 쿼리의 결과 대신 쿼리를 저장하는 변수입니다. 더 구체적으로, 쿼리 변수는 항상 `foreach` 문에서 반복될 때 요소 시퀀스를 생성하거나 해당 `IEnumerator.MoveNext()` 메서드에 대한 직접 호출을 생성하는 열거형 형식입니다.

### ① 참고

이 문서의 예제에서는 다음 데이터 원본 및 샘플 데이터를 사용합니다.

C#

```
record City(string Name, long Population);
record Country(string Name, double Area, long Population, List<City>
    Cities);
record Product(string Name, string Category);
```

C#

```
static readonly City[] cities = [
    new City("Tokyo", 37_833_000),
    new City("Delhi", 30_290_000),
    new City("Shanghai", 27_110_000),
    new City("São Paulo", 22_043_000),
    new City("Mumbai", 20_412_000),
    new City("Beijing", 20_384_000),
    new City("Cairo", 18_772_000),
    new City("Dhaka", 17_598_000),
    new City("Osaka", 19_281_000),
    new City("New York-Newark", 18_604_000),
    new City("Karachi", 16_094_000),
    new City("Chongqing", 15_872_000),
    new City("Istanbul", 15_029_000),
    new City("Buenos Aires", 15_024_000),
    new City("Kolkata", 14_850_000),
    new City("Lagos", 14_368_000),
    new City("Kinshasa", 14_342_000),
    new City("Manila", 13_923_000),
```

```

        new City("Rio de Janeiro", 13_374_000),
        new City("Tianjin", 13_215_000)
    ];

    static readonly Country[] countries = [
        new Country ("Vatican City", 0.44, 526, [new City("Vatican City",
826)]),
        new Country ("Monaco", 2.02, 38_000, [new City("Monte Carlo", 38_000)]),
        new Country ("Nauru", 21, 10_900, [new City("Yaren", 1_100)]),
        new Country ("Tuvalu", 26, 11_600, [new City("Funafuti", 6_200)]),
        new Country ("San Marino", 61, 33_900, [new City("San Marino", 4_500)]),
        new Country ("Liechtenstein", 160, 38_000, [new City("Vaduz", 5_200)]),
        new Country ("Marshall Islands", 181, 58_000, [new City("Majuro",
28_000)]),
        new Country ("Saint Kitts & Nevis", 261, 53_000, [new City("Basseterre",
13_000)])
    ];

```

다음 코드 예제는 하나의 데이터 소스, 하나의 필터링 절, 하나의 순서 지정 절, 변환 없는 원본 요소로 간단한 쿼리 식을 보여 줍니다. `select` 절은 쿼리를 종료합니다.

C#

```

// Data source.
int[] scores = [90, 71, 82, 93, 75, 82];

// Query Expression.
IQueryable<int> scoreQuery = //query variable
    from score in scores //required
    where score > 80 // optional
    orderby score descending // optional
    select score; //must end with select or group

// Execute the query to produce the results
foreach (var testScore in scoreQuery)
{
    Console.WriteLine(testScore);
}

// Output: 93 90 82 82

```

이전 예제에서 `scoreQuery`는 쿼리 변수이며, 간단히 쿼리라고 하는 경우도 있습니다. 쿼리 변수는 `foreach` 루프에서 생성되는 실제 결과 데이터를 저장하지 않습니다. 그리고 `foreach` 문이 실행되면 쿼리 결과가 쿼리 변수 `scoreQuery` 을(를) 통해 반환되지 않습니다. 대신 반복 변수 `testScore`(을)를 통해 반환됩니다. `scoreQuery` 변수는 두 번째 `foreach` 루프에서 반복될 수 있습니다. 자체적 수정 또는 데이터 원본이 모두 수정되지 않은 한 동일한 결과를 생성합니다.

쿼리 변수는 쿼리 구문 또는 메서드 구문으로 표현되는 쿼리 또는 둘의 조합을 저장할 수 있습니다. 다음 예제에서는 `queryMajorCities` 및 `queryMajorCities2` 모두 쿼리 변수입니다.

C#

```
City[] cities = [
    new City("Tokyo", 37_833_000),
    new City("Delhi", 30_290_000),
    new City("Shanghai", 27_110_000),
    new City("São Paulo", 22_043_000)
];

//Query syntax
IEnumerable<City> queryMajorCities =
    from city in cities
    where city.Population > 100000
    select city;

// Execute the query to produce the results
foreach (City city in queryMajorCities)
{
    Console.WriteLine(city);
}

// Output:
// City { Population = 120000 }
// City { Population = 112000 }
// City { Population = 150340 }

// Method-based syntax
IEnumerable<City> queryMajorCities2 = cities.Where(c => c.Population >
100000);
```

반면에 다음 두 예제에서는 각각 쿼리를 사용하여 초기화되더라도 쿼리 변수가 아닌 변수를 보여 줍니다. 이들은 결과를 저장하기 때문에 쿼리 변수가 아닙니다.

C#

```
var highestScore = (
    from score in scores
    select score
).Max();

// or split the expression
IEnumerable<int> scoreQuery =
    from score in scores
    select score;

var highScore = scoreQuery.Max();
```

```
// the following returns the same result  
highScore = scores.Max();
```

C#

```
var largeCitiesList = (  
    from country in countries  
    from city in country.Cities  
    where city.Population > 10000  
    select city  
).ToList();  
  
// or split the expression  
IEnumerable<City> largeCitiesQuery =  
    from country in countries  
    from city in country.Cities  
    where city.Population > 10000  
    select city;  
var largeCitiesList2 = largeCitiesQuery.ToList();
```

## 쿼리 변수의 명시적 형식 및 암시적 형식

이 문서는 일반적으로 쿼리 변수와 `select` 절 간의 형식 관계를 표시하기 위해 쿼리 변수의 명시적 형식을 제공합니다. 그러나 `var` 키워드를 사용하여 컴파일 시간에 쿼리 변수(또는 다른 지역 변수)의 형식을 추론하도록 컴파일러에 지시할 수도 있습니다. 예를 들어 이 문서의 앞에 표시된 쿼리 예제는 암시적 입력을 사용하여 표현할 수도 있습니다.

C#

```
var queryCities =  
    from city in cities  
    where city.Population > 100000  
    select city;
```

앞의 예제에서 `var`의 사용은 선택 사항입니다. `queryCities`(은)는 암시적으로 또는 명시적으로 입력되었는지 여부에 관계없이 `IEnumerable<City>`입니다.

## 쿼리 식 시작

쿼리 식은 `from` 절로 시작해야 합니다. 쿼리 식은 범위 변수와 함께 데이터 소스를 지정합니다. 범위 변수는 소스 시퀀스가 트래버스할 때 소스 시퀀스의 각 연속 요소를 나타냅니다. 범위 변수는 데이터 소스의 요소 형식을 기반으로 강력하게 형식이 지정됩니다. 다음 예제에서 `countries`는 `Country` 개체의 배열이므로 범위 변수의 형식도 `Country`로 지정

됩니다. 범위 변수의 형식은 강력하게 지정되므로 사용 가능한 형식 멤버에 액세스하기 위해 점 연산자를 사용할 수 있습니다.

C#

```
IEnumerable<Country> countryAreaQuery =  
    from country in countries  
    where country.Area > 500000 //sq km  
    select country;
```

쿼리가 세미콜론 또는 [continuation](#) 절로 종료될 때까지 범위 변수는 범위 내에 있습니다.

쿼리 식에는 여러 `from` 절이 포함될 수 있습니다. 소스 시퀀스의 각 요소가 컬렉션이거나 컬렉션을 포함하는 경우 더 많은 `from` 절을 사용합니다. 예를 들어 `Country` 개체의 컬렉션이 있으며, 각각에 `Cities`라는 이름의 `City` 개체 컬렉션이 포함되어 있다고 가정해 보겠습니다. 각 `Country`에서 `City` 개체를 쿼리하려면 다음과 같이 두 개의 `from` 절을 사용합니다.

C#

```
IEnumerable<City> cityQuery =  
    from country in countries  
    from city in country.Cities  
    where city.Population > 10000  
    select city;
```

자세한 내용은 [from 절](#)을 참조하세요.

## 쿼리 식 종료

쿼리 식은 `group` 절 또는 `select` 절로 끝나야 합니다.

### group 절

지정한 키로 구성되는 그룹의 시퀀스를 생성하려면 `group` 절을 사용합니다. 키의 데이터 형식은 무엇이든 가능합니다. 예를 들어, 다음 쿼리는 하나 이상의 `Country` 개체를 포함하며 키가 `char` 형식이고 값은 국가 이름의 첫 번째 문자인 그룹의 시퀀스를 만듭니다.

C#

```
var queryCountryGroups =  
    from country in countries  
    group country by country.Name[0];
```

그룹화에 대한 자세한 내용은 [group 절](#)을 참조하세요.

## select 절

다른 모든 시퀀스 형식을 생성하려면 `select` 절을 사용합니다. 간단한 `select` 절은 데이터 소스에 포함된 개체와 동일한 개체 형식의 시퀀스를 생성합니다. 이 예제에서는 데이터 소스에 `Country` 개체가 포함됩니다. `orderby` 절은 단지 요소를 새로운 순서로 정렬하고, `select` 절은 다시 정렬된 `Country` 개체의 시퀀스를 생성합니다.

C#

```
IEnumerable<Country> sortedQuery =  
    from country in countries  
    orderby country.Area  
    select country;
```

소스 데이터를 새 형식의 시퀀스로 변환하려면 `select` 절을 사용할 수 있습니다. 이 변환을 **프로젝션**이라고도 합니다. 다음 예제에서 `select` 절은 원래 요소에 있는 필드의 하위 집합만 포함하는 무명 형식의 시퀀스를 **프로젝션**합니다. 새 개체는 개체 이니셜라이저를 사용하여 초기화됩니다.

C#

```
var queryNameAndPop =  
    from country in countries  
    select new  
    {  
        Name = country.Name,  
        Pop = country.Population  
    };
```

따라서 이 예제에서는 쿼리가 무명 형식을 생성하기 때문에 `var`(이)가 필요합니다.

소스 데이터를 변환하기 위해 `select` 절을 사용하는 모든 방법에 대한 자세한 내용은 [select 절](#)을 참조하세요.

## into를 사용한 연속

`select` 또는 `group` 절에서 `into` 키워드를 사용하여 쿼리를 저장하는 임시 식별자를 만들 수 있습니다. 그룹화 또는 선택 작업 후에 쿼리에 대해 추가 쿼리 작업을 수행해야 하는 경우 `into` 절을 사용합니다. 다음 예제에서 `countries`(은)는 1,000만 범위의 인구에 따라 그룹화됩니다. 이러한 그룹을 만든 후에는 더 많은 절이 일부 그룹을 필터링한 다음 그

룹을 오름차순으로 정렬합니다. 이러한 추가 작업을 수행하려면 `countryGroup`(으)로 표현되는 연속 작업이 필요합니다.

C#

```
// percentileQuery is an IEnumerable<IGrouping<int, Country>>
var percentileQuery =
    from country in countries
    let percentile = (int)country.Population / 10_000_000
    group country by percentile into countryGroup
    where countryGroup.Key >= 20
    orderby countryGroup.Key
    select countryGroup;

// grouping is an IGrouping<int, Country>
foreach (var grouping in percentileQuery)
{
    Console.WriteLine(grouping.Key);
    foreach (var country in grouping)
    {
        Console.WriteLine(country.Name + ":" + country.Population);
    }
}
```

자세한 내용은 [into](#)를 참조하세요.

## 필터링, 정렬 및 조인

시작 절 `from`과 종료 절 `select` 또는 `group` 사이의 다른 모든 절(`where`, `join`, `orderby`, `from`, `let`)은 선택 사항입니다. 선택적 절은 쿼리 본문에서 0번 또는 여러 번 사용될 수 있습니다.

### where 절

하나 이상의 조건자식을 기반으로 소스 데이터에서 요소를 필터링하려면 `where` 절을 사용합니다. 다음 예제의 `where` 절에는 조건자 하나와 조건 두 개가 있습니다.

C#

```
IEnumerable<City> queryCityPop =
    from city in cities
    where city.Population is < 200000 and > 100000
    select city;
```

자세한 내용은 [where 절](#)을 참조하세요.

## orderby 절

결과를 오름차순 또는 내림차순으로 정렬하려면 `orderby` 절을 사용합니다. 2차 정렬 순서를 지정할 수도 있습니다. 다음 예제에서는 `Area` 속성을 사용하여 `country` 개체에 대해 1차 정렬을 수행합니다. 그런 다음 `Population` 속성을 사용하여 2차 정렬을 수행합니다.

C#

```
IEnumerable<Country> querySortedCountries =  
    from country in countries  
    orderby country.Area, country.Population descending  
    select country;
```

`ascending` 키워드는 선택 사항입니다. 지정된 순서가 없는 경우 기본 정렬 순서입니다. 자세한 내용은 [orderby 절](#)을 참조하세요.

## join 절

각 요소의 지정된 키 간 동일성 비교에 따라 하나의 데이터 소스의 요소를 다른 데이터 소스의 요소와 연결하거나 결합하려면 `join` 절을 사용합니다. LINQ에서는 요소의 형식이 서로 다른 개체의 시퀀스에 대해 조인 작업이 수행됩니다. 두 시퀀스를 조인한 후에는 `select` 또는 `group` 문을 사용하여 출력 시퀀스에 저장할 요소를 지정해야 합니다. 연결된 각 요소 집합의 속성을 출력 시퀀스의 새 형식으로 결합하는 데에도 무명 형식을 사용할 수 있습니다. 다음 예제는 `Category` 속성이 `categories` 문자열 배열의 범주 중 하나와 일치하는 `prod` 개체를 연결합니다. `Category(이)`가 `categories`의 문자열과 일치하지 않는 제품은 필터링됩니다. `select` 문은 속성을 `cat` 및 `prod`에서 가져오는 새 형식을 프로젝션합니다.

C#

```
var categoryQuery =  
    from cat in categories  
    join prod in products on cat equals prod.Category  
    select new  
    {  
        Category = cat,  
        Name = prod.Name  
    };
```

`into` 키워드를 사용하여 `join` 작업의 결과를 임시 변수에 저장함으로써 그룹 조인을 수행할 수 있습니다. 자세한 내용은 [join 절](#)을 참조하세요.

## let 절

식의 결과(예: 메서드 호출)를 새 범위 변수에 저장하려면 `let` 절을 사용합니다. 다음 예제에서 범위 변수 `firstName`(은)는 `Split`에서 반환하는 문자열 배열의 첫 번째 요소를 저장합니다.

C#

```
string[] names = ["Svetlana Omelchenko", "Claire O'Donnell", "Sven  
Mortensen", "Cesar Garcia"];  
IEnumerable<string> queryFirstNames =  
    from name in names  
    let firstName = name.Split(' ')[0]  
    select firstName;  
  
foreach (var s in queryFirstNames)  
{  
    Console.Write(s + " ");  
}  
  
//Output: Svetlana Claire Sven Cesar
```

자세한 내용은 [let 절](#)을 참조하세요.

## 쿼리 식의 하위 쿼리

쿼리 절 자체에는 [하위 쿼리](#)라고도 하는 쿼리 식이 포함될 수 있습니다. 각 하위 쿼리는 첫 번째 `from` 절에서 반드시 동일한 데이터 원본을 가리키지 않는 고유한 `from` 절로 시작합니다. 예를 들어 다음 쿼리는 그룹화 작업의 결과를 검색하기 위해 `select` 문에서 사용되는 쿼리 식을 보여 줍니다.

C#

```
var queryGroupMax =  
    from student in students  
    group student by student.Year into studentGroup  
    select new  
    {  
        Level = studentGroup.Key,  
        HighestScore = (  
            from student2 in studentGroup  
            select student2.ExamScores.Average()  
        ).Max()  
    };
```

자세한 내용은 [그룹화 작업에서 하위 쿼리를 수행하는 방법](#)을 참조하세요.

# 참고 항목

- 쿼리 키워드(LINQ)
- 표준 쿼리 연산자 개요

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

💡 설명서 문제 열기

↗️ 제품 사용자 의견 제공

# 데이터 쿼리를 위한 C# LINQ 쿼리 작성

아티클 • 2024. 04. 28.

LINQ(Language Integrated Query) 소개 설명서에 있는 대부분의 쿼리는 LINQ 선언적 쿼리 구문을 사용하여 작성되었습니다. 그러나 쿼리 구문은 코드를 컴파일할 때 .NET CLR(공용 언어 런타임)에 대한 메서드 호출로 변환해야 합니다. 이러한 메서드 호출은 `Where`, `Select`, `GroupBy`, `Join`, `Max`, `Average` 등과 같은 표준 쿼리 연산자를 호출합니다. 사용자는 쿼리 구문 대신 메서드 구문을 사용하여 연산자를 직접 호출할 수 있습니다.

쿼리 구문과 메서드 구문은 의미 체계적으로 동일하지만 쿼리 구문이 종종 더 간단하고 읽기 쉽습니다. 일부 쿼리는 메서드 호출로 표현해야 합니다. 예를 들어, 지정된 조건과 일치하는 요소 수를 검색하는 쿼리를 표현하려면 메서드 호출을 사용해야 합니다. 또한 소스 시퀀스에서 최대값을 갖는 요소를 검색하는 쿼리에 대해서도 메서드 호출을 사용해야 합니다. [System.Linq](#) 네임스페이스의 표준 쿼리 연산자에 대한 참조 문서는 일반적으로 메서드 구문을 사용합니다. 쿼리 및 쿼리 식 자체에서 메서드 구문을 사용하는 방법을 숙지해야 합니다.

## 표준 쿼리 연산자 확장 메서드

다음 예제는 간단한 쿼리 식 및 메서드 기반 쿼리로서 작성된, 의미상 동등한 쿼리를 보여줍니다.

C#

```
int[] numbers = [ 5, 10, 8, 3, 6, 12 ];

//Query syntax:
IQueryable<int> numQuery1 =
    from num in numbers
    where num % 2 == 0
    orderby num
    select num;

//Method syntax:
IQueryable<int> numQuery2 = numbers.Where(num => num % 2 == 0).OrderBy(n => n);

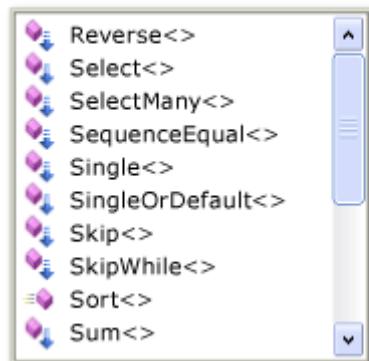
foreach (int i in numQuery1)
{
    Console.Write(i + " ");
}
Console.WriteLine(System.Environment.NewLine);
foreach (int i in numQuery2)
{
```

```
        Console.WriteLine(i + " ");
    }
```

두 예제에서 출력은 동일합니다. 쿼리 변수의 형식이 두 양식에서 모두 동일한 것을 알 수 있습니다([IEnumarable<T>](#)).

메서드 기반 쿼리를 이해할 수 있도록 더 자세히 살펴보겠습니다. 식의 오른쪽에서 이제 `where` 절이 `IEnumerable<int>` 형식인 `numbers` 개체의 인스턴스 메서드로 표현됩니다. 제네릭 `IEnumerable<T>` 인터페이스에 익숙한 경우 `where` 메서드가 없다는 것을 알 것입니다. 그러나 Visual Studio IDE에서 IntelliSense 완성 목록을 호출하는 경우 `where` 메서드뿐만 아니라 `Select`, `SelectMany`, `Join` 및 `Orderby`와 같은 다른 많은 메서드가 표시됩니다. 이러한 메서드는 표준 쿼리 연산자를 구현합니다.

```
List<string> list = new List<string>();
list.|
```



`IEnumerable<T>`에 더 많은 메서드가 포함된 것처럼 보이지만 그렇지 않습니다. 표준 쿼리 연산자는 확장 메서드로 구현됩니다. 확장 메서드는 기존 형식을 "확장"하며, 마치 형식에 대한 인스턴스 메서드인 것처럼 호출할 수 있습니다. 표준 쿼리 연산자는 `IEnumerable<T>`을 확장하므로 `numbers.Where(...)`를 작성할 수 있습니다.

확장 메서드를 사용하려면 `using` 지시문을 사용하여 범위로 가져옵니다. 애플리케이션의 관점에서는 일반 인스턴스 메서드와 확장 메서드가 동일합니다.

확장 메서드에 대한 자세한 내용은 [확장 메서드](#)를 참조하세요. 표준 쿼리 연산자에 대한 자세한 내용은 [표준 쿼리 연산자 개요\(C#\)](#)를 참조하세요. Entity Framework 및 LINQ to XML과 같은 일부 LINQ 공급자는 `IEnumerable<T>` 이외의 다른 형식에 대해 고유한 표준 쿼리 연산자 및 확장 메서드를 구현합니다.

## 람다 식

이전 예제에서는 조건식(`num % 2 == 0`)이 `Enumerable.Where` 메서드에 인라인 인수로 전달됩니다. `Where(num => num % 2 == 0)`. 이 인라인 식은 [람다 식](#)입니다. 그렇지 않으면 더 번거로운 형식으로 작성해야 하는 코드를 작성하는 편리한 방법입니다. 연산자 왼쪽의 `num`(은)은 쿼리 식의 `num`에 해당하는 입력 변수입니다. 컴파일러는 `numbers`가 제네릭

`IEnumerable<T>` 형식이라는 것을 알고 있으므로 `num`의 형식을 유추할 수 있습니다. 람다의 본문은 쿼리 구문 또는 다른 C# 식 또는 문의 식과 동일합니다. 메서드 호출 및 기타 복잡한 논리를 포함할 수 있습니다. 반환 값은 식 결과일 뿐입니다. 특정 쿼리는 메서드 구문으로만 표현할 수 있으며 그 중 일부는 람다 식이 필요합니다. 람다 식은 LINQ 도구 상자에서 강력하고 유연한 도구입니다.

## 쿼리 작성 가능성

이전 코드 예제에서 `Enumerable.OrderBy` 메서드는 `Where` 호출 시 점 연산자를 사용하여 호출됩니다. `Where`(은)는 필터링된 시퀀스를 생성한 다음 `Where`에서 생성되는 시퀀스를 `OrderBy` 정렬합니다. 쿼리는 `IEnumerable`을 반환하기 때문에, 사용자는 메서드 호출을 함께 연결하여 메서드 구문에서 쿼리를 작성합니다. 컴파일러는 쿼리 구문을 사용하여 쿼리를 작성할 때 이 컴파지션을 수행합니다. 쿼리 변수는 쿼리 결과를 저장하지 않으므로 쿼리를 실행한 후에도 언제든지 쿼리를 수정하거나 새 쿼리의 기준으로 사용할 수 있습니다.

다음 예제에서는 앞서 나열한 각 방법을 사용하여 몇몇 간단한 LINQ 쿼리를 보여 줍니다.

### ① 참고

이러한 쿼리는 간단한 메모리 내 컬렉션에서 작동합니다. 그러나 기본 구문은 LINQ to Entities 및 LINQ to XML에서 사용되는 구문과 동일합니다.

## 예제 - 쿼리 구문

쿼리 구문을 사용하여 대부분의 쿼리를 작성하여 쿼리 식을 만듭니다. 다음 예제는 세 개의 쿼리 식을 보여 줍니다. 첫 번째 쿼리 식은 `where` 절과 함께 조건을 적용하여 결과를 필터링 또는 제한하는 방법을 보여 줍니다. 이 식은 값이 7보다 크거나 3보다 작은 소수 시퀀스의 모든 요소를 반환합니다. 두 번째 식은 반환된 결과를 정렬하는 방법을 보여 줍니다. 세 번째 식은 키에 따라 결과를 그룹화하는 방법을 보여 줍니다. 이 쿼리는 단어의 첫 글자를 기반으로 두 그룹을 반환합니다.

C#

```
List<int> numbers = [5, 4, 1, 3, 9, 8, 6, 7, 2, 0];

// The query variables can also be implicitly typed by using var

// Query #1.
IQueryable<int> filteringQuery =
    from num in numbers
    where num is < 3 or > 7
```

```

    select num;

// Query #2.
IEnumerable<int> orderingQuery =
    from num in numbers
    where num is < 3 or > 7
    orderby num ascending
    select num;

// Query #3.
string[] groupingQuery = ["carrots", "cabbage", "broccoli", "beans",
"barley"];
IEnumerable<IGrouping<char, string>> queryFoodGroups =
    from item in groupingQuery
    group item by item[0];

```

쿼리의 형식은 `IEnumerable<T>`입니다. 다음 예제와 같이 이러한 모든 쿼리는 `var`을 사용해 작성할 수 있습니다.

```
var query = from num in numbers...
```

이전의 각 예제에서는 `foreach` 문 또는 기타 문의 쿼리 변수를 반복할 때까지 쿼리가 실제로 실행되지 않습니다.

## 예제 - 메서드 구문

일부 쿼리 작업은 메서드 호출로 표현해야 합니다. 이러한 가장 일반적인 메서드는 `Sum`, `Max`, `Min`, `Average` 등과 같은 싱글톤 숫자 값을 반환하는 메서드입니다. 이러한 메서드는 단일 값을 반환하고 추가 쿼리 작업의 원본으로 사용할 수 없으므로 항상 모든 쿼리에서 마지막으로 호출되어야 합니다. 다음 예제는 쿼리 식에서의 메서드 호출을 보여 줍니다.

C#

```

List<int> numbers1 = [5, 4, 1, 3, 9, 8, 6, 7, 2, 0];
List<int> numbers2 = [15, 14, 11, 13, 19, 18, 16, 17, 12, 10];

// Query #4.
double average = numbers1.Average();

// Query #5.
IEnumerable<int> concatenationQuery = numbers1.Concat(numbers2);

```

메서드에 `System.Action` 또는 `System.Func<TResult>` 매개 변수가 있는 경우 다음 예제와 같이 이러한 인수는 람다 식형식으로 제공됩니다.

C#

```
// Query #6.  
IEnumerable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
```

이전 쿼리에서는 제네릭 `IEnumerable<T>` 컬렉션이 아닌 단일 값을 반환하기 때문에 쿼리 #4만 즉시 실행됩니다. 메서드 자체는 해당 값을 계산하기 위해 `foreach` 또는 유사한 코드를 사용합니다.

이전 쿼리 각각은 다음 예제와 같이 '`var`"(으)로 암시적 입력을 사용하여 작성할 수 있습니다.

C#

```
// var is used for convenience in these queries  
double average = numbers1.Average();  
var concatenationQuery = numbers1.Concat(numbers2);  
var largeNumbersQuery = numbers2.Where(c => c > 15);
```

## 예제 - 혼합된 쿼리 및 메서드 구문

이 예제는 쿼리 절의 결과에서 메서드 구문을 사용하는 방법을 보여 줍니다. 쿼리 식을 괄호로 묶은 다음 점 연산자를 적용하고 메서드를 호출하면 됩니다. 다음 예제에서 쿼리 #7은 값이 3과 7 사이인 숫자의 수를 반환합니다. 그러나 일반적으로 두 번째 변수를 사용하여 메서드 호출의 결과를 저장하는 것이 좋습니다. 이렇게 하면 쿼리가 쿼리의 결과와 혼동될 가능성이 줄어듭니다.

C#

```
// Query #7.  
  
// Using a query expression with method syntax  
var numCount1 = (  
    from num in numbers1  
    where num is > 3 and < 7  
    select num  
).Count();  
  
// Better: Create a new variable to store  
// the method call result  
IEnumerable<int> numbersQuery =  
    from num in numbers1  
    where num is > 3 and < 7  
    select num;  
  
var numCount2 = numbersQuery.Count();
```

쿼리 #7은 컬렉션이 아닌 단일 값을 반환하므로 쿼리가 즉시 실행됩니다.

이전 쿼리는 다음과 같이 `var`과 함께 암시적 형식을 사용하여 작성할 수 있습니다.

```
C#
```

```
var numCount = (from num in numbers...
```

다음과 같이 메서드 구문에서 작성할 수 있습니다.

```
C#
```

```
var numCount = numbers.Count(n => n is > 3 and < 7);
```

다음과 같이 명시적 형식을 사용하여 작성할 수 있습니다.

```
C#
```

```
int numCount = numbers.Count(n => n is > 3 and < 7);
```

## 런타임에 동적으로 조건자 필터 지정

경우에 따라 `where` 절의 소스 요소에 적용해야 하는 조건자 수를 런타임할 때까지 알 수 없습니다. 다음 예제와 같이 여러 조건자 필터를 동적으로 지정하는 한 가지 방법은 `Contains` 메서드를 사용하는 것입니다. 쿼리는 쿼리가 실행될 때 `id` 값을 기반으로 다른 결과를 반환합니다.

```
C#
```

```
int[] ids = [111, 114, 112];

var queryNames =
    from student in students
    where ids.Contains(student.ID)
    select new
    {
        student.LastName,
        student.ID
    };

foreach (var name in queryNames)
{
    Console.WriteLine($"{name.LastName}: {name.ID}");
}

/* Output:
```

```

Garcia: 114
O'Donnell: 112
Omelchenko: 111
*/
// Change the ids.
ids = [122, 117, 120, 115];

// The query will now return different results
foreach (var name in queryNames)
{
    Console.WriteLine($"{name.LastName}: {name.ID}");
}

/* Output:
   Adams: 120
   Feng: 117
   Garcia: 115
   Tucker: 122
*/

```

`if... else` 또는 `switch`와 같은 제어 흐름 문을 사용하여 미리 결정된 대체 쿼리 중에서 선택할 수 있습니다. 다음 예에서 `oddYear`의 런타임 값이 `true` 또는 `false`인 경우 `studentQuery`는 다른 `where` 절을 사용합니다.

C#

```

void FilterByYearType(bool oddYear)
{
    IEnumerable<Student> studentQuery = oddYear
        ? (from student in students
           where student.Year is GradeLevel.FirstYear or
GradeLevel.ThirdYear
           select student)
        : (from student in students
           where student.Year is GradeLevel.SecondYear or
GradeLevel.FourthYear
           select student);
    var descr = oddYear ? "odd" : "even";
    Console.WriteLine($"The following students are at an {descr} year
level:");
    foreach (Student name in studentQuery)
    {
        Console.WriteLine($"{name.LastName}: {name.ID}");
    }
}

FilterByYearType(true);

/* Output:
   The following students are at an odd year level:
   Fakhouri: 116

```

```

Feng: 117
Garcia: 115
Mortensen: 113
Tucker: 119
Tucker: 122
*/
FilterByYearType(false);

/* Output:
The following students are at an even year level:
Adams: 120
Garcia: 114
Garcia: 118
O'Donnell: 112
Omelchenko: 111
Zabokritski: 121
*/

```

## 쿼리 식의 Null 값 처리

이 예제에서는 소스 컬렉션에서 가능한 null 값을 처리하는 방법을 보여 줍니다. `IEnumerable<T>` 등의 개체 컬렉션에는 값이 `null`인 요소가 포함될 수 있습니다. 소스 컬렉션이 `null` 이거나 값이 `null`인 요소를 포함하고 사용 중인 쿼리가 `null` 값을 처리하지 않는 경우 쿼리를 실행하면 `NullReferenceException`이 throw됩니다.

다음 예제와 같이 null 참조 예외를 피하도록 방어적으로 코딩할 수 있습니다.

C#

```

var query1 =
    from c in categories
    where c != null
    join p in products on c.ID equals p?.CategoryID
    select new
    {
        Category = c.Name,
        Name = p.Name
    };

```

이전 예제에서 `where` 절은 범주 시퀀스에서 모든 null 요소를 필터링합니다. 이 방법은 `join` 절의 null 확인과 관계가 없습니다. `Products.CategoryID` 가 `Nullable<int>`의 축약형인 `int?` 형식이므로 이 예제에서는 null이 있는 조건식이 적용됩니다.

`join` 절에서 비교 키 중 하나만 null 허용 값 형식인 경우에는 쿼리 식에서 다른 키를 null 허용 형식으로 캐스팅할 수 있습니다. 다음 예제에서는 `EmployeeID` 가 `int?` 형식의 값이 포함된 열이라고 가정합니다.

C#

```
var query =
    from o in db.Orders
    join e in db.Employees
        on o.EmployeeID equals (int?)e.EmployeeID
    select new { o.OrderID, e.FirstName };
```

각 예제에서 `equals` 쿼리 키워드를 사용합니다. `is null` 및 `is not null`에 대한 패턴을 포함하는 [패턴 일치](#)를 사용할 수도 있습니다. 쿼리 공급자가 새 C# 구문을 올바르게 해석하지 못할 수 있으므로 LINQ 쿼리에서는 이러한 패턴을 사용하지 않는 것이 좋습니다. 쿼리 공급자는 C# 쿼리 식을 Entity Framework Core와 같은 네이티브 데이터 형식으로 변환하는 라이브러리입니다. 쿼리 공급자는 `System.Linq.IQueryProvider` 인터페이스를 구현하여 `System.Linq.IQueryable<T>` 인터페이스를 구현하는 데이터 소스를 만듭니다.

## 쿼리 식의 예외 처리

쿼리 식의 컨텍스트에서 모든 메서드를 호출할 수 있습니다. 데이터 원본의 콘텐츠를 수정하거나 예외를 `throw`하는 등의 부작용을 일으킬 수 있는 쿼리 식의 메서드를 호출하지 마세요. 이 예제에서는 예외 처리에 대한 일반적인 .NET 지침을 위반하지 않고 쿼리 식에서 메서드를 호출할 때 예외 발생을 방지하는 방법을 보여 줍니다. 해당 지침에 의하면 특정 컨텍스트에서 예외가 `throw`된 이유를 이해할 경우 이 예외를 `catch`할 수 있습니다. 자세한 내용은 [최선의 예외 구현 방법](#)을 참조하세요.

마지막 예제에서는 쿼리 실행 중에 예외를 `throw`해야 할 경우 사례를 처리하는 방법을 보여 줍니다.

다음 예제에서는 예외 처리 코드를 쿼리 식 외부로 이동하는 방법을 보여 줍니다. 이 리팩터링은 메서드가 쿼리에 대한 로컬 변수에 의존하지 않는 경우에만 가능합니다. 쿼리 식 외부에서 예외를 처리하는 것이 더 쉽습니다.

C#

```
// A data source that is very likely to throw an exception!
IEnumerable<int> GetData() => throw new InvalidOperationException();

// DO THIS with a datasource that might
// throw an exception.
IEnumerable<int>? dataSource = null;
try
{
    dataSource = GetData();
}
catch (InvalidOperationException)
{
    Console.WriteLine("Invalid operation");
```

```

}

if (dataSource is not null)
{
    // If we get here, it is safe to proceed.
    var query =
        from i in dataSource
        select i * i;

    foreach (var i in query)
    {
        Console.WriteLine(i.ToString());
    }
}

```

이전 예의 `catch (InvalidOperationException)` 블록에서 애플리케이션에 적절한 방식으로 예외를 처리합니다(또는 처리하지 않습니다).

몇몇 경우에는 쿼리 내에서 `throw`된 예외에 대한 가장 좋은 응답은 쿼리 실행을 즉시 중지하는 것입니다. 다음 예제에서는 쿼리 본문 내부에서 `throw`된 예외를 처리하는 방법을 보여 줍니다. `SomeMethodThatMightThrow`가 잠재적으로 쿼리 실행을 중지해야 하는 예외를 일으킬 수 있다고 가정합니다.

`try` 블록은 쿼리 자체가 아닌 `foreach` 루프를 묶습니다. `foreach` 루프는 쿼리가 실행되는 지점입니다. 쿼리가 실행될 때 런타임 예외가 `throw`됩니다. 따라서 `foreach` 루프에서 처리되어야 합니다.

C#

```

// Not very useful as a general purpose method.
string SomeMethodThatMightThrow(string s) =>
    s[4] == 'C' ?
        throw new InvalidOperationException() :
        @"C:\newFolder\" + s;

// Data source.
string[] files = ["fileA.txt", "fileB.txt", "fileC.txt"];

// Demonstration query that throws.
var exceptionDemoQuery =
    from file in files
    let n = SomeMethodThatMightThrow(file)
    select n;

try
{
    foreach (var item in exceptionDemoQuery)
    {
        Console.WriteLine($"Processing {item}");
    }
}

```

```
}

catch (InvalidOperationException e)
{
    Console.WriteLine(e.Message);
}

/* Output:
Processing C:\newFolder\fileA.txt
Processing C:\newFolder\fileB.txt
Operation is not valid due to the current state of the object.
*/
```

발생할 것으로 예상되는 모든 예외를 catch하거나 `finally` 블록에서 필요한 정리를 수행하는 것을 기억합니다.

## 참고 항목

- 연습: C#에서 쿼리 작성
- where 절
- 런타임 상태에 따라 쿼리
- `Nullable<T>`
- `Nullable` 값 형식

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

[설명서 문제 열기](#)

[제품 사용자 의견 제공](#)

# LINQ 쿼리 작업의 형식 관계(C#)

아티클 • 2023. 12. 15.

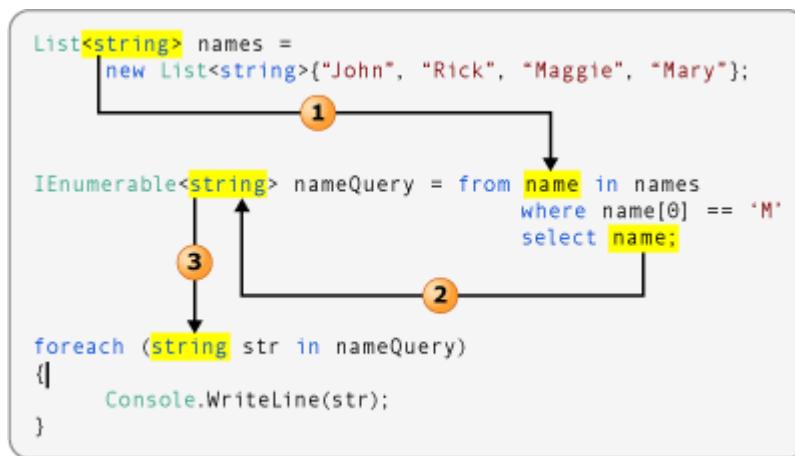
쿼리를 효과적으로 작성하려면 전체 쿼리 작업의 변수 형식이 모두 어떻게 서로 관련되는지를 이해해야 합니다. 이러한 관계를 이해하면 설명서의 LINQ 샘플 및 코드 예제를 더 쉽게 이해할 수 있습니다. 또한 변수를 사용하여 `var` 암시적으로 형식화할 때 발생하는 작업을 이해합니다.

LINQ 쿼리 작업은 데이터 소스, 쿼리 자체 및 쿼리 실행에서 강력하게 형식화됩니다. 쿼리의 변수 형식은 데이터 소스의 요소 형식 및 `foreach` 문의 반복 변수 형식과 호환되어야 합니다. 이 강력한 형식화는 사용자가 발견하기 전에 수정될 수 있도록 컴파일 시간에 형식 오류가 catch되도록 합니다.

이러한 형식 관계를 보여 주기 위해 뒤에 나오는 대부분의 예제에서는 모든 변수에 명시적 형식화를 사용합니다. 마지막 예제에서는 암시적 입력을 사용하는 `var` 경우에도 동일한 원칙이 적용되는 방법을 보여 줍니다.

## 소스 데이터를 변환하지 않는 쿼리

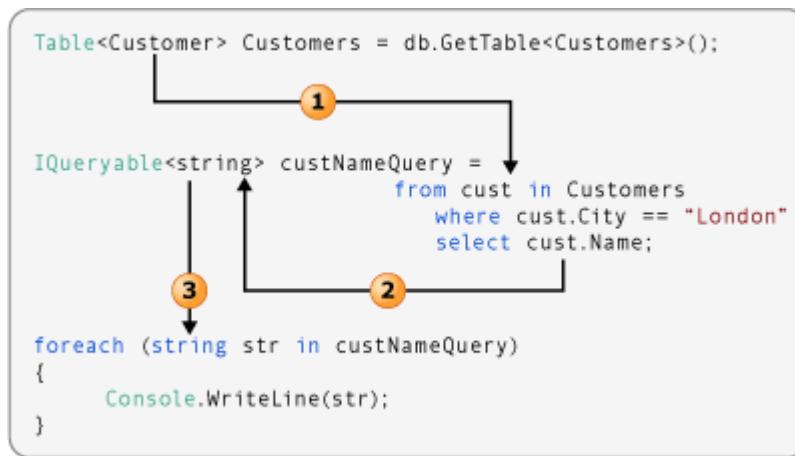
다음 그림에서는 데이터 변환을 수행하지 않는 LINQ to Objects 쿼리 작업을 보여 줍니다. 소스에는 문자열 시퀀스가 포함되어 있고, 쿼리 출력도 문자열 시퀀스입니다.



- 데이터 소스의 형식 인수에 따라 범위 변수의 형식이 결정됩니다.
- 선택된 개체의 형식에 따라 쿼리 변수의 형식이 결정됩니다. 여기서 `name`은 문자열입니다. 따라서 쿼리 변수는 `IEnumerable<string>`입니다.
- 쿼리 변수는 `foreach` 문에서 반복됩니다. 쿼리 변수가 문자열 시퀀스이기 때문에 반복 변수도 문자열입니다.

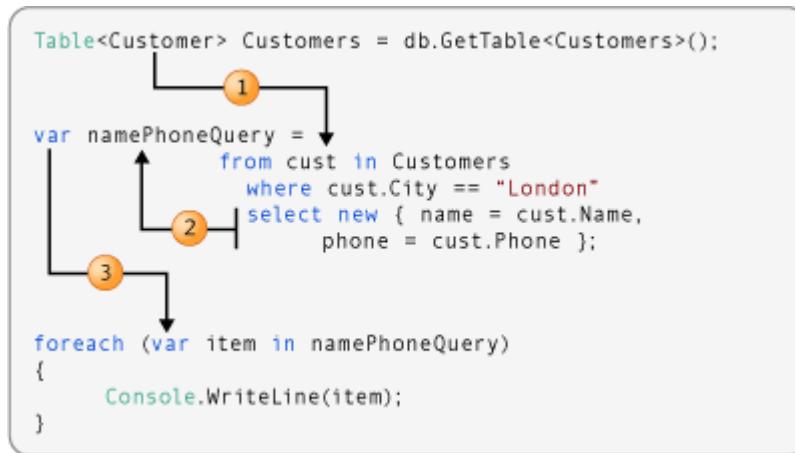
## 소스 데이터를 변환하는 쿼리

다음 그림에서는 데이터에 대한 간단한 변환을 수행하는 LINQ to SQL 쿼리 작업을 보여 줍니다. 쿼리는 `Customer` 개체 시퀀스를 입력으로 사용하고 결과에서 `Name` 속성만 선택합니다. `Name`이 문자열이기 때문에 쿼리에서 출력으로 문자열 시퀀스를 생성합니다.



- 데이터 소스의 형식 인수에 따라 범위 변수의 형식이 결정됩니다.
- `select` 문은 전체 `Customer` 개체가 아니라 `Name` 속성을 반환합니다. `Name`이 문자열 이므로 `custNameQuery`의 형식 인수는 `Customer`가 아니라 `string`입니다.
- `custNameQuery`가 문자열 시퀀스이므로 `foreach` 루프의 반복 변수도 `string`이어야 합니다.

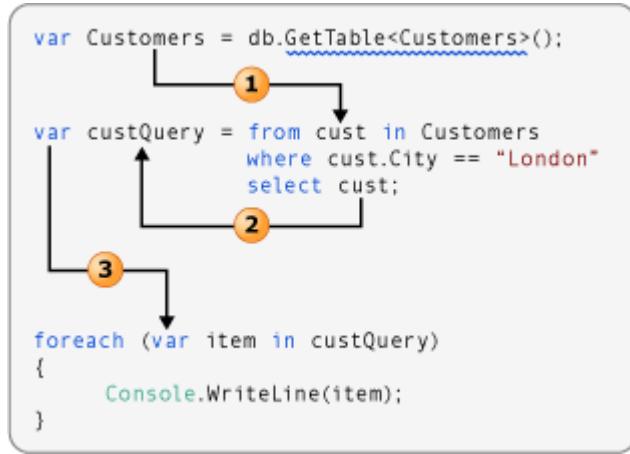
다음 그림에서는 약간 더 복잡한 변환을 보여 줍니다. `select` 문은 원래 `Customer` 개체의 두 멤버만 캡처하는 무명 형식을 반환합니다.



- 데이터 소스의 형식 인수는 항상 쿼리의 범위 변수 형식입니다.
- `select` 문이 무명 형식을 생성하기 때문에 `var`을 사용하여 쿼리 변수를 암시적으로 형식화해야 합니다.
- 쿼리 변수의 형식이 암시적이기 때문에 `foreach` 루프의 반복 변수도 암시적이어야 합니다.

## 컴파일러에서 형식 정보를 유추하도록 허용

쿼리 작업의 형식 관계를 이해해야 하지만 컴파일러가 모든 작업을 대신 수행하도록 하는 옵션도 있습니다. `var` 키워드를 쿼리 작업의 모든 지역 변수에 사용할 수 있습니다. 다음 그림은 앞에서 설명한 예제 번호 2와 유사합니다. 그러나 컴파일러는 쿼리 작업의 각 변수에 대해 강력한 형식을 제공합니다.



## LINQ 및 제네릭 형식(C#)

LINQ 쿼리는 제네릭 형식을 기반으로 합니다. 제네릭에 대한 세부 지식이 없어도 쿼리 작성은 시작할 수 있습니다. 그러나 다음 두 가지 기본 개념은 이해하는 것이 좋습니다.

1. `List<T>` 같은 제네릭 컬렉션 클래스의 인스턴스를 만들 때 “T”를 목록에 포함할 개체 형식으로 대체합니다. 예를 들어 문자열 목록은 `List<string>`으로 표현되고, `Customer` 개체 목록은 `List<Customer>`로 표현됩니다. 제네릭 목록은 강력한 형식이어야 하며 해당 요소를 `Object`로 저장하는 컬렉션에 비해 많은 장점을 제공합니다. `List<string>`에 `Customer`를 추가하려고 하면 컴파일 시간에 오류가 발생합니다. 런타임 형식 캐스팅을 수행할 필요가 없기 때문에 제네릭 컬렉션을 사용하기가 쉽습니다.
2. `IEnumerable<T>`은 `foreach` 문을 사용하여 제네릭 컬렉션 클래스를 열거할 수 있는 인터페이스입니다. 제네릭 컬렉션 클래스는 `ArrayList` 등의 제네릭이 아닌 컬렉션이 `IEnumerable`을 지원하는 것처럼 `IEnumerable<T>`을 지원합니다.

제네릭에 대한 자세한 내용은 [제네릭](#)을 참조하세요.

## LINQ 쿼리의 `IEnumerable<T>` 변수

LINQ 쿼리 변수는 `IEnumerable<T>` 또는 파생 형식(예: `IQueryable<T>`)으로 형식화됩니다. 형식이 `IEnumerable<Customer>`인 쿼리 변수가 표시되면 쿼리가 실행될 때 0개 이상의 `Customer` 개체 시퀀스를 생성한다는 의미입니다.

```
IEnumerable<Customer> customerQuery =  
    from cust in customers  
    where cust.City == "London"  
    select cust;  
  
foreach (Customer customer in customerQuery)  
{  
    Console.WriteLine($"{customer.LastName}, {customer.FirstName}");  
}
```

## 컴파일러에서 제네릭 형식 선언을 처리하도록 허용

원하는 경우 `var` 키워드를 사용하여 제네릭 구문을 방지할 수 있습니다. `var` 키워드는 `from` 절에 지정된 데이터 소스를 확인하여 쿼리 변수의 형식을 유추하도록 컴파일러에 지시합니다. 다음 예제에서는 이전 예제와 동일하게 컴파일된 코드를 생성합니다.

C#

```
var customerQuery2 =  
    from cust in customers  
    where cust.City == "London"  
    select cust;  
  
foreach(var customer in customerQuery2)  
{  
    Console.WriteLine($"{customer.LastName}, {customer.FirstName}");  
}
```

`var` 키워드는 변수의 형식이 명확하거나 그룹 쿼리에 의해 생성되는 형식과 같이 중첩된 제네릭 형식을 명시적으로 지정하는 것이 중요하지 않은 경우에 유용합니다. 일반적으로 `var`을 사용하는 경우 다른 사용자가 코드를 읽기가 더 어려워질 수 있습니다. 자세한 내용은 [암시적으로 형식화된 지역 변수](#)를 참조하세요.

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

☞ 설명서 문제 열기

☞ 제품 사용자 의견 제공

# LINQ를 지원하는 C# 기능

아티클 • 2024. 04. 30.

## 쿼리 식

쿼리 식은 SQL 또는 XQuery와 유사한 선언적 구문을 사용하여

`System.Collections.Generic.IEnumerable<T>` 컬렉션을 쿼리합니다. 컴파일 시간에 쿼리 구문은 LINQ 공급자의 표준 쿼리 메서드 구현에 대한 메서드 호출로 변환됩니다. 애플리케이션은 `using` 지시문으로 적절한 네임스페이스를 지정하여 범위 내에 있는 표준 쿼리 연산자를 제어합니다. 다음 쿼리 식은 문자열의 배열을 사용하고 문자열의 첫 번째 문자에 따라 그룹화하며 그룹의 순서를 지정합니다.

C#

```
var query = from str in stringArray
            group str by str[0] into stringGroup
            orderby stringGroup.Key
            select stringGroup;
```

## 암시적으로 형식화된 변수(var)

다음과 같이 `var` 한정자를 사용하여 컴파일러에 형식을 유추하고 할당하도록 지시할 수 있습니다.

C#

```
var number = 5;
var name = "Virginia";
var query = from str in stringArray
            where str[0] == 'm'
            select str;
```

`var`(으)로 선언된 변수는 명시적으로 지정하는 형식의 변수와 마찬가지로 강력한 형식입니다. `var`(을)를 사용하면 무명 형식을 만들 수 있지만 지역 변수에 대해서만 만들 수 있습니다. 자세한 내용은 [암시적으로 형식화된 지역 변수](#)를 참조하세요.

## 개체 및 컬렉션 이니셜라이저

개체 및 컬렉션 이니셜라이저를 사용하면 개체에 대한 생성자를 명시적으로 호출하지 않고 개체를 초기화할 수 있습니다. 일반적으로 이니셜라이저는 소스 데이터를 새 데이터

형식으로 프로젝션할 때 쿼리 식에서 사용됩니다. public `Name` 및 `Phone` 속성이 있는 `Customer`라는 클래스를 가정할 경우 다음 코드에서처럼 개체 이니셜라이저를 사용할 수 있습니다.

C#

```
var cust = new Customer { Name = "Mike", Phone = "555-1212" };
```

`Customer` 클래스를 계속 진행하면서 `IncomingOrders`(이)라는 데이터 원본이 있으며, 큰 `OrderSize`(이)가 있는 각 주문에 대해 해당 순서를 기반으로 새 `Customer`(을)를 만들려고 있다고 가정합니다. LINQ 쿼리는 이 데이터 소스에서 실행하고 개체 초기화를 사용하여 컬렉션을 채울 수 있습니다.

C#

```
var newLargeOrderCustomers = from o in IncomingOrders
                               where o.OrderSize > 5
                               select new Customer { Name = o.Name, Phone =
o.Phone };
```

데이터 원본에는 `OrderSize` 와(과) 같은 `Customer` 클래스보다 더 많은 속성이 정의되어 있을 수 있지만 개체 초기화를 사용하면 쿼리에서 반환된 데이터가 원하는 데이터 형식으로 성형됩니다. 클래스와 관련된 데이터를 선택합니다. 결과적으로, 이제 원하는 새 `Customer`(으)로 채워진 `System.Collections.Generic.IEnumerable<T>`(이)가 있습니다. 앞의 예제는 LINQ의 메서드 구문으로 작성할 수도 있습니다.

C#

```
var newLargeOrderCustomers = IncomingOrders.Where(x => x.OrderSize >
5).Select(y => new Customer { Name = y.Name, Phone = y.Phone });
```

C# 12부터는 [컬렉션 식](#)을 사용하여 컬렉션을 초기화할 수 있습니다.

자세한 내용은 다음을 참조하세요.

- [개체 이니셜라이저 및 컬렉션 이니셜라이저](#)
- [표준 쿼리 연산자의 쿼리 식 구문](#)

## 익명 형식

컴파일러는 [무명 형식](#)을 생성합니다. 형식 이름은 컴파일러에서만 사용할 수 있습니다. 무명 형식은 별도의 명명된 형식을 정의하지 않고 쿼리 결과에서 일시적으로 속성 집합

을 그룹화하는 편리한 방법을 제공합니다. 무명 형식은 다음과 같이 새로운 식과 개체 이니셜라이저를 사용하여 초기화됩니다.

C#

```
select new {name = cust.Name, phone = cust.Phone};
```

C# 7부터 [튜플](#)을 사용하여 명명되지 않은 형식을 만들 수 있습니다.

## 확장 메서드

확장 메서드는 형식과 연결할 수 있는 정적 메서드이므로 형식의 인스턴스 메서드인 것처럼 호출할 수 있습니다. 이 기능을 사용하면 실제로 수정하지 않고도 기존 형식에 새 메서드를 "추가"할 수 있습니다. 표준 쿼리 연산자는 [IEnumerable<T>](#)을 구현하는 모든 형식에 대해 LINQ 쿼리 기능을 제공하는 확장 메서드 집합입니다.

## 람다 식

람다 식은 => 연산자를 사용하여 입력 매개 변수를 함수 본문과 분리하고 컴파일 시간에 대리자 또는 식 트리로 변환할 수 있는 인라인 함수입니다. LINQ 프로그래밍에서는 표준 쿼리 연산자에 대한 메서드를 직접 호출할 때 람다 식이 나타납니다.

## 데이터로서의 식

쿼리 개체는 구성 가능하므로 메서드에서 쿼리를 반환할 수 있습니다. 쿼리를 나타내는 개체는 결과 컬렉션을 저장하지 않고, 대신 필요할 때 결과를 생성하는 단계를 저장합니다. 메서드에서 쿼리 개체를 반환하는 경우 메서드를 추가로 작성하거나 수정할 수 있다는 이점이 있습니다. 따라서 쿼리를 반환하는 메서드의 반환 값 또는 [out](#) 매개 변수도 해당 형식을 가지고 있어야 합니다. 메서드가 쿼리를 구체적인 [List<T>](#) 또는 [Array](#) 형식으로 구체화하는 경우 쿼리 자체가 아니라 쿼리 결과를 반환합니다. 메서드에서 반환되는 쿼리 변수는 여전히 구성 또는 수정 가능합니다.

다음 예제에서 첫 번째 메서드 [QueryMethod1](#)은 쿼리를 반환 값으로 반환하고 두 번째 메서드 [QueryMethod2](#)는 쿼리를 [out](#) 매개 변수로 반환합니다(예제의 [returnQ](#)). 두 경우 모두 쿼리 결과가 아니라 쿼리가 반환됩니다.

C#

```
IEnumerable<string> QueryMethod1(int[] ints) =>
    from i in ints
    where i > 4
```

```
    select i.ToString();

void QueryMethod2(int[] ints, out IEnumerable<string> returnQ) =>
    returnQ =
        from i in ints
        where i < 4
        select i.ToString();

int[] nums = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

var myQuery1 = QueryMethod1(nums);
```

쿼리 `myQuery1` 은 다음 foreach 루프에서 실행됩니다.

C#

```
foreach (var s in myQuery1)
{
    Console.WriteLine(s);
}
```

마우스 포인터를 `myQuery1`에 놓아 형식을 확인합니다.

`myQuery1` 을 사용하지 않고 `QueryMethod1`에서 반환된 쿼리를 직접 실행할 수도 있습니다.

C#

```
foreach (var s in QueryMethod1(nums))
{
    Console.WriteLine(s);
}
```

마우스 포인터를 `QueryMethod1` 호출 위에 놓아 반환 형식을 확인합니다.

`QueryMethod2` 는 쿼리를 `out` 매개 변수의 값으로 반환합니다.

C#

```
QueryMethod2(nums, out IEnumerable<string> myQuery2);

// Execute the returned query.
foreach (var s in myQuery2)
{
    Console.WriteLine(s);
}
```

쿼리 컴파지션을 사용하여 쿼리를 수정할 수 있습니다. 이 경우 이전 쿼리 개체를 사용하여 새 쿼리 개체를 만듭니다. 이 새 개체는 원래 쿼리 개체와 다른 결과를 반환합니다.

C#

```
myQuery1 =  
    from item in myQuery1  
    orderby item descending  
    select item;  
  
// Execute the modified query.  
Console.WriteLine("\nResults of executing modified myQuery1:");  
foreach (var s in myQuery1)  
{  
    Console.WriteLine(s);  
}
```

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

💡 설명서 문제 열기

↗️ 제품 사용자 의견 제공

# Tutorial: Write queries in C# using language integrated query (LINQ)

Article • 04/25/2024

In this tutorial, you create a data source and write several LINQ queries. You can experiment with the query expressions and see the differences in the results. This walkthrough demonstrates the C# language features that are used to write LINQ query expressions. You can follow along and build the app and experiment with the queries yourself. This article assumes you've installed the latest .NET SDK. If not, go to the [.NET Downloads page](#) and install the latest version on your machine.

First, create the application. From the console, type the following command:

.NET CLI

```
dotnet new console -o WalkthroughWritingLinqQueries
```

Or, if you prefer Visual Studio, create a new console application named *WalkthroughWritingLinqQueries*.

## Create an in-memory data source

The first step is to create a data source for your queries. The data source for the queries is a simple list of `Student` records. Each `Student` record has a first name, family name, and an array of integers that represents their test scores in the class. Add a new file named *students.cs*, and copy the following code into that file:

C#

```
namespace WalkthroughWritingLinqQueries;

public record Student(string First, string Last, int ID, int[] Scores);
```

Note the following characteristics:

- The `Student` record consists of autoimplemented properties.
- Each student in the list is initialized with the primary constructor.
- The sequence of scores for each student is initialized with a primary constructor.

Next, create a sequence of `Student` records that serves as the source of this query. Open *Program.cs*, and remove the following boilerplate code:

C#

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

Replace it with the following code that creates a sequence of `Student` records:

C#

```
using WalkthroughWritingLinqQueries;

// Create a data source by using a collection initializer.
IEnumerable<Student> students =
[
    new Student(First: "Svetlana", Last: "Omelchenko", ID: 111, Scores: [97, 92, 81, 60]),
    new Student(First: "Claire", Last: "O'Donnell", ID: 112, Scores: [75, 84, 91, 39]),
    new Student(First: "Sven", Last: "Mortensen", ID: 113, Scores: [88, 94, 65, 91]),
    new Student(First: "Cesar", Last: "Garcia", ID: 114, Scores: [97, 89, 85, 82]),
    new Student(First: "Debra", Last: "Garcia", ID: 115, Scores: [35, 72, 91, 70]),
    new Student(First: "Fadi", Last: "Fakhouri", ID: 116, Scores: [99, 86, 90, 94]),
    new Student(First: "Hanying", Last: "Feng", ID: 117, Scores: [93, 92, 80, 87]),
    new Student(First: "Hugo", Last: "Garcia", ID: 118, Scores: [92, 90, 83, 78]),

    new Student("Lance", "Tucker", 119, [68, 79, 88, 92]),
    new Student("Terry", "Adams", 120, [99, 82, 81, 79]),
    new Student("Eugene", "Zabokritski", 121, [96, 85, 91, 60]),
    new Student("Michael", "Tucker", 122, [94, 92, 91, 91])
];
```

- The sequence of students is initialized with a collection expression.
- The `Student` record type holds the static list of all students.
- Some of the constructor calls use `named arguments` to clarify which argument matches which constructor parameter.

Try adding a few more students with different test scores to the list of students to get more familiar with the code so far.

## Create the query

Next, you create your first query. Your query, when you execute it, produces a list of all students whose score on the first test was greater than 90. Because the whole `Student` object is selected, the type of the query is `IEnumerable<Student>`. Although the code could also use implicit typing by using the `var` keyword, explicit typing is used to clearly illustrate results. (For more information about `var`, see [Implicitly Typed Local Variables](#).) Add the following code to `Program.cs`, after the code that creates the sequence of students:

C#

```
// Create the query.  
// The first line could also be written as "var studentQuery ="  
IEnumerable<Student> studentQuery =  
    from student in students  
    where student.Scores[0] > 90  
    select student;
```

The query's range variable, `student`, serves as a reference to each `Student` in the source, providing member access for each object.

## Run the query

Now write the `foreach` loop that causes the query to execute. Each element in the returned sequence is accessed through the iteration variable in the `foreach` loop. The type of this variable is `Student`, and the type of the query variable is compatible, `IEnumerable<Student>`. After you added the following code, build and run the application to see the results in the **Console** window.

C#

```
// Execute the query.  
// var could be used here also.  
foreach (Student student in studentQuery)  
{  
    Console.WriteLine($"{student.Last}, {student.First}");  
  
// Output:  
// Omelchenko, Svetlana  
// Garcia, Cesar  
// Fakhouri, Fadi  
// Feng, Hanying  
// Garcia, Hugo  
// Adams, Terry
```

```
// Zabokritski, Eugene  
// Tucker, Michael
```

To further refine the query, you can combine multiple Boolean conditions in the `where` clause. The following code adds a condition so that the query returns those students whose first score was over 90 and whose last score was less than 80. The `where` clause should resemble the following code.

C#

```
where student.Scores[0] > 90 && student.Scores[3] < 80
```

Try the preceding `where` clause, or experiment yourself with other filter conditions. For more information, see [where clause](#).

## Order the query results

It's easier to scan the results if they are in some kind of order. You can order the returned sequence by any accessible field in the source elements. For example, the following `orderby` clause orders the results in alphabetical order from A to Z according to the family name of each student. Add the following `orderby` clause to your query, right after the `where` statement and before the `select` statement:

C#

```
orderby student.Last ascending
```

Now change the `orderby` clause so that it orders the results in reverse order according to the score on the first test, from the highest score to the lowest score.

C#

```
orderby student.Scores[0] descending
```

Change the `WriteLine` format string so that you can see the scores:

C#

```
Console.WriteLine($"{student.Last}, {student.First} {student.Scores[0]}");
```

For more information, see [orderby clause](#).

# Group the results

Grouping is a powerful capability in query expressions. A query with a group clause produces a sequence of groups, and each group itself contains a `key` and a sequence that consists of all the members of that group. The following new query groups the students by using the first letter of their family name as the key.

C#

```
IEnumerable<IGrouping<char, Student>> studentQuery =  
    from student in students  
    group student by student.Last[0];
```

The type of the query changed. It now produces a sequence of groups that have a `char` type as a key, and a sequence of `student` objects. The code in the `foreach` execution loop also must change:

C#

```
foreach (IGrouping<char, Student> studentGroup in studentQuery)  
{  
    Console.WriteLine(studentGroup.Key);  
    foreach (Student student in studentGroup)  
    {  
        Console.WriteLine($"    {student.Last}, {student.First}");  
    }  
}  
// Output:  
// O  
//     Omelchenko, Svetlana  
//     O'Donnell, Claire  
// M  
//     Mortensen, Sven  
// G  
//     Garcia, Cesar  
//     Garcia, Debra  
//     Garcia, Hugo  
// F  
//     Fakhouri, Fadi  
//     Feng, Hanying  
// T  
//     Tucker, Lance  
//     Tucker, Michael  
// A  
//     Adams, Terry  
// Z  
//     Zabokritski, Eugene
```

Run the application and view the results in the **Console** window. For more information, see [group clause](#).

Explicitly coding `IEnumerables` of `IGroupings` can quickly become tedious. Write the same query and `foreach` loop much more conveniently by using `var`. The `var` keyword doesn't change the types of your objects; it just instructs the compiler to infer the types. Change the type of `studentQuery` and the iteration variable `group` to `var` and rerun the query. In the inner `foreach` loop, the iteration variable is still typed as `Student`, and the query works as before. Change the `student` iteration variable to `var` and run the query again. You see that you get exactly the same results.

C#

```
IEnumerable<IGrouping<char, Student>> studentQuery =
    from student in students
    group student by student.Last[0];

foreach (IGrouping<char, Student> studentGroup in studentQuery)
{
    Console.WriteLine(studentGroup.Key);
    foreach (Student student in studentGroup)
    {
        Console.WriteLine($"    {student.Last}, {student.First}");
    }
}
```

For more information about `var`, see [Implicitly Typed Local Variables](#).

## Order the groups by their key value

The groups in the previous query aren't in alphabetical order. You can provide an `orderby` clause after the `group` clause. But to use an `orderby` clause, you first need an identifier that serves as a reference to the groups created by the `group` clause. You provide the identifier by using the `into` keyword, as follows:

C#

```
var studentQuery4 =
    from student in students
    group student by student.Last[0] into studentGroup
    orderby studentGroup.Key
    select studentGroup;

foreach (var groupOfStudents in studentQuery4)
{
    Console.WriteLine(groupOfStudents.Key);
```

```

        foreach (var student in groupOfStudents)
    {
        Console.WriteLine($"    {student.Last}, {student.First}");
    }
}

// Output:
//A
//    Adams, Terry
//F
//    Fakhouri, Fadi
//    Feng, Hanying
//G
//    Garcia, Cesar
//    Garcia, Debra
//    Garcia, Hugo
//M
//    Mortensen, Sven
//O
//    Omelchenko, Svetlana
//    O'Donnell, Claire
//T
//    Tucker, Lance
//    Tucker, Michael
//Z
//    Zabokritski, Eugene

```

Run this query, and the groups are now sorted in alphabetical order.

You can use the `let` keyword to introduce an identifier for any expression result in the query expression. This identifier can be a convenience, as in the following example. It can also enhance performance by storing the results of an expression so that it doesn't have to be calculated multiple times.

C#

```

// This query returns those students whose
// first test score was higher than their
// average score.
var studentQuery5 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where totalScore / 4 < student.Scores[0]
    select $"{student.Last}, {student.First}";

foreach (string s in studentQuery5)
{
    Console.WriteLine(s);
}

// Output:

```

```
// Omelchenko, Svetlana
// O'Donnell, Claire
// Mortensen, Sven
// Garcia, Cesar
// Fakhouri, Fadi
// Feng, Hanying
// Garcia, Hugo
// Adams, Terry
// Zabokritski, Eugene
// Tucker, Michael
```

For more information, see the article on the [let clause](#).

## Use method syntax in a query expression

As described in [Query Syntax and Method Syntax in LINQ](#), some query operations can only be expressed by using method syntax. The following code calculates the total score for each `Student` in the source sequence, and then calls the `Average()` method on the results of that query to calculate the average score of the class.

C#

```
var studentQuery =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    select totalScore;

double averageScore = studentQuery.Average();
Console.WriteLine("Class average score = {0}", averageScore);

// Output:
// Class average score = 334.166666666667
```

## To transform or project in the select clause

It's common for a query to produce a sequence whose elements differ from the elements in the source sequences. Delete or comment out your previous query and execution loop, and replace it with the following code. The query returns a sequence of strings (not `Students`), and this fact is reflected in the `foreach` loop.

C#

```
IEnumerable<string> studentQuery =
    from student in students
    where student.Last == "Garcia"
```

```
select student.First;

Console.WriteLine("The Garcias in the class are:");
foreach (string s in studentQuery)
{
    Console.WriteLine(s);
}

// Output:
// The Garcias in the class are:
// Cesar
// Debra
// Hugo
```

Code earlier in this walkthrough indicated that the average class score is approximately 334. To produce a sequence of `Students` whose total score is greater than the class average, together with their `Student ID`, you can use an anonymous type in the `select` statement:

C#

```
var aboveAverageQuery =
    from student in students
    let x = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where x > averageScore
    select new { id = student.ID, score = x };

foreach (var item in aboveAverageQuery)
{
    Console.WriteLine("Student ID: {0}, Score: {1}", item.id, item.score);
}

// Output:
// Student ID: 113, Score: 338
// Student ID: 114, Score: 353
// Student ID: 116, Score: 369
// Student ID: 117, Score: 352
// Student ID: 118, Score: 343
// Student ID: 120, Score: 341
// Student ID: 122, Score: 368
```

 Collaborate with us on  
GitHub

The source for this content can  
be found on GitHub, where you  
can also create and review



.NET feedback

.NET is an open source project.  
Select a link to provide feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 Open a documentation issue

 Provide product feedback

# Standard Query Operators Overview

Article • 05/31/2024

The *standard query operators* are the keywords and methods that form the LINQ pattern. The C# language defines [LINQ query keywords](#) that you use for the most common query expression. The compiler translates expressions using these keywords to the equivalent method calls. The two forms are synonymous. Other methods that are part of the [System.Linq](#) namespace don't have equivalent query keywords. In those cases, you must use the method syntax. This section covers all the query operator keywords. The runtime and other NuGet packages add more methods designed to work with LINQ queries each release. The most common methods, including those that have query keyword equivalents are covered in this section. For the full list of query methods supported by the .NET Runtime, see the [System.Linq.Enumerable](#) API documentation. In addition to the methods covered here, this class contains methods for concatenating data sources, computing a single value from a data source, such as a sum, average, or other value.

## Important

These samples use an [System.Collections.Generic.IEnumerable<T>](#) data source. Data sources based on [System.Linq.IQueryProvider](#) use [System.Linq.IQueryable<T>](#) data sources and [expression trees](#). Expression trees have [limitations](#) on the allowed C# syntax. Furthermore, each [IQueryProvider](#) data source, such as [EF Core](#) may impose more restrictions. Check the documentation for your data source.

Most of these methods operate on sequences, where a sequence is an object whose type implements the [IEnumerable<T>](#) interface or the [IQueryable<T>](#) interface. The standard query operators provide query capabilities including filtering, projection, aggregation, sorting and more. The methods that make up each set are static members of the [Enumerable](#) and [Queryable](#) classes, respectively. They're defined as [extension methods](#) of the type that they operate on.

The distinction between [IEnumerable<T>](#) and [IQueryable<T>](#) sequences determines how the query is executed at runtime.

For [IEnumerable<T>](#), the returned enumerable object captures the arguments that were passed to the method. When that object is enumerated, the logic of the query operator is employed and the query results are returned.

For `IQueryable<T>`, the query is translated into an [expression tree](#). The expression tree can be translated to a native query when the data source can optimize the query. Libraries such as [Entity Framework](#) translate LINQ queries into native SQL queries that execute at the database.

The following code example demonstrates how the standard query operators can be used to obtain information about a sequence.

C#

```
string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words.
    GroupBy(w => w.Length, w => w.ToUpper()).
    Select(g => new { Length = g.Key, Words = g }).
    OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine("Words of length {0}:", obj.Length);
    foreach (string word in obj.Words)
        Console.WriteLine(word);
}

// This code example produces the following output:
//
// Words of length 3:
// THE
// FOX
// THE
// DOG
// Words of length 4:
// OVER
// LAZY
// Words of length 5:
// QUICK
// BROWN
// JUMPS
```

Where possible, the queries in this section use a sequence of words or numbers as the input source. For queries where more complicated relationships between objects are used, the following sources that model a school are used:

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

Each `Student` has a grade level, a primary department, and a series of scores. A `Teacher` also has a `City` property that identifies the campus where the teacher holds classes. A `Department` has a name, and a reference to a `Teacher` who serves as the department head.

## Types of query operators

The standard query operators differ in the timing of their execution, depending on whether they return a singleton value or a sequence of values. Those methods that return a singleton value (such as `Average` and `Sum`) execute immediately. Methods that return a sequence defer the query execution and return an enumerable object. You can

use the output sequence of one query as the input sequence to another query. Calls to query methods can be chained together in one query, which enables queries to become arbitrarily complex.

## Query operators

In a LINQ query, the first step is to specify the data source. In a LINQ query, the `from` clause comes first in order to introduce the data source (`customers`) and the *range variable* (`cust`).

C#

```
//queryAllStudents is an IEnumerable<Student>
var queryAllStudents = from student in students
                        select student;
```

The range variable is like the iteration variable in a `foreach` loop except that no actual iteration occurs in a query expression. When the query is executed, the range variable serves as a reference to each successive element in `customers`. Because the compiler can infer the type of `cust`, you don't have to specify it explicitly. You can introduce more range variables in a `let` clause. For more information, see [let clause](#).

### ⓘ Note

For non-generic data sources such as [ArrayList](#), the range variable must be explicitly typed. For more information, see [How to query an ArrayList with LINQ \(C#\) and from clause](#).

Once you obtain a data source, you can perform any number of operations on that data source:

- [Filter data](#) using the `where` keyword.
- [Order data](#) using the `orderby` and optionally `descending` keywords.
- [Group data](#) using the `group` and optionally `into` keywords.
- [Join data](#) using the `join` keyword.
- [Project data](#) using the `select` keyword.

## Query Expression Syntax Table

The following table lists the standard query operators that have equivalent query expression clauses.

[Expand table](#)

Method	C# query expression syntax
Cast	Use an explicitly typed range variable:  <code>from int i in numbers</code>  (For more information, see <a href="#">from clause</a> .)
GroupBy	<code>group ... by</code>  -or-  <code>group ... by ... into ...</code>  (For more information, see <a href="#">group clause</a> .)
<code>GroupJoin&lt;TOuter,TInner,TKey,TResult&gt;(IEnumerable&lt;TOuter&gt;, IEnumerable&lt;TInner&gt;, Func&lt;TOuter,TKey&gt;, Func&lt;TInner,TKey&gt;, Func&lt;TOuter,IEnumerable&lt;TInner&gt;, TResult&gt;)</code>	<code>join ... in ... on ... equals ... into ...</code>  (For more information, see <a href="#">join clause</a> .)
<code>Join&lt;TOuter,TInner,TKey,TResult&gt;(IEnumerable&lt;TOuter&gt;, IEnumerable&lt;TInner&gt;, Func&lt;TOuter,TKey&gt;, Func&lt;TInner,TKey&gt;, Func&lt;TOuter,TInner,TResult&gt;)</code>	<code>join ... in ... on ... equals ...</code>  (For more information, see <a href="#">join clause</a> .)
<code>OrderBy&lt;TSource,TKey&gt;(IEnumerable&lt;TSource&gt;, Func&lt;TSource,TKey&gt;)</code>	<code>orderby</code>  (For more information, see <a href="#">orderby clause</a> .)

Method	C# query expression syntax
OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	<code>orderby ...</code> <code>descending</code>  (For more information, see <a href="#">orderby clause</a> .)
Select	<code>select</code>  (For more information, see <a href="#">select clause</a> .)
SelectMany	Multiple <code>from</code> clauses.  (For more information, see <a href="#">from clause</a> .)
ThenBy<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	<code>orderby ..., ...</code>  (For more information, see <a href="#">orderby clause</a> .)
ThenByDescending<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	<code>orderby ..., ...</code> <code>descending</code>  (For more information, see <a href="#">orderby clause</a> .)
Where	<code>where</code>  (For more information, see <a href="#">where clause</a> .)

## Data Transformations with LINQ

Language-Integrated Query (LINQ) isn't only about retrieving data. It's also a powerful tool for transforming data. By using a LINQ query, you can use a source sequence as input and modify it in many ways to create a new output sequence. You can modify the sequence itself without modifying the elements themselves by sorting and grouping.

But perhaps the most powerful feature of LINQ queries is the ability to create new types. The `select` clause creates an output element from an input element. You use it to transform an input element into an output element:

- Merge multiple input sequences into a single output sequence that has a new type.
- Create output sequences whose elements consist of only one or several properties of each element in the source sequence.
- Create output sequences whose elements consist of the results of operations performed on the source data.
- Create output sequences in a different format. For example, you can transform data from SQL rows or text files into XML.

These transformations can be combined in various ways in the same query. Furthermore, the output sequence of one query can be used as the input sequence for a new query.

The following example transforms objects in an in-memory data structure into XML elements.

C#

```
// Create the query.
var studentsToXML = new XElement("Root",
    from student in students
    let scores = string.Join(", ", student.Scores)
    select new XElement("student",
        new XElement("First", student.FirstName),
        new XElement("Last", student.LastName),
        new XElement("Scores", scores)
    ) // end "student"
); // end "Root"

// Execute the query.
Console.WriteLine(studentsToXML);
```

The code produces the following XML output:

XML

```
<Root>
  <student>
    <First>Svetlana</First>
    <Last>Omelchenko</Last>
    <Scores>97,90,73,54</Scores>
  </student>
  <student>
    <First>Claire</First>
    <Last>O'Donnell</Last>
```

```

<Scores>56,78,95,95</Scores>
</student>
...
<student>
<First>Max</First>
<Last>Lindgren</Last>
<Scores>86,88,96,63</Scores>
</student>
<student>
<First>Arina</First>
<Last>Ivanova</Last>
<Scores>93,63,70,80</Scores>
</student>
</Root>

```

For more information, see [Creating XML Trees in C# \(LINQ to XML\)](#).

You can use the results of one query as the data source for a subsequent query. This example shows how to order the results of a join operation. This query creates a group join, and then sorts the groups based on the category element, which is still in scope. Inside the anonymous type initializer, a subquery orders all the matching elements from the products sequence.

C#

```

var orderedQuery = from department in departments
                   join student in students on department.ID equals
                   student.DepartmentID into studentGroup
                   orderby department.Name
                   select new
                   {
                       DepartmentName = department.Name,
                       Students = from student in studentGroup
                                   orderby student.LastName
                                   select student
                   };

foreach (var departmentList in orderedQuery)
{
    Console.WriteLine(departmentList.DepartmentName);
    foreach (var student in departmentList.Students)
    {
        Console.WriteLine($" {student.LastName,-10}
{student.FirstName,-10}");
    }
}
/* Output:
Chemistry
Balzan      Josephine
Fakhouri   Fadi
Popov       Innocenty
Seleznyova Sofiya

```

Vella	Carmen
<b>Economics</b>	
Adams	Terry
Adaobi	Izuchukwu
Berggren	Jeanette
Garcia	Cesar
Ifeoma	Nwanneka
Jamuike	Ifeanacho
Larsson	Naima
Svensson	Noel
Ugomma	Ifunanya
<b>Engineering</b>	
Axelsson	Erik
Berg	Veronika
Engström	Nancy
Hicks	Cassie
Keever	Bruce
Micallef	Nicholas
Mortensen	Sven
Nilsson	Erna
Tucker	Michael
Yermolayeva Anna	
<b>English</b>	
Andersson	Sarah
Feng	Hanying
Ivanova	Arina
Jakobsson	Jesper
Jensen	Christiane
Johansson	Mark
Kolpakova	Nadezhda
Omelchenko	Svetlana
Urquhart	Donald
<b>Mathematics</b>	
Frost	Gaby
Garcia	Hugo
Hedlund	Anna
Kovaleva	Katerina
Lindgren	Max
Maslova	Evgeniya
Olsson	Ruth
Sammut	Maria
Sazonova	Anastasiya
<b>Physics</b>	
Åkesson	Sami
Edwards	Amy E.
Falzon	John
Garcia	Debra
Hansson	Sanna
Mattsson	Martina
Richardson	Don
Zabokritski	Eugene

\*/

The equivalent query using method syntax is shown in the following code:

C#

```
var orderedQuery = departments
    .GroupJoin(students, department => department.ID, student =>
student.DepartmentID,
    (department, studentGroup) => new
    {
        DepartmentName = department.Name,
        Students = studentGroup.OrderBy(student => student.LastName)
    })
    .OrderBy(department => department.DepartmentName);

foreach (var departmentList in orderedQuery)
{
    Console.WriteLine(departmentList.DepartmentName);
    foreach (var student in departmentList.Students)
    {
        Console.WriteLine($" {student.LastName,-10}{student.FirstName,-10}");
    }
}
```

Although you can use an `orderby` clause with one or more of the source sequences before the join, generally we don't recommend it. Some LINQ providers might not preserve that ordering after the join. For more information, see [join clause](#).

## See also

- [Enumerable](#)
- [Queryable](#)
- [select clause](#)
- [Extension Methods](#)
- [Query Keywords \(LINQ\)](#)
- [Anonymous Types](#)

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

### .NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

# LINQ를 사용하여 C#으로 데이터 필터링

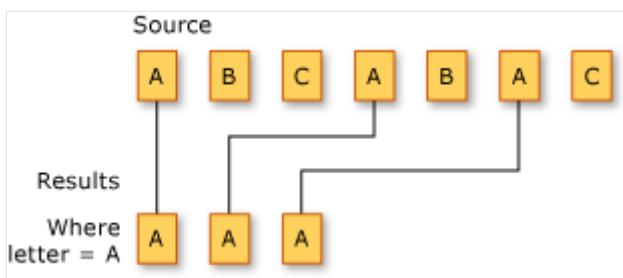
아티클 • 2024. 06. 03.

필터링은 지정된 조건을 충족하는 요소만 포함하도록 결과 집합을 제한하는 작업을 가리킵니다. 지정된 조건과 일치하는 요소를 선택한다고도 합니다.

## ① 중요

이 샘플은 [System.Collections.Generic.IEnumerable<T>](#) 데이터 원본을 사용합니다. [System.Linq.IQueryProvider](#) 기반 데이터 원본은 [System.Linq.IQueryable<T>](#) 데이터 원본과 [식 트리](#)를 사용합니다. 식 트리에는 허용되는 C# 구문에 대한 [제한 사항](#)이 있습니다. 또한 [EF Core](#)와 같은 각 [IQueryProvider](#) 데이터 원본에는 더 많은 제한이 적용될 수 있습니다. 데이터 원본에 대한 설명서를 확인합니다.

다음 그림에서는 문자 시퀀스를 필터링한 결과를 보여 줍니다. 필터링 작업에 대한 조건자는 문자가 'A'가 되도록 지정합니다.



선택을 수행하는 표준 쿼리 연산자 메서드가 다음 표에 나와 있습니다.

### 테이블 확장

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
OfType	지정된 형식으로 캐스트할 수 있는지 여부에 따라 값을 선택합니다.	해당 없음. Enumerable.OfType Queryable.OfType	
Where	조건자 함수를 기반으로 하는 값을 선택합니다.	where Enumerable.Where Queryable.Where	

다음 예제에서는 `where` 절을 사용하여 배열에서 특정 길이의 문자열을 필터링합니다.

C#

```

string[] words = ["the", "quick", "brown", "fox", "jumps"];

IQueryable<string> query = from word in words
                           where word.Length == 3
                           select word;

foreach (string str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:

    the
    fox
*/

```

메서드 구문을 사용하는 동일한 쿼리는 다음 코드에 나와 있습니다.

C#

```

string[] words = ["the", "quick", "brown", "fox", "jumps"];

IQueryable<string> query =
    words.Where(word => word.Length == 3);

foreach (string str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:

    the
    fox
*/

```

## 참고 항목

- [System.Linq](#)
- [where 절](#)
- [리플렉션을 사용하여 어셈블리의 메타데이터를 쿼리하는 방법\(LINQ\)\(C#\)](#)
- [지정된 특성 또는 이름을 갖는 파일을 쿼리하는 방법\(C#\)](#)
- [단어 또는 필드에 따라 텍스트 데이터를 정렬하거나 필터링하는 방법\(LINQ\)\(C#\)](#)

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 프로젝션 작업(C#)

아티클 • 2024. 06. 03.

프로젝션은 종종 나중에 사용되는 속성으로만 구성되는 새 형식으로 개체를 변환하는 작업을 나타냅니다. 프로젝션을 사용하면 각 개체를 기반으로 만들어지는 새 형식을 생성할 수 있습니다. 속성을 프로젝션하고 속성에서 수학 함수를 수행할 수 있습니다. 원래 개체를 변경하지 않고 프로젝션할 수도 있습니다.

## ⓘ 중요

이 샘플은 `System.Collections.Generic.IEnumerable<T>` 데이터 원본을 사용합니다. `System.Linq.IQueryProvider` 기반 데이터 원본은 `System.Linq.IQueryable<T>` 데이터 원본과 식 트리를 사용합니다. 식 트리에는 허용되는 C# 구문에 대한 제한 사항이 있습니다. 또한 EF Core와 같은 각 `IQueryProvider` 데이터 원본에는 더 많은 제한이 적용될 수 있습니다. 데이터 원본에 대한 설명서를 확인합니다.

다음 섹션에는 프로젝션을 수행하는 표준 쿼리 연산자 메서드가 나와 있습니다.

## 메서드

[+] 테이블 확장

메서드 이름	설명	C# 쿼리 식 구문	자세한 정보
선택	변환 함수를 기반으로 하는 값을 프로젝션 합니다.	<code>select</code>	<code>Enumerable.Select</code> <code>Queryable.Select</code>
SelectMany	변환 함수를 기반으로 하는 값의 시퀀스를 프로젝션한 다음 하나의 시퀀스로 평면화 합니다.	여러 <code>from</code> 절 사용	<code>Enumerable.SelectMany</code> <code>Queryable.SelectMany</code>
Zip	지정된 2~3개 시퀀스의 요소를 사용하여 튜플 시퀀스를 생성합니다.	해당 사항 없음	<code>Enumerable.Zip</code> <code>Queryable.Zip</code>

## Select

다음 예제에서는 `select` 절을 사용하여 문자열 목록의 각 문자열에서 첫 글자를 프로젝션합니다.

C#

```
List<string> words = ["an", "apple", "a", "day"];  
  
var query = from word in words  
            select word.Substring(0, 1);  
  
foreach (string s in query)  
{  
    Console.WriteLine(s);  
}  
  
/* This code produces the following output:  
  
    a  
    a  
    a  
    d  
*/
```

메서드 구문을 사용하는 동일한 쿼리는 다음 코드에 나와 있습니다.

```
C#  
  
List<string> words = ["an", "apple", "a", "day"];  
  
var query = words.Select(word => word.Substring(0, 1));  
  
foreach (string s in query)  
{  
    Console.WriteLine(s);  
}  
  
/* This code produces the following output:  
  
    a  
    a  
    a  
    d  
*/
```

## SelectMany

다음 예제에서는 여러 `from` 절을 사용하여 문자열 목록의 각 문자열에서 각 단어를 프로젝션합니다.

```
C#  
  
List<string> phrases = ["an apple a day", "the quick brown fox"];  
  
var query = from phrase in phrases
```

```
        from word in phrase.Split(' ')
        select word;

foreach (string s in query)
{
    Console.WriteLine(s);
}

/* This code produces the following output:

an
apple
a
day
the
quick
brown
fox
*/
```

메서드 구문을 사용하는 동일한 쿼리는 다음 코드에 나와 있습니다.

C#

```
List<string> phrases = ["an apple a day", "the quick brown fox"];

var query = phrases.SelectMany(phrases => phrases.Split(' '));

foreach (string s in query)
{
    Console.WriteLine(s);
}

/* This code produces the following output:

an
apple
a
day
the
quick
brown
fox
*/
```

SelectMany 메서드는 첫 번째 시퀀스의 모든 항목을 두 번째 시퀀스의 모든 항목과 일치시키는 조합을 형성할 수도 있습니다.

C#

```
var query = from number in numbers
            from letter in letters
            select (number, letter);

foreach (var item in query)
{
    Console.WriteLine(item);
}
```

메서드 구문을 사용하는 동일한 쿼리는 다음 코드에 나와 있습니다.

C#

```
var method = numbers
    .SelectMany(number => letters,
    (number, letter) => (number, letter));

foreach (var item in method)
{
    Console.WriteLine(item);
}
```

## Zip

`Zip` 프로젝션 연산자에 대한 오버로드가 여러 개 있습니다. 모든 `Zip` 메서드는 둘 이상의 이종 형식의 시퀀스에 대해 작동합니다. 처음 두 오버로드는 지정된 시퀀스의 해당 위치 형식과 함께 튜플을 반환합니다.

다음과 같은 컬렉션을 생각해 보겠습니다.

C#

```
// An int array with 7 elements.
IEnumerable<int> numbers = [1, 2, 3, 4, 5, 6, 7];
// A char array with 6 elements.
IEnumerable<char> letters = ['A', 'B', 'C', 'D', 'E', 'F'];
```

이러한 시퀀스를 함께 프로젝션하려면 `Enumerable.Zip<TFirst,TSecond>` (`IEnumerable<TFirst>, IEnumerable<TSecond>`) 연산자를 사용합니다.

C#

```
foreach ((int number, char letter) in numbers.Zip(letters))
{
    Console.WriteLine($"Number: {number} zipped with letter: '{letter}'");
}
```

```
// This code produces the following output:  
// Number: 1 zipped with letter: 'A'  
// Number: 2 zipped with letter: 'B'  
// Number: 3 zipped with letter: 'C'  
// Number: 4 zipped with letter: 'D'  
// Number: 5 zipped with letter: 'E'  
// Number: 6 zipped with letter: 'F'
```

## ① 중요

zip 작업의 결과 시퀀스는 길이가 최단 시퀀스보다 길지 않습니다. `numbers` 및 `letters` 컬렉션은 길이가 다르며, 결과 시퀀스에서는 `numbers` 컬렉션의 마지막 요소가 생략됩니다. 압축할 항목이 없기 때문입니다.

두 번째 오버로드는 `third` 시퀀스를 허용합니다. 또 다른 컬렉션 `emoji`를 만들어 보겠습니다.

C#

```
// A string array with 8 elements.  
IEnumerable<string> emoji = [ "👀", "🔥", "🎉", "👀", "⭐", "❤️", "✓",  
"💯" ];
```

이러한 시퀀스를 함께 프로젝션하려면 `Enumerable.Zip<TFirst,TSecond,TThird>` (`IEnumerable<TFirst>`, `IEnumerable<TSecond>`, `IEnumerable<TThird>`) 연산자를 사용합니다.

C#

```
foreach ((int number, char letter, string em) in numbers.Zip(letters,  
emoji))  
{  
    Console.WriteLine(  
        $"Number: {number} is zipped with letter: '{letter}' and emoji:  
{em}");  
}  
  
// This code produces the following output:  
// Number: 1 is zipped with letter: 'A' and emoji: 👀  
// Number: 2 is zipped with letter: 'B' and emoji: 🔥  
// Number: 3 is zipped with letter: 'C' and emoji: 🎉  
// Number: 4 is zipped with letter: 'D' and emoji: 👀  
// Number: 5 is zipped with letter: 'E' and emoji: ⭐  
// Number: 6 is zipped with letter: 'F' and emoji: ❤️
```

이전 오버로드와 마찬가지로 `Zip` 메서드가 튜플을 프로젝션하지만 이번에는 세 개의 요소가 있습니다.

세 번째 오버로드는 결과 선택기 역할을 하는 `Func<TFirst, TSecond, TResult>` 인수를 허용합니다. 압축되는 시퀀스에서 새 결과 시퀀스를 예상할 수 있습니다.

C#

```
foreach (string result in
    numbers.Zip(letters, (number, letter) => $"{number} = {letter}"
    $({(int)letter})) )
{
    Console.WriteLine(result);
}
// This code produces the following output:
//      1 = A (65)
//      2 = B (66)
//      3 = C (67)
//      4 = D (68)
//      5 = E (69)
//      6 = F (70)
```

앞의 `Zip` 오버로드를 사용하여 지정된 함수를 해당 요소 `numbers` 및 `letter`에 적용하여 `string` 결과의 시퀀스를 생성합니다.

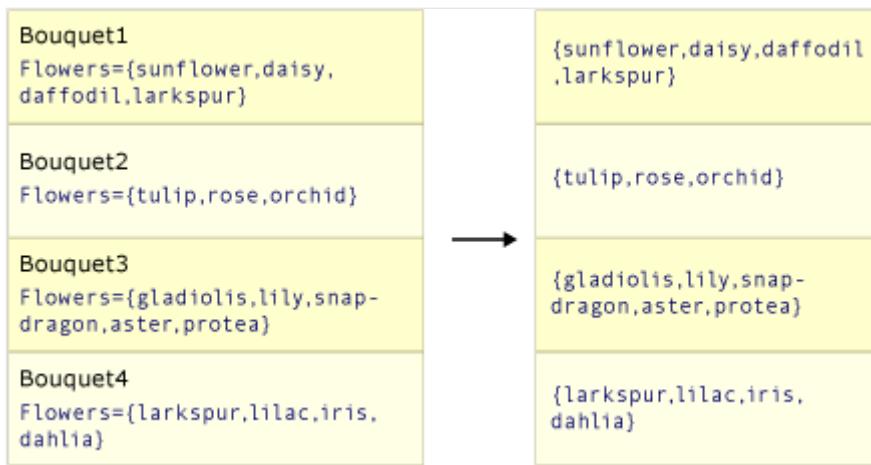
## Select 및 SelectMany

`Select` 및 `SelectMany` 둘 다의 작업은 소스 값에서 결과 값을 생성하는 것입니다. `Select` 는 모든 소스 값에 대해 하나의 결과 값을 생성합니다. 따라서 전체 결과는 소스 컬렉션과 동일한 개수의 요소가 들어 있는 컬렉션입니다. 반면, `SelectMany` 은(는) 각 원본 값에서 연결된 하위 컬렉션을 포함하는 단일 전체 결과를 생성합니다. `SelectMany`에 대한 인수로 전달되는 변환 함수는 각 소스 값에 대해 열거 가능한 값 시퀀스를 반환해야 합니다.

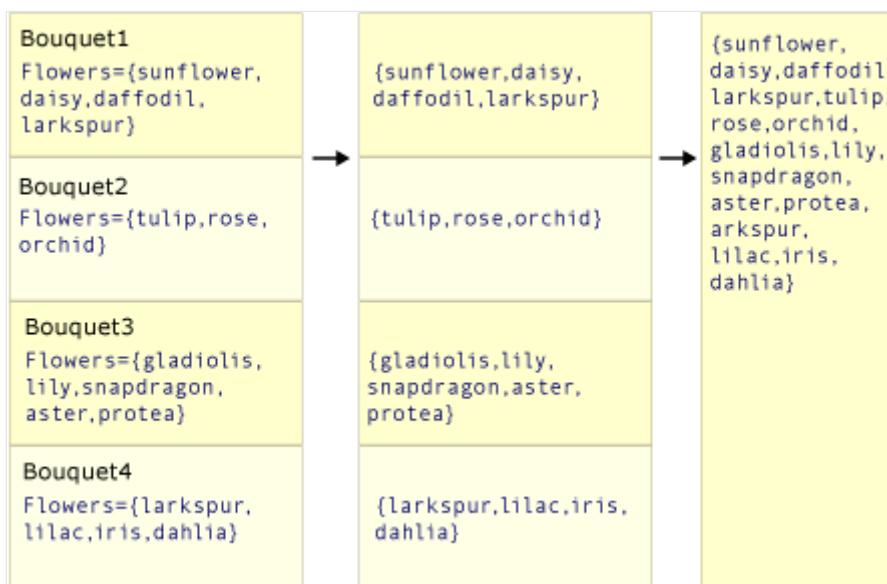
`SelectMany` 은(는) 이러한 열거 가능한 시퀀스를 연결하여 하나의 큰 시퀀스를 만듭니다.

다음 두 그림은 이러한 두 메서드의 작업 간에 개념적 차이를 보여 줍니다. 각각의 경우에 서 선택기(변환) 함수는 각 소스 값에서 꽃의 배열을 선택한다고 가정합니다.

이 그림은 `Select`에서 소스 컬렉션과 동일한 개수의 요소가 들어 있는 컬렉션을 반환하는 방법을 보여 줍니다.



이 그림은 `SelectMany`에서 배열의 중간 시퀀스를 각 중간 배열의 각 값이 포함된 하나의 최종 결과 값으로 연결하는 방법을 보여 줍니다.



## 코드 예제

다음 예제에서는 `Select` 및 `SelectMany`의 동작을 비교합니다. 코드는 소스 컬렉션의 각 꽃 이름 목록에서 해당 항목을 사용하여 꽃 '부케'를 만듭니다. 다음 예제에서는 변환 함수 `Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)`에서 사용하는 "단일 값"은 값의 컬렉션입니다. 이 예제에서는 각 하위 시퀀스의 각 문자열을 열거하기 위해 추가 `foreach` 루프가 필요합니다.

```
C#
```

```

class Bouquet
{
    public required List<string> Flowers { get; init; }
}

static void SelectVsSelectMany()
{
    List<Bouquet> bouquets =

```

```

[

    new Bouquet { Flowers = [ "sunflower", "daisy", "daffodil",
"larkspur" ] },
    new Bouquet { Flowers = [ "tulip", "rose", "orchid" ] },
    new Bouquet { Flowers = [ "gladiolus", "lily", "snapdragon", "aster",
"protea" ] },
    new Bouquet { Flowers = [ "larkspur", "lilac", "iris", "dahlia" ] }

];

IEnumerable<List<string>> query1 = bouquets.Select(bq => bq.Flowers);

IEnumerable<string> query2 = bouquets.SelectMany(bq => bq.Flowers);

Console.WriteLine("Results by using Select():");
// Note the extra foreach loop here.
foreach (IEnumerable<string> collection in query1)
{
    foreach (string item in collection)
    {
        Console.WriteLine(item);
    }
}

Console.WriteLine("\nResults by using SelectMany():");
foreach (string item in query2)
{
    Console.WriteLine(item);
}
}

```

## 참고 항목

- [System.Linq](#)
- [select 절](#)
- [여러 소스로 개체 컬렉션을 채우는 방법\(LINQ\)\(C#\)](#)
- [그룹을 사용하여 파일을 여러 파일로 분할하는 방법\(LINQ\)\(C#\)](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.



### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

#### 설명서 문제 열기

#### 제품 사용자 의견 제공

# 집합 작업(C#)

아티클 • 2024. 06. 03.

LINQ의 집합 작업은 동일하거나 별도의 컬렉션 내에 동등한 요소가 있는지 여부에 따라 결과 집합을 생성하는 쿼리 작업을 나타냅니다.

## ① 중요

이 샘플은 `System.Collections.Generic.IEnumerable<T>` 데이터 원본을 사용합니다. `System.Linq.IQueryProvider` 기반 데이터 원본은 `System.Linq.IQueryable<T>` 데이터 원본과 식 트리를 사용합니다. 식 트리에는 허용되는 C# 구문에 대한 제한 사항이 있습니다. 또한 EF Core와 같은 각 `IQueryProvider` 데이터 원본에는 더 많은 제한이 적용될 수 있습니다. 데이터 원본에 대한 설명서를 확인합니다.

## 데이터 확장

메서드 이름	설명	C# 쿼리 식 구문	자세한 정보
<code>Distinct</code> 또는 <code>DistinctBy</code>	컬렉션에서 중복 값을 제거합니다.	해당 없음.	<code>Enumerable.Distinct</code> <code>Enumerable.DistinctBy</code> <code>Queryable.Distinct</code> <code>Queryable.DistinctBy</code>
<code>Except</code> 또는 <code>ExceptBy</code>	두 번째 컬렉션에 표시되지 않는 한 컬렉션의 요소를 의미하는 차집합을 반환합니다.	해당 없음.	<code>Enumerable.Except</code> <code>Enumerable.ExceptBy</code> <code>Queryable.Except</code> <code>Queryable.ExceptBy</code>
<code>Intersect</code> 또는 <code>IntersectBy</code>	두 컬렉션에 각각 표시되는 요소를 의미하는 교집합을 반환합니다.	해당 없음.	<code>Enumerable.Intersect</code> <code>Enumerable.IntersectBy</code> <code>Queryable.Intersect</code> <code>Queryable.IntersectBy</code>
<code>Union</code> 또는 <code>UnionBy</code>	두 컬렉션 중 하나에 표시되는 고유한 요소를 의미하는 합집합을 반환합니다.	해당 없음.	<code>Enumerable.Union</code> <code>Enumerable.UnionBy</code> <code>Queryable.Union</code> <code>Queryable.UnionBy</code>

## Distinct 및 DistinctBy

다음 예제에서는 문자열 시퀀스에 대한 `Enumerable.Distinct` 메서드의 동작을 보여 줍니다. 반환된 시퀀스에는 입력 시퀀스의 고유한 요소가 포함됩니다.



C#

```
string[] words = ["the", "quick", "brown", "fox", "jumped", "over", "the",
"lazy", "dog"];

IEnumerable<string> query = from word in words.Distinct()
                            select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * the
 * quick
 * brown
 * fox
 * jumped
 * over
 * lazy
 * dog
 */
```

`DistinctBy`는 `Distinct`에 대한 대체 방법으로, `keySelector`를 사용합니다. `keySelector`는 원본 형식의 비교 판별자로 사용됩니다. 다음 코드에서 단어는 `Length`에 따라 구별되고 각 길이의 첫 번째 단어가 표시됩니다.

C#

```
string[] words = ["the", "quick", "brown", "fox", "jumped", "over", "the",
"lazy", "dog"];

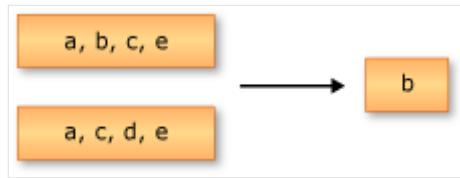
foreach (string word in words.DistinctBy(p => p.Length))
{
    Console.WriteLine(word);

}

// This code produces the following output:
//      the
//      quick
//      jumped
//      over
```

## Except 및 ExceptBy

다음 예제에서는 `Enumerable.Except`의 동작을 보여줍니다. 반환된 시퀀스에는 두 번째 입력 시퀀스에 없는 첫 번째 입력 시퀀스의 요소만 포함됩니다.



이 문서의 다음 예제에서는 이 영역에 대한 공통 데이터 원본을 사용합니다.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

각 `Student`에는 학년 수준, 기본 부서 및 일련의 점수가 있습니다. `Teacher`에는 교사가 수업을 진행하는 캠퍼스를 식별하는 `city` 속성도 있습니다. `Department`에는 이름이 있고 부서장 역할을 하는 `Teacher`에 대한 참조가 있습니다.

C#

```
string[] words1 = ["the", "quick", "brown", "fox"];
string[] words2 = ["jumped", "over", "the", "lazy", "dog"];

IEnumerable<string> query = from word in words1.Except(words2)
                             select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * quick
 * brown
 * fox
 */
```

`ExceptBy` 메서드는 이종 형식의 두 시퀀스와 `keySelector`를 사용하는 `Except`에 대한 대체 접근 방식입니다. `keySelector`은(는) 첫 번째 컬렉션의 형식과 동일한 형식입니다. 제외할 다음 `Teacher` 배열 및 교사 ID를 고려합니다. 두 번째 컬렉션에 없는 교사를 첫 번째 컬렉션에서 찾으려면 교사의 ID를 두 번째 컬렉션에 투영할 수 있습니다.

C#

```
int[] teachersToExclude =
[
    901,    // English
    965,    // Mathematics
    932,    // Engineering
    945,    // Economics
    987,    // Physics
    901     // Chemistry
];

foreach (Teacher teacher in
    teachers.ExceptBy(
        teachersToExclude, teacher => teacher.ID))
{
    Console.WriteLine($"{teacher.First} {teacher.Last}");
}
```

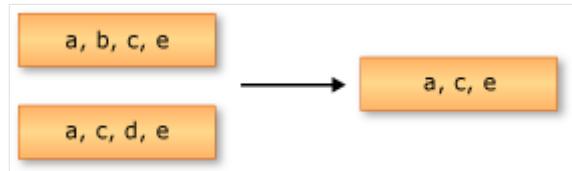
위의 C# 코드에서:

- `teachers` 배열은 `teachersToExclude` 배열에 없는 교사로만 필터링됩니다.
- `teachersToExclude` 배열에는 모든 부서장의 `ID` 값이 포함됩니다.
- `ExceptBy` 을(를) 호출하면 콘솔에 기록되는 새로운 값 집합이 생성됩니다.

새 값 집합은 첫 번째 컬렉션의 형식인 `Teacher` 형식입니다. `teachersToExclude` 배열에 해당 ID 값이 없는 `teachers` 배열의 각 `teacher`이(가) 콘솔에 기록됩니다.

## Intersect 및 IntersectBy

다음 예제에서는 `Enumerable.Intersect`의 동작을 보여줍니다. 반환된 시퀀스에는 입력 시퀀스 둘 다에 공통적으로 있는 요소가 포함됩니다.



C#

```
string[] words1 = ["the", "quick", "brown", "fox"];
string[] words2 = ["jumped", "over", "the", "lazy", "dog"];

IQueryable<string> query = from word in words1.Intersect(words2)
                            select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * the
 */
```

`IntersectBy` 메서드는 이종 형식의 두 시퀀스와 `keySelector`를 사용하는 `Intersect`에 대한 대체 접근 방식입니다. `keySelector`는 두 번째 컬렉션 형식의 비교 판별자로 사용됩니다. 다음 학생 및 교사 배열을 고려합니다. 쿼리는 각 시퀀스의 항목을 이름별로 일치시켜 교사는 아닌 학생을 찾습니다.

C#

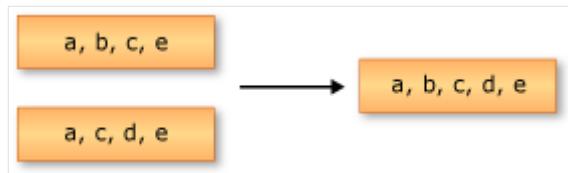
```
foreach (Student person in
        students.IntersectBy(
            teachers.Select(t => (t.First, t.Last)), s => (s.FirstName,
s.LastName)))
{
    Console.WriteLine($"{person.FirstName} {person.LastName}");
}
```

위의 C# 코드에서:

- 쿼리는 이름과 비교하여 `Teacher` 과(와) `Student`의 교차를 생성합니다.
- 두 배열 모두에서 찾은 사람만 결과 시퀀스에 나타납니다.
- 결과 `Student` 인스턴스는 콘솔에 기록됩니다.

## Union 및 UnionBy

다음 예제에서는 두 개의 문자열 시퀀스에 대한 합집합을 보여줍니다. 반환된 시퀀스에는 두 입력 시퀀스의 고유한 요소가 모두 포함됩니다.



C#

```

string[] words1 = ["the", "quick", "brown", "fox"];
string[] words2 = ["jumped", "over", "the", "lazy", "dog"];

IQueryable<string> query = from word in words1.Union(words2)
                            select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * the
 * quick
 * brown
 * fox
 * jumped
 * over
 * the
 * lazy
 * dog
 */
  
```

`UnionBy` 메서드는 동일한 형식의 두 시퀀스와 `keySelector`를 사용하는 `Union`에 대한 대체 접근 방식입니다. `keySelector`는 원본 형식의 비교 판별자로 사용됩니다. 다음 쿼리는 학생 또는 교사인 모든 사람의 목록을 생성합니다. 교사이기도 한 학생은 합집합에 한 번만 추가됩니다.

C#

```
foreach (var person in
    students.Select(s => (s.FirstName, s.LastName)).UnionBy(
        teachers.Select(t => (FirstName: t.First, LastName: t.Last)), s =>
    (s.FirstName, s.LastName)))
{
    Console.WriteLine($"{person.FirstName} {person.LastName}");
}
```

위의 C# 코드에서:

- `teachers` 및 `students` 배열은 해당 이름을 키 선택기로 사용하여 함께 짜여집니다.
- 결과 이름은 콘솔에 기록됩니다.

## 참고 항목

- [System.Linq](#)
- [두 목록 간의 차집합을 구하는 방법\(LINQ\)\(C#\)](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 데이터 정렬(C#)

아티클 • 2024. 06. 03.

정렬 작업은 하나 이상의 특성을 기준으로 시퀀스의 요소를 정렬합니다. 첫 번째 정렬 기준은 요소에 대해 기본 정렬을 수행합니다. 두 번째 정렬 기준을 지정하면 각 기본 정렬 그룹 내의 요소를 정렬할 수 있습니다.

## ① 중요

이 샘플은 `System.Collections.Generic.IEnumerable<T>` 데이터 원본을 사용합니다. `System.Linq.IQueryProvider` 기반 데이터 원본은 `System.Linq.IQueryable<T>` 데이터 원본과 식 트리를 사용합니다. 식 트리에는 허용되는 C# 구문에 대한 제한 사항이 있습니다. 또한 EF Core와 같은 각 `IQueryProvider` 데이터 원본에는 더 많은 제한이 적용될 수 있습니다. 데이터 원본에 대한 설명서를 확인합니다.

다음 그림은 문자 시퀀스에 대한 사전순 정렬 작업의 결과를 보여 줍니다.



다음 섹션에는 데이터를 정렬하는 표준 쿼리 연산자 메서드가 나와 있습니다.

## 메서드

[+] 테이블 확장

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
OrderBy	값을 오름차순으로 정렬합니다.	<code>orderby</code>	<code>Enumerable.OrderBy</code> <code>Queryable.OrderBy</code>
OrderByDescending	값을 내림차순으로 정렬합니다.	<code>orderby ...</code> <code>descending</code>	<code>Enumerable.OrderByDescending</code> <code>Queryable.OrderByDescending</code>
ThenBy	2차 정렬을 오름차순으로 수행합니다.	<code>orderby ..., ...</code>	<code>Enumerable.ThenBy</code> <code>Queryable.ThenBy</code>

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
ThenByDescending	2차 정렬을 내림차순으로 수행합니다.	orderby ..., descending	Enumerable.ThenByDescending Queryable.ThenByDescending
Reverse	컬렉션에서 요소의 순서를 반대로 바꿉니다.	해당 없음.	Enumerable.Reverse Queryable.Reverse

이 문서의 다음 예제에서는 이 영역에 대한 공통 데이터 원본을 사용합니다.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

각 `Student`에는 학년 수준, 기본 부서 및 일련의 점수가 있습니다. `Teacher`에는 교사가 수업을 진행하는 캠퍼스를 식별하는 `City` 속성도 있습니다. `Department`에는 부서장 역할을

하는 Teacher에 대한 참조와 이름이 있습니다.

## 1차 오름차순 정렬

다음 예제에서는 LINQ 쿼리에 orderby 절을 사용하여 교사 배열을 성을 기준으로 오름차순으로 정렬하는 방법을 보여 줍니다.

C#

```
IEnumerable<string> query = from teacher in teachers
                                orderby teacher.Last
                                select teacher.Last;

foreach (string str in query)
{
    Console.WriteLine(str);
}
```

메서드 구문을 사용하여 작성된 이와 동일한 쿼리는 다음 코드에 나와 있습니다.

C#

```
IEnumerable<string> query = teachers
    .OrderBy(teacher => teacher.Last)
    .Select(teacher => teacher.Last);

foreach (string str in query)
{
    Console.WriteLine(str);
}
```

## 1차 내림차순 정렬

다음 예제에서는 LINQ 쿼리에 orderby descending 절을 사용하여 교사를 성의 내림차순으로 정렬하는 방법을 보여 줍니다.

C#

```
IEnumerable<string> query = from teacher in teachers
                                orderby teacher.Last descending
                                select teacher.Last;

foreach (string str in query)
{
```

```
        Console.WriteLine(str);
    }
```

메서드 구문을 사용하여 작성된 이와 동일한 쿼리는 다음 코드에 나와 있습니다.

C#

```
IEnumerable<string> query = teachers
    .OrderByDescending(teacher => teacher.Last)
    .Select(teacher => teacher.Last);

foreach (string str in query)
{
    Console.WriteLine(str);
}
```

## 2차 오름차순 정렬

다음 예제에서는 LINQ 쿼리에 `orderby` 절을 사용하여 1차 및 2차 정렬을 수행하는 방법을 보여 줍니다. 교사는 주로 도시를 기준으로 정렬되고 부차적으로 성으로 정렬되며, 둘 다 오름차순으로 정렬됩니다.

C#

```
IEnumerable<(string, string)> query = from teacher in teachers
                                         orderby teacher.City, teacher.Last
                                         select (teacher.Last, teacher.City);

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

메서드 구문을 사용하여 작성된 이와 동일한 쿼리는 다음 코드에 나와 있습니다.

C#

```
IEnumerable<(string, string)> query = teachers
    .OrderBy(teacher => teacher.City)
    .ThenBy(teacher => teacher.Last)
    .Select(teacher => (teacher.Last, teacher.City));

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

## 2차 내림차순 정렬

다음 예제에서는 LINQ 쿼리에 `orderby descending` 절을 사용하여 1차 정렬을 오름차순으로 수행한 다음 2차 정렬을 내림차순으로 수행하는 방법을 보여 줍니다. 교사는 주로 도시를 기준으로 정렬되고 부차적으로 성으로 정렬됩니다.

C#

```
IEnumerable<(string, string)> query = from teacher in teachers
                                             orderby teacher.City, teacher.Last descending
                                             select (teacher.Last, teacher.City);

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

메서드 구문을 사용하여 작성된 이와 동일한 쿼리는 다음 코드에 나와 있습니다.

C#

```
IEnumerable<(string, string)> query = teachers
    .OrderBy(teacher => teacher.City)
    .ThenByDescending(teacher => teacher.Last)
    .Select(teacher => (teacher.Last, teacher.City));

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

## 참고 항목

- [System.Linq](#)
- `orderby` 절
- [단어 또는 필드에 따라 텍스트 데이터를 정렬하거나 필터링하는 방법\(LINQ\)\(C#\)](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

### 설명서 문제 열기

수도 있습니다. 자세한 내용은  
참여자 가이드를 참조하세요.

☞ 제품 사용자 의견 제공

# LINQ의 수량자 작업(C#)

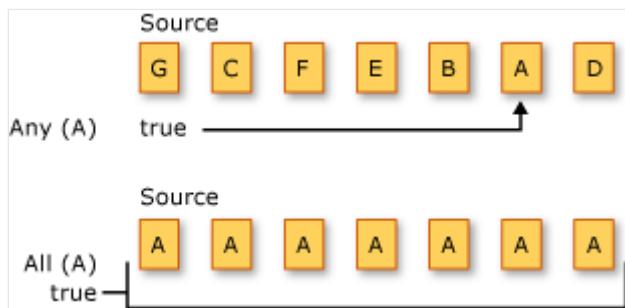
아티클 • 2024. 06. 03.

수량자 작업은 시퀀스에서 조건을 충족하는 요소가 일부인지 전체인지를 나타내는 Boolean 값을 반환합니다.

## ① 중요

이 샘플은 `System.Collections.Generic.IEnumerable<T>` 데이터 원본을 사용합니다. `System.Linq.IQueryProvider` 기반 데이터 원본은 `System.Linq.IQueryable<T>` 데이터 원본과 식 트리를 사용합니다. 식 트리에는 허용되는 C# 구문에 대한 제한 사항이 있습니다. 또한 EF Core와 같은 각 `IQueryProvider` 데이터 원본에는 더 많은 제한이 적용될 수 있습니다. 데이터 원본에 대한 설명서를 확인합니다.

다음 그림은 두 개의 서로 다른 소스 시퀀스에 대한 두 개의 서로 다른 수량자 작업을 보여 줍니다. 첫 번째 작업에서는 요소 중 문자 'A'가 있는지 묻습니다. 두 번째 작업에서는 모든 요소가 문자 'A'인지 묻습니다. 이 예에서는 두 메서드 모두 `true`를 반환합니다.



## [] 테이블 확장

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
모두	시퀀스의 모든 요소가 조건을 만족하는지를 확인합니다.	해당 없음.	<code>Enumerable.All</code> <code>Queryable.All</code>
모두	시퀀스의 임의의 요소가 조건을 만족하는지를 확인합니다.	해당 없음.	<code>Enumerable.Any</code> <code>Queryable.Any</code>
포함	시퀀스에 지정된 요소가 들어 있는지를 확인합니다.	해당 없음.	<code>Enumerable.Contains</code> <code>Queryable.Contains</code>

## 모두

다음 예제에서는 `All`을 사용하여 모든 시험에서 70점 이상의 점수를 받은 학생을 찾습니다.

C#

```
IEnumerable<string> names = from student in students
                                where student.Scores.All(score => score > 70)
                                select $"{student.FirstName} {student.LastName}":
{string.Join(", ", student.Scores.Select(s => s.ToString()))}";

foreach (string name in names)
{
    Console.WriteLine($"{name}");
}

// This code produces the following output:
//
// Cesar Garcia: 71, 86, 77, 97
// Nancy Engström: 75, 73, 78, 83
// Ifunanya Ugomma: 84, 82, 96, 80
```

## 모두

다음 예제에서는 `Any`를 사용하여 어떤 시험에서든 95점보다 높은 점수를 받은 학생을 찾습니다.

C#

```
IEnumerable<string> names = from student in students
                                where student.Scores.Any(score => score > 95)
                                select $"{student.FirstName} {student.LastName}":
{student.Scores.Max()}";

foreach (string name in names)
{
    Console.WriteLine($"{name}");
}

// This code produces the following output:
//
// Svetlana Omelchenko: 97
// Cesar Garcia: 97
// Debra Garcia: 96
// Ifeanacho Jamuike: 98
// Ifunanya Ugomma: 96
// Michelle Caruana: 97
// Nwanneka Ifeoma: 98
// Martina Mattsson: 96
// Anastasiya Sazonova: 96
```

```
// Jesper Jakobsson: 98  
// Max Lindgren: 96
```

## 포함

다음 예제에서는 `Contains`를 사용하여 시험에서 정확한 95점을 받은 학생을 찾습니다.

C#

```
IEnumerable<string> names = from student in students  
                           where student.Scores.Contains(95)  
                           select $"{student.FirstName} {student.LastName}":  
{string.Join(", ", student.Scores.Select(s => s.ToString()))};  
  
foreach (string name in names)  
{  
    Console.WriteLine($"{name}");  
}  
  
// This code produces the following output:  
//  
// Claire O'Donnell: 56, 78, 95, 95  
// Donald Urquhart: 92, 90, 95, 57
```

## 참고 항목

- [System.Linq](#)
- [런타임에 동적으로 조건자 필터 지정](#)
- [지정된 단어 집합이 들어 있는 문장을 쿼리하는 방법\(LINQ\)\(C#\)](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 데이터 분할(C#)

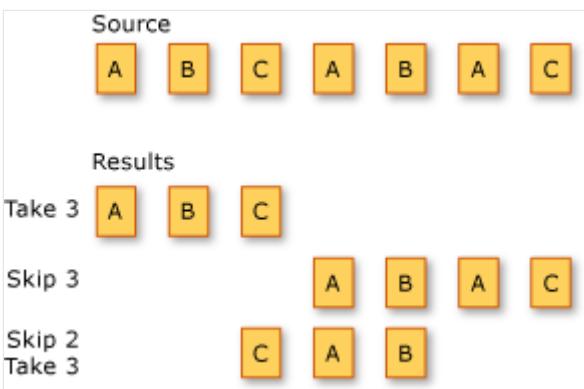
아티클 • 2024. 06. 03.

LINQ의 분할은 요소를 다시 정렬한 후 섹션 중 하나를 반환하지 않고 입력 시퀀스를 두 개의 섹션으로 나누는 작업을 가리킵니다.

## ① 중요

이 샘플은 `System.Collections.Generic.IEnumerable<T>` 데이터 원본을 사용합니다. `System.Linq.IQueryProvider` 기반 데이터 원본은 `System.Linq.IQueryable<T>` 데이터 원본과 식 트리를 사용합니다. 식 트리에는 허용되는 C# 구문에 대한 제한 사항이 있습니다. 또한 EF Core와 같은 각 `IQueryProvider` 데이터 원본에는 더 많은 제한이 적용될 수 있습니다. 데이터 원본에 대한 설명서를 확인합니다.

다음 그림은 문자 시퀀스에 대한 세 가지 분할 작업의 결과를 보여 줍니다. 첫 번째 작업은 시퀀스에서 처음 세 개의 요소를 반환합니다. 두 번째 작업은 처음 세 개의 요소를 건너뛰고 나머지 요소를 반환합니다. 세 번째 작업은 시퀀스에서 처음 두 개의 요소를 건너뛰고 다음 세 개의 요소를 반환합니다.



시퀀스를 분할하는 표준 쿼리 연산자 메서드가 다음 섹션에 나와 있습니다.

## 연산자

[+] 테이블 확장

메서드 이름	설명	C# 쿼리 식 구문	자세한 정보
Skip	시퀀스에서 지정한 위치까지 요소를 건너뜁니다.	해당 없음.	<code>Enumerable.Skip</code> <code>Queryable.Skip</code>

메서드 이름	설명	C# 쿼리식 구문	자세한 정보
SkipWhile	요소가 조건을 충족하지 않을 때까지 조건자 함수를 기반으로 하여 요소를 건너뜁니다.	해당 없음.	<a href="#">Enumerable.SkipWhile</a> <a href="#">Queryable.SkipWhile</a>
Take	시퀀스에서 지정된 위치까지 요소를 사용합니다.	해당 없음.	<a href="#">Enumerable.Take</a> <a href="#">Queryable.Take</a>
TakeWhile	요소가 조건을 충족하지 않을 때까지 조건자 함수를 기반으로 하여 요소를 사용합니다.	해당 없음.	<a href="#">Enumerable.TakeWhile</a> <a href="#">Queryable.TakeWhile</a>
Chunk	시퀀스의 구성 요소를 지정된 최대 크기의 청크로 분할합니다.	해당 없음.	<a href="#">Enumerable.Chunk</a> <a href="#">Queryable.Chunk</a>

다음 예제는 [Enumerable.Range\(Int32, Int32\)](#)를 사용하여 모두 0에서 7까지의 숫자 시퀀스를 생성합니다.

시퀀스의 첫 번째 요소만 사용하려면 `Take` 메서드를 사용합니다.

C#

```
foreach (int number in Enumerable.Range(0, 8).Take(3))
{
    Console.WriteLine(number);
}
// This code produces the following output:
// 0
// 1
// 2
```

시퀀스의 첫 번째 요소를 건너뛰고 다시 기본 요소를 사용하려면 `Skip` 메서드를 사용합니다.

C#

```
foreach (int number in Enumerable.Range(0, 8).Skip(3))
{
    Console.WriteLine(number);
}
// This code produces the following output:
// 3
// 4
// 5
// 6
// 7
```

또한 `TakeWhile` 및 `SkipWhile` 메서드는 시퀀스의 요소를 사용하고 건너뜁니다. 그러나 이러한 메서드는 설정된 수의 요소 대신 조건에 따라 요소를 건너뛰거나 사용합니다. `TakeWhile`은 요소가 조건과 일치하지 않을 때까지 시퀀스의 요소를 사용합니다.

C#

```
foreach (int number in Enumerable.Range(0, 8).TakeWhile(n => n < 5))
{
    Console.WriteLine(number);
}
// This code produces the following output:
// 0
// 1
// 2
// 3
// 4
```

`SkipWhile`은 조건이 true이면 첫 번째 요소를 건너뜁니다. 조건과 일치하지 않는 첫 번째 요소와 모든 후속 요소가 반환됩니다.

C#

```
foreach (int number in Enumerable.Range(0, 8).SkipWhile(n => n < 5))
{
    Console.WriteLine(number);
}
// This code produces the following output:
// 5
// 6
// 7
```

`Chunk` 연산자는 지정된 `size`에 따라 시퀀스의 요소를 분할하는 데 사용됩니다.

C#

```
int chunkNumber = 1;
foreach (int[] chunk in Enumerable.Range(0, 8).Chunk(3))
{
    Console.WriteLine($"Chunk {chunkNumber++}:");
    foreach (int item in chunk)
    {
        Console.WriteLine($"    {item}");
    }

    Console.WriteLine();
}
// This code produces the following output:
// Chunk 1:
//     0
```

```
//    1
//    2
//
//Chunk 2:
//    3
//    4
//    5
//
//Chunk 3:
//    6
//    7
```

위의 C# 코드에서:

- `Enumerable.Range(Int32, Int32)`를 사용하여 숫자 시퀀스를 생성합니다.
- `Chunk` 연산자를 적용하여 시퀀스를 최대 크기가 3인 청크로 분할합니다.

## 참고 항목

- [System.Linq](#)



### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.



### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

[설명서 문제 열기](#)

[제품 사용자 의견 제공](#)

# 데이터 형식 변환(C#)

아티클 • 2024. 06. 04.

변환 메서드는 입력 개체의 형식을 변경합니다.

## ① 중요

이 샘플은 `System.Collections.Generic.IEnumerable<T>` 데이터 원본을 사용합니다. `System.Linq.IQueryProvider` 기반 데이터 원본은 `System.Linq.IQueryable<T>` 데이터 원본과 식 트리를 사용합니다. 식 트리에는 허용되는 C# 구문에 대한 제한 사항이 있습니다. 또한 EF Core와 같은 각 `IQueryProvider` 데이터 원본에는 더 많은 제한이 적용될 수 있습니다. 데이터 원본에 대한 설명서를 확인합니다.

LINQ 쿼리의 변환 작업은 다양한 애플리케이션에서 유용합니다. 다음은 몇 가지 예제입니다.

- `Enumerable.AsEnumerable` 메서드는 표준 쿼리 연산자의 형식 사용자 지정 구현을 숨기는 데 사용될 수 있습니다.
- `Enumerable.OfType` 메서드는 LINQ 쿼리에 대해 매개 변수가 없는 컬렉션을 사용하도록 설정하는 데 사용될 수 있습니다.
- `Enumerable.ToArray`, `Enumerable.ToDictionary`, `Enumerable.ToList`, `Enumerable.ToLookup` 메서드는 쿼리가 열거될 때까지 연기하는 대신 강제로 쿼리를 즉시 실행하는 데 사용될 수 있습니다.

## 메서드

다음 표에는 데이터-형식 변환을 수행하는 표준 쿼리 연산자 메서드가 나와 있습니다.

이 표에서 이름이 "As"로 시작하는 변환 메서드는 소스 컬렉션의 정적 형식을 변경하지만 열거하지는 않습니다. 이름이 "To"로 시작하는 메서드는 소스 컬렉션을 열거하고 항목을 해당하는 컬렉션 형식에 삽입합니다.

### 테이블 확장

메서드 이름	설명	C# 쿼리 식	추가 정보
		구문	
AsEnumerable	<code>IEnumerable&lt;T&gt;</code> 로 형식화된 입력을 반환합니다.	해당 없음.	<code>Enumerable.AsEnumerable</code>
AsQueryable	(제네릭) <code>IEnumerable</code> 을 (제네릭)	해당 없음.	<code>Queryable.AsQueryable</code>

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
	<b>IQueryable</b> 로 변환합니다.		
캐스트	컬렉션의 요소를 지정된 형식으로 캐스트합니다.	명시적 형식 범위 변수를 사용합니다. 예시:  <code>from string str in words</code>	<a href="#">Enumerable.Cast</a> <a href="#">Queryable.Cast</a>
OfType	지정된 형식으로 캐스트할 수 있는지 여부에 따라 값을 필터링합니다.	해당 없음.	<a href="#">Enumerable.OfType</a> <a href="#">Queryable.OfType</a>
ToArray	컬렉션을 배열로 변환합니다. 이 메서드는 쿼리를 강제로 실행합니다.	해당 없음.	<a href="#">Enumerable.ToArray</a>
ToDictionary	키 선택기 함수에 따라 <a href="#">Dictionary&lt; TKey, TValue &gt;</a> 에 요소를 배치합니다. 이 메서드는 쿼리를 강제로 실행합니다.	해당 없음.	<a href="#">Enumerable.ToDictionary</a>
ToList	컬렉션을 <a href="#">List&lt; T &gt;</a> 로 변환합니다. 이 메서드는 쿼리를 강제로 실행합니다.	해당 없음.	<a href="#">Enumerable.ToList</a>
ToLookup	키 선택기 함수에 따라 <a href="#">Lookup&lt; TKey, TElement &gt;</a> (일 대 다 사전)에 요소를 배치합니다. 이 메서드는 쿼리를 강제로 실행합니다.	해당 없음.	<a href="#">Enumerable.ToLookup</a>

이 문서의 다음 예제에서는 이 영역에 대한 공통 데이터 원본을 사용합니다.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
```

```

public required int ID { get; init; }

public required GradeLevel Year { get; init; }
public required List<int> Scores { get; init; }

public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}

```

각 `Student`에는 학년 수준, 기본 부서 및 일련의 점수가 있습니다. `Teacher`에는 교사가 수업을 진행하는 캠퍼스를 식별하는 `city` 속성도 있습니다. `Department`에는 부서장 역할을 하는 `Teacher`에 대한 참조와 이름이 있습니다.

## 쿼리식 구문 예제

다음 코드 예제에서는 명시적 형식 범위 변수를 사용하여 하위 형식에서만 사용할 수 있는 멤버에 액세스하기 전에 형식을 하위 형식으로 캐스트합니다.

C#

```

IQueryable people = students;

var query = from Student student in students
            where student.Year == GradeLevel.ThirdYear
            select student;

foreach (Student student in query)
{
    Console.WriteLine(student.FirstName);
}

```

이와 동일한 쿼리는 다음 예제와 같이 메서드 구문을 사용하여 표현할 수 있습니다.

C#

```
IEnumerable people = students;

var query = people
    .Cast<Student>()
    .Where(student => student.Year == GradeLevel.ThirdYear);

foreach (Student student in query)
{
    Console.WriteLine(student.FirstName);
}
```

## 참고 항목

- [System.Linq](#)
- [from 절](#)

### ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

☞ 설명서 문제 열기

☞ 제품 사용자 의견 제공

# Join LINQ의 작업

아티클 • 2024. 05. 29.

두 데이터 소스를 조인하는 것은 한 데이터 소스의 개체를 공통 특성을 공유하는 다른 데이터 소스의 개체와 연결하는 것입니다.

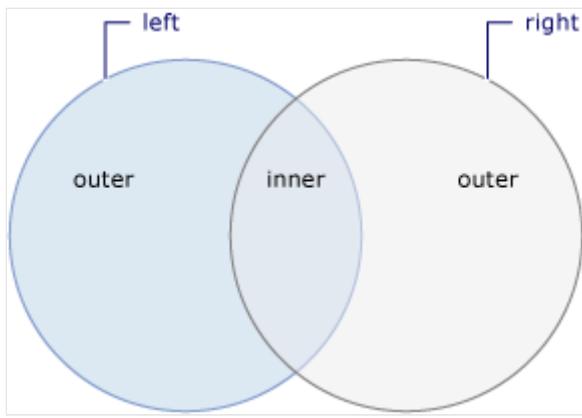
## ① 중요

이 샘플은 `System.Collections.Generic.IEnumerable<T>` 데이터 원본을 사용합니다. `System.Linq.IQueryProvider` 기반 데이터 원본은 `System.Linq.IQueryable<T>` 데이터 원본과 식 트리를 사용합니다. 식 트리에는 허용되는 C# 구문에 대한 제한 사항이 있습니다. 또한 EF Core와 같은 각 `IQueryProvider` 데이터 원본에는 더 많은 제한이 적용될 수 있습니다. 데이터 원본에 대한 설명서를 확인합니다.

서로 간의 관계를 직접 적용할 수 없는 데이터 원본을 대상으로 하는 쿼리에서는 `Join`이 중요한 작업입니다. 개체 지향 프로그래밍에서 조인한다는 것은 모델링되지 않은 개체 간에 상관 관계가 있음을 의미할 수 있습니다(예: 단방향 관계에서 반대 방향을 사용). 단방향 관계의 예로는 `student` 클래스가 주를 나타내는 `Department` 형식 속성을 포함하는데 `Department` 클래스는 `Student` 개체의 컬렉션인 속성을 포함하지 않는 경우를 들 수 있습니다. `Department` 개체 목록이 있는 경우 각 부서의 모든 학생을 찾으려면 조인 작업을 사용하면 됩니다.

LINQ 프레임워크에 제공되는 조인 메서드는 `Join` 및 `GroupJoin`입니다. 이러한 메서드는 키가 같은지 여부에 따라 두 데이터 소스의 일치 여부를 확인하는 조인인 동등 조인을 수행합니다. 비교를 위해 Transact-SQL에서는 `equals` 이 아닌 `less than` 연산자와 같은 조인 연산자를 지원합니다. 관계형 데이터베이스 용어에서 `Join`은 내부 조인을 구현합니다. 내부 조인은 다른 데이터 집합에 일치하는 항목이 있는 개체만 반환하는 유형의 조인입니다. `GroupJoin` 메서드에는 관계형 데이터베이스 측면에 직접 상응하는 기능이 없지만 내부 조인 및 왼쪽 우선 외부 조인의 상위 집합을 구현합니다. 왼쪽 우선 외부 조인은 다른 데이터 소스에 서로 관련된 요소가 없더라도 첫 번째(왼쪽) 데이터 소스의 각 요소를 반환하는 조인입니다.

다음 그림에서는 내부 조인 또는 왼쪽 우선 외부 조인에 포함된 두 집합 및 해당 집합 내의 요소를 개념적으로 보여줍니다.



## 메서드

테이블 확장

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
Join	키 선택기 함수를 기준으로 두 시퀀스를 Join한 다음 값 쌍을 추출합니다.	<code>join ... in ... on ... equals ...</code>	<code>Enumerable.Join</code> <code>Queryable.Join</code>
GroupJoin	키 선택기 함수를 기준으로 두 시퀀스를 Join한 다음 결과로 생성된 일치 항목을 요소마다 그룹화합니다.	<code>join ... in ... on ... equals ... into ...</code>	<code>Enumerable.GroupJoin</code> <code>Queryable.GroupJoin</code>

이 문서의 다음 예제에서는 이 영역에 대한 공통 데이터 원본을 사용합니다.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}
```

```

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}
public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}

```

각 `Student`에는 학년 수준, 기본 부서 및 일련의 점수가 있습니다. `Teacher`에는 교사가 수업을 진행하는 캠퍼스를 식별하는 `city` 속성도 있습니다. `Department`에는 부서장 역할을 하는 `Teacher`에 대한 참조와 이름이 있습니다.

다음 예제에서는 `join ... in ... on ... equals ...` 절을 사용하여 특정 값을 기준으로 두 시퀀스를 조인합니다.

C#

```

var query = from student in students
            join department in departments on student.DepartmentID equals
department.ID
            select new { Name = $"{student.FirstName} {student.LastName}",
DepartmentName = department.Name };

foreach (var item in query)
{
    Console.WriteLine($"{item.Name} - {item.DepartmentName}");
}

```

위의 쿼리는 다음 코드와 같이 메서드 구문을 사용하여 표현할 수 있습니다.

C#

```

var query = students.Join(departments,
    student => student.DepartmentID, department => department.ID,
    (student, department) => new { Name = $"{student.FirstName}
{student.LastName}", DepartmentName = department.Name });

foreach (var item in query)
{
    Console.WriteLine($"{item.Name} - {item.DepartmentName}");
}

```

다음 예제에서는 `join ... in ... on ... equals ... into ...` 절을 사용하여 특정 값을 기준으로 두 시퀀스를 조인하고 각 요소에 대해 결과 일치 항목을 그룹화합니다.

C#

```
IEnumerable<IEnumerable<Student>> studentGroups = from department in departments
    join student in students on department.ID equals
    student.DepartmentID into studentGroup
    select studentGroup;

foreach (IEnumerable<Student> studentGroup in studentGroups)
{
    Console.WriteLine("Group");
    foreach (Student student in studentGroup)
    {
        Console.WriteLine($" - {student.FirstName}, {student.LastName}");
    }
}
```

위의 쿼리는 다음 예제와 같이 메서드 구문을 사용하여 표현할 수 있습니다.

C#

```
// Join department and student based on DepartmentId and grouping result
IEnumerable<IEnumerable<Student>> studentGroups =
departments.GroupJoin(students,
    department => department.ID, student => student.DepartmentID,
    (department, studentGroup) => studentGroup);

foreach (IEnumerable<Student> studentGroup in studentGroups)
{
    Console.WriteLine("Group");
    foreach (Student student in studentGroup)
    {
        Console.WriteLine($" - {student.FirstName}, {student.LastName}");
    }
}
```

## 내부 조인 수행

관계형 데이터베이스 용어에서 내부 조인은 첫 번째 컬렉션의 각 요소가 두 번째 컬렉션에서 일치하는 모든 요소에 대해 한 번 표시되는 결과 집합을 생성합니다. 첫 번째 컬렉션의 요소에 일치하는 요소가 없는 경우에는 결과 집합에 표시되지 않습니다. C#에서 `join` 절에 의해 호출되는 `Join` 메서드는 내부 조인을 구현합니다. 다음 예제에서는 내부 조인의 네 가지 변형을 수행하는 방법을 보여 줍니다.

- 단순 키에 따라 두 데이터 소스의 요소를 상호 연결하는 간단한 내부 조인

- 복합 키에 따라 두 데이터 소스의 요소를 상호 연결하는 내부 조인. 둘 이상의 값으로 구성된 키인 복합 키를 사용하면 둘 이상의 속성에 따라 요소를 상호 연결할 수 있습니다.
- 연속 조인 작업이 서로 추가되는 여러 조인
- 그룹 조인을 사용하여 구현되는 내부 조인

## 단일 키 조인

다음 예제에서는 해당 `TeacherId`와 `Teacher`가 일치하는 `Deparment`를 `Teacher`와 일치시킵니다. C#의 `select` 절은 결과 개체의 모양을 정의합니다. 다음 예제에서 결과 개체는 부서 이름과 부서를 이끄는 교사의 이름으로 구성된 익명 형식입니다.

C#

```
var query = from department in departments
            join teacher in teachers on department.TeacherID equals
teacher.ID
            select new
{
    DepartmentName = department.Name,
    TeacherName = $"{teacher.First} {teacher.Last}"
};

foreach (var departmentAndTeacher in query)
{
    Console.WriteLine($"{departmentAndTeacher.DepartmentName} is managed by
{departmentAndTeacher.TeacherName}");
}
```

Join 메서드 구문을 사용하여 동일한 결과를 얻을 수 있습니다.

C#

```
var query = teachers
    .Join(departments, teacher => teacher.ID, department =>
department.TeacherID,
    (teacher, department) =>
    new { DepartmentName = department.Name, TeacherName = $""
{teacher.First} {teacher.Last}" });

foreach (var departmentAndTeacher in query)
{
    Console.WriteLine($"{departmentAndTeacher.DepartmentName} is managed by
{departmentAndTeacher.TeacherName}");
}
```

부서장이 아닌 교사는 최종 결과에 나타나지 않습니다.

## 복합 키 조인

속성 하나만 기준으로 요소를 상호 연결하는 대신 복합 키를 사용하여 여러 속성을 기준으로 요소를 비교할 수 있습니다. 비교하려는 속성으로 구성된 무명 형식을 반환할 각 컬렉션에 대한 키 선택기 함수를 지정합니다. 속성에 레이블을 지정하는 경우 각 키의 무명 형식에 동일한 레이블이 있어야 합니다. 또한 속성은 동일한 순서로 나타나야 합니다.

다음 예제에서는 `Teacher` 개체 목록과 `Student` 개체 목록을 사용하여 학생이기도 한 교사를 확인합니다. 이러한 두 형식에는 각 사람의 이름과 성을 나타내는 속성이 있습니다. 각 목록의 요소에서 조인 키를 만드는 함수는 속성으로 구성된 무명 형식을 반환합니다. 조인 작업은 이러한 복합 키가 같은지 비교하고 각 목록에서 이름과 성이 둘 다 일치하는 개체 쌍을 반환합니다.

C#

```
// Join the two data sources based on a composite key consisting of first
and last name,
// to determine which employees are also students.
IEnumerable<string> query =
    from teacher in teachers
    join student in students on new
    {
        FirstName = teacher.First,
        LastName = teacher.Last
    } equals new
    {
        student.FirstName,
        student.LastName
    }
    select teacher.First + " " + teacher.Last;

string result = "The following people are both teachers and students:\r\n";
foreach (string name in query)
{
    result += $"{name}\r\n";
}
Console.WriteLine(result);
```

다음 예제와 같이 `Join` 메서드를 사용할 수 있습니다.

C#

```
IEnumerable<string> query = teachers
    .Join(students,
        teacher => new { FirstName = teacher.First, LastName = teacher.Last
    },
        student => new { student.FirstName, student.LastName },
        (teacher, student) => $"{teacher.First} {teacher.Last}"
    );
```

```
Console.WriteLine("The following people are both teachers and students:");
foreach (string name in query)
{
    Console.WriteLine(name);
}
```

## 여러 조인

개수에 제한없이 조인 작업을 서로 추가하여 여러 조인을 수행할 수 있습니다. C#의 각 `join` 절은 지정된 데이터 소스를 이전 조인의 결과와 상호 연결합니다.

첫 번째 `join` 절은 개체와 `Department` 개체의 `ID`와 일치하는 `Student` 개체의 `DepartmentID`에 따라 학생과 부서를 일치시킵니다. `Student` 개체 및 `Department` 개체를 포함하는 무명 형식의 시퀀스를 반환합니다.

두 번째 `join` 절은 부서장 ID와 일치하는 해당 교사의 ID를 토대로 첫 번째 조인에서 반환된 익명 형식과 `Teacher` 개체 간의 상관 관계를 지정합니다. 학생 이름, 부서 이름 및 부서장 이름을 포함하는 익명 형식의 시퀀스를 반환합니다. 이 작업은 내부 조인이기 때문에 두 번째 데이터 원본에 일치 항목이 있는 첫 번째 데이터 원본의 개체만 반환됩니다.

C#

```
// The first join matches Department.ID and Student.DepartmentID from the
list of students and
// departments, based on a common ID. The second join matches teachers who
lead departments
// with the students studying in that department.
var query = from student in students
            join department in departments on student.DepartmentID equals
department.ID
            join teacher in teachers on department.TeacherID equals teacher.ID
            select new {
                StudentName = $"{student.FirstName} {student.LastName}",
                DepartmentName = department.Name,
                TeacherName = $"{teacher.First} {teacher.Last}"
            };
foreach (var obj in query)
{
    Console.WriteLine($"The student \"{obj.StudentName}\" studies in the
department run by \"{obj.TeacherName}\".");
}
```

여러 [Join](#) 메서드를 사용하는 동등한 방법은 익명 형식과 동일한 방법을 사용합니다.

C#

```

var query = students
    .Join(departments, student => student.DepartmentID, department =>
department.ID,
        (student, department) => new { student, department })
    .Join(teachers, commonDepartment =>
commonDepartment.department.TeacherID, teacher => teacher.ID,
        (commonDepartment, teacher) => new
        {
            StudentName = $"{commonDepartment.student.FirstName}"
{commonDepartment.student.LastName}",
            DepartmentName = commonDepartment.department.Name,
            TeacherName = $"{teacher.First} {teacher.Last}"
        });
}

foreach (var obj in query)
{
    Console.WriteLine($"The student \"{obj.StudentName}\" studies in the
department run by \"{obj.TeacherName}\".");
}

```

## 그룹화된 조인을 사용하는 내부 조인

다음 예제에서는 그룹 조인을 사용하여 내부 조인을 구현하는 방법을 보여 줍니다.

`Department` 개체 목록은 `Student.DepartmentID` 속성과 일치하는 `Department.ID`를 기준으로 `Student` 개체 목록에 그룹 조인됩니다. 그룹 조인은 각 그룹이 `Department` 개체 및 일치하는 `student` 개체 시퀀스로 구성된 중간 그룹 컬렉션을 만듭니다. 두 번째 `from` 절은 이 시퀀스를 하나의 긴 시퀀스로 결합하거나 평면화합니다. `select` 절은 최종 시퀀스의 요소 형식을 지정합니다. 이 형식은 학생 이름과 일치하는 부서 이름으로 구성된 익명 형식입니다.

C#

```

var query1 =
    from department in departments
    join student in students on department.ID equals student.DepartmentID
    into gj
    from subStudent in gj
    select new
    {
        DepartmentName = department.Name,
        StudentName = $"{subStudent.FirstName} {subStudent.LastName}"
    };
Console.WriteLine("Inner join using GroupJoin():");
foreach (var v in query1)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}

```

다음과 같이 GroupJoin 메서드를 사용하여 동일한 결과를 달성할 수 있습니다.

C#

```
var queryMethod1 = departments
    .GroupJoin(students, department => department.ID, student =>
student.DepartmentID,
    (department, gj) => new { department, gj })
    .SelectMany(departmentAndStudent => departmentAndStudent.gj,
    (departmentAndStudent, subStudent) => new
    {
        DepartmentName = departmentAndStudent.department.Name,
        StudentName = $"{subStudent.FirstName} {subStudent.LastName}"
    });
Console.WriteLine("Inner join using GroupJoin():");
foreach (var v in queryMethod1)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}
```

결과는 `into` 절 없이 `join` 절을 사용해 내부 조인을 수행하여 얻은 결과 집합과 같습니다. 다음 코드는 이와 동등한 쿼리를 보여 줍니다.

C#

```
var query2 = from department in departments
    join student in students on department.ID equals student.DepartmentID
    select new
    {
        DepartmentName = department.Name,
        StudentName = $"{student.FirstName} {student.LastName}"
    };
Console.WriteLine("The equivalent operation using Join():");
foreach (var v in query2)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}
```

체인을 방지하기 위해 다음과 같이 단일 `Join` 메서드를 사용할 수 있습니다.

C#

```
var queryMethod2 = departments.Join(students, departments => departments.ID,
student => student.DepartmentID,
    (department, student) => new
    {
        DepartmentName = department.Name,
        StudentName = $"{student.FirstName} {student.LastName}"
    });
Console.WriteLine("The equivalent operation using Join():");
foreach (var v in queryMethod2)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}
```

```
});  
  
Console.WriteLine("The equivalent operation using Join():");  
foreach (var v in queryMethod2)  
{  
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");  
}
```

## 그룹화 조인 수행

그룹 조인은 계층적 데이터 구조를 생성하는 데 유용합니다. 첫 번째 컬렉션의 각 요소와 두 번째 컬렉션에서 상관 관계가 지정된 요소 집합을 쌍으로 구성합니다.

### ① 참고

첫 번째 컬렉션의 각 요소는 상관 관계가 지정된 요소가 두 번째 컬렉션에 있는지 여부에 관계없이 그룹 조인의 결과 집합에 표시됩니다. 상관 관계가 지정된 요소가 없는 경우 해당 요소에 대해 상관 관계가 지정된 요소의 시퀀스가 비어 있습니다. 따라서 결과 선택기에서 첫 번째 컬렉션의 모든 요소에 액세스할 수 있습니다. 이는 두 번째 컬렉션에 일치 항목이 없는 첫 번째 컬렉션의 요소에 액세스할 수 없는 비그룹 조인의 결과 선택기와 다릅니다.

### ⚠ 경고

[Enumerable.GroupJoin](#)에는 기존 관계형 데이터베이스 용어에 직접적으로 해당하는 항목이 없습니다. 그러나 이 메서드는 내부 조인 및 왼쪽 우선 외부 조인의 상위 집합을 구현합니다. 이러한 작업은 모두 그룹화된 조인과 관련하여 작성할 수 있습니다. 자세한 내용은 [Entity Framework Core, GroupJoin](#)을 참조하세요.

이 문서의 첫 번째 예제에서는 그룹 조인을 수행하는 방법을 보여 줍니다. 두 번째 예제에서는 그룹 조인을 사용하여 XML 요소를 만드는 방법을 보여 줍니다.

## 그룹 조인

다음 예제에서는 `Student.DepartmentID` 속성과 일치하는 `Department.ID`에 따라 `Department` 및 `Student` 형식 개체의 그룹 조인을 수행합니다. 각 일치 항목에 대한 요소 쌍을 생성하는 비그룹 조인과 달리 그룹 조인은 첫 번째 컬렉션의 각 요소에 대해 하나의 결과 개체(이 예제에서는 `Department` 개체)를 생성합니다. 두 번째 컬렉션의 해당 요소(이 예제에서는 `Student` 개체)는 컬렉션으로 그룹화됩니다. 마지막으로, 결과 선택기 함수는

`Department.Name` 및 `Student` 개체 컬렉션으로 구성된 각 일치 항목에 대해 무명 형식을 만듭니다.

C#

```
var query = from department in departments
            join student in students on department.ID equals student.DepartmentID
            into studentGroup
            select new
            {
                DepartmentName = department.Name,
                Students = studentGroup
            };

foreach (var v in query)
{
    // Output the department's name.
    Console.WriteLine($"{v.DepartmentName}:");

    // Output each of the students in that department.
    foreach (Student? student in v.Students)
    {
        Console.WriteLine($" {student.FirstName} {student.LastName}");
    }
}
```

위의 예제에서 `query` 변수에는 각 요소가 부서의 이름과 해당 부서에서 학습하는 학생의 컬렉션을 포함하는 익명 형식인 목록을 만드는 쿼리가 포함되어 있습니다.

메서드 구문을 사용하는 동일한 쿼리는 다음 코드에 나와 있습니다.

C#

```
var query = departments.GroupJoin(students, department => department.ID,
                                    student => student.DepartmentID,
                                    (department, Students) => new { DepartmentName = department.Name,
                                    Students });

foreach (var v in query)
{
    // Output the department's name.
    Console.WriteLine($"{v.DepartmentName}:");

    // Output each of the students in that department.
    foreach (Student? student in v.Students)
    {
        Console.WriteLine($" {student.FirstName} {student.LastName}");
    }
}
```

## XML을 만들기 위한 그룹 조인

그룹 조인은 LINQ to XML을 사용하여 XML을 만드는데 적합합니다. 다음 예제는 무명 형식을 만드는 대신 결과 선택기 함수가 조인된 개체를 나타내는 XML 요소를 만든다는 점을 제외하고 앞의 예제와 비슷합니다.

C#

```
XElement departmentsAndStudents = new("DepartmentEnrollment",
    from department in departments
    join student in students on department.ID equals student.DepartmentID
    into studentGroup
    select new XElement("Department",
        new XAttribute("Name", department.Name),
        from student in studentGroup
        select new XElement("Student",
            new XAttribute("FirstName", student.FirstName),
            new XAttribute("LastName", student.LastName)
        )
    )
);
Console.WriteLine(departmentsAndStudents);
```

메서드 구문을 사용하는 동일한 쿼리는 다음 코드에 나와 있습니다.

C#

```
XElement departmentsAndStudents = new("DepartmentEnrollment",
    departments.GroupJoin(students, department => department.ID, student =>
student.DepartmentID,
    (department, Students) => new XElement("Department",
        new XAttribute("Name", department.Name),
        from student in Students
        select new XElement("Student",
            new XAttribute("FirstName", student.FirstName),
            new XAttribute("LastName", student.LastName)
        )
    )
);
Console.WriteLine(departmentsAndStudents);
```

## 왼쪽 우선 외부 조인 수행

왼쪽 우선 외부 조인은 두 번째 컬렉션에 상호 연결된 요소가 있는지 여부에 관계없이 첫 번째 컬렉션의 각 요소가 반환되는 조인입니다. LINQ를 통해 그룹 조인의 결과에서

`DefaultIfEmpty` 메서드를 호출하여 왼쪽 우선 외부 조인을 수행할 수 있습니다.

다음 예제에서는 그룹 조인의 결과에서 `DefaultIfEmpty` 메서드를 사용하여 왼쪽 우선 외부 조인을 수행하는 방법을 보여 줍니다.

두 컬렉션의 왼쪽 우선 외부 조인을 생성하는 첫 번째 단계는 그룹 조인을 사용하여 내부 조인을 수행하는 것입니다. 이 프로세스에 대한 설명은 [내부 조인 수행](#)을 참조하세요. 이 예제에서 `Department` 개체 목록은 학생의 `DepartmentID`와 일치하는 `Department` 개체의 ID를 기준으로 `Student` 개체 목록에 내부 조인됩니다.

두 번째 단계는 오른쪽 컬렉션에 일치하는 항목이 없는 경우에도 첫 번째(왼쪽) 컬렉션의 각 요소를 결과 집합에 포함하는 것입니다. 이렇게 하려면 그룹 조인에서 일치하는 요소의 각 시퀀스에 대해 `DefaultIfEmpty`를 호출합니다. 이 예제에서는 일치하는 `Student` 개체의 각 시퀀스에서 `DefaultIfEmpty`를 호출합니다. 메서드는 `Department` 개체에 대해 일치하는 `Student` 개체의 시퀀스가 비어 있는 경우 단일 기본값을 포함하는 컬렉션을 반환하여 각 `Department` 개체가 결과 컬렉션에 반환되도록 합니다.

### ① 참고

참조 형식의 기본값은 `null` 이므로 예제에서는 각 `Student` 컬렉션의 각 요소에 액세스하기 전에 `null` 참조를 확인합니다.

C#

```
var query =
    from student in students
    join department in departments on student.DepartmentID equals
department.ID into gj
    from subgroup in gj.DefaultIfEmpty()
    select new
    {
        student.FirstName,
        student.LastName,
        Department = subgroup?.Name ?? string.Empty
    };

foreach (var v in query)
{
    Console.WriteLine($"{v.FirstName:-15} {v.LastName:-15}:
{v.Department}");
}
```

메서드 구문을 사용하는 동일한 쿼리는 다음 코드에 나와 있습니다.

C#

```

var query = students.GroupJoin(departments, student => student.DepartmentID,
department => department.ID,
    (student, departmentList) => new { student, subgroup =
departmentList.AsQueryable() })
    .SelectMany(joinedSet => joinedSet.subgroup, (student, department) =>
new
{
    student.student.FirstName,
    student.student.LastName,
    Department = department.Name
});

foreach (var v in query)
{
    Console.WriteLine($"{v.FirstName:-15} {v.LastName:-15}:
{v.Department}");
}

```

## 참고 항목

- [Join](#)
- [GroupJoin](#)
- [무명 형식](#)
- [Join 및 교차곱 쿼리 작성](#)
- [join 절](#)
- [group 절](#)
- [서로 다른 파일의 콘텐츠를 조인하는 방법\(LINQ\)\(C#\)](#)
- [여러 소스로 개체 컬렉션을 채우는 방법\(LINQ\)\(C#\)](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.



### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

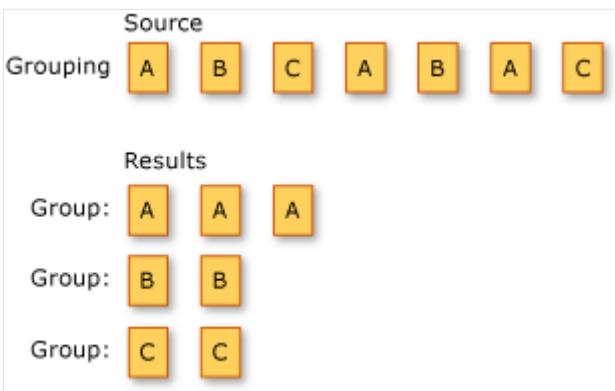
 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# 데이터 그룹화(C#)

아티클 • 2024. 06. 03.

그룹화는 데이터를 그룹에 넣어 각 그룹의 요소가 공통 특성을 공유하게 하는 작업을 가리킵니다. 다음 그림은 문자 시퀀스를 그룹화한 결과를 보여 줍니다. 각 그룹에 대한 키는 문자입니다.



## ① 중요

이 샘플은 [System.Collections.Generic.IEnumerable<T>](#) 데이터 원본을 사용합니다. [System.Linq.IQueryProvider](#) 기반 데이터 원본은 [System.Linq.IQueryable<T>](#) 데이터 원본과 [식 트리](#)를 사용합니다. 식 트리에는 허용되는 C# 구문에 대한 [제한 사항](#)이 있습니다. 또한 [EF Core](#)와 같은 각 [IQueryProvider](#) 데이터 원본에는 더 많은 제한이 적용될 수 있습니다. 데이터 원본에 대한 설명서를 확인합니다.

데이터 요소를 그룹화하는 표준 쿼리 연산자 메서드가 다음 표에 나와 있습니다.

### ② 테이블 확장

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
GroupBy	공통 특성을 공유하는 요소를 그룹화합니다. <a href="#">IGrouping&lt; TKey, TElement &gt;</a> 개체는 각 그룹을 나타냅니다.	<code>group ... by</code>	<a href="#">Enumerable.GroupBy</a> <a href="#">Queryable.GroupBy</a> 또는  <code>group ... by ... into</code> ...
ToLookup	키 선택기 함수에 따라 <a href="#">Lookup&lt; TKey, TElement &gt;</a> (일대다 사전)에 요소를 삽입합니다.	해당 없음.	<a href="#">Enumerable.ToLookup</a>

다음 코드 예제에서는 `group by` 절을 사용하여 짝수 또는 홀수인지에 따라 목록에서 정수를 그룹화합니다.

C#

```
List<int> numbers = [35, 44, 200, 84, 3987, 4, 199, 329, 446, 208];

IEnumerable<IGrouping<int, int>> query = from number in numbers
                                             group number by number % 2;

foreach (var group in query)
{
    Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd
numbers:");
    foreach (int i in group)
    {
        Console.WriteLine(i);
    }
}
```

메서드 구문을 사용하는 동일한 쿼리는 다음 코드에 나와 있습니다.

C#

```
List<int> numbers = [35, 44, 200, 84, 3987, 4, 199, 329, 446, 208];

IEnumerable<IGrouping<int, int>> query = numbers
    .GroupBy(number => number % 2);

foreach (var group in query)
{
    Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd
numbers:");
    foreach (int i in group)
    {
        Console.WriteLine(i);
    }
}
```

이 문서의 다음 예제에서는 이 영역에 대한 공통 데이터 원본을 사용합니다.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};
```

```

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}

```

각 `Student`에는 학년 수준, 기본 부서 및 일련의 점수가 있습니다. `Teacher`에는 교사가 수업을 진행하는 캠퍼스를 식별하는 `city` 속성도 있습니다. `Department`에는 부서장 역할을 하는 `Teacher`에 대한 참조와 이름이 있습니다.

## 쿼리 결과 그룹화

그룹화는 LINQ의 가장 강력한 기능 중 하나입니다. 다음 예제에서는 다양한 방법으로 데이터를 그룹화하는 방법을 보여 줍니다.

- 단일 속성 사용.
- 문자열 속성의 첫 문자 사용.
- 계산된 숫자 범위 사용.
- 부울 조건자 또는 기타 식 사용.
- 복합 키 사용.

또한 마지막 두 개의 쿼리는 학생의 이름과 성만 포함된 새 무명 형식으로 결과를 프로젝션합니다. 자세한 내용은 [group 절](#)을 참조하세요.

## 단일 속성으로 그룹화 예제

다음 예제에서는 요소의 단일 속성을 그룹 키로 사용하여 소스 요소를 그룹화하는 방법을 보여 줍니다. 핵심은 학교에서 학생의 학년을 나타내는 `enum`입니다. 그룹화 작업에는 형식에 대한 기본 같은 비교자가 사용됩니다.

C#

```
var groupByYearQuery =
    from student in students
    group student by student.Year into newGroup
    orderby newGroup.Key
    select newGroup;

foreach (var yearGroup in groupByYearQuery)
{
    Console.WriteLine($"Key: {yearGroup.Key}");
    foreach (var student in yearGroup)
    {
        Console.WriteLine($"{student.LastName}, {student.FirstName}");
    }
}
```

메서드 구문을 사용하는 동일한 코드가 다음 예제에 나와 있습니다.

C#

```
// Variable groupByLastNamesQuery is an IEnumerable<IGrouping<string,
// DataClass.Student>>.
var groupByYearQuery = students
    .GroupBy(student => student.Year)
    .OrderBy(newGroup => newGroup.Key);

foreach (var yearGroup in groupByYearQuery)
{
    Console.WriteLine($"Key: {yearGroup.Key}");
    foreach (var student in yearGroup)
    {
        Console.WriteLine($"{student.LastName}, {student.FirstName}");
    }
}
```

## 값으로 그룹화 예제

다음 예제에서는 개체의 속성이 아닌 다른 항목을 그룹 키에 사용하여 소스 요소를 그룹화하는 방법을 보여 줍니다. 이 예제에서 키는 학생 성의 첫 번째 문자입니다.

C#

```

var groupByFirstLetterQuery =
    from student in students
    let firstLetter = student.LastName[0]
    group student by firstLetter;

foreach (var studentGroup in groupByFirstLetterQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup)
    {
        Console.WriteLine($"{student.LastName}, {student.FirstName}");
    }
}

```

그룹 항목에 액세스하려면 중첩된 foreach가 필요합니다.

메서드 구문을 사용하는 동일한 코드가 다음 예제에 나와 있습니다.

C#

```

var groupByFirstLetterQuery = students
    .GroupBy(student => student.LastName[0]);

foreach (var studentGroup in groupByFirstLetterQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup)
    {
        Console.WriteLine($"{student.LastName}, {student.FirstName}");
    }
}

```

## 범위로 그룹화 예제

다음 예제에서는 숫자 범위를 그룹 키로 사용하여 소스 요소를 그룹화하는 방법을 보여줍니다. 그런 다음 쿼리는 이름과 성 및 학생이 속한 백분위수 범위만 포함된 무명 형식으로 결과를 프로젝션합니다. 결과를 표시하는 데 완전한 `Student` 개체를 사용할 필요가 없으므로 무명 형식이 사용됩니다. `GetPercentile`은 학생의 평균 점수를 기준으로 백분위수를 계산하는 도우미 함수입니다. 이 메서드는 0~10 사이의 정수를 반환합니다.

C#

```

static int GetPercentile(Student s)
{
    double avg = s.Scores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

```

```

var groupByPercentileQuery =
    from student in students
    let percentile = GetPercentile(student)
    group new
    {
        student.FirstName,
        student.LastName
    } by percentile into percentGroup
    orderby percentGroup.Key
    select percentGroup;

foreach (var studentGroup in groupByPercentileQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key * 10}");
    foreach (var item in studentGroup)
    {
        Console.WriteLine($"{item.LastName}, {item.FirstName}");
    }
}

```

그룹 및 그룹 항목을 반복하려면 중첩된 foreach가 필요합니다. 메서드 구문을 사용하는 동일한 코드가 다음 예제에 나와 있습니다.

C#

```

static int GetPercentile(Student s)
{
    double avg = s.Scores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

var groupByPercentileQuery = students
    .Select(student => new { student, percentile = GetPercentile(student) })
    .GroupBy(student => student.percentile)
    .Select(percentGroup => new
    {
        percentGroup.Key,
        Students = percentGroup.Select(s => new { s.student.FirstName,
s.student.LastName })
    })
    .OrderBy(percentGroup => percentGroup.Key);

foreach (var studentGroup in groupByPercentileQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key * 10}");
    foreach (var item in studentGroup.Students)
    {
        Console.WriteLine($"{item.LastName}, {item.FirstName}");
    }
}

```

## 비교로 그룹화 예제

다음 예제에서는 부울 비교 식을 사용하여 소스 요소를 그룹화하는 방법을 보여 줍니다. 이 예제에서는 부울 식은 학생의 평균 시험 점수가 75보다 큰지 테스트합니다. 이전 예제처럼 완전한 소스 요소가 필요하지 않으므로 결과는 무명 형식으로 프로젝션됩니다. 익명 형식의 속성은 `Key` 멤버의 속성이 됩니다.

C#

```
var groupByHighAverageQuery =
    from student in students
    group new
    {
        student.FirstName,
        student.LastName
    } by student.Scores.Average() > 75 into studentGroup
    select studentGroup;

foreach (var studentGroup in groupByHighAverageQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup)
    {
        Console.WriteLine($"{student.FirstName} {student.LastName}");
    }
}
```

메서드 구문을 사용하는 동일한 쿼리는 다음 코드에 나와 있습니다.

C#

```
var groupByHighAverageQuery = students
    .GroupBy(student => student.Scores.Average() > 75)
    .Select(group => new
    {
        group.Key,
        Students = group.AsEnumerable().Select(s => new { s.FirstName,
s.LastName })
    });

foreach (var studentGroup in groupByHighAverageQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup.Students)
    {
        Console.WriteLine($"{student.FirstName} {student.LastName}");
    }
}
```

## 무명 형식으로 그룹화

다음 예제에서는 무명 형식을 사용하여 여러 값이 포함된 키를 캡슐화하는 방법을 보여줍니다. 이 예제에서 첫 번째 키 값은 학생 성의 첫 번째 문자입니다. 두 번째 키 값은 첫 번째 시험에서 학생의 점수가 85보다 큰지 여부를 지정하는 부울입니다. 키의 속성을 기준으로 그룹 순서를 지정할 수 있습니다.

C#

```
var groupByCompoundKey =
    from student in students
    group student by new
    {
        FirstLetterOfLastName = student.LastName[0],
        IsScoreOver85 = student.Scores[0] > 85
    } into studentGroup
    orderby studentGroup.Key.FirstLetterOfLastName
    select studentGroup;

foreach (var scoreGroup in groupByCompoundKey)
{
    var s = scoreGroup.Key.IsScoreOver85 ? "more than 85" : "less than 85";
    Console.WriteLine($"Name starts with
{scoreGroup.Key.FirstLetterOfLastName} who scored {s}");
    foreach (var item in scoreGroup)
    {
        Console.WriteLine($"{item.FirstName} {item.LastName}");
    }
}
```

메서드 구문을 사용하는 동일한 쿼리는 다음 코드에 나와 있습니다.

C#

```
var groupByCompoundKey = students
    .GroupBy(student => new
    {
        FirstLetterOfLastName = student.LastName[0],
        IsScoreOver85 = student.Scores[0] > 85
    })
    .OrderBy(studentGroup => studentGroup.Key.FirstLetterOfLastName);

foreach (var scoreGroup in groupByCompoundKey)
{
    var s = scoreGroup.Key.IsScoreOver85 ? "more than 85" : "less than 85";
    Console.WriteLine($"Name starts with
{scoreGroup.Key.FirstLetterOfLastName} who scored {s}");
    foreach (var item in scoreGroup)
    {
        Console.WriteLine($"{item.FirstName} {item.LastName}");
    }
}
```

```
    }  
}
```

## 중첩 그룹 만들기

다음 예제에서는 LINQ 쿼리 식에서 중첩 그룹을 만드는 방법을 보여 줍니다. 학년 또는 성적 수준에 따라 만들어진 각 그룹은 개인의 이름에 따라 하위 그룹으로 추가로 구분됩니다.

C#

```
var nestedGroupsQuery =  
    from student in students  
    group student by student.Year into newGroup1  
    from newGroup2 in  
    from student in newGroup1  
    group student by student.LastName  
    group newGroup2 by newGroup1.Key;  
  
foreach (var outerGroup in nestedGroupsQuery)  
{  
    Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");  
    foreach (var innerGroup in outerGroup)  
    {  
        Console.WriteLine($"{innerGroup.Key}");  
        foreach (var innerGroupElement in innerGroup)  
        {  
            Console.WriteLine($"{innerGroupElement.LastName}  
{innerGroupElement.FirstName}");  
        }  
    }  
}
```

중첩 그룹의 내부 요소를 반복하려면 세 개의 중첩 `foreach` 루프가 필요합니다.  
(실제 형식을 보려면 반복 변수 `outerGroup`, `innerGroup` 및 `innerGroupElement` 위에 마우스 커서를 올리세요.)

메서드 구문을 사용하는 동일한 쿼리는 다음 코드에 나와 있습니다.

C#

```
var nestedGroupsQuery =  
    students  
    .GroupBy(student => student.Year)  
    .Select(newGroup1 => new  
    {  
        newGroup1.Key,  
        NestedGroup = newGroup1
```

```

        .GroupBy(student => student.LastName)
    });

foreach (var outerGroup in nestedGroupsQuery)
{
    Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
    foreach (var innerGroup in outerGroup.NestedGroup)
    {
        Console.WriteLine($"\\tNames that begin with: {innerGroup.Key}");
        foreach (var innerGroupElement in innerGroup)
        {
            Console.WriteLine($"\\t\\t{innerGroupElement.LastName}
{innerGroupElement.FirstName}");
        }
    }
}

```

## 그룹화 작업에서 하위 쿼리 수행

이 문서에서는 소스 데이터를 그룹으로 정렬한 다음, 각 그룹에 대해 개별적으로 하위 쿼리를 수행하는 쿼리를 만드는 두 가지 방법을 보여 줍니다. 각 예제의 기본적인 방법은 `newGroup`이라는 연속을 사용하고 `newGroup`에 대한 새 하위 쿼리를 생성하여 소스 요소를 그룹화하는 것입니다. 이 하위 쿼리는 외부 쿼리에 의해 만들어지는 각 새로운 그룹에 대해 실행됩니다. 이 특정 예제에서 최종 출력은 그룹이 아니라 무명 형식의 플랫 시퀀스입니다.

그룹화하는 방법에 대한 자세한 내용은 [group 절](#)을 참조하세요. 연속에 대한 자세한 내용은 [into](#)를 참조하세요. 다음 예제에서는 메모리 내 데이터 구조를 데이터 소스로 사용하지만 모든 종류의 LINQ 데이터 소스에 대해 동일한 원칙이 적용됩니다.

C#

```

var queryGroupMax =
    from student in students
    group student by student.Year into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore = (
            from student2 in studentGroup
            select student2.Scores.Average()
        ).Max()
    };
    count = queryGroupMax.Count();
    Console.WriteLine($"Number of groups = {count}");

    foreach (var item in queryGroupMax)
    {

```

```
        Console.WriteLine($" {item.Level} Highest Score={item.HighestScore}");
    }
```

위의 코드 조각에 있는 쿼리는 메서드 구문을 사용하여 작성할 수도 있습니다. 다음 코드 조각은 메서드 구문을 사용하여 작성된 의미상 동일한 쿼리입니다.

C#

```
var queryGroupMax =
    students
        .GroupBy(student => student.Year)
        .Select(studentGroup => new
    {
        Level = studentGroup.Key,
        HighestScore = studentGroup.Max(student2 =>
            student2.Scores.Average())
    });

var count = queryGroupMax.Count();
Console.WriteLine($"Number of groups = {count}");

foreach (var item in queryGroupMax)
{
    Console.WriteLine($" {item.Level} Highest Score={item.HighestScore}");
}
```

## 참고 항목

- [System.Linq](#)
- [GroupBy](#)
- [IGrouping<TKey,TElement>](#)
- [group 절](#)
- [그룹을 사용하여 파일을 여러 파일로 분할하는 방법\(LINQ\)\(C#\)](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

[설명서 문제 열기](#)

[제품 사용자 의견 제공](#)

# 방법: LINQ를 사용하여 파일 및 디렉터리 쿼리

아티클 • 2024. 04. 28.

많은 파일 시스템 작업은 기본적으로 쿼리이므로 LINQ 접근 방식에 적합합니다. 이러한 쿼리는 비파괴적입니다. 원본 파일이나 폴더의 콘텐츠는 변경되지 않습니다. 쿼리로 인해 부작용이 발생해서는 안 됩니다. 일반적으로 원본 데이터를 수정하는 모든 코드(만들기/업데이트/삭제 작업을 수행하는 쿼리 포함)는 데이터를 쿼리만하는 코드와 별도로 유지되어야 합니다.

파일 시스템의 내용을 정확하게 나타내고 예외를 정상적으로 처리하는 데이터 원본 만들기와 관련하여 몇 가지 복잡한 부분이 있습니다. 이 섹션의 예제에서는 지정된 루트 폴더와 모든 하위 폴더에 있는 모든 파일을 나타내는 [FileInfo](#) 개체의 스냅샷 컬렉션을 만듭니다. 각 [FileInfo](#)의 실제 상태는 쿼리 실행을 시작하고 종료하는 시간 사이에 변경될 수 있습니다. 예를 들어 [FileInfo](#) 개체 목록을 만들어 데이터 소스로 사용할 수 있습니다. 쿼리에서 `Length` 속성에 액세스하려고 하면 [FileInfo](#) 개체는 `Length` 값을 업데이트하기 위해 파일 시스템에 액세스하려고 시도합니다. 파일이 더 이상 존재하지 않으면 파일 시스템을 직접 쿼리하지 않더라도 쿼리에 [FileNotFoundException](#)이 표시됩니다.

## 지정된 특성 또는 이름을 사용하여 파일을 쿼리하는 방법

이 예제에서는 지정된 디렉터리 트리에서 지정된 파일 이름 확장명(예: ".txt")을 가진 파일을 모두 찾는 방법을 보여 줍니다. 또한 생성 시간을 기준으로 트리에서 가장 최신 파일이나 가장 오래된 파일을 반환하는 방법을 보여 줍니다. Windows, Mac 또는 Linux 시스템에서 이 코드를 실행하는지 여부에 관계없이 많은 샘플의 첫 번째 줄을 수정해야 할 수도 있습니다.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

DirectoryInfo dir = new DirectoryInfo(startFolder);
var fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

var fileQuery = from file in fileList
                where file.Extension == ".txt"
                orderby file.Name
                select file;
```

```

// Uncomment this block to see the full query
// foreach (FileInfo fi in fileQuery)
// {
//     Console.WriteLine(fi.FullName);
// }

var newestFile = (from file in fileQuery
                  orderby file.CreationTime
                  select new { file.FullName, file.CreationTime })
                  .Last();

Console.WriteLine($"\\r\\nThe newest .txt file is {newestFile.FullName}.
Creation time: {newestFile.CreationTime}");

```

## 확장명을 기준으로 파일을 그룹화하는 방법

이 예제에서는 LINQ를 사용하여 파일 또는 폴더 목록에 대해 고급 그룹화 및 정렬 작업을 수행하는 방법을 보여 줍니다. 또한 `Skip` 및 `Take` 메서드를 사용하여 콘솔 창에서 출력을 페이지징하는 방법을 보여 줍니다.

다음 쿼리는 지정된 디렉터리 트리의 내용을 파일 이름 확장명으로 그룹화하는 방법을 보여 줍니다.

C#

```

string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

int trimLength = startFolder.Length;

DirectoryInfo dir = new DirectoryInfo(startFolder);

var fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

var queryGroupByExt = from file in fileList
                      group file by file.Extension.ToLower() into fileGroup
                      orderby fileGroup.Count(), fileGroup.Key
                      select fileGroup;

// Iterate through the outer collection of groups.
foreach (var filegroup in queryGroupByExt.Take(5))
{
    Console.WriteLine($"Extension: {filegroup.Key}");
    var resultPage = filegroup.Take(20);

    //Execute the resultPage query
    foreach (var f in resultPage)
    {
        Console.WriteLine($"\\t{f.FullName.Substring(trimLength)}");
    }
}

```

```
        }
        Console.WriteLine();
    }
```

이 프로그램의 출력은 로컬 파일 시스템의 세부 정보 및 `startFolder`의 설정에 따라 길어질 수 있습니다. 모든 결과를 볼 수 있도록, 이 예제에서는 결과를 페이지하는 방법을 보여줍니다. 각 그룹이 별도로 열거되므로 중첩된 `foreach` 루프가 필요합니다.

## 폴더 집합의 총 바이트 수를 쿼리하는 방법

이 예제에서는 지정된 폴더 및 모든 하위 폴더의 모든 파일에서 사용된 총 바이트 수를 검색하는 방법을 보여 줍니다. `Sum` 메서드는 `select` 절에서 선택된 모든 항목의 값을 더합니다. `Sum` 대신 `Min` 또는 `Max` 메서드를 호출하여 지정된 디렉터리 트리에서 가장 큰 파일이나 가장 작은 파일을 검색하도록 이 쿼리를 수정할 수 있습니다.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

var fileList = Directory.GetFiles(startFolder, "*.*",
SearchOption.AllDirectories);

var fileQuery = from file in fileList
    let fileLen = new FileInfo(file).Length
    where fileLen > 0
    select fileLen;

// Cache the results to avoid multiple trips to the file system.
long[] fileLengths = fileQuery.ToArray();

// Return the size of the largest file
long largestFile = fileLengths.Max();

// Return the total number of bytes in all the files under the specified
// folder.
long totalBytes = fileLengths.Sum();

Console.WriteLine($"There are {totalBytes} bytes in {fileList.Count()} files
under {startFolder}");
Console.WriteLine($"The largest file is {largestFile} bytes.");
```

이 예제에서는 이전 예를 확장하여 다음을 수행합니다.

- 가장 큰 파일의 크기(바이트)를 검색하는 방법입니다.
- 가장 작은 파일의 크기(바이트)를 검색하는 방법입니다.

- 지정된 루트 폴더 아래의 하나 이상 폴더에서 `FileInfo` 개체의 가장 큰 파일이나 가장 작은 파일을 검색하는 방법입니다.
- 가장 큰 파일 10개 등의 시퀀스를 검색하는 방법입니다.
- 지정된 크기보다 작은 파일을 무시하고 해당 파일 크기(바이트)에 따라 파일을 그룹으로 정렬하는 방법입니다.

다음 예제에서는 파일 크기(바이트)에 따라 파일을 쿼리 및 그룹화하는 방법을 보여 주는 5개의 개별 쿼리가 포함되어 있습니다. `FileInfo` 개체의 다른 속성을 기반으로 쿼리를 작성하도록 이러한 예를 수정할 수 있습니다.

C#

```
// Return the FileInfo object for the largest file
// by sorting and selecting from beginning of list
FileInfo longestFile = (from file in fileList
                        let fileInfo = new FileInfo(file)
                        where fileInfo.Length > 0
                        orderby fileInfo.Length descending
                        select fileInfo
                        ).First();

Console.WriteLine($"The largest file under {startFolder} is
{longestFile.FullName} with a length of {longestFile.Length} bytes");

//Return the FileInfo of the smallest file
FileInfo smallestFile = (from file in fileList
                        let fileInfo = new FileInfo(file)
                        where fileInfo.Length > 0
                        orderby fileInfo.Length ascending
                        select fileInfo
                        ).First();

Console.WriteLine($"The smallest file under {startFolder} is
{smallestFile.FullName} with a length of {smallestFile.Length} bytes");

//Return the FileInfos for the 10 largest files
var queryTenLargest = (from file in fileList
                        let fileInfo = new FileInfo(file)
                        let len = fileInfo.Length
                        orderby len descending
                        select fileInfo
                        ).Take(10);

Console.WriteLine($"The 10 largest files under {startFolder} are:");

foreach (var v in queryTenLargest)
{
    Console.WriteLine($"{v.FullName}: {v.Length} bytes");
}

// Group the files according to their size, leaving out
// files that are less than 200000 bytes.
```

```

var querySizeGroups = from file in fileList
    let fileInfo = new FileInfo(file)
    let len = fileInfo.Length
    where len > 0
    group fileInfo by (len / 100000) into fileGroup
    where fileGroup.Key >= 2
    orderby fileGroup.Key descending
    select fileGroup;

foreach (var filegroup in querySizeGroups)
{
    Console.WriteLine($"{filegroup.Key}0000");
    foreach (var item in filegroup)
    {
        Console.WriteLine($"\\t{item.Name}: {item.Length}");
    }
}

```

전체 `FileInfo` 개체를 하나 이상 반환하기 위해 쿼리는 먼저 데이터 소스에서 각 개체를 검사한 다음 해당 `Length` 속성 값을 기준으로 정렬해야 합니다. 그런 다음 길이가 가장 큰 단일 개체나 시퀀스를 반환할 수 있습니다. 목록의 첫 번째 요소를 반환하려면 `First`를 사용합니다. 처음 n개의 요소를 반환하려면 `Take`를 사용합니다. 목록의 시작 부분에 가장 작은 요소를 배치하려면 내림차순 정렬 순서를 지정합니다.

## 디렉터리 트리에서 중복 파일을 쿼리하는 방법

때로는 동일한 이름을 가진 파일이 둘 이상의 폴더에 있을 수 있습니다. 이 예제에서는 지정된 루트 폴더 아래에서 이러한 중복 파일 이름을 쿼리하는 방법을 보여 줍니다. 두 번째 예제에서는 크기 및 `LastWrite` 시간도 일치하는 파일을 쿼리하는 방법을 보여 줍니다.

C#

```

string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

DirectoryInfo dir = new DirectoryInfo(startFolder);

IEnumerable<FileInfo> fileList = dir.GetFiles("*.*",
SearchOption.AllDirectories);

// used in WriteLine to keep the lines shorter
int charsToSkip = startFolder.Length;

// var can be used for convenience with groups.
var queryDupNames = from file in fileList
                    group file.FullName.Substring(charsToSkip) by file.Name
                    into fileGroup
                    where fileGroup.Count() > 1

```

```

        select fileGroup;

foreach (var queryDup in queryDupNames.Take(20))
{
    Console.WriteLine($"Filename = {{(queryDup.Key.ToString() == string.Empty
? "[none]" : queryDup.Key.ToString())}}");

    foreach (var fileName in queryDup.Take(10))
    {
        Console.WriteLine($"{\t}{fileName}");
    }
}

```

첫 번째 쿼리에서는 키를 사용하여 일치 항목을 확인합니다. 이름은 같지만 콘텐츠가 다를 수 있는 파일을 찾습니다. 두 번째 쿼리는 복합 키를 사용하여 [FileInfo](#) 개체의 세 가지 속성과 비교합니다. 이 쿼리가 이름이 같고 내용이 비슷하거나 동일한 파일을 찾을 가능성이 훨씬 더 큽니다.

C#

```

string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

// Make the lines shorter for the console display
int charsToSkip = startFolder.Length;

// Take a snapshot of the file system.
DirectoryInfo dir = new DirectoryInfo(startFolder);
IEnumerable<FileInfo> fileList = dir.GetFiles(".*",
SearchOption.AllDirectories);

// Note the use of a compound key. Files that match
// all three properties belong to the same group.
// A named type is used to enable the query to be
// passed to another method. Anonymous types can also be used
// for composite keys but cannot be passed across method boundaries
//
var queryDupFiles = from file in fileList
                    group file.FullName.Substring(charsToSkip) by
                    (Name: file.Name, LastWriteTime: file.LastWriteTime,
Length: file.Length )
                    into fileGroup
                    where fileGroup.Count() > 1
                    select fileGroup;

foreach (var queryDup in queryDupFiles.Take(20))
{
    Console.WriteLine($"Filename = {{(queryDup.Key.ToString() ==
string.Empty ? "[none]" : queryDup.Key.ToString())}}");

    foreach (var fileName in queryDup)

```

```

    {
        Console.WriteLine($"\\t{fileName}");
    }
}

```

## 폴더에 있는 텍스트 파일의 콘텐츠를 쿼리하는 방법

이 예제에서는 지정된 디렉터리 트리에 있는 모든 파일을 쿼리하고 각 파일을 연 다음 내용을 검사하는 방법을 보여 줍니다. 이러한 유형의 기술을 사용하여 디렉터리 트리 내용의 인덱스 또는 역방향 인덱스를 만들 수 있습니다. 이 예제에서는 단순 문자열 검색이 수행됩니다. 그러나 정규식을 사용하면 더 복잡한 유형의 패턴 일치를 수행할 수 있습니다.

C#

```

string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

DirectoryInfo dir = new DirectoryInfo(startFolder);

var fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

string searchTerm = "change";

var queryMatchingFiles = from file in fileList
                        where file.Extension == ".txt"
                        let fileText = File.ReadAllText(file.FullName)
                        where fileText.Contains(searchTerm)
                        select file.FullName;

// Execute the query.
Console.WriteLine($"\"The term \"{searchTerm}\" was found in:\"");
foreach (string filename in queryMatchingFiles)
{
    Console.WriteLine(filename);
}

```

## 두 폴더의 내용을 비교하는 방법

이 예제에서는 두 파일 목록을 비교하는 세 가지 방법을 보여 줍니다.

- 두 파일 목록이 똑같은지 여부를 지정하는 부울 값 쿼리.
- 양쪽 폴더에 있는 파일을 검색하기 위해 교집합 쿼리.
- 두 개 중 한 폴더에만 있는 파일을 검색하기 위해 차집합 쿼리.

여기 표시된 방법은 형식에 관계없이 개체의 시퀀스를 비교하도록 조정될 수 있습니다.

여기 표시된 `FileComparer` 클래스는 표준 쿼리 연산자와 함께 사용자 지정 비교자 클래스를 사용하는 방법을 보여 줍니다. 이 클래스는 실제 시나리오에서 사용하기 위한 것이 아닙니다. 단지 각 파일의 이름 및 길이(바이트)를 사용하여 각 폴더의 내용이 똑같은지 여부를 확인합니다. 실제 시나리오에서는 더 엄격한 일치 검사를 수행하도록 이 비교자를 수정해야 합니다.

C#

```
// This implementation defines a very simple comparison
// between two FileInfo objects. It only compares the name
// of the files being compared and their length in bytes.
class FileCompare : IEqualityComparer<FileInfo>
{
    public bool Equals(FileInfo? f1, FileInfo? f2)
    {
        return (f1?.Name == f2?.Name &&
                f1?.Length == f2?.Length);
    }

    // Return a hash that reflects the comparison criteria. According to the
    // rules for IEqualityComparer<T>, if Equals is true, then the hash
    // codes must
    // also be equal. Because equality as defined here is a simple value
    // equality, not
    // reference identity, it is possible that two or more objects will
    // produce the same
    // hash code.
    public int GetHashCode(FileInfo fi)
    {
        string s = $"{fi.Name}{fi.Length}";
        return s.GetHashCode();
    }
}

public static void CompareDirectories()
{
    string pathA = """C:\Program Files\dotnet\sdk\8.0.104""";
    string pathB = """C:\Program Files\dotnet\sdk\8.0.204""";

    DirectoryInfo dir1 = new DirectoryInfo(pathA);
    DirectoryInfo dir2 = new DirectoryInfo(pathB);

    IEnumerable<FileInfo> list1 = dir1.GetFiles("*.",
SearchOption.AllDirectories);
    IEnumerable<FileInfo> list2 = dir2.GetFiles("*.",
SearchOption.AllDirectories);

    //A custom file comparer defined below
    FileCompare myFileCompare = new FileCompare();
```

```

// This query determines whether the two folders contain
// identical file lists, based on the custom file comparer
// that is defined in the FileCompare class.
// The query executes immediately because it returns a bool.
bool areIdentical = list1.SequenceEqual(list2, myFileCompare);

if (areIdentical == true)
{
    Console.WriteLine("the two folders are the same");
}
else
{
    Console.WriteLine("The two folders are not the same");
}

// Find the common files. It produces a sequence and doesn't
// execute until the foreach statement.
var queryCommonFiles = list1.Intersect(list2, myFileCompare);

if (queryCommonFiles.Any())
{
    Console.WriteLine($"The following files are in both folders (total
number = {queryCommonFiles.Count()}):");
    foreach (var v in queryCommonFiles.Take(10))
    {
        Console.WriteLine(v.Name); //shows which items end up in result
list
    }
}
else
{
    Console.WriteLine("There are no common files in the two folders.");
}

// Find the set difference between the two folders.
var queryList1Only = (from file in list1
                      select file)
                      .Except(list2, myFileCompare);

Console.WriteLine();
Console.WriteLine($"The following files are in list1 but not list2
(total number = {queryList1Only.Count()}):");
foreach (var v in queryList1Only.Take(10))
{
    Console.WriteLine(v.FullName);
}

var queryList2Only = (from file in list2
                      select file)
                      .Except(list1, myFileCompare);

Console.WriteLine();
Console.WriteLine($"The following files are in list2 but not list1
(total number = {queryList2Only.Count()}):");
foreach (var v in queryList2Only.Take(10))

```

```
{  
    Console.WriteLine(v.FullName);  
}  
}
```

## 구분된 파일의 필드를 다시 정렬하는 방법

쉼표로 구분된 값(CSV) 파일은 스프레드시트 데이터 또는 행과 열로 표현되는 다른 표 형식 데이터를 저장하는 데 자주 사용되는 텍스트 파일입니다. [Split](#) 메서드를 사용하여 필드를 분리하면 LINQ를 사용하여 CSV 파일을 쉽게 쿼리하고 조작할 수 있습니다. 실제로 동일한 방법을 사용하여 모든 구조적 텍스트 줄의 일부를 다시 정렬할 수 있습니다. CSV 파일로 제한되지 않습니다.

다음 예에서는 세 개의 열이 학생의 "성", "이름" 및 "ID"를 나타낸다고 가정합니다. 필드는 학생의 성을 기준으로 사전순으로 되어 있습니다. 쿼리는 ID 열이 첫 번째로 표시되고, 학생의 이름과 성을 결합하는 두 번째 열이 뒤에 오는 새 시퀀스를 생성합니다. ID 필드에 따라 줄이 다시 정렬됩니다. 결과는 새 파일에 저장되며 원본 데이터는 수정되지 않습니다. 다음 텍스트는 다음 예에 사용된 *spreadsheet1.csv* 파일의 콘텐츠를 보여 줍니다.

txt

```
Adams,Terry,120  
Fakhouri,Fadi,116  
Feng,Hanying,117  
Garcia,Cesar,114  
Garcia,Debra,115  
Garcia,Hugo,118  
Mortensen,Sven,113  
O'Donnell,Claire,112  
Omelchenko,Svetlana,111  
Tucker,Lance,119  
Tucker,Michael,122  
Zabokritski,Eugene,121
```

다음 코드는 원본 파일을 읽고 CSV 파일의 각 열을 다시 정렬하여 열 순서를 다시 정렬합니다.

C#

```
string[] lines = File.ReadAllLines("spreadsheet1.csv");  
  
// Create the query. Put field 2 first, then  
// reverse and combine fields 0 and 1 from the old field  
IEnumerable<string> query = from line in lines  
    let fields = line.Split(',')  
    orderby fields[2]  
    select $"{fields[2]}, {fields[1]} {fields[0]}";
```

```
File.WriteAllLines("spreadsheet2.csv", query.ToArray());  
  
/* Output to spreadsheet2.csv:  
111, Svetlana Omelchenko  
112, Claire O'Donnell  
113, Sven Mortensen  
114, Cesar Garcia  
115, Debra Garcia  
116, Fadi Fakhouri  
117, Hanying Feng  
118, Hugo Garcia  
119, Lance Tucker  
120, Terry Adams  
121, Eugene Zabokritski  
122, Michael Tucker  
*/
```

## 그룹을 사용하여 파일을 여러 파일로 분할하는 방법

이 예제에서는 두 파일의 내용을 병합한 다음 새로운 방식으로 데이터를 구성하는 새 파일 집합을 만드는 한 가지 방법을 보여 줍니다. 쿼리는 두 파일의 콘텐츠를 사용합니다. 다음 텍스트는 첫 번째 파일인 *names1.txt*의 콘텐츠를 보여 줍니다.

```
txt  
  
Bankov, Peter  
Holm, Michael  
Garcia, Hugo  
Potra, Cristina  
Noriega, Fabricio  
Aw, Kam Foo  
Beebe, Ann  
Toyoshima, Tim  
Guy, Wey Yuan  
Garcia, Debra
```

두 번째 파일인 *names2.txt*에는 다른 이름 집합이 포함되어 있으며 그중 일부는 첫 번째 집합과 공통됩니다.

```
txt  
  
Liu, Jinghao  
Bankov, Peter  
Holm, Michael  
Garcia, Hugo  
Beebe, Ann
```

Gilchrist, Beth  
Myrcha, Jacek  
Giakoumakis, Leo  
McLin, Nkenge  
El Yassir, Mehdi

다음 코드는 두 파일을 모두 쿼리하고 두 파일의 합집합을 가져온 다음 성의 첫 문자로 정의된 각 그룹에 대해 새 파일을 작성합니다.

C#

```
string[] fileA = File.ReadAllLines("names1.txt");
string[] fileB = File.ReadAllLines("names2.txt");

// Concatenate and remove duplicate names
var mergeQuery = fileA.Union(fileB);

// Group the names by the first letter in the last name.
var groupQuery = from name in mergeQuery
                  let n = name.Split(',')[0]
                  group name by n[0] into g
                  orderby g.Key
                  select g;

foreach (var g in groupQuery)
{
    string fileName = $"testFile_{g.Key}.txt";

    Console.WriteLine(g.Key);

    using StreamWriter sw = new StreamWriter(fileName);
    foreach (var item in g)
    {
        sw.WriteLine(item);
        // Output to console for example purposes.
        Console.WriteLine($"    {item}");
    }
}
/* Output:
   A
     Aw, Kam Foo
   B
     Bankov, Peter
     Beebe, Ann
   E
     El Yassir, Mehdi
   G
     Garcia, Hugo
     Guy, Wey Yuan
     Garcia, Debra
     Gilchrist, Beth
     Giakoumakis, Leo
   H
```

```
Holm, Michael
L
Liu, Jinghao
M
Myrcha, Jacek
McLin, Nkenge
N
Noriega, Fabricio
P
Potra, Cristina
T
Toyoshima, Tim
*/
```

## 서로 다른 파일의 콘텐츠를 조인하는 방법

이 예제에서는 일치하는 키로 사용되는 공통 값을 공유하는 두 개의 쉼표로 구분된 파일의 데이터를 조인하는 방법을 보여 줍니다. 이 방법은 두 스프레드시트나 한 스프레드시트와 다른 형식으로 된 파일의 데이터를 하나의 새 파일로 결합해야 하는 경우에 유용할 수 있습니다. 모든 종류의 구조적 텍스트에서 작동하도록 예제를 수정할 수 있습니다.

다음 텍스트는 *scores.csv*의 콘텐츠를 보여 줍니다. 파일은 스프레드시트 데이터를 나타냅니다. 열 1은 학생 ID이고, 열 2-5는 시험 점수입니다.

```
txt

111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

다음 텍스트는 *names.csv*의 콘텐츠를 보여 줍니다. 이 파일은 학생의 성, 이름, 학생 ID가 포함된 스프레드시트를 나타냅니다.

```
txt

Omelchenko,Svetlana,111
O'Donnell,Claire,112
Mortensen,Sven,113
Garcia,Cesar,114
```

```
Garcia,Debra,115
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Hugo,118
Tucker,Lance,119
Adams,Terry,120
Zabokritski,Eugene,121
Tucker,Michael,122
```

관련 정보가 포함된 서로 다른 파일의 콘텐츠를 조인합니다. 파일 *names.csv*에는 학생 이름과 ID 번호가 포함되어 있습니다. 파일 *scores.csv*에는 ID와 4개의 테스트 점수 집합이 포함되어 있습니다. 다음 쿼리는 ID를 일치 키로 사용하여 점수를 학생 이름에 조인합니다. 코드는 다음 예에 표시됩니다.

C#

```
string[] names = File.ReadAllLines(@"names.csv");
string[] scores = File.ReadAllLines(@"scores.csv");

var scoreQuery = from name in names
                 let nameFields = name.Split(',')
                 from id in scores
                 let scoreFields = id.Split(',')
                 where Convert.ToInt32(nameFields[2]) ==
                       Convert.ToInt32(scoreFields[0])
                 select $"{nameFields[0]},{scoreFields[1]},"
                         $"{scoreFields[2]},${scoreFields[3]},${scoreFields[4]}";

Console.WriteLine("\r\nMerge two spreadsheets:");
foreach (string item in scoreQuery)
{
    Console.WriteLine(item);
}
Console.WriteLine("{0} total names in list", scoreQuery.Count());
/* Output:
Merge two spreadsheets:
Omelchenko, 97, 92, 81, 60
O'Donnell, 75, 84, 91, 39
Mortensen, 88, 94, 65, 91
Garcia, 97, 89, 85, 82
Garcia, 35, 72, 91, 70
Fakhouri, 99, 86, 90, 94
Feng, 93, 92, 80, 87
Garcia, 92, 90, 83, 78
Tucker, 68, 79, 88, 92
Adams, 99, 82, 81, 79
Zabokritski, 96, 85, 91, 60
Tucker, 94, 92, 91, 91
12 total names in list
*/
```

# CSV 텍스트 파일에서 열 값을 계산하는 방법

이 예제에서는 .csv 파일의 열에 대해 Sum, Average, Min 및 Max 등의 집계 계산을 수행하는 방법을 보여 줍니다. 여기 표시된 예제 원칙은 다른 형식의 구조화된 텍스트에 적용할 수 있습니다.

다음 텍스트는 `scores.csv`의 콘텐츠를 보여 줍니다. 첫 번째 열은 학생 ID를 나타내고 후속 열은 4개 시험의 점수를 나타낸다고 가정합니다.

txt

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

다음 텍스트는 [Split](#) 메서드를 사용하여 텍스트의 각 줄을 배열로 변환하는 방법을 보여 줍니다. 각 배열 요소는 열을 나타냅니다. 마지막으로 각 열의 텍스트가 숫자 표현으로 변환됩니다.

C#

```
public class SumColumns
{
    public static void SumCSVColumns(string fileName)
    {
        string[] lines = File.ReadAllLines(fileName);

        // Specifies the column to compute.
        int exam = 3;

        // Spreadsheet format:
        // Student ID    Exam#1  Exam#2  Exam#3  Exam#4
        // 111,          97,     92,     81,     60

        // Add one to exam to skip over the first column,
        // which holds the student ID.
        SingleColumn(lines, exam + 1);
        Console.WriteLine();
        MultiColumns(lines);
    }
}
```



```

    // Execute the query and cache the results to improve
    // performance.
    // ToArray could be used instead of ToList.
    var results = multiColQuery.ToList();

    // Find out how many columns you have in results.
    int columnCount = results[0].Count();

    // Perform aggregate calculations Average, Max, and
    // Min on each column.
    // Perform one iteration of the loop for each column
    // of scores.
    // You can use a for loop instead of a foreach loop
    // because you already executed the multiColQuery
    // query by calling ToList.
    for (int column = 0; column < columnCount; column++)
    {
        var results2 = from row in results
                      select row.ElementAt(column);
        double average = results2.Average();
        int max = results2.Max();
        int min = results2.Min();

        // Add one to column because the first exam is Exam #1,
        // not Exam #0.
        Console.WriteLine($"Exam #{column + 1} Average: {average:##.##}");
        High Score: {max} Low Score: {min}");
    }
}
/* Output:
Single Column Query:
Exam #4: Average:76.92 High Score:94 Low Score:39

Multi Column Query:
Exam #1 Average: 86.08 High Score: 99 Low Score: 35
Exam #2 Average: 86.42 High Score: 94 Low Score: 72
Exam #3 Average: 84.75 High Score: 91 Low Score: 65
Exam #4 Average: 76.92 High Score: 94 Low Score: 39
*/

```

탭으로 구분된 파일인 경우 `Split` 메서드의 인수를 `\t`로 업데이트하세요.

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할



## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

## 설명서 문제 열기

수도 있습니다. 자세한 내용은  
참여자 가이드를 참조하세요.

☞ 제품 사용자 의견 제공

# 방법: LINQ를 사용하여 문자열 쿼리

아티클 • 2024. 05. 02.

문자열은 문자 시퀀스로 저장됩니다. 문자 시퀀스로서 문자열은 LINQ를 사용하여 쿼리할 수 있습니다. 이 문서에서는 여러 다른 문자 또는 단어의 문자열을 쿼리하거나, 문자열을 필터링하거나, 쿼리를 정규식과 혼합하는 몇 가지 예제 쿼리를 보여줍니다.

## 문자열의 문자를 쿼리하는 방법

다음 예제에서는 문자열을 쿼리하여 문자열에 포함된 숫자 자릿수를 확인합니다.

C#

```
string aString = "ABCDE99F-J74-12-89A";

// Select only those characters that are numbers
var stringQuery = from ch in aString
                  where Char.IsDigit(ch)
                  select ch;

// Execute the query
foreach (char c in stringQuery)
    Console.Write(c + " ");

// Call the Count method on the existing query.
int count = stringQuery.Count();
Console.WriteLine($"Count = {count}");

// Select all characters before the first '-'
var stringQuery2 = aString.TakeWhile(c => c != '-');

// Execute the second query
foreach (char c in stringQuery2)
    Console.Write(c);
/* Output:
   Output: 9 9 7 4 1 2 8 9
   Count = 8
   ABCDE99F
*/
```

앞의 쿼리는 문자열을 문자 시퀀스로 처리하는 방법을 보여줍니다.

## 문자열에서 단어가 나오는 횟수를 세는 방법

다음 예제에서는 LINQ 쿼리를 사용하여 문자열에서 지정된 단어의 발생 수를 계산하는 방법을 보여줍니다. 계산을 수행하기 위해 먼저 [Split](#) 메서드를 호출하여 단어 배열을 만

듭니다. [Split](#) 메서드를 사용하는 경우 성능이 저하됩니다. 문자열 작업이 단어 단어 개수 계산인 경우 [Matches](#) 또는 [IndexOf](#) 메서드를 대신 사용하는 것이 좋습니다.

C#

```
string text = """
    Historically, the world of data and the world of objects
    have not been well integrated. Programmers work in C# or Visual Basic
    and also in SQL or XQuery. On the one side are concepts such as classes,
    objects, fields, inheritance, and .NET APIs. On the other side
    are tables, columns, rows, nodes, and separate languages for dealing
with
    them. Data types often require translation between the two worlds; there
are
    different standard functions. Because the object world has no notion of
query, a
    query can only be represented as a string without compile-time type
checking or
    IntelliSense support in the IDE. Transferring data from SQL tables or
XML trees to
    objects in memory is often tedious and error-prone.
""";
```

```
string searchTerm = "data";

//Convert the string into an array of words
char[] separators = ['.', '?', '!', ' ', ';', ':', ','];
string[] source = text.Split(separators,
StringSplitOptions.RemoveEmptyEntries);

// Create the query. Use the InvariantCultureIgnoreCase comparison to match
//"data" and "Data"
var matchQuery = from word in source
                 where word.Equals(searchTerm,
StringComparison.InvariantCultureIgnoreCase)
                 select word;

// Count the matches, which executes the query.
int wordCount = matchQuery.Count();
Console.WriteLine($""""{wordCount} occurrences(s) of the search term "
{searchTerm}" were found."""");
/* Output:
   3 occurrences(s) of the search term "data" were found.
*/
```

앞의 쿼리는 문자열을 단어 시퀀스로 분할한 후 단어 시퀀스로 보는 방법을 보여줍니다.

## 단어 또는 필드를 기준으로 텍스트 데이터를 정렬하거나 필터링하는 방법

다음 예제에서는 줄의 필드를 기준으로 쉼표로 구분된 값 등의 구조적 텍스트 줄을 정렬하는 방법을 보여 줍니다. 필드는 런타임에 동적으로 지정할 수 있습니다. scores.csv의 필드가 학생의 ID 번호와 일련의 시험 점수 4개를 나타낸다고 가정합니다.

txt

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

다음 쿼리는 두 번째 열에 저장된 첫 번째 시험의 점수를 기준으로 줄을 정렬합니다.

C#

```
// Create an IEnumerable data source
string[] scores = File.ReadAllLines("scores.csv");

// Change this to any value from 0 to 4.
int sortField = 1;

Console.WriteLine($"Sorted highest to lowest by field [{sortField}]:");

// Split the string and sort on field[num]
var scoreQuery = from line in scores
                  let fields = line.Split(',')
                  orderby fields[sortField] descending
                  select line;

foreach (string str in scoreQuery)
{
    Console.WriteLine(str);
}
/* Output (if sortField == 1):
Sorted highest to lowest by field [1]:
116, 99, 86, 90, 94
120, 99, 82, 81, 79
111, 97, 92, 81, 60
114, 97, 89, 85, 82
121, 96, 85, 91, 60
122, 94, 92, 91, 91
117, 93, 92, 80, 87
118, 92, 90, 83, 78
113, 88, 94, 65, 91
```

```
112, 75, 84, 91, 39
119, 68, 79, 88, 92
115, 35, 72, 91, 70
*/
```

앞의 쿼리는 문자열을 필드로 분할하고 개별 필드를 쿼리하여 문자열을 조작하는 방법을 보여줍니다.

## 특정 단어가 포함된 문장을 쿼리하는 방법

다음 예제에서는 지정된 각 단어 집합에 대해 일치하는 항목이 포함된 문장을 텍스트 파일에서 찾는 방법을 보여줍니다. 검색 용어의 배열이 하드 코딩되어 있지만 런타임에 동적으로 채워질 수도 있습니다. 쿼리가 "Historically" "data" 및 "integrated" 단어가 포함된 문장을 반환합니다.

C#

```
string text = """
Historically, the world of data and the world of objects
have not been well integrated. Programmers work in C# or Visual Basic
and also in SQL or XQuery. On the one side are concepts such as classes,
objects, fields, inheritance, and .NET APIs. On the other side
are tables, columns, rows, nodes, and separate languages for dealing with
them. Data types often require translation between the two worlds; there are
different standard functions. Because the object world has no notion of
query, a
query can only be represented as a string without compile-time type checking
or
IntelliSense support in the IDE. Transferring data from SQL tables or XML
trees to
objects in memory is often tedious and error-prone.
""";
```

```
// Split the text block into an array of sentences.
string[] sentences = text.Split(['.', '?', '!']);
```

```
// Define the search terms. This list could also be dynamically populated at
run time.
string[] wordsToMatch = [ "Historically", "data", "integrated" ];
```

```
// Find sentences that contain all the terms in the wordsToMatch array.
// Note that the number of terms to match is not specified at compile time.
char[] separators = [ '.', '?', '!', ' ', ';' , ':', ',' ];
var sentenceQuery = from sentence in sentences
    let w =
        sentence.Split(separators, StringSplitOptions.RemoveEmptyEntries)
            where w.Distinct().Intersect(wordsToMatch).Count() ==
wordsToMatch.Count()
    select sentence;
```

```
foreach (string str in sentenceQuery)
{
    Console.WriteLine(str);
}
/* Output:
Historically, the world of data and the world of objects have not been well
integrated
*/
```

쿼리는 먼저 텍스트를 문장으로 분할한 다음, 각 문장을 각 단어를 포함하는 문자열 배열로 분할합니다. 각 배열에 대해 `Distinct` 메서드가 모든 중복 단어를 제거한 다음 쿼리가 단어 배열 및 `wordsToMatch` 배열에 대해 `Intersect` 작업을 수행합니다. 교집합의 개수가 `wordsToMatch` 배열의 개수와 같으면 단어에서 모든 단어가 발견된 것이며 원래 문장이 반환됩니다.

`Split` 호출은 문자열에서 구분 기호를 제거하기 위해 문장 부호를 구분 기호로 사용합니다. 문장 부호를 제거하지 않은 경우, 예를 들자면 `wordsToMatch` 배열의 "Historically"와 일치하지 않는 "Historically," 문자열이 있을 수 있습니다. 소스 텍스트에서 찾은 문장 부호 유형에 따라 추가 구분 기호를 사용해야 할 수도 있습니다.

## LINQ 쿼리와 정규식을 결합하는 방법

다음 예제에서는 `Regex` 클래스를 사용하여 더 복잡한 텍스트 문자열 일치를 찾는 정규식을 작성하는 방법을 보여줍니다. LINQ 쿼리를 사용하면 손쉽게 정규식을 통해 검색하려는 파일을 정확히 필터링하고 결과를 구성할 수 있습니다.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

// Take a snapshot of the file system.
var fileList = from file in Directory.GetFiles(startFolder, "*.*",
SearchOption.AllDirectories)
    let fileInfo = new FileInfo(file)
    select fileInfo;

// Create the regular expression to find all things "Visual".
System.Text.RegularExpressions.Regex searchTerm =
    new System.Text.RegularExpressions.Regex(@"microsoft.net.
(sdk|workload)"); 

// Search the contents of each .htm file.
// Remove the where clause to find even more matchedValues!
// This query produces a list of files where a match
// was found, and a list of the matchedValues in that file.
// Note: Explicit typing of "Match" in select clause.
```

```

// This is required because MatchCollection is not a
// generic IEnumerable collection.
var queryMatchingFiles =
    from file in fileList
    where file.Extension == ".txt"
    let fileText = File.ReadAllText(file.FullName)
    let matches = searchTerm.Matches(fileText)
    where matches.Count > 0
    select new
    {
        name = file.FullName,
        matchedValues = from System.Text.RegularExpressions.Match match in
matches
            select match.Value
    };

// Execute the query.
Console.WriteLine($"""The term \"{searchTerm}\" was found in:""");

foreach (var v in queryMatchingFiles)
{
    // Trim the path a bit, then write
    // the file name in which a match was found.
    string s = v.name.Substring(startFolder.Length - 1);
    Console.WriteLine(s);

    // For this file, write out all the matching strings
    foreach (var v2 in v.matchedValues)
    {
        Console.WriteLine($"  {v2}");
    }
}

```

RegEx 검색에서 반환되는 MatchCollection 개체를 쿼리할 수도 있습니다. 각 일치 항목의 값만 결과로 생성됩니다. 하지만 LINQ를 사용하여 해당 컬렉션에 대한 모든 종류의 필터링, 정렬 및 그룹화를 수행할 수도 있습니다. MatchCollection은 제네릭이 아닌 IEnumerable 컬렉션이므로 쿼리에 범위 변수의 형식을 명시해야 합니다.

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

[설명서 문제 열기](#)

[제품 사용자 의견 제공](#)

# LINQ 및 컬렉션

아티클 • 2024. 04. 28.

대부분의 컬렉션은 요소의 시퀀스를 모델링합니다. LINQ를 사용하여 모든 컬렉션 형식을 쿼리할 수 있습니다. 다른 LINQ 메서드는 컬렉션의 요소를 찾고, 컬렉션의 요소에서 값을 계산하거나, 컬렉션 또는 해당 요소를 수정합니다. 이러한 예는 LINQ 메서드에 대해 알아보고 컬렉션이나 기타 데이터 원본에 이를 사용하는 방법을 배우는 데 도움이 됩니다.

## 두 목록 간의 차집합을 구하는 방법

이 예에서는 LINQ를 사용하여 두 개의 문자열 목록을 비교하고 첫 번째 컬렉션에는 있지만 두 번째 컬렉션에는 없는 줄을 출력하는 방법을 보여 줍니다. 첫 번째 이름 컬렉션은 *names1.txt* 파일에 저장됩니다.

txt

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

두 번째 이름 컬렉션은 *names2.txt* 파일에 저장됩니다. 일부 이름은 두 시퀀스 모두에 나타납니다.

txt

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkengne
El Yassir, Mehdi
```

다음 코드는 [Enumerable.Except](#) 메서드를 사용하여 두 번째 목록에 없는 첫 번째 목록의 요소를 찾는 방법을 보여 줍니다.

C#

```
// Create the IEnumerable data sources.  
string[] names1 = File.ReadAllLines("names1.txt");  
string[] names2 = File.ReadAllLines("names2.txt");  
  
// Create the query. Note that method syntax must be used here.  
var differenceQuery = names1.Except(names2);  
  
// Execute the query.  
Console.WriteLine("The following lines are in names1.txt but not  
names2.txt");  
foreach (string s in differenceQuery)  
    Console.WriteLine(s);  
/* Output:  
The following lines are in names1.txt but not names2.txt  
Potra, Cristina  
Noriega, Fabricio  
Aw, Kam Foo  
Toyoshima, Tim  
Guy, Wey Yuan  
Garcia, Debra  
*/
```

Except, Distinct, Union 및 Concat과 같은 일부 형식의 쿼리 작업은 메서드 기반 구문으로만 표현할 수 있습니다.

## 문자열 컬렉션을 결합하고 비교하는 방법

이 예제에서는 텍스트 줄이 포함된 파일을 병합하고 결과를 정렬하는 방법을 보여 줍니다. 특히, 두 개의 텍스트 줄 집합에 대한 연결, 합집합 및 교집합을 수행하는 방법을 보여 줍니다. 앞의 예에 표시된 것과 동일한 두 개의 텍스트 파일을 사용합니다. 코드는 Enumerable.Concat, Enumerable.Union 및 Enumerable.Except의 예를 보여 줍니다.

C#

```
//Put text files in your solution folder  
string[] fileA = File.ReadAllLines("names1.txt");  
string[] fileB = File.ReadAllLines("names2.txt");  
  
//Simple concatenation and sort. Duplicates are preserved.  
var concatQuery = fileA.Concat(fileB).OrderBy(s => s);  
  
// Pass the query variable to another function for execution.  
OutputQueryResults(concatQuery, "Simple concatenate and sort. Duplicates are  
preserved:");  
  
// Concatenate and remove duplicate names based on  
// default string comparer.
```

```

var uniqueNamesQuery = fileA.Union(fileB).OrderBy(s => s);
OutputQueryResults(uniqueNamesQuery, "Union removes duplicate names:");

// Find the names that occur in both files (based on
// default string comparer).
var commonNamesQuery = fileA.Intersect(fileB);
OutputQueryResults(commonNamesQuery, "Merge based on intersect:");

// Find the matching fields in each list. Merge the two
// results by using Concat, and then
// sort using the default string comparer.
string nameMatch = "Garcia";

var tempQuery1 = from name in fileA
    let n = name.Split(',')
    where n[0] == nameMatch
    select name;

var tempQuery2 = from name2 in fileB
    let n2 = name2.Split(',')
    where n2[0] == nameMatch
    select name2;

var nameMatchQuery = tempQuery1.Concat(tempQuery2).OrderBy(s => s);
OutputQueryResults(nameMatchQuery, $"""Concat based on partial name match "
{nameMatch}":""");

static void OutputQueryResults(IEnumerable<string> query, string message)
{
    Console.WriteLine(Environment.NewLine + message);
    foreach (string item in query)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine($"{query.Count()} total names in list");
}
/* Output:
   Simple concatenate and sort. Duplicates are preserved:
   Aw, Kam Foo
   Bankov, Peter
   Bankov, Peter
   Beebe, Ann
   Beebe, Ann
   El Yassir, Mehdi
   Garcia, Debra
   Garcia, Hugo
   Garcia, Hugo
   Giakoumakis, Leo
   Gilchrist, Beth
   Guy, Wey Yuan
   Holm, Michael
   Holm, Michael
   Liu, Jinghao
   McLin, Nkeng
   Myrcha, Jacek

```

```
Noriega, Fabricio  
Potra, Cristina  
Toyoshima, Tim  
20 total names in list
```

```
Union removes duplicate names:
```

```
Aw, Kam Foo  
Bankov, Peter  
Beebe, Ann  
El Yassir, Mehdi  
Garcia, Debra  
Garcia, Hugo  
Giakoumakis, Leo  
Gilchrist, Beth  
Guy, Wey Yuan  
Holm, Michael  
Liu, Jinghao  
McLin, Nkeng  
Myrcha, Jacek  
Noriega, Fabricio  
Potra, Cristina  
Toyoshima, Tim  
16 total names in list
```

```
Merge based on intersect:
```

```
Bankov, Peter  
Holm, Michael  
Garcia, Hugo  
Beebe, Ann  
4 total names in list
```

```
Concat based on partial name match "Garcia":
```

```
Garcia, Debra  
Garcia, Hugo  
Garcia, Hugo  
3 total names in list
```

```
*/
```

## 여러 소스로 개체 컬렉션을 채우는 방법

이 예제에서는 여러 소스의 데이터를 새 형식의 시퀀스에 병합하는 방법을 보여 줍니다.

### ① 참고

메모리 내 데이터 또는 파일 시스템의 데이터와 아직 데이터베이스에 있는 데이터를 조인하지 마세요. 이러한 도메인 간 조인을 사용하면 데이터베이스 쿼리 및 다른 소스 유형에 대해 조인 작업이 정의될 수 있는 다양한 방법으로 인해 정의되지 않은 결과가 발생할 수 있습니다. 또한 데이터베이스의 데이터 양이 충분히 큰 경우 이러한 작업으로 인해 메모리 부족 예외가 발생할 수 있는 위험이 있습니다. 데이터베이스

의 데이터를 메모리 내 데이터에 조인하려면 먼저 데이터베이스 쿼리에서 `ToList` 또는 `ToDictionary`를 호출한 다음 반환된 컬렉션에서 조인을 수행합니다.

이 예에서는 두 개의 파일을 사용합니다. 첫 번째 파일인 `names.csv`에는 학생 이름과 학생 ID가 포함되어 있습니다.

txt

```
Omelchenko,Svetlana,111
O'Donnell,Claire,112
Mortensen,Sven,113
Garcia,Cesar,114
Garcia,Debra,115
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Hugo,118
Tucker,Lance,119
Adams,Terry,120
Zabokritski,Eugene,121
Tucker,Michael,122
```

두 번째인 `scores.csv`에는 첫 번째 열에 학생 ID가 포함되고 그 뒤에 시험 점수가 표시됩니다.

txt

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

다음 예제에서는 명명된 레코드 `Student`를 사용하여 스프레드시트 데이터를 시뮬레이트하는 두 개의 메모리 내 문자열 컬렉션의 병합된 데이터를 .csv 형식으로 저장하는 방법을 보여 줍니다. ID는 학생의 점수를 매핑하는 열쇠로 사용됩니다.

C#

```
// Each line of names.csv consists of a last name, a first name, and an
// ID number, separated by commas. For example, Omelchenko,Svetlana,111
string[] names = File.ReadAllLines("names.csv");
```

```
// Each line of scores.csv consists of an ID number and four test
// scores, separated by commas. For example, 111, 97, 92, 81, 60
string[] scores = File.ReadAllLines("scores.csv");

// Merge the data sources using a named type.
// var could be used instead of an explicit type. Note the dynamic
// creation of a list of ints for the ExamScores member. The first item
// is skipped in the split string because it is the student ID,
// not an exam score.
IEnumerable<Student> queryNamesScores = from nameLine in names
                                             let splitName = nameLine.Split(',')
                                             from scoreLine in scores
                                             let splitScoreLine =
scoreLine.Split(',')
== Convert.ToInt32(splitScoreLine[0])
                                             select new Student
(
    FirstName: splitName[0],
    LastName: splitName[1],
    ID:
Convert.ToInt32(splitName[2]),
    ExamScores: (from scoreAsText in
splitScoreLine.Skip(1)
                                             select
Convert.ToInt32(scoreAsText)
).ToArray()
);

// Optional. Store the newly created student objects in memory
// for faster access in future queries. This could be useful with
// very large data files.
List<Student> students = queryNamesScores.ToList();

// Display each student's name and exam score average.
foreach (var student in students)
{
    Console.WriteLine($"The average score of {student.FirstName}
{student.LastName} is {student.ExamScores.Average()}.");
}
/* Output:
The average score of Omelchenko Svetlana is 82.5.
The average score of O'Donnell Claire is 72.25.
The average score of Mortensen Sven is 84.5.
The average score of Garcia Cesar is 88.25.
The average score of Garcia Debra is 67.
The average score of Fakhouri Fadi is 92.25.
The average score of Feng Hanying is 88.
The average score of Garcia Hugo is 85.75.
The average score of Tucker Lance is 81.75.
The average score of Adams Terry is 85.25.
The average score of Zabokritski Eugene is 83.
The average score of Tucker Michael is 92.
*/
```

`select` 절에서 각각의 새로운 `Student` 개체는 두 원본의 데이터에서 초기화됩니다.

쿼리 결과를 저장할 필요가 없다면 튜플이나 무명 형식이 명명된 형식보다 더 편리할 수 있습니다. 다음 예에서는 이전 예와 동일한 작업을 실행하지만 명명된 형식 대신 튜플을 사용합니다.

C#

```
// Merge the data sources by using an anonymous type.  
// Note the dynamic creation of a list of ints for the  
// ExamScores member. We skip 1 because the first string  
// in the array is the student ID, not an exam score.  
var queryNamesScores2 = from nameLine in names  
    let splitName = nameLine.Split(',')  
    from scoreLine in scores  
    let splitScoreLine = scoreLine.Split(',')  
    where Convert.ToInt32(splitName[2]) ==  
        Convert.ToInt32(splitScoreLine[0])  
    select (FirstName: splitName[0],  
            LastName: splitName[1],  
            ExamScores: (from scoreAsText in  
splitScoreLine.Skip(1)  
                select  
Convert.ToInt32(scoreAsText))  
            .ToList()  
        );  
  
// Display each student's name and exam score average.  
foreach (var student in queryNamesScores2)  
{  
    Console.WriteLine($"The average score of {student.FirstName}  
{student.LastName} is {student.ExamScores.Average()}.");  
}
```

## LINQ를 사용하여 ArrayList를 쿼리하는 방법

LINQ를 사용하여 `ArrayList` 등의 제네릭이 아닌 `IEnumerable` 컬렉션을 쿼리하는 경우 컬렉션에 있는 개체의 특정 형식을 반영하도록 범위 변수의 형식을 명시적으로 선언해야 합니다. `Student` 개체의 `ArrayList`가 있는 경우 `from` 절은 다음과 같아야 합니다.

C#

```
var query = from Student s in arrList  
//...
```

범위 변수의 형식을 지정하여 `ArrayList`의 각 항목을 `Student`로 캐스팅합니다.

명시적 형식 범위 변수를 쿼리 식에 사용하는 것은 [Cast](#) 메서드 호출과 같습니다. 지정된 캐스트를 수행할 수 없으면 [Cast](#)에서 예외가 throw됩니다. [Cast](#) 및 [OfType](#)은 제네릭이 아닌 [IEnumerable](#) 형식에서 작동하는 두 가지 표준 쿼리 연산자 메서드입니다. 자세한 내용은 [LINQ 쿼리 작업의 형식 관계](#)를 참조하세요. 다음 예에서는 [ArrayList](#)에 대한 쿼리를 보여 줍니다.

C#

```
ArrayList arrList = new ArrayList();
arrList.Add(
    new Student
    (
        FirstName: "Svetlana",
        LastName: "Omelchenko",
        ExamScores: new int[] { 98, 92, 81, 60 }
    ));
arrList.Add(
    new Student
    (
        FirstName: "Claire",
        LastName: "O'Donnell",
        ExamScores: new int[] { 75, 84, 91, 39 }
    ));
arrList.Add(
    new Student
    (
        FirstName: "Sven",
        LastName: "Mortensen",
        ExamScores: new int[] { 88, 94, 65, 91 }
    ));
arrList.Add(
    new Student
    (
        FirstName: "Cesar",
        LastName: "Garcia",
        ExamScores: new int[] { 97, 89, 85, 82 }
    ));

var query = from Student student in arrList
            where student.ExamScores[0] > 95
            select student;

foreach (Student s in query)
    Console.WriteLine(s.LastName + ":" + s.ExamScores[0]);
```

 GitHub에서 Microsoft와 공동 작업

.NET

.NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

제공해 주세요.

💡 설명서 문제 열기

↗️ 제품 사용자 의견 제공

# LINQ를 확장하는 방법

아티클 • 2024. 05. 02.

모든 LINQ 기반 메서드는 두 가지 유사한 패턴 중 하나를 따릅니다. 이러한 메서드는 열거 가능한 시퀀스를 수행합니다. 다른 시퀀스 또는 단일 값을 반환합니다. 모양의 일관성을 사용하면 비슷한 모양의 메서드를 작성하여 LINQ를 확장할 수 있습니다. 실제로 .NET 라이브러리는 LINQ가 처음 도입된 이후 많은 .NET 릴리스에서 새로운 메서드를 얻었습니다. 이 문서에서는 동일한 패턴을 따르는 사용자 고유의 메서드를 작성하여 LINQ를 확장하는 예를 확인합니다.

## LINQ 쿼리에 대한 사용자 지정 메서드 추가

`IEnumerable<T>` 인터페이스에 확장 메서드를 추가하여 LINQ 쿼리에 사용하는 메서드 세트를 확장합니다. 예를 들어 표준 평균 또는 최대 작업 외에 사용자 지정 집계 메서드를 만들어 값 시퀀스에서 단일 값을 계산합니다. 값 시퀀스에 대한 특정 데이터 변환 또는 사용자 지정 필터로 작동하고 새 시퀀스를 반환하는 메서드도 만듭니다. 이러한 메서드의 예로는 `Distinct`, `Skip` 및 `Reverse`가 있습니다.

`IEnumerable<T>` 인터페이스를 확장하면 사용자 지정 메서드를 열거 가능한 컬렉션에 적용할 수 있습니다. 자세한 내용은 [확장 메서드](#)를 참조하세요.

집계 메서드는 값 집합에서 하나의 값을 계산합니다. LINQ는 `Average`, `Min` 및 `Max`를 포함하여 여러 집계 메서드를 제공합니다. `IEnumerable<T>` 인터페이스에 확장 메서드를 추가하여 고유한 집계 메서드를 만들 수 있습니다.

다음 코드 예제에서는 `double` 형식의 숫자 시퀀스에 대한 중앙값을 계산하는 `Median`이라는 확장 메서드를 만드는 방법을 보여 줍니다.

C#

```
public static class EnumerableExtension
{
    public static double Median(this IEnumerable<double>? source)
    {
        if (source is null || !source.Any())
        {
            throw new InvalidOperationException("Cannot compute median for a
null or empty set.");
        }

        var sortedList =
            source.OrderBy(number => number).ToList();

        int itemIndex = sortedList.Count / 2;
    }
}
```

```

        if (sortedList.Count % 2 == 0)
    {
        // Even number of items.
        return (sortedList[itemIndex] + sortedList[itemIndex - 1]) / 2;
    }
    else
    {
        // Odd number of items.
        return sortedList[itemIndex];
    }
}
}

```

`IEnumerable<T>` 인터페이스에서 다른 집계 메서드를 호출하는 것과 같은 방식으로 열거 가능한 컬렉션에 대해 이 확장 메서드를 호출합니다.

다음 코드 예제에서는 `double` 형식의 배열에 대해 `Median` 메서드를 사용하는 방법을 보여 줍니다.

C#

```

double[] numbers = [1.9, 2, 8, 4, 5.7, 6, 7.2, 0];
var query = numbers.Median();

Console.WriteLine($"double: Median = {query}");
// This code produces the following output:
//      double: Median = 4.85

```

다양한 형식의 시퀀스를 허용하도록 집계 메서드를 오버로드할 수 있습니다. 표준 접근법은 각 형식에 대한 오버로드를 만드는 것입니다. 또 다른 접근법은 제네릭 형식을 사용하고 대리자를 통해 이를 특정 형식으로 변환할 오버로드를 만드는 것입니다. 두 가지 접근법을 결합할 수도 있습니다.

지원하려는 각 형식에 대한 특정 오버로드를 만들 수 있습니다. 다음 코드 예제에서는 `int` 형식에 대한 `Median` 메서드의 오버로드를 보여 줍니다.

C#

```

// int overload
public static double Median(this IEnumerable<int> source) =>
    (from number in source select (double)number).Median();

```

이제 다음 코드와 같이 `integer` 및 `double` 형식에 대한 `Median` 오버로드를 호출할 수 있습니다.

C#

```

double[] numbers1 = [1.9, 2, 8, 4, 5.7, 6, 7.2, 0];
var query1 = numbers1.Median();

Console.WriteLine($"double: Median = {query1}");

int[] numbers2 = [1, 2, 3, 4, 5];
var query2 = numbers2.Median();

Console.WriteLine($"int: Median = {query2}");
// This code produces the following output:
//      double: Median = 4.85
//      int: Median = 3

```

개체의 제네릭 시퀀스를 허용하는 오버로드를 만들 수도 있습니다. 이 오버로드는 대리자를 매개 변수로 사용하여 제네릭 형식의 개체 시퀀스를 특정 형식으로 변환합니다.

다음 코드에서는 `Func<T,TResult>` 대리자를 매개 변수로 사용하는 `Median` 메서드의 오버로드를 보여 줍니다. 이 대리자는 제네릭 형식 `T`의 개체를 사용하고 `double` 형식의 개체를 반환합니다.

C#

```

// generic overload
public static double Median<T>(
    this IEnumerable<T> numbers, Func<T, double> selector) =>
    (from num in numbers select selector(num)).Median();

```

이제 임의 형식의 개체 시퀀스에 대해 `Median` 메서드를 호출할 수 있습니다. 형식에 자체 메서드 오버로드가 없으면 대리자 매개 변수를 전달해야 합니다. C#에서는 이 목적으로 람다 식을 사용할 수 있습니다. 또한 Visual Basic에서만, 메서드 호출 대신 `Aggregate` 또는 `Group By` 절을 사용할 경우 범위 내에 있는 값 또는 식을 이 절에 전달할 수 있습니다.

다음 예제 코드에서는 정수 배열 및 문자열 배열에 대해 `Median` 메서드를 호출하는 방법을 보여 줍니다. 문자열의 경우 배열에서 문자열 길이의 중앙값이 계산됩니다. 예제에서는 각 경우에 `Func<T,TResult>` 대리자 매개 변수를 `Median` 메서드에 전달하는 방법을 보여 줍니다.

C#

```

int[] numbers3 = [1, 2, 3, 4, 5];

/*
    You can use the num => num lambda expression as a parameter for the
    Median method
        so that the compiler will implicitly convert its value to double.
        If there is no implicit conversion, the compiler will display an error

```

```

message.
*/
var query3 = numbers3.Median(num => num);

Console.WriteLine($"int: Median = {query3}");

string[] numbers4 = ["one", "two", "three", "four", "five"];

// With the generic overload, you can also use numeric properties of
// objects.
var query4 = numbers4.Median(str => str.Length);

Console.WriteLine($"string: Median = {query4}");
// This code produces the following output:
//      int: Median = 3
//      string: Median = 4

```

값 시퀀스를 반환하는 사용자 지정 쿼리 메서드를 사용하여 `IEnumerable<T>` 인터페이스를 확장할 수 있습니다. 이 경우 메서드는 `IEnumerable<T>` 형식의 컬렉션을 반환해야 합니다. 이러한 메서드는 필터 또는 데이터 변환을 값 시퀀스에 적용하는 데 사용될 수 있습니다.

다음 예제에서는 모든 기타 요소를 첫 번째 인수부터 반환하는 `AlternateElements` 확장 메서드를 만드는 방법을 보여 줍니다.

C#

```

// Extension method for the IEnumerable<T> interface.
// The method returns every other element of a sequence.
public static IEnumerable<T> AlternateElements<T>(this IEnumerable<T>
source)
{
    int index = 0;
    foreach (T element in source)
    {
        if (index % 2 == 0)
        {
            yield return element;
        }

        index++;
    }
}

```

다음 코드와 같이 `IEnumerable<T>` 인터페이스에서 다른 메서드를 호출할 때와 같은 방 법으로 열거 가능한 모든 컬렉션에 대해 이 확장 메서드를 호출할 수 있습니다.

C#

```

string[] strings = ["a", "b", "c", "d", "e"];

var query5 = stringsAlternateElements();

foreach (var element in query5)
{
    Console.WriteLine(element);
}

// This code produces the following output:
//      a
//      c
//      e

```

## 연속 키를 기준으로 결과 그룹화

다음 예제에서는 연속 키의 하위 시퀀스를 나타내는 청크로 요소를 그룹화하는 방법을 보여 줍니다. 예를 들어 다음과 같은 키-값 쌍의 다음 시퀀스가 제공된다고 가정합니다.

 테이블 확장

키	값
A	3x3 이미지 등과 같은
A	think
A	that
B	Linq
C	is
A	really
B	cool
B	!

다음과 같은 그룹이 이 순서로 만들어집니다.

1. We, think, that
2. Linq
3. is
4. really
5. cool, !

솔루션은 스트리밍 방식으로 결과를 반환하는 스레드로부터 안전한 확장 메서드로 구현됩니다. 솔루션은 소스 시퀀스를 거치면서 그룹을 생성합니다. `group` 또는 `orderby` 연산자와 달리 전체 시퀀스를 읽기 전에 호출자에 그룹을 반환하기 시작할 수 있습니다. 다음 예제에서는 확장 메서드 및 이를 사용하는 클라이언트 코드를 보여 줍니다.

C#

```
public static class ChunkExtensions
{
    public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSouce,
    TKey>(
        this IEnumberable<TSouce> source,
        Func<TSouce, TKey> keySelector) =>
            source.ChunkBy(keySelector, EqualityComparer<TKey>.Default);

    public static IEnumberable<IGrouping<TKey, TSource>> ChunkBy<TSouce,
    TKey>(
        this IEnumberable<TSouce> source,
        Func<TSouce, TKey> keySelector,
        IEqualityComparer<TKey> comparer)
    {
        // Flag to signal end of source sequence.
        const bool noMoreSourceElements = true;

        // Auto-generated iterator for the source array.
        IEnumberator<TSouce>? enumerator = source.GetEnumberator();

        // Move to the first element in the source sequence.
        if (!enumerator.MoveNext())
        {
            yield break;           // source collection is empty
        }

        while (true)
        {
            var key = keySelector(enumerator.Current);

            Chunk<TKey, TSource> current = new(key, enumerator, value =>
comparer.Equals(key, keySelector(value)));

            yield return current;

            if (current.CopyAllChunkElements() == noMoreSourceElements)
            {
                yield break;
            }
        }
    }
}
```

C#

```

public static class GroupByContiguousKeys
{
    // The source sequence.
    static readonly KeyValuePair<string, string>[] list = [
        new("A", "We"),
        new("A", "think"),
        new("A", "that"),
        new("B", "LINQ"),
        new("C", "is"),
        new("A", "really"),
        new("B", "cool"),
        new("B", "!")
    ];

    // Query variable declared as class member to be available
    // on different threads.
    static readonly IEnumerable<IGrouping<string, KeyValuePair<string,
    string>>> query =
        list.ChunkBy(p => p.Key);

    public static void GroupByContiguousKeys1()
    {
        // ChunkBy returns IGrouping objects, therefore a nested
        // foreach loop is required to access the elements in each "chunk".
        foreach (var item in query)
        {
            Console.WriteLine($"Group key = {item.Key}");
            foreach (var inner in item)
            {
                Console.WriteLine($"{inner.Value}");
            }
        }
    }
}

```

## ChunkExtensions 클래스

제공된 `ChunkExtensions` 클래스 구현 코드에서 `ChunkBy` 메서드의 `while(true)` 루프가 소스 시퀀스에서 반복되고 각 청크의 복사본을 만듭니다. 각 패스에서 반복기는 소스 시퀀스에서 `Chunk` 개체로 표시되는 다음 "청크"의 첫 번째 요소로 이동합니다. 이 루프는 큐리를 실행하는 외부 `foreach` 루프에 해당합니다. 이 루프에서 코드는 다음 작업을 수행합니다.

1. 현재 청크의 키를 가져와서 `key` 변수에 할당합니다. 소스 반복기는 일치하지 않는 키가 있는 요소를 찾을 때까지 소스 시퀀스를 사용합니다.
2. 새 청크(그룹) 개체를 만들고 `current` 변수에 저장합니다. 여기에는 현재 소스 요소의 복사본인 `GroupItem`이 하나 있습니다.

3. 해당 청크를 반환합니다. 청크는 `ChunkBy` 메서드의 반환 값인 `IGrouping< TKey, TSource >` 입니다. 청크에는 소스 시퀀스의 첫 번째 요소만 있습니다. 나머지 요소는 클라이언트 코드 `foreach`가 이 청크 위에 있는 경우에만 반환됩니다. 자세한 내용은 `Chunk.GetEnumerator`를 참조하세요.
4. 다음 여부를 확인합니다.

- 청크가 모든 소스 요소의 복사본을 만들었는지 여부, 또는
- 반복기가 소스 시퀀스의 끝에 도달했는지 여부.

5. 호출자가 모든 청크 항목을 열거한 경우 `Chunk.GetEnumerator` 메서드는 모든 청크 항목을 복사합니다. `Chunk.GetEnumerator` 루프가 청크의 모든 요소를 열거하지 않은 경우 별도의 스레드에서 호출할 수 있는 클라이언트의 반복기가 손상되지 않도록 지금 수행해야 합니다.

## Chunk 클래스

`Chunk` 클래스는 키가 같은 하나 이상의 원본 요소로 구성된 연속 그룹입니다. 청크에는 키와 원본 시퀀스의 요소 복사본인 `ChunkItem` 개체 목록이 있습니다.

C#

```
class Chunk< TKey, TSource > : IGrouping< TKey, TSource >
{
    // INVARIANT: DoneCopyingChunk == true ||
    // (predicate != null && predicate(enumerator.Current) &&
    current.Value == enumerator.Current)

    // A Chunk has a linked list of ChunkItems, which represent the elements
    // in the current chunk. Each ChunkItem
    // has a reference to the next ChunkItem in the list.
    class ChunkItem
    {
        public ChunkItem(TSource value) => Value = value;
        public readonly TSource Value;
        public ChunkItem? Next;
    }

    TKey Key { get; }

    // Stores a reference to the enumerator for the source sequence
    private IEnumerator< TSource > enumerator;

    // A reference to the predicate that is used to compare keys.
    private Func< TSource, bool > predicate;

    // Stores the contents of the first source element that
    // belongs with this chunk.
    private readonly ChunkItem head;
```

```

// End of the list. It is repositioned each time a new
// ChunkItem is added.
private ChunkItem? tail;

// Flag to indicate the source iterator has reached the end of the
source sequence.
internal bool isLastSourceElement;

// Private object for thread synchronization
private readonly object m_Lock;

// REQUIRES: enumerator != null && predicate != null
public Chunk(TKey key, [DisallowNull] IEnumrator<TSource> enumerator,
[DisallowNull] Func<TSource, bool> predicate)
{
    Key = key;
    this.enumerator = enumerator;
    this.predicate = predicate;

    // A Chunk always contains at least one element.
    head = new ChunkItem(enumerator.Current);

    // The end and beginning are the same until the list contains > 1
elements.
    tail = head;

    m_Lock = new object();
}

// Indicates that all chunk elements have been copied to the list of
ChunkItems.
private bool DoneCopyingChunk => tail == null;

// Adds one ChunkItem to the current group
// REQUIRES: !DoneCopyingChunk && lock(this)
private void CopyNextChunkElement()
{
    // Try to advance the iterator on the source sequence.
    isLastSourceElement = !enumerator.MoveNext();

    // If we are (a) at the end of the source, or (b) at the end of the
current chunk
    // then null out the enumerator and predicate for reuse with the
next chunk.
    if (isLastSourceElement || !predicate(enumerator.Current))
    {
        enumerator = default!;
        predicate = default!;
    }
    else
    {
        tail!.Next = new ChunkItem(enumerator.Current);
    }
}

```

```

        // tail will be null if we are at the end of the chunk elements
        // This check is made in DoneCopyingChunk.
        tail = tail!.Next;
    }

    // Called after the end of the last chunk was reached.
    internal bool CopyAllChunkElements()
    {
        while (true)
        {
            lock (m_Lock)
            {
                if (DoneCopyingChunk)
                {
                    return isLastSourceElement;
                }
                else
                {
                    CopyNextChunkElement();
                }
            }
        }
    }

    // Stays just one step ahead of the client requests.
    public IEnumrator<TSource> GetEnumrator()
    {
        // Specify the initial element to enumerate.
        ChunkItem? current = head;

        // There should always be at least one ChunkItem in a Chunk.
        while (current != null)
        {
            // Yield the current item in the list.
            yield return current.Value;

            // Copy the next item from the source sequence,
            // if we are at the end of our local list.
            lock (m_Lock)
            {
                if (current == tail)
                {
                    CopyNextChunkElement();
                }
            }

            // Move to the next ChunkItem in the list.
            current = current.Next;
        }
    }

    System.Collections.IEnumrator
    System.Collections.IEnumerable.GetEnumrator() => GetEnumrator();
}

```

각 `ChunkItem` (`ChunkItem` 클래스로 표시됨)에는 목록에 다음 `ChunkItem`에 대한 참조가 있습니다. 목록은 이 청크에 속하는 첫 번째 소스 요소의 내용을 저장하는 `head`와 목록의 끝인 `tail`로 구성됩니다. 새 `chunkItem`이 추가될 때마다 꼬리의 위치가 변경됩니다. 다음 요소의 키가 현재 청크의 키와 일치하지 않거나 소스에 더 이상 요소가 없는 경우, 연결된 목록의 꼬리가 `CopyNextChunkElement` 메서드에서 `null`로 설정됩니다.

`Chunk` 클래스의 `CopyNextChunkElement` 메서드는 현재 항목 그룹에 하나의 `chunkItem`을 추가합니다. 소스 시퀀스에서 반복기를 진행하려고 합니다. `MoveNext()` 메서드가 `false`를 반환하면 반복은 끝에 있고 `isLastSourceElement` 가 `true`로 설정됩니다.

`CopyAllChunkElements` 메서드는 마지막 청크의 끝에 도달한 후 호출됩니다. 먼저 소스 시퀀스에 더 많은 요소가 있는지 확인합니다. 더 많은 요소가 있는 경우 이 청크에 대한 열거자가 소진되면 `true`를 반환합니다. 이 메서드에서 프라이빗 `DoneCopyingChunk` 필드가 `true`로 확인될 때 `isLastSourceElement`가 `false`이면 외부 반복기에 반복을 계속하도록 신호를 보냅니다.

내부 `foreach` 루프가 `Chunk` 클래스의 `GetEnumerator` 메서드를 호출합니다. 이 메서드는 클라이언트 요청보다 한 요소 앞서 있습니다. 클라이언트가 목록의 이전 마지막 요소를 요청한 후에만 청크의 다음 요소를 추가합니다.

### ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.



### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

#### 🌟 설명서 문제 열기

#### ↗️ 제품 사용자 의견 제공

# 런타임 상태 기반 쿼리

아티클 • 2024. 04. 28.

대부분의 LINQ 쿼리에서 쿼리의 일반적인 모양은 코드에서 설정됩니다. `where` 절을 사용하여 항목을 필터링하고, `orderby`를 사용하여 출력 컬렉션을 정렬하고, 항목을 그룹화하거나 일부 계산을 수행할 수 있습니다. 코드에서는 필터, 정렬 키 또는 쿼리의 일부인 기타식에 대한 매개 변수를 제공할 수 있습니다. 그러나 쿼리의 전체 모양은 변경할 수 없습니다. 이 문서에서는 런타임 시 쿼리 형태를 수정하기 위해 `System.Linq.IQueryable<T>` 인터페이스와 이를 구현하는 형식을 사용하는 기술을 알아봅니다.

이러한 기술을 사용하여 런타임에 쿼리를 빌드합니다. 여기서 일부 사용자 입력 또는 런타임 상태는 쿼리의 일부로 사용하려는 쿼리 메서드를 변경합니다. 쿼리 절을 추가, 제거 또는 편집하여 쿼리를 편집하려고 합니다.

## ① 참고

`using System.Linq.Expressions;` 및 `using static System.Linq.Expressions.Expression;` 을 .cs 파일의 맨 위에 추가해야 합니다.

데이터 원본에 대해 `IQueryable` 또는 `IQueryable<T>`를 정의하는 코드를 고려합니다.

C#

```
string[] companyNames = [
    "Consolidated Messenger", "Alpine Ski House", "Southridge Video",
    "City Power & Light", "Coho Winery", "Wide World Importers",
    "Graphic Design Institute", "Adventure Works", "Humongous Insurance",
    "Woodgrove Bank", "Margie's Travel", "Northwind Traders",
    "Blue Yonder Airlines", "Trey Research", "The Phone Company",
    "Wingtip Toys", "Lucerne Publishing", "Fourth Coffee"
];

// Use an in-memory array as the data source, but the IQueryable could have
// come
// from anywhere -- an ORM backed by a database, a web request, or any other
// LINQ provider.
IQueryable<string> companyNamesSource = companyNames.AsQueryable();
var fixedQry = companyNames.OrderBy(x => x);
```

앞의 코드를 실행할 때마다 동일한 쿼리가 실행됩니다. 쿼리를 확장하거나 수정하는 방법을 알아봅시다. 기본적으로 `IQueryable`에는 다음 두 가지 구성 요소가 있습니다.

- `Expression`—식 트리 형식으로 현재 쿼리 구성 요소의 언어 중립적 및 데이터 소스 중립적 표현입니다.

- **Provider** 현재 쿼리를 값 또는 값 세트로 구체화하는 방법을 인식하는 LINQ 공급자 의 인스턴스입니다.

동적 쿼리의 컨텍스트에서 공급자는 일반적으로 동일하게 유지됩니다. 쿼리의 식 트리는 쿼리마다 다릅니다.

식 트리는 변경할 수 없습니다. 다른 식 트리와 다른 쿼리를 원하는 경우 기존 식 트리를 새 식 트리로 변환해야 합니다. 다음 섹션에서는 런타임 상태에 대한 응답으로 다르게 쿼리하는 특정 기술을 설명합니다.

- 식 트리 내에서 런타임 상태 사용
- 추가 LINQ 메서드 호출
- LINQ 메서드에 전달된 식 트리 다양화
- Expression에서 팩터리 메서드를 사용하여 `Expression<TDelegate>` 식 트리 생성
- `IQueryable`의 식 트리에 메서드 호출 노드 추가
- 문자열을 생성하고 [동적 LINQ 라이브러리](#) 사용

각 기술은 더 많은 기능을 제공하지만 복잡성이 증가합니다.

## 식 트리 내에서 런타임 상태 사용

동적으로 쿼리하는 가장 간단한 방법은 다음 코드 예의 `length`와 같은 폐쇄형 변수를 통해 쿼리에서 직접 런타임 상태를 참조하는 것입니다.

C#

```
var length = 1;
var qry = companyNamesSource
    .Select(x => x.Substring(0, length))
    .Distinct();

Console.WriteLine(string.Join(", ", qry));
// prints: C, A, S, W, G, H, M, N, B, T, L, F

length = 2;
Console.WriteLine(string.Join(", ", qry));
// prints: Co, Al, So, Ci, Wi, Gr, Ad, Hu, Wo, Ma, No, Bl, Tr, Th, Lu, Fo
```

내부 식 트리와 쿼리는 수정되지 않습니다. 쿼리는 `length` 값이 변경되었기 때문에 다른 값을 반환합니다.

## 추가 LINQ 메서드 호출

일반적으로 `Queryable`의 기본 제공 LINQ 메서드는 다음 두 단계를 수행합니다.

- 메서드 호출을 나타내는 [MethodCallExpression](#)으로 현재 식 트리를 래핑합니다.
- 래핑된 식 트리를 공급자에게 다시 전달하여 공급자의 [IQueryProvider.Execute](#) 메서드를 통해 값을 반환하거나 [IQueryProvider.CreateQuery](#) 메서드를 통해 변환된 쿼리 개체를 반환합니다.

원래 쿼리를 [System.Linq.IQueryable<T>](#) 반환 메서드의 결과로 바꿔서 새 쿼리를 가져올 수 있습니다. 다음 예와 같이 런타임 상태를 사용할 수 있습니다.

```
C#  
  
// bool sortByLength = /* ... */;  
  
var qry = companyNamesSource;  
if (sortByLength)  
{  
    qry = qry.OrderBy(x => x.Length);  
}
```

## LINQ 메서드에 전달된 식 트리 다양화

런타임 상태에 따라 다양한 식을 LINQ 메서드에 전달할 수 있습니다.

```
C#  
  
// string? startsWith = /* ... */;  
// string? endsWith = /* ... */;  
  
Expression<Func<string, bool>> expr = (startsWith, endsWith) switch  
{  
    ("" or null, "" or null) => x => true,  
    (_, "" or null) => x => x.StartsWith(startsWith),  
    ("" or null, _) => x => x.EndsWith(endsWith),  
    (_, _) => x => x.StartsWith(startsWith) || x.EndsWith(endsWith)  
};  
  
var qry = companyNamesSource.Where(expr);
```

[LinqKit](#)의 [PredicateBuilder](#)와 같은 다른 라이브러리를 사용하여 다양한 하위 식을 구성할 수도 있습니다.

```
C#  
  
// This is functionally equivalent to the previous example.  
  
// using LinqKit;  
// string? startsWith = /* ... */;  
// string? endsWith = /* ... */;
```

```

Expression<Func<string, bool>>? expr = PredicateBuilder.New<string>(false);
var original = expr;
if (!string.IsNullOrEmpty(startsWith))
{
    expr = expr.Or(x => x.StartsWith(startsWith));
}
if (!string.IsNullOrEmpty(endsWith))
{
    expr = expr.Or(x => x.EndsWith(endsWith));
}
if (expr == original)
{
    expr = x => true;
}

var qry = companyNamesSource.Where(expr);

```

## 팩터리 메서드를 사용하여 식 트리 및 쿼리 생성

지금까지의 모든 예에서는 컴파일 시간의 요소 형식(`string`)과 쿼리 형식(`IQueryable<string>`)을 알 수 있었습니다. 요소 형식에 따라 모든 요소 형식의 쿼리에 구성 요소를 추가하거나 다른 구성 요소를 추가할 수 있습니다.

`System.Linq.Expressions.Expression`에서 팩터리 메서드를 사용하여 처음부터 식 트리를 만들어서 런타임에 식을 특정 요소 형식에 맞게 조정할 수 있습니다.

## 식<TDelegate> 구문

LINQ 메서드 중 하나에 전달할 식을 생성하는 경우 실제로는 `System.Linq.Expressions.Expression<TDelegate>`의 인스턴스를 생성하는 것입니다. 여기서 `TDelegate`는 `Func<string, bool>`, `Action` 또는 사용자 지정 대리자 형식과 같은 대리자 형식입니다.

`System.Linq.Expressions.Expression<TDelegate>`는 다음 예와 같이 완전한 람다 식을 나타내는 `LambdaExpression`에서 상속됩니다.

C#

```
Expression<Func<string, bool>> expr = x => x.StartsWith("a");
```

`LambdaExpression`에는 다음 두 가지 구성 요소가 있습니다.

1. 매개 변수 목록(`(string x)`)은 `Parameters` 속성으로 표시됩니다.
2. 본문(`x.StartsWith("a")`)은 `Body` 속성으로 표시됩니다.

`Expression<TDelegate>`를 생성하는 기본 단계는 다음과 같습니다.

1. `Parameter` 팩터리 메서드를 사용하여 람다 식에서 각 매개 변수(있는 경우)에 대해 `ParameterExpression` 개체를 정의합니다.

C#

```
ParameterExpression x = Parameter(typeof(string), "x");
```

2. 정의된 `ParameterExpression`과 `Expression`의 팩터리 메서드를 사용하여 `LambdaExpression`의 본문을 구성합니다. 예를 들어 `x.StartsWith("a")`를 나타내는 식은 다음과 같이 생성할 수 있습니다.

C#

```
Expression body = Call(
    x,
    typeof(string).GetMethod("StartsWith", [typeof(string)])!,
    Constant("a")
);
```

3. 적절한 `Lambda` 팩터리 메서드 오버로드를 사용하여 컴파일 시간 형식의 `Expression<TDelegate>`에 매개 변수와 본문을 래핑합니다.

C#

```
Expression<Func<string, bool>> expr = Lambda<Func<string, bool>>(body,
    x);
```

다음 섹션에서는 LINQ 메서드에 전달하기 위해 `Expression<TDelegate>`를 생성하려는 시나리오를 설명합니다. 팩터리 메서드를 사용하여 이를 수행하는 방법에 대한 완전한 예를 제공합니다.

## 런타임 시 전체 쿼리 구문

여러 엔터티 형식에 작동하는 쿼리를 작성하려고 합니다.

C#

```
record Person(string LastName, string FirstName, DateTime DateOfBirth);
record Car(string Model, int Year);
```

해당 엔터티 형식에 대해 해당 `string` 필드 중 하나에 지정된 텍스트가 포함된 엔터티만 필터링하고 반환하려고 합니다. `Person`의 경우 `FirstName` 및 `LastName` 속성을 검색하려고 합니다.

C#

```
string term = /* ... */;
var personsQry = new List<Person>()
    .AsQueryable()
    .Where(x => x.FirstName.Contains(term) || x.LastName.Contains(term));
```

그러나 `Car`의 경우 `Model` 속성만 검색하려고 합니다.

C#

```
string term = /* ... */;
var carsQry = new List<Car>()
    .AsQueryable()
    .Where(x => x.Model.Contains(term));
```

`IQueryable<Person>` 및 `IQueryable<Car>`에 대해 하나씩 사용자 지정 함수를 작성할 수 있지만, 다음 함수는 특정 요소 형식과 관계없이 이 필터링을 기준 쿼리에 추가합니다.

C#

```
// using static System.Linq.Expressions.Expression;

IQueryable<T> TextFilter<T>(IQueryable<T> source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }

    // T is a compile-time placeholder for the element type of the query.
    Type elementType = typeof(T);

    // Get all the string properties on this specific type.
    PropertyInfo[] stringProperties = elementType
        .GetProperties()
        .Where(x => x.PropertyType == typeof(string))
        .ToArray();
    if (!stringProperties.Any()) { return source; }

    // Get the right overload of String.Contains
    MethodInfo containsMethod = typeof(string).GetMethod("Contains",
    [typeof(string)])!;

    // Create a parameter for the expression tree:
    // the 'x' in 'x => x.PropertyName.Contains("term")'
    // The type of this parameter is the query's element type
    ParameterExpression prm = Parameter(elementType);

    // Map each property to an expression tree node
    IEnumerable<Expression> expressions = stringProperties
        .Select(prp =>
            // For each property, we have to construct an expression tree
            // node like x.PropertyName.Contains("term")
```

```

        Call(                  // .Contains(...)
          Property(           // .PropertyName
            prm,              // x
            prp
          ),
          containsMethod,
          Constant(term)     // "term"
        )
      );

// Combine all the resultant expression nodes using ||
Expression body = expressions
  .Aggregate((prev, current) => Or(prev, current));

// Wrap the expression body in a compile-time-typed lambda expression
Expression<Func<T, bool>> lambda = Lambda<Func<T, bool>>(body, prm);

// Because the lambda is compile-time-typed (albeit with a generic
// parameter), we can use it with the Where method
return source.Where(lambda);
}

```

`TextFilter` 함수는 `IQueryable<T>`만이 아니라 `IQueryable`을 사용하고 반환하기 때문에 텍스트 필터 뒤에 컴파일 시간 형식의 쿼리 요소를 더 추가할 수 있습니다.

C#

```

var qry = TextFilter(
  new List<Person>().AsQueryable(),
  "abcd"
)
.Where(x => x.DateOfBirth < new DateTime(2001, 1, 1));

var qry1 = TextFilter(
  new List<Car>().AsQueryable(),
  "abcd"
)
.Where(x => x.Year == 2010);

```

## IQueryable<TDelegate>의 식 트리에 메서드 호출 노드를 추가합니다.

`IQueryable` 대신 `IQueryable<T>`가 있는 경우 제네릭 LINQ 메서드를 직접 호출할 수 없습니다. 한 가지 대안은 이전 예에 표시된 대로 내부 식 트리를 빌드하고 리플렉션을 사용하여 식 트리를 전달하는 동안 적절한 LINQ 메서드를 호출하는 것입니다.

LINQ 메서드 호출을 나타내는 `MethodCallExpression`으로 트리 전체를 래핑하여 LINQ 메서드의 기능을 복제할 수도 있습니다.

C#

```
IQueryable TextFilter_Untyped(IQueryable source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }
    Type elementType = source.ElementType;

    // The logic for building the ParameterExpression and the
    // LambdaExpression's body is the same as in the previous example,
    // but has been refactored into the constructBody function.
    (Expression? body, ParameterExpression? prm) =
        constructBody(elementType, term);
    if (body is null) { return source; }

    Expression filteredTree = Call(
        typeof(Queryable),
        "Where",
        [elementType],
        source.Expression,
        Lambda(body, prm!))
);

    return source.Provider.CreateQuery(filteredTree);
}
```

이 경우 컴파일 시간 `T` 제네릭 자리 표시자가 없으므로 컴파일 시간 형식 정보가 필요하지 않고 `Lambda` 대신 `LambdaExpression`을 생성하는 `Expression<TDelegate>` 오버로드를 사용합니다.

## 동적 LINQ 라이브러리

팩터리 메서드를 사용하여 식 트리를 생성하는 작업은 비교적 복잡하며, 문자열을 작성하는 작업이 더 쉽습니다. [동적 LINQ 라이브러리](#)는 `Queryable`에서 표준 LINQ 메서드에 해당하며 식 트리 대신 [특수 구문](#)에서 문자열을 허용하는 `IQueryable`에 확장 메서드 세트를 공개합니다. 라이브러리는 문자열에서 적절한 식 트리를 생성하며 결과로 변환된 `IQueryable`을 반환할 수 있습니다.

예를 들어 이전 예제를 다음과 같이 다시 작성할 수 있습니다.

C#

```
// using System.Linq.Dynamic.Core

IQueryable TextFilter.Strings(IQueryable source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }

    var elementType = source.ElementType;
```

```
// Get all the string property names on this specific type.
var stringProperties =
    elementType.GetProperties()
        .Where(x => x.PropertyType == typeof(string))
        .ToArray();
if (!stringProperties.Any()) { return source; }

// Build the string expression
string filterExpr = string.Join(
    " || ",
    stringProperties.Select(prp => $"{prp.Name}.Contains(@0)")
);

return source.Where(filterExpr, term);
}
```

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

💡 설명서 문제 열기

↗️ 제품 사용자 의견 제공

# async 및 await를 사용한 비동기 프로그래밍

아티클 • 2024. 02. 21.

TAP(Task 비동기 프로그래밍) 모델은 비동기 코드에 대한 추상화를 제공합니다. 항상 그 렇듯이 코드는 일련의 명령문으로 작성합니다. 다음 명령문이 시작되기 전에 각 명령문이 완료되는 것처럼 해당 코드를 읽을 수 있습니다. 이러한 명령문 중 일부에서 작업을 시작하고 진행 중인 작업을 나타내는 Task를 반환할 수 있으므로 컴파일러는 여러 가지 변환을 수행합니다.

이 구문의 목표는 일련의 명령문처럼 읽지만 외부 리소스 할당과 작업 완료 시점에 따라 훨씬 더 복잡한 순서로 실행되는 코드를 사용하도록 설정하는 것입니다. 사람이 비동기 작업이 포함된 프로세스에 대한 지침을 제공하는 방법과 비슷합니다. 이 문서에서는 `async` 및 `await` 키워드를 사용하여 일련의 비동기 명령이 포함된 코드를 쉽게 추론하는 방법을 알아보기 위해 아침 식사를 준비하기 위한 지침의 예를 사용합니다. 아침 식사를 준비하는 방법을 설명하기 위해 작성하는 지침은 다음 목록과 같습니다.

1. 커피 한 잔을 따릅니다.
2. 팬을 가열한 다음 계란 두 개를 볶습니다.
3. 베이컨 세 조각을 튀깁니다.
4. 빵 두 조각을 굽습니다.
5. 토스트에 버터와잼을 바릅니다.
6. 오렌지 주스 한잔을 따릅니다.

요리에 대한 경험이 있는 경우 이러한 지침은 **비동기적으로** 실행됩니다. 계란 프라이를 위해 팬을 데우기 시작한 다음, 베이컨을 시작합니다. 토스터에 빵을 넣고 계란 프라이를 시작합니다. 프로세스의 각 단계에서 작업을 시작한 다음, 주의가 필요한 작업에 주의를 돌립니다.

아침을 요리하는 것은 병렬로 수행되지 않는 비동기 작업의 좋은 예입니다. 한 사람(또는 스레드)이 이러한 모든 작업을 처리할 수 있습니다. 아침 식사 비유를 계속하면 첫 번째 작업이 완료되기 전에 다음 작업을 시작하여 한 사람이 비동기적으로 아침 식사를 만들 수 있습니다. 누군가 보고 있는지 여부에 관계없이 요리는 계속됩니다. 계란 프라이를 위해 팬을 데우기 시작하자마자 베이컨을 튀기기 시작할 수 있습니다. 베이컨 튀김이 시작되면 토스터에 빵을 넣을 수 있습니다.

병렬 알고리즘의 경우 여러 요리사(또는 스레드)가 필요합니다. 한 사람은 계란을 만들고 또 한 사람은 베이컨을 만드는 방식으로 진행될 것입니다. 즉 각각은 하나의 작업에만 집중할 것입니다. 각 요리사(또는 스레드)는 베이컨이 뒤집을 준비가 되거나 토스트가 나올 때까지 동기적으로 차단됩니다.

이제 C# 문으로 작성된 동일한 명령을 고려합니다.

C#

```
using System;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    // These classes are intentionally empty for the purpose of this
    // example. They are simply marker classes for the purpose of demonstration,
    // contain no properties, and serve no other purpose.
    internal class Bacon { }
    internal class Coffee { }
    internal class Egg { }
    internal class Juice { }
    internal class Toast { }

    class Program
    {
        static void Main(string[] args)
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            Egg eggs = FryEggs(2);
            Console.WriteLine("eggs are ready");

            Bacon bacon = FryBacon(3);
            Console.WriteLine("bacon is ready");

            Toast toast = ToastBread(2);
            ApplyButter(toast);
            ApplyJam(toast);
            Console.WriteLine("toast is ready");

            Juice oj = PourOJ();
            Console.WriteLine("oj is ready");
            Console.WriteLine("Breakfast is ready!");
        }

        private static Juice PourOJ()
        {
            Console.WriteLine("Pouring orange juice");
            return new Juice();
        }

        private static void ApplyJam(Toast toast) =>
            Console.WriteLine("Putting jam on the toast");

        private static void ApplyButter(Toast toast) =>
            Console.WriteLine("Putting butter on the toast");

        private static Toast ToastBread(int slices)
```

```

    {
        for (int slice = 0; slice < slices; slice++)
        {
            Console.WriteLine("Putting a slice of bread in the
toaster");
        }
        Console.WriteLine("Start toasting...");
        Task.Delay(3000).Wait();
        Console.WriteLine("Remove toast from toaster");

        return new Toast();
    }

    private static Bacon FryBacon(int slices)
    {
        Console.WriteLine($"putting {slices} slices of bacon in the
pan");
        Console.WriteLine("cooking first side of bacon...");
        Task.Delay(3000).Wait();
        for (int slice = 0; slice < slices; slice++)
        {
            Console.WriteLine("flipping a slice of bacon");
        }
        Console.WriteLine("cooking the second side of bacon...");
        Task.Delay(3000).Wait();
        Console.WriteLine("Put bacon on plate");

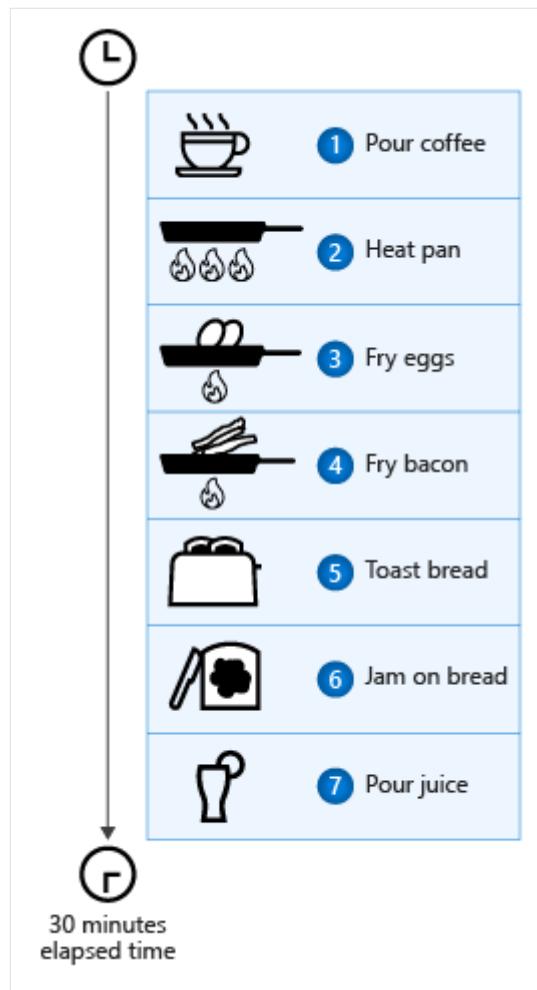
        return new Bacon();
    }

    private static Egg FryEggs(int howMany)
    {
        Console.WriteLine("Warming the egg pan...");
        Task.Delay(3000).Wait();
        Console.WriteLine($"cracking {howMany} eggs");
        Console.WriteLine("cooking the eggs ...");
        Task.Delay(3000).Wait();
        Console.WriteLine("Put eggs on plate");

        return new Egg();
    }

    private static Coffee PourCoffee()
    {
        Console.WriteLine("Pouring coffee");
        return new Coffee();
    }
}

```



동기적으로 준비된 아침 식사는 합계가 각 작업의 합계이기 때문에 약 30분이 걸렸습니다.

컴퓨터에서는 사람들이 수행하는 것과 같은 방식으로 이러한 명령을 해석하지 않습니다. 다음 명령문으로 이동하기 전에 작업이 완료될 때까지 컴퓨터는 각 명령문에서 차단됩니다. 이로 인해 불만족스러운 아침 식사를 만듭니다. 이전 작업이 완료될 때까지 이후 작업을 시작할 수 없었습니다. 아침 식사를 만드는데 훨씬 더 오래 걸리고, 일부 음식은 식은 채로 제공되었을 것입니다.

컴퓨터에서 위의 명령을 비동기적으로 실행하게 하려면 비동기 코드를 작성해야 합니다.

이러한 문제는 현재 작성하는 프로그램에 중요합니다. 클라이언트 프로그램을 작성할 때 UI에서 사용자 입력에 응답해야 합니다. 웹에서 데이터를 다운로드하는 동안 애플리케이션에서 휴대폰이 중지된 것처럼 표시하면 안 됩니다. 서버 프로그램을 작성하는 경우 스레드가 차단되지 않도록 합니다. 이러한 스레드는 다른 요청을 처리할 수 있습니다. 비동기 대안이 있을 때 동기 코드를 사용하면 비용이 적게 드는 규모 확장 기능이 저하됩니다. 차단된 스레드에 대한 비용을 지불합니다.

성공적인 최신 애플리케이션에는 비동기 코드가 필요합니다. 언어 지원 없이 비동기 코드를 작성하는 경우 콜백, 완료 이벤트 또는 코드의 원래 의도를 모호하게 하는 다른 수단이 필요했습니다. 동기 코드의 이점은 단계별 작업을 통해 쉽게 검사하고 이해할 수 있다는

점입니다. 기존의 비동기 모델에서는 코드의 기본 동작이 아니라 코드의 비동기적 특성에 집중할 수 밖에 없었습니다.

## 차단하는 대신 대기

앞의 코드에서는 동기 코드를 구성하여 비동기 작업을 수행하는 잘못된 사례를 보여 줍니다. 작성한 대로 이 코드는 실행되는 스레드에서 다른 작업을 수행하지 못하도록 차단합니다. 작업이 진행되는 동안에는 중단되지 않습니다. 마치 빵을 넣은 후 토스터를 쳐다보는 것과 같습니다. 토스트가 나오기 전까지 아무하고도 대화하지 않을 것입니다.

먼저 이 코드를 업데이트하여 작업이 실행되는 동안 스레드가 차단되지 않도록 하겠습니다. `await` 키워드는 작업을 차단하지 않는 방식으로 시작한 다음, 해당 작업이 완료되면 실행을 계속합니다. 간단한 비동기 버전의 아침 식사 준비 코드는 다음과 같습니다.

C#

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    Egg eggs = await FryEggsAsync(2);
    Console.WriteLine("eggs are ready");

    Bacon bacon = await FryBaconAsync(3);
    Console.WriteLine("bacon is ready");

    Toast toast = await ToastBreadAsync(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

### ① 중요

총 경과 시간은 초기 동기 버전과 거의 같습니다. 이 코드에서는 아직 비동기 프로그래밍의 몇 가지 주요 기능을 활용하지 않았습니다.

### 💡 팁

`FryEggsAsync`, `FryBaconAsync` 및 `ToastBreadAsync`의 메서드 본문은 각각 `Task<Egg>`, `Task<Bacon>` 및 `Task<Toast>`를 반환하도록 모두 업데이트되었습니다. 이 메서드들은 원래 버전에서 “`Async`” 접미사를 포함하도록 이름이 바뀌었습니다. 해당 구현은 이 문서의 뒷부분에 나오는 최종 버전의 일부로 표시됩니다.

## ① 참고

`Main` 메서드는 `return` 식이 없더라도 기본적으로 `Task` 이(가) 반환 됩니다. 자세한 내용은 void 반환 비동기 함수 평가를 참조하세요.

이 코드는 계란이나 베이컨을 요리하는 동안 차단되지 않습니다. 하지만 이 코드는 다른 작업을 시작하지 않습니다. 토스트가 토스터에 넣어져 나올 때까지 쳐다보고 있습니다. 그러나 적어도 주의를 끌려고 하는 누구에게나 응답할 수는 있습니다. 여러 주문을 받는 식당에서 요리사는 첫 번째 요리를 하는 동안 또 다른 아침 식사 준비를 시작할 수 있습니다.

시작했지만 아직 완료되지 않은 작업을 기다리는 동안 아침 식사 작업 스레드가 차단되지 않습니다. 일부 애플리케이션의 경우 이 변경만으로 충분합니다. GUI 애플리케이션은 이 변경만으로도 사용자에게 응답합니다. 그러나 지금 시나리오에서는 더 많은 작업이 필요합니다. 각 구성 요소 작업이 순차적으로 실행되지 않도록 해야 합니다. 이전 작업이 완료되기를 기다리기 전에 각 구성 요소 작업을 시작하는 것이 좋습니다.

## 동시에 작업 시작

대부분의 시나리오에서는 독립적인 몇 가지 작업을 즉시 시작하려고 합니다. 그런 다음, 각 작업이 완료되면 준비된 다른 작업을 계속할 수 있습니다. 아침 식사 비유에서 이는 아침 식사를 더 빨리 준비하는 방법입니다. 모든 것을 거의 동시에 완료할 수 있습니다. 이에 따라 따뜻한 아침 식사가 준비됩니다.

`System.Threading.Tasks.Task` 및 관련 형식은 진행 중인 작업을 추론하는 데 사용할 수 있는 클래스입니다. 이를 통해 아침 식사를 만드는 방법과 더 유사한 코드를 작성할 수 있습니다. 계란, 베이컨 및 토스트 요리를 동시에 시작할 수 있을 것입니다. 각 요리에 필요한 작업이 있으므로 해당 작업에 주의를 기울이고, 다음 작업을 처리한 다음, 주의가 필요한 다른 작업을 기다립니다.

작업을 시작하고, 해당 작업을 나타내는 `Task` 개체를 유지합니다. 결과를 사용하기 전에 각 작업을 기다립니다(`await`).

아침 식사 코드를 이처럼 변경해 보겠습니다. 첫 번째 단계는 작업을 기다리지 않고 시작될 때 해당 작업을 저장하는 것입니다.

C#

```
Coffee cup = PourCoffee();
Console.WriteLine("Coffee is ready");

Task<Egg> eggsTask = FryEggsAsync(2);
Egg eggs = await eggsTask;
Console.WriteLine("Eggs are ready");

Task<Bacon> baconTask = FryBaconAsync(3);
Bacon bacon = await baconTask;
Console.WriteLine("Bacon is ready");

Task<Toast> toastTask = ToastBreadAsync(2);
Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("Toast is ready");

Juice oj = PourOJ();
Console.WriteLine("OJ is ready");
Console.WriteLine("Breakfast is ready!");
```

앞의 코드는 아침 식사를 더 빨리 준비하지 않습니다. 작업이 시작되자마자 모든 `await` 작업이 완료됩니다. 다음으로, 아침 식사를 제공하기 전에 베이컨과 달걀에 대한 `await` 문을 메서드 끝으로 이동할 수 있습니다.

C#

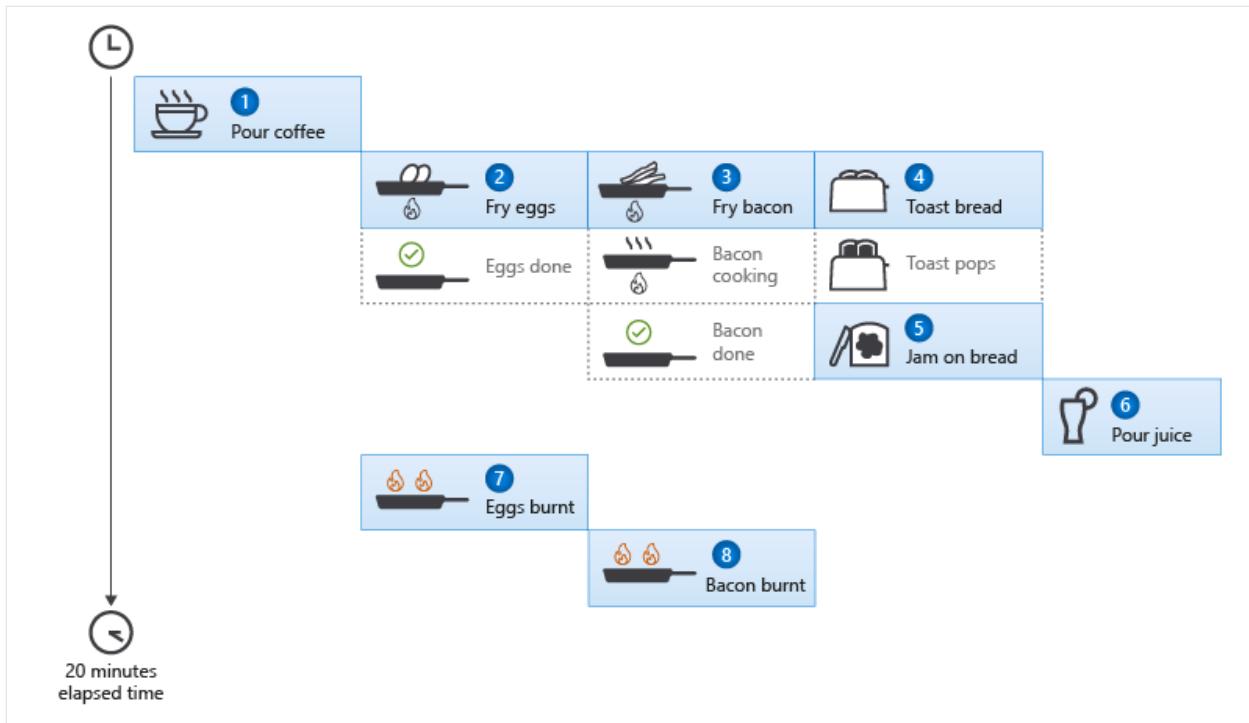
```
Coffee cup = PourCoffee();
Console.WriteLine("Coffee is ready");

Task<Egg> eggsTask = FryEggsAsync(2);
Task<Bacon> baconTask = FryBaconAsync(3);
Task<Toast> toastTask = ToastBreadAsync(2);

Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("Toast is ready");
Juice oj = PourOJ();
Console.WriteLine("OJ is ready");

Egg eggs = await eggsTask;
Console.WriteLine("Eggs are ready");
Bacon bacon = await baconTask;
Console.WriteLine("Bacon is ready");

Console.WriteLine("Breakfast is ready!");
```



비동기적으로 준비된 아침 식사에는 대략 20분이 걸렸는데, 일부 작업이 동시에 실행되었기 때문에 이렇게 시간을 절약할 수 있는 것입니다.

앞의 코드가 더 잘 작동합니다. 모든 비동기 작업을 한 번에 시작합니다. 결과가 필요할 때만 각 작업을 기다립니다. 앞의 코드는 다른 마이크로서비스를 요청한 다음, 결과를 단일 페이지로 결합하는 웹 애플리케이션의 코드와 비슷할 수 있습니다. 모든 요청을 즉시 수행한 다음, 이러한 모든 작업을 기다리고(`await`) 웹 페이지를 구성합니다.

## 작업 구성

토스트를 제외한 모든 아침 식사가 동시에 준비되었습니다. 토스트를 만드는 것은 비동기 작업(빵 굽기)과 동기 작업(버터와 잼 바르기)의 구성입니다. 이 코드를 업데이트하면 중요한 개념을 알 수 있습니다.

### ⓘ 중요

동기 작업이 뒤따르는 비동기 작업으로 구성된 작업은 비동기 작업입니다. 즉 작업의 일부가 비동기이면 전체 작업이 비동기입니다.

이전 코드에서는 `Task` 또는 `Task<TResult>` 개체를 사용하여 실행 중인 작업을 유지할 수 있음을 보여 주었습니다. 결과를 사용하기 전에 각 작업을 기다립니다(`await`). 다음 단계는 다른 작업의 결합을 나타내는 메서드를 만드는 것입니다. 아침 식사를 제공하기 전에 빵을 구운 후에 버터와 잼을 바르는 것을 나타내는 작업을 기다리려고 합니다. 이 작업은 다음 코드를 사용하여 나타낼 수 있습니다.

C#

```
static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);

    return toast;
}
```

앞의 메서드에서 해당 시그니처에는 `async` 한정자가 있습니다. 이 경우 이 메서드에서 비동기 작업이 포함된 `await` 문을 포함하고 있다고 컴파일러에 알립니다. 이 메서드는 빵을 구운 다음, 버터와 잼을 바르는 작업을 나타내며, 이러한 세 가지 작업의 구성을 나타내는 `Task<TResult>`를 반환합니다. 이제 main 코드 블록은 다음과 같습니다.

C#

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = MakeToastWithButterAndJamAsync(2);

    var eggs = await eggsTask;
    Console.WriteLine("eggs are ready");

    var bacon = await baconTask;
    Console.WriteLine("bacon is ready");

    var toast = await toastTask;
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

앞의 변경에서는 비동기 코드를 사용하는 데 있어 중요한 기술을 보여 주었습니다. 작업을 반환하는 새 메서드로 구분하여 작업을 구성합니다. 해당 작업을 기다리는 시기를 선택할 수 있습니다. 다른 작업을 동시에 시작할 수 있습니다.

## 비동기 예외

이 시점까지 이러한 모든 작업이 성공적으로 완료된다고 암시적으로 가정했습니다. 비동기 메서드는 동기 메서드와 마찬가지로 예외를 throw합니다. 예외 및 오류 처리에 대한 비동기 지원은 일반적인 비동기 지원과 같은 목표를 달성하려고 합니다. 즉, 일련의 동기 문처럼 읽는 코드를 작성해야 합니다. 작업은 성공적으로 완료될 수 없는 경우 예외를 throw합니다. 시작된 작업이 `awaited`인 경우 클라이언트 코드에서 해당 예외를 catch할 수 있습니다. 예를 들어 토스트를 만드는 동안 토스터에 불이 난다고 가정해 보겠습니다. `ToastBreadAsync` 메서드를 다음 코드와 일치하도록 수정하여 이 상황을 시뮬레이션할 수 있습니다.

C#

```
private static async Task<Toast> ToastBreadAsync(int slices)
{
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("Putting a slice of bread in the toaster");
    }
    Console.WriteLine("Start toasting...");
    await Task.Delay(2000);
    Console.WriteLine("Fire! Toast is ruined!");
    throw new InvalidOperationException("The toaster is on fire");
    await Task.Delay(1000);
    Console.WriteLine("Remove toast from toaster");

    return new Toast();
}
```

## ① 참고

연결할 수 없는 코드에 대해 앞의 코드를 컴파일하면 경고가 표시됩니다. 토스터에 불이 나면 작업이 정상적으로 진행되지 않으므로 이는 의도적입니다.

이러한 변경을 수행한 후 애플리케이션을 실행하면 다음 텍스트와 유사하게 출력됩니다.

콘솔

```
Pouring coffee
Coffee is ready
Warming the egg pan...
putting 3 slices of bacon in the pan
Cooking first side of bacon...
Putting a slice of bread in the toaster
Putting a slice of bread in the toaster
Start toasting...
Fire! Toast is ruined!
Flipping a slice of bacon
Flipping a slice of bacon
```

```
Flipping a slice of bacon
Cooking the second side of bacon...
Cracking 2 eggs
Cooking the eggs ...
Put bacon on plate
Put eggs on plate
Eggs are ready
Bacon is ready
Unhandled exception. System.InvalidOperationException: The toaster is on
fire
  at AsyncBreakfast.Program.ToastBreadAsync(Int32 slices) in
Program.cs:line 65
  at AsyncBreakfast.Program.MakeToastWithButterAndJamAsync(Int32 number) in
Program.cs:line 36
  at AsyncBreakfast.Program.Main(String[] args) in Program.cs:line 24
  at AsyncBreakfast.Program.<Main>(String[] args)
```

토스터에 불이 붙은 시점과 예외가 관찰되는 시점 사이에 꽤 많은 작업이 완료되었음을 알 수 있습니다. 비동기적으로 실행되는 작업에서 예외를 throw하면 해당 Task가 **오류** 상태가 됩니다. Task 개체는 [Task.Exception](#) 속성에서 throw된 예외를 포함합니다. 오류 상태인 작업이 대기되면 예외를 throw합니다.

이해해야 할 두 가지 중요한 메커니즘이 있습니다. 하나는 예외가 오류 상태인 작업에 저장되는 방식이고 다른 하나는 코드가 오류 상태인 작업을 대기할 때 예외가 패키지 해제되었다가 다시 throw되는 방식입니다.

비동기적으로 실행되는 코드가 예외를 throw하면 해당 예외는 `Task`에 저장됩니다. 비동기 작업 중에는 둘 이상의 예외가 throw될 수 있으므로 [Task.Exception](#) 속성은 [System.AggregateException](#)입니다. throw된 모든 예외는 [AggregateException.InnerExceptions](#) 컬렉션에 추가됩니다. 해당 `Exception` 속성이 null이면 새 `AggregateException`이 만들어지고 throw된 예외는 컬렉션의 첫 번째 항목이 됩니다.

오류 상태인 작업의 가장 일반적인 시나리오는 `Exception` 속성이 정확히 하나의 예외를 포함하는 것입니다. 코드가 오류 상태인 작업을 `awaits`하면 [AggregateException.InnerExceptions](#) 컬렉션의 첫 번째 예외가 다시 throw됩니다. 그렇기 때문에 이 예제의 출력에 `AggregateException` 대신 `InvalidOperationException`가 표시되는 것입니다. 첫 번째 내부 예외를 추출하면 동기 메서드로 작업하는 것과 최대한 유사하게 비동기 메서드로 작업할 수 있습니다. 시나리오에서 여러 예외를 생성할 수 있는 경우 코드에서 `Exception` 속성을 검사할 수 있습니다.

## 💡 팁

인수 유효성 검사 예외는 작업 반환 메서드에서 동기적으로 나타나는 것이 좋습니다. 자세한 내용과 이 작업을 수행하는 방법의 예는 [작업 반환 메서드의 예외](#)를 참조

하세요.

계속하기 전에 `ToastBreadAsync` 메서드에서 다음 두 줄을 주석으로 처리합니다. 또 다른 불이 시작되기를 원치는 않으니까요.

C#

```
Console.WriteLine("Fire! Toast is ruined!");
throw new InvalidOperationException("The toaster is on fire");
```

## 효율적인 작업 대기

`Task` 클래스의 메서드를 사용하여 앞의 코드 끝에 있는 일련의 `await` 문을 향상시킬 수 있습니다. 이러한 API 중 하나인 `WhenAll`은 다음 코드와 같이 인수 목록의 모든 작업이 완료되면 완료된 `Task`를 반환합니다.

C#

```
await Task.WhenAll(eggsTask, baconTask, toastTask);
Console.WriteLine("Eggs are ready");
Console.WriteLine("Bacon is ready");
Console.WriteLine("Toast is ready");
Console.WriteLine("Breakfast is ready");
```

또 다른 옵션으로, 인수가 완료되면 완료된 `Task<Task>`를 반환하는 `WhenAny`를 사용하는 것입니다. 반환된 작업은 이미 완료되었음을 알고 있으므로 기다릴 수 있습니다. 다음 코드에서는 `WhenAny`를 사용하여 첫 번째 작업이 완료될 때까지 기다린 다음, 결과를 처리하는 방법을 보여 줍니다. 완료된 작업의 결과가 처리되면 완료된 작업을 `WhenAny`에 전달된 작업 목록에서 제거합니다.

C#

```
var breakfastTasks = new List<Task> { eggsTask, baconTask, toastTask };
while (breakfastTasks.Count > 0)
{
    Task finishedTask = await Task.WhenAny(breakfastTasks);
    if (finishedTask == eggsTask)
    {
        Console.WriteLine("Eggs are ready");
    }
    else if (finishedTask == baconTask)
    {
        Console.WriteLine("Bacon is ready");
    }
    else if (finishedTask == toastTask)
```

```
{  
    Console.WriteLine("Toast is ready");  
}  
await finishedTask;  
breakfastTasks.Remove(finishedTask);  
}
```

끝부분에 줄 `await finishedTask;` 이(가) 표시됩니다. 이 줄 `await Task.WhenAny` 은(는) 완료된 작업을 기다리지 않습니다. `await Task.WhenAny`에서 반환된 `Task`입니다.

`Task.WhenAny` 결과는 완료되었거나 오류가 발생한 작업입니다. 실행이 완료된 것을 알고 있더라도 해당 작업을 다시 `await`해야 합니다. 이것이 결과를 검색하거나 오류를 일으키는 예외가 발생하는지 확인하는 방법입니다.

변경 내용을 모두 적용한 후 코드의 최종 버전은 다음과 같습니다.

C#

```
using System;  
using System.Collections.Generic;  
using System.Threading.Tasks;  
  
namespace AsyncBreakfast  
{  
    // These classes are intentionally empty for the purpose of this  
    // example. They are simply marker classes for the purpose of demonstration,  
    // contain no properties, and serve no other purpose.  
    internal class Bacon { }  
    internal class Coffee { }  
    internal class Egg { }  
    internal class Juice { }  
    internal class Toast { }  
  
    class Program  
    {
```

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    var eggsTask = FryEggsAsync(2);
    var baconTask = FryBaconAsync(3);
    var toastTask = MakeToastWithButterAndJamAsync(2);

    var breakfastTasks = new List<Task> { eggsTask, baconTask,
toastTask };
    while (breakfastTasks.Count > 0)
    {
        Task finishedTask = await Task.WhenAny(breakfastTasks);
        if (finishedTask == eggsTask)
        {
            Console.WriteLine("eggs are ready");
        }
        else if (finishedTask == baconTask)
        {
            Console.WriteLine("bacon is ready");
        }
        else if (finishedTask == toastTask)
        {
            Console.WriteLine("toast is ready");
        }
        await finishedTask;
        breakfastTasks.Remove(finishedTask);
    }

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}

static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);

    return toast;
}

private static Juice PourOJ()
{
    Console.WriteLine("Pouring orange juice");
    return new Juice();
}

private static void ApplyJam(Toast toast) =>
    Console.WriteLine("Putting jam on the toast");

private static void ApplyButter(Toast toast) =>
    Console.WriteLine("Putting butter on the toast");
```

```
private static async Task<Toast> ToastBreadAsync(int slices)
{
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("Putting a slice of bread in the
toaster");
    }
    Console.WriteLine("Start toasting...");
    await Task.Delay(3000);
    Console.WriteLine("Remove toast from toaster");

    return new Toast();
}

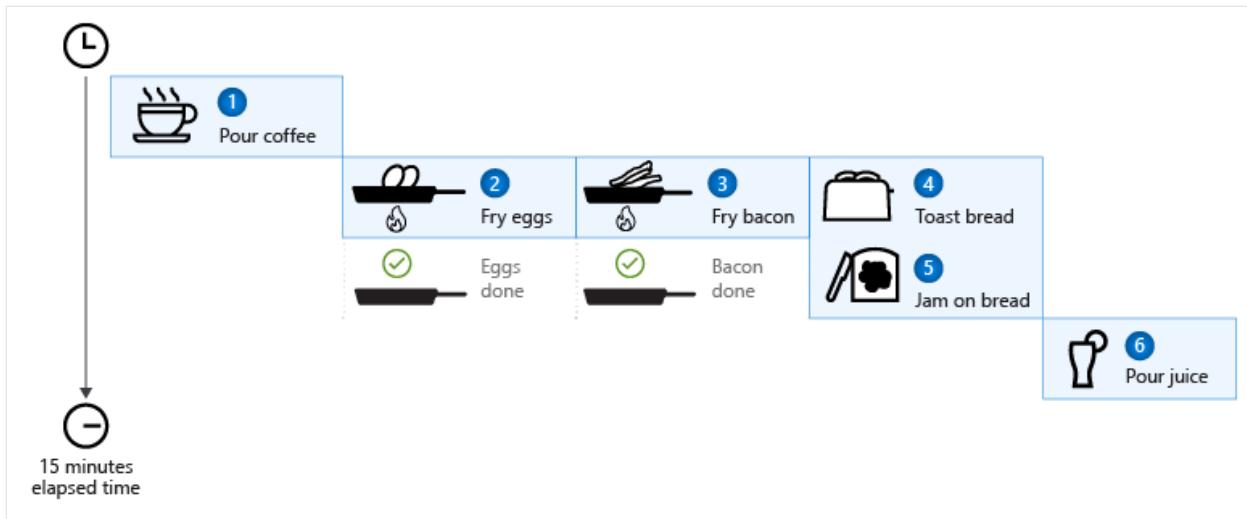
private static async Task<Bacon> FryBaconAsync(int slices)
{
    Console.WriteLine($"putting {slices} slices of bacon in the
pan");
    Console.WriteLine("cooking first side of bacon...");
    await Task.Delay(3000);
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("flipping a slice of bacon");
    }
    Console.WriteLine("cooking the second side of bacon...");
    await Task.Delay(3000);
    Console.WriteLine("Put bacon on plate");

    return new Bacon();
}

private static async Task<Egg> FryEggsAsync(int howMany)
{
    Console.WriteLine("Warming the egg pan...");
    await Task.Delay(3000);
    Console.WriteLine($"cracking {howMany} eggs");
    Console.WriteLine("cooking the eggs ...");
    await Task.Delay(3000);
    Console.WriteLine("Put eggs on plate");

    return new Egg();
}

private static Coffee PourCoffee()
{
    Console.WriteLine("Pouring coffee");
    return new Coffee();
}
}
```



비동기적으로 준비된 아침 식사의 최종 버전에는 대략 6분이 걸렸는데, 일부 작업을 동시에 실행하고 코드가 여러 작업을 한 번에 모니터링하고 필요한 경우에만 작업을 수행했기 때문입니다.

이 최종 코드는 비동기입니다. 이 코드는 아침 식사를 요리하는 방법을 더 정확하게 반영하고 있습니다. 앞의 코드를 이 문서의 첫 번째 코드 샘플과 비교해 보세요. 핵심 작업은 코드를 읽어 파악할 수 있습니다. 이 코드는 이 문서의 시작 부분에 나와 있는 아침 식사 준비 지침을 읽는 것과 동일한 방식으로 읽을 수 있습니다. `async` 및 `await` 언어 기능을 사용하면 모든 사용자가 작성된 이러한 지침을 따를 수 있습니다. 가능한 한 작업을 시작하지만 작업이 완료될 때까지 기다리는 것을 차단하지 않도록 합니다.

## 다음 단계

### 비동기 프로그램의 실제 시나리오 살펴보기

#### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.



#### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

[설명서 문제 열기](#)

[제품 사용자 의견 제공](#)

# 비동기 프로그래밍

아티클 • 2023. 04. 08.

I/O 바인딩된 요구 사항이 있는 경우(예: 네트워크에 데이터 요청, 데이터베이스 액세스 또는 파일 시스템 읽기 및 쓰기) 비동기 프로그래밍을 활용하는 것이 좋습니다. 부담이 큰 계산을 수행하는 것과 같이 CPU 바인딩된 코드가 있을 수도 있으며 이는 비동기 코드 작성의 좋은 시나리오이기도 합니다.

C#에는 콜백을 조작하거나 비동기를 지원하는 라이브러리를 따를 필요 없이 비동기 코드를 쉽게 작성할 수 있는 언어 수준 비동기 프로그래밍 모델이 있습니다. 이 모델은 [TAP\(작업 기반 비동기 패턴\)](#)을 따릅니다.

## 비동기 모델 개요

비동기 프로그래밍의 핵심은 비동기 작업을 모델링하는 `Task` 및 `Task<T>` 개체입니다. 이러한 개체는 `async` 및 `await` 키워드를 통해 지원됩니다. 대부분의 경우 모델은 매우 간단합니다.

- I/O 바인딩된 코드에서는 `async` 메서드의 내부에서 `Task` 또는 `Task<T>`를 반환하는 작업을 기다립니다.
- CPU 바인딩된 코드에서는 `Task.Run` 메서드로 백그라운드 스레드에서 시작되는 작업을 기다립니다.

`await` 키워드가 마법이 일어나는 곳입니다. `await`를 수행한 메서드의 호출자에게 제어를 넘기고, 궁극적으로 UI가 응답하거나 서비스가 탄력적일 수 있도록 합니다. `async` 및 `await` 외에 비동기 코드를 사용하는 [여러 방법](#)이 있지만, 이 문서에서는 언어 수준 구문을 집중적으로 설명합니다.

## I/O 바인딩 예제: 웹 서비스에서 데이터 다운로드

단추가 눌릴 때 웹 서비스에서 일부 데이터를 다운로드해야 할 수 있지만 UI 스레드를 차단하지 않으려고 합니다. 클래스를 사용하여 다음과 같이 수행할 수 있습니다.  
[System.Net.Http.HttpClient](#)

C#

```
private readonly HttpClient _httpClient = new HttpClient();

downloadButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI as the request
    // from the web service is happening.
```

```
//  
// The UI thread is now free to perform other work.  
var stringData = await _httpClient.GetStringAsync(URL);  
DoSomethingWithData(stringData);  
};
```

이 코드는 `Task` 개체 조작 시 위험에 빠지지 않고 의도(데이터를 비동기식으로 다운로드)를 표현합니다.

## CPU 바인딩 예제: 게임에 대한 계산 수행

단추를 누르면 화면의 많은 적에게 손상을 입힐 수 있는 모바일 게임을 작성한다고 가정합니다. 손상 계산을 수행하는 것은 부담이 클 수 있고 UI 스레드에서 이 작업을 수행하면 계산이 수행될 때 게임이 일시 중지되는 것처럼 보입니다.

이 작업을 처리하는 가장 좋은 방법은 `Task.Run`을 사용하여 작업을 수행하는 백그라운드 스레드를 시작하고 `await`를 사용하여 결과를 기다리는 것입니다. 이렇게 하면 작업이 수행되는 동안 UI가 매끄럽게 느껴질 수 있습니다.

C#

```
private DamageResult CalculateDamageDone()  
{  
    // Code omitted:  
    //  
    // Does an expensive calculation and returns  
    // the result of that calculation.  
}  
  
calculateButton.Clicked += async (o, e) =>  
{  
    // This line will yield control to the UI while CalculateDamageDone()  
    // performs its work. The UI thread is free to perform other work.  
    var damageResult = await Task.Run(() => CalculateDamageDone());  
    DisplayDamage(damageResult);  
};
```

이 코드는 단추 클릭 이벤트의 의도를 표현하고 백그라운드 스레드를 수동으로 관리할 필요가 없고 비차단 방식으로 작업을 수행합니다.

## 백그라운드에서 수행되는 작업

C#에서는 컴파일러가 해당 코드를, `await`에 도달할 때 실행을 양도하고 백그라운드 작업이 완료될 때 실행을 다시 시작하는 것과 같은 작업을 추적하는 상태 시스템으로 변환합니다.

이론적으로 보면 이 변환은 [비동기 프라미스 모델](#)입니다.

## 이해해야 할 주요 부분

- 비동기 코드는 I/O 바인딩된 코드와 CPU 바인딩된 코드에 둘 다 사용할 수 있지만 시나리오마다 다르게 사용됩니다.
- 비동기 코드는 백그라운드에서 수행되는 작업을 모델링하는 데 사용되는 구문인 `Task<T>` 및 `Task`를 사용합니다.
- `async` 키워드는 본문에서 `await` 키워드를 사용할 수 있는 비동기 메서드로 메서드를 변환합니다.
- `await` 키워드가 적용되면 이 키워드는 호출 메서드를 일시 중단하고 대기 작업이 완료할 때까지 제어 권한을 다시 호출자에게 양도합니다.
- `await`는 비동기 메서드 내부에서만 사용할 수 있습니다.

## CPU 바인딩된 작업 및 I/O 바인딩된 작업 인식

이 가이드의 처음 두 예제에서는 I/O 바인딩된 작업과 CPU 바인딩된 작업에 `async` 및 `await`를 사용하는 방법을 설명했습니다. 이 방법은 수행해야 하는 작업이 I/O 바인딩된 작업 또는 CPU 바인딩된 작업일 경우 이를 식별할 수 있는 키입니다. 이 방법이 코드 성능에 큰 영향을 미칠 수 있고 잠재적으로 특정 구문을 잘못 사용하게 될 수 있기 때문입니다.

다음은 코드를 작성하기 전에 질문해야 하는 두 가지 질문입니다.

1. 코드가 데이터베이스의 데이터와 같은 무엇인가를 "기다리게" 되나요?

대답이 "예"이면 **I/O 바인딩된** 작업입니다.

2. 코드가 비용이 높은 계산을 수행하게 되나요?

대답이 "예"이면 **CPU 바인딩된** 작업입니다.

I/O 바인딩된 작업이 있는 경우 및 `await`를 사용하지 `async` 않습니다 `Task.Run`. 작업 병렬 라이브러리를 사용하면 안 됩니다.

있는 작업이 **CPU에 바인딩되어** 있고 응답성에 관심이 있는 경우 및 를 `await` 사용하지만 를 사용하여 `async` 다른 스레드 `Task.Run`에서 작업을 생성합니다. 작업이 동시성 및 병렬 처리에 해당할 경우 **작업 병렬 라이브러리**를 사용할 것을 고려할 수도 있습니다.

또한 항상 코드 실행을 측정해야 합니다. 예를 들어 CPU 바인딩된 작업이 다중 스레딩 시 컨텍스트 전환의 오버헤드에 의해 부담이 크지 않은 상황이 될 수 있습니다. 모든 선택에는 절충점이 있습니다. 상황에 맞는 올바른 절충점을 선택해야 합니다.

# 추가 예제

다음 예제에서는 C#에서 비동기 코드를 작성할 수 있는 다양한 방법을 보여 줍니다. 예제에서는 발생할 수 있는 몇 가지 시나리오를 다룹니다.

## 네트워크에서 데이터 추출

이 코드 조각은 홈페이지 <https://dotnetfoundation.org>에서 HTML을 다운로드하고 문자열 ".NET"이 HTML에서 발생하는 횟수를 계산합니다. 이 작업을 수행하고 횟수를 반환하는 Web API 컨트롤러 메서드를 정의하기 위해 ASP.NET을 사용합니다.

### ① 참고

프로덕션 코드에서 HTML 구문 분석을 수행하려는 경우 정규식을 사용하지 마세요.  
대신 구문 분석 라이브러리를 사용하세요.

C#

```
private readonly HttpClient _httpClient = new HttpClient();

[HttpGet, Route("DotNetCount")]
public async Task<int> GetDotNetCount()
{
    // Suspends GetDotNetCount() to allow the caller (the web server)
    // to accept another request, rather than blocking on this one.
    var html = await
    _httpClient.GetStringAsync("https://dotnetfoundation.org");

    return Regex.Matches(html, @"\.\.NET").Count;
}
```

다음은 단추가 눌릴 때 같은 작업을 수행하는 유니버설 Windows 앱용으로 작성된 동일한 시나리오입니다.

C#

```
private readonly HttpClient _httpClient = new HttpClient();

private async void OnSeeTheDotNetsButtonClick(object sender, RoutedEventArgs e)
{
    // Capture the task handle here so we can await the background task
    // later.
    var getDotNetFoundationHtmlTask =
    _httpClient.GetStringAsync("https://dotnetfoundation.org");
```

```

    // Any other work on the UI thread can be done here, such as enabling a
    Progress Bar.
    // This is important to do here, before the "await" call, so that the
    user
    // sees the progress bar before execution of this method is yielded.
    NetworkProgressBar.IsEnabled = true;
    NetworkProgressBar.Visibility = Visibility.Visible;

    // The await operator suspends OnSeeTheDotNetsButtonClick(), returning
    control to its caller.
    // This is what allows the app to be responsive and not block the UI
    thread.
    var html = await getDotNetFoundationHtmlTask;
    int count = Regex.Matches(html, @"\.\.NET").Count;

    DotNetCountLabel.Text = $"Number of .NETs on dotnetfoundation.org:
{count}";

    NetworkProgressBar.IsEnabled = false;
    NetworkProgressBar.Visibility = Visibility.Collapsed;
}

```

## 여러 작업이 완료될 때까지 대기

동시에 데이터의 여러 부분을 검색해야 하는 상황이 될 수 있습니다. `Task` API에는 여러 백그라운드 작업에서 비차단 대기를 수행하는 비동기 코드를 작성할 수 있는 `Task.WhenAll` 및 `Task.WhenAny` 메서드가 포함됩니다.

이 예제에서는 `userId` 집합에 대한 `User` 데이터를 확인하는 방법을 보여 줍니다.

C#

```

public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.
}

public static async Task<IEnumerable<User>> GetUsersAsync(IEnumerable<int>
userIds)
{
    var getUserTasks = new List<Task<User>>();
    foreach (int userId in userIds)
    {
        getUserTasks.Add(GetUserAsync(userId));
    }
}

```

```
        return await Task.WhenAll(getUserTasks);
    }
```

다음은 LINQ를 사용하여 이 코드를 보다 간결하게 작성하는 또 다른 방법입니다.

C#

```
public async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.
}

public static async Task<User[]> GetUsersAsync(IEnumerable<int> userIds)
{
    var getUserTasks = userIds.Select(id => GetUserAsync(id)).ToArray();
    return await Task.WhenAll(getUserTasks);
}
```

코드 양은 더 적지만 LINQ를 비동기 코드와 함께 사용할 때는 주의하세요. LINQ는 연기된(지연) 실행을 사용하므로, `.ToList()` 또는 `.ToArray()` 호출을 반복하도록 생성된 시퀀스를 적용해야 비동기 호출이 `foreach` 루프에서 수행되면 즉시 비동기 호출이 발생합니다. 위의 예제에서는 `Enumerable.ToArray` 쿼리를 열심히 수행하고 결과를 배열에 저장합니다. 코드를 강제로 `id => GetUserAsync(id)` 실행하고 작업을 시작합니다.

## 중요한 정보 및 조언

비동기 프로그래밍을 사용하는 경우 예기치 않은 동작을 방지할 수 있는 몇 가지 세부 정보를 고려해야 합니다.

- `async` 메서드에 다음이 있어야 `await` 합니다. 키워드(keyword) 자신의 몸에 또는 그 들은 양보하지 않습니다!

기억해야 할 중요한 정보입니다. `await` 가 `async` 메서드의 본문에서 사용되지 않으면 C# 컴파일러가 경고를 생성하지만 코드는 일반 메서드인 것처럼 컴파일 및 실행됩니다. 이는 C# 컴파일러가 비동기 메서드에 대해 생성한 상태 시스템이 아무것도 수행하지 않기 때문에 매우 비효율적입니다.

- 작성하는 모든 비동기 메서드 이름의 접미사로 “`Async`”를 추가합니다.

이 규칙을 .NET에서 사용하여 동기 및 비동기 메서드를 더 쉽게 구별할 수 있습니다. 코드에서 명시적으로 호출되지 않은 특정 메서드(예: 이벤트 처리기 또는 웹 컨트롤

러 메서드)가 반드시 적용되는 것은 아닙니다. 이러한 메서드는 코드에서 명시적으로 호출되지 않으므로 명시적으로 명명하는 것은 별로 중요하지 않습니다.

- `async void`는 이벤트 처리기에만 사용해야 합니다.

이벤트에는 반환 형식이 없어서 `Task` 및 `Task<T>`를 사용할 수 없으므로 비동기 이벤트 처리기가 작동하도록 허용하는 유일한 방법은 `async void`입니다. `async void`의 다른 사용은 TAP 모델을 따르지 않고 다음과 같이 사용이 어려울 수 있습니다.

- `async void` 메서드에서 `throw`된 예외는 해당 메서드 외부에서 `catch`될 수 없습니다.
- `async void` 메서드는 테스트하기가 어렵습니다.
- 호출자가 `async void` 메서드를 비동기로 예상하지 않을 경우 이러한 메서드는 의도하지 않은 잘못된 결과를 일으킬 수 있습니다.

- LINQ 식에서 비동기 람다를 사용할 경우 신중하게 스레드

LINQ의 람다 식은 연기된 실행을 사용합니다. 즉, 예상치 않은 시점에 코드 실행이 끝날 수 있습니다. 이 코드에 차단 작업을 도입하면 코드가 제대로 작성되지 않은 경우 교착 상태가 쉽게 발생할 수 있습니다. 또한 이 코드처럼 비동기 코드를 중첩하면 코드 실행에 대해 추론하기가 훨씬 더 어려울 수도 있습니다. 비동기 및 LINQ는 강력하지만 가능한 한 신중하고 분명하게 함께 사용되어야 합니다.

- 비차단 방식으로 작업을 기다리는 코드 작성

`Task` 가 완료될 때까지 대기하는 수단으로 현재 스레드를 차단하면 교착 상태가 발생하고 컨텍스트 스레드가 차단될 수 있고 더 복잡한 오류 처리가 필요할 수 있습니다. 다음 표에서는 비차단 방식으로 작업 대기를 처리하는 방법에 대한 지침을 제공합니다.

사용 방법	대체 방법	수행할 작업
<code>await</code>	<code>Task.Wait</code> 또는 <code>Task.Result</code>	백그라운드 작업의 결과 검색
<code>await Task.WhenAny</code>	<code>Task.WaitAny</code>	작업이 완료될 때까지 대기
<code>await Task.WhenAll</code>	<code>Task.WaitAll</code>	모든 작업이 완료될 때까지 대기
<code>await Task.Delay</code>	<code>Thread.Sleep</code>	일정 기간 대기

- 사용 `ValueTask` 고려 가능한 경우

비동기 메서드에서 `Task` 개체를 반환하면 특정 경로에 성능 병목 현상이 발생할 수 있습니다. `Task`는 참조 형식이므로 이를 사용하는 것은 개체 할당을 의미합니다.

`async` 한정자로 선언된 메서드가 캐시된 결과를 반환하거나 동기적으로 완료된 경

우 코드의 성능이 중요한 섹션에서 추가 할당에 상당한 시간이 소요될 수 있습니다. 연속 루프에서 이러한 할당이 발생하면 부담이 될 수 있습니다. 자세한 내용은 [일반화된 비동기 반환 형식을 참조하세요](#).

- **사용 고려** `ConfigureAwait(false)`

일반적인 질문은 "언제 `Task.ConfigureAwait(Boolean)` 메서드를 사용해야 하는가"입니다. 이 메서드를 사용하면 `Task` 인스턴스가 awaiter를 구성할 수 있습니다. 이는 중요한 고려 사항이며 잘못 설정할 경우 성능에 영향을 미칠 수 있고 심지어 교착 상태가 발생할 수도 있습니다. `ConfigureAwait`에 대한 자세한 내용은 [ConfigureAwait FAQ](#)를 참조하세요.

- **상태 저장 코드 작성 분량 감소**

전역 개체의 상태나 특정 메서드의 실행에 의존하지 마세요. 대신, 메서드의 반환 값에만 의존합니다. 이유

- 코드를 더 쉽게 추론할 수 있습니다.
- 코드를 더 쉽게 테스트할 수 있습니다.
- 비동기 및 동기 코드를 훨씬 더 쉽게 혼합할 수 있습니다.
- 일반적으로 함께 경합 상태를 피할 수 있습니다.
- 반환 값에 의존하면 비동기 코드를 간단히 조정할 수 있습니다.
- (이점) 이 방법은 실제로 종속성 주입에도 잘 작동합니다.

권장되는 목적은 코드에서 완전하거나 거의 완전한 [참조 투명성](#)을 달성하는 것입니다. 이렇게 하면 예측 가능하고 테스트 가능하고 유지 관리 가능한 코드베이스가 생성됩니다.

## 기타 리소스

- [작업 비동기 프로그래밍 모델\(C#\)](#).

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# 작업 비동기 프로그래밍 모델

아티클 • 2023. 03. 28.

비동기 프로그래밍을 사용하여 성능 병목 현상을 방지하고 애플리케이션의 전체적인 응답성을 향상할 수 있습니다. 그러나 비동기 애플리케이션을 쓰는 일반적인 기술이 복잡하여 해당 애플리케이션을 쓰고, 디버깅하고, 유지 관리하기 어려울 수 있습니다.

C#은 .NET 런타임에 비동기 지원을 활용하는 간단한 비동기 프로그래밍을 새로 도입했습니다. 컴파일러는 개발자가 하던 어려운 작업을 수행하고, 애플리케이션은 동기 코드와 비슷한 논리 구조를 유지합니다. 따라서 약간의 노력만으로도 비동기 프로그래밍의 모든 장점을 누릴 수 있습니다.

이 항목에서는 비동기 프로그래밍을 사용하는 시기 및 방법에 대한 개요를 제공하고 특정 세부 정보 및 예제가 포함된 지원 항목에 대한 링크가 포함되어 있습니다.

## 반응성을 향상시키는 비동기

비동기는 웹 액세스와 같이 차단 가능성이 있는 작업에 반드시 필요합니다. 웹 리소스에 대한 액세스 속도가 느리거나 지연됩니다. 동기 프로세스 안에서 이러한 활동이 차단되면 전체 애플리케이션이 기다려야 합니다. 비동기 프로세스에서 애플리케이션은 잠재적인 차단 작업이 완료될 때까지 웹 리소스에 의존하지 않는 다른 작업을 계속 수행할 수 있습니다.

다음 표에는 비동기 프로그래밍으로 응답성이 향상되는 일반적인 영역이 나와 있습니다. .NET 및 Windows 런타임에서 나열된 API에는 비동기 프로그래밍을 지원하는 메서드가 포함되어 있습니다.

애플리케이션 영역	비동기 메서드가 있는 .NET 형식	비동기 메서드가 있는 Windows 런타임 형식
웹 액세스	HttpClient	Windows.Web.Http.HttpClient SyndicationClient
파일 작업	JsonSerializer StreamReader StreamWriter XmlReader XmlWriter	StorageFile
이미지 작업		MediaCapture BitmapEncoder BitmapDecoder
WCF 프로그래밍	동기 및 비동기 작업	

모든 UI 관련 작업이 대체로 스레드 한 개를 공유하므로 비동기는 특히 UI 스레드에 액세스하는 애플리케이션의 변수를 증명합니다. 동기 애플리케이션에서 임의의 프로세스가 차단되면 모든 프로세스가 차단됩니다. 애플리케이션이 응답을 중지하면 기다리지 않고 애플리케이션이 실패한 것으로 결론을 내릴 것입니다.

비동기 메서드를 사용하면 애플리케이션이 UI에 계속 응답합니다. 예를 들어 창의 크기를 조정하거나 최소화할 수 있습니다. 또는 애플리케이션이 완료될 때까지 기다리고 싶지 않다면 애플리케이션을 종료할 수 있습니다.

비동기 기반 접근 방식을 사용하면 비동기 작업을 디자인할 때 선택할 수 있는 옵션 목록에 자동 전송과 동일한 기능을 추가할 수 있습니다. 즉, 더 적은 개발자의 노력으로 기존 비동기 프로그래밍의 이점을 모두 활용할 수 있습니다.

## 작성이 간편한 비동기 메서드

C#의 `async` 및 `await` 키워드는 비동기 프로그래밍의 핵심입니다. 이 두 개의 키워드를 사용하면 .NET Framework, .NET Core 또는 Windows 런타임의 리소스를 사용하여 동기 메서드를 만드는 것만큼 쉽게 비동기 메서드를 만들 수 있습니다. `async` 키워드를 사용하여 정의하는 비동기 메서드를 *비동기 메서드*라고 합니다.

다음 예제에서는 비동기 메서드를 보여줍니다. 코드의 거의 모든 내용이 익숙할 것입니다.

[async 및 await를 사용한 비동기 프로그래밍\(C#\)](#)에서 다운로드 가능한 전체 WPF(Windows Presentation Foundation) 예제를 찾을 수 있습니다.

C#

```
public async Task<int> GetUrlContentLengthAsync()
{
    var client = new HttpClient();

    Task<string> getStringTask =
        client.GetStringAsync("https://learn.microsoft.com/dotnet");

    DoIndependentWork();

    string contents = await getStringTask;

    return contents.Length;
}

void DoIndependentWork()
{
    Console.WriteLine("Working...");
}
```

위의 샘플에서 몇 가지 사례를 알아볼 수 있습니다. 메서드 서명부터 시작해봅시다. 여기에는 `async` 한정자가 포함됩니다. 반환 형식은 `Task<int>`입니다(추가 옵션은 "반환 형식" 섹션 참조). 메서드 이름은 `Async`로 끝납니다. 메서드의 본문에서 `GetStringAsync` 가 `Task<string>`을 반환합니다. 즉, 작업을 `await`하는 경우 `string`을 받게 됩니다 (`contents`). 작업을 대기하기 전에 `GetStringAsync`의 `string`을 사용하지 않는 작업을 수행할 수 있습니다.

`await` 연산자에 주의하세요. `GetUrlContentLengthAsync`를 일시 중단합니다.

- `GetUrlContentLengthAsync`는 `getStringTask`가 완료될 때까지 계속할 수 없습니다.
- 반면 제어는 `GetUrlContentLengthAsync`의 호출자에 반환됩니다.
- `getStringTask`가 완료되면 컨트롤이 다시 시작됩니다.
- 그런 다음, `await` 연산자가 `getStringTask`에서 `string` 결과를 검색합니다.

반환 문은 정수 결과를 지정합니다. `GetUrlContentLengthAsync`를 대기하는 메서드는 길이 값을 검색합니다.

`GetUrlContentLengthAsync`에 `GetStringAsync` 호출과 해당 완료 대기 사이에 수행할 수 있는 작업이 없는 경우 다음 단일 문을 호출하고 대기하여 코드를 단순화할 수 있습니다.

C#

```
string contents = await  
client.GetStringAsync("https://learn.microsoft.com/dotnet");
```

이전 예제가 비동기 메서드인 이유는 다음과 같은 특성 때문입니다.

- 메서드 시그니처에 `async` 한정자가 포함됩니다.
- 비동기 메서드의 이름은 규칙에 따라 "Async" 접미사로 끝납니다.
- 반환 형식은 다음 형식 중 하나입니다.
  - 메서드에 연산자 형식이 `TResult`인 Return 문이 있는 경우 `Task<TResult>`입니다.
  - 메서드에 반환 문이 포함되지 않았거나 피연산자가 없는 반환 문이 포함된 경우 `Task`입니다.
  - 비동기 이벤트 처리기를 작성하는 경우 `void`입니다.
  - `GetAwaiter` 메서드가 포함된 모든 기타 형식.

자세한 내용은 [반환 형식 및 매개 변수](#) 섹션을 참조하세요.

- 메서드는 일반적으로 비동기 작업이 완료될 때까지 메서드가 계속될 수 없는 지점을 표시하는 하나 이상의 `await` 표현을 포함하고 있습니다. 한편, 메서드가 일시 중

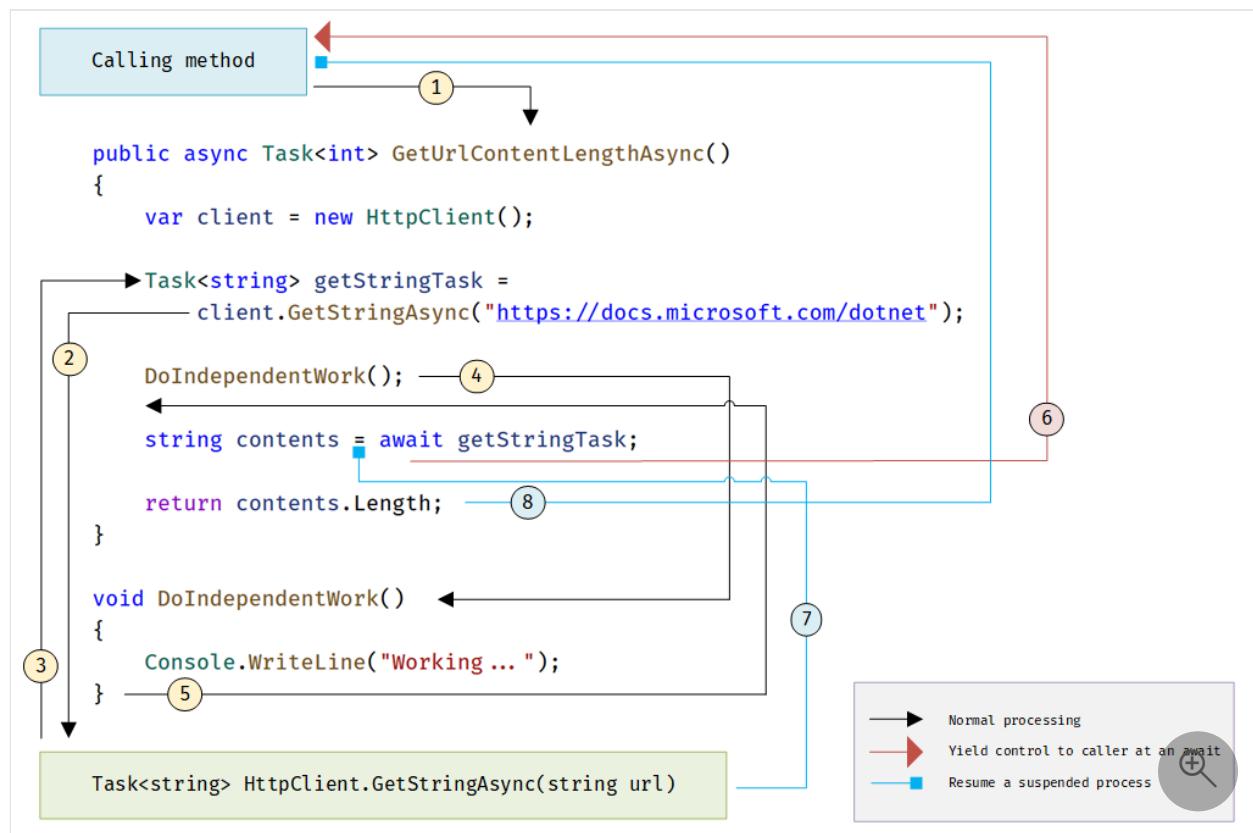
단되고 컨트롤이 메서드의 호출자로 반환됩니다. 이 항목의 다음 단원은 일시 중단 지점에서 발생하는 상황을 보여줍니다.

비동기 메서드에서는 제공된 키워드 및 형식을 사용해서 수행하려는 작업을 나타내고, 컴파일러는 일시 중단된 메서드에서 컨트롤이 대기 지점으로 반환될 때 수행되어야 하는 항목을 추적하는 등의 나머지 작업을 수행합니다. 루프 및 예외 처리와 같은 일부 루틴 프로세스는 기존의 비동기 코드에서 처리하기 어려울 수 있습니다. 비동기 메서드에서는 동기 솔루션에서와 같이 필요한 만큼 이러한 요소를 작성하여 문제를 해결합니다.

이전 버전의 .NET Framework의 비동기에 관한 자세한 내용은 [TPL 및 일반적인 .NET Framework 비동기 프로그래밍](#)을 참조하세요.

## 비동기 메서드에서 수행되는 작업

비동기 프로그래밍을 이해하는 데 있어 가장 중요한 점은 메서드에서 메서드로 제어 흐름을 이동하는 방법입니다. 다음 다이어그램에서 과정을 안내합니다.



다이어그램의 숫자는 호출하는 메서드가 비동기 메서드를 호출할 때 시작되는 다음 단계에 해당합니다.

- 호출하는 메서드는 `GetUrlContentLengthAsync` 비동기 메서드를 호출하고 기다립니다.
- `GetUrlContentLengthAsync`는 `HttpClient` 인스턴스를 만들고 `GetStringAsync` 비동기 메서드를 호출하여 웹 사이트의 내용을 문자열로 다운로드합니다.

3. `GetStringAsync`에서 특정 작업이 발생하여 진행이 일시 중단됩니다. 웹 사이트에서 다운로드 또는 다른 차단 작업을 수행할 때까지 기다려야 할 수 있습니다. 리소스를 차단하지 않기 위해 `GetStringAsync`는 해당 호출자인 `GetUrlContentLengthAsync`에 제어 권한을 양도합니다.

`GetStringAsync`는 `TResult` 가 문자열인 `Task<TResult>`를 반환하고, `GetUrlContentLengthAsync`는 `getStringTask` 변수에 작업을 할당합니다. 이 작업은 작업이 완료될 때 실제 문자열 값을 생성하기 위한 코드와 함께 `GetStringAsync`를 호출하는 지속적인 프로세스를 나타냅니다.

4. `getStringTask`가 아직 대기되지 않았으므로 `GetUrlContentLengthAsync`가 `GetStringAsync`의 최종 결과에 무관한 다른 작업을 계속할 수 있습니다. 이 작업은 동기 메서드 `DoIndependentWork`를 호출하여 나타냅니다.

5. `DoIndependentWork`는 작업을 수행하고 호출자에게 반환하는 동기 메서드입니다.

6. `GetUrlContentLengthAsync`에 `getStringTask` 결과 없이 수행할 수 있는 작업이 없습니다. 다음으로 `GetUrlContentLengthAsync`는 다운로드한 문자열의 길이를 계산하여 반환하려 하지만, 메서드가 문자열을 확인할 때까지 해당 값을 계산할 수 없습니다.

따라서 `GetUrlContentLengthAsync`는 `await` 연산자를 사용해서 해당 프로세스를 일시 중단하고 `GetUrlContentLengthAsync`를 호출한 메서드에 제어 권한을 양도합니다. `GetUrlContentLengthAsync`는 `Task<int>`를 호출자에게 반환합니다. 작업은 다운로드한 문자열의 길이인 정수 결과를 만든다는 약속을 나타냅니다.

### ① 참고

`GetUrlContentLengthAsync`가 대기하기 전에 `GetStringAsync`(및 `getStringTask`)가 완료되면 `GetUrlContentLengthAsync`에서 컨트롤이 유지됩니다. 호출된 비동기 프로세스 `getStringTask`가 이미 완료되었고 `GetUrlContentLengthAsync`가 최종 결과를 기다릴 필요가 없다면 일시 중단한 다음, `GetUrlContentLengthAsync`로 돌아가는 비용이 낭비됩니다.

호출하는 메서드 내에서 패턴 처리가 계속됩니다. 호출자가 해당 결과를 기다리거나 즉시 기다리기 전에 `GetUrlContentLengthAsync`에서 결과에 의존하지 않는 다른 작업을 수행할 수 있습니다. 호출하는 메서드는 `GetUrlContentLengthAsync`를 기다리고 있으며 `GetUrlContentLengthAsync`는 `GetStringAsync`를 기다리고 있습니다.

7. `GetStringAsync`가 완료되고 문자열 결과를 생성합니다. `GetStringAsync`를 호출할 경우 문자열 결과가 예상대로 반환되지 않습니다. (메서드가 이미 3단계에서 작업을 반환했습니다.) 대신 문자열 결과가 메서드 `getStringTask`의 완료를 나타내는 작업

에 저장됩니다. `await` 연산자가 `getStringTask`에서 결과를 검색합니다. 할당 문은 검색된 결과를 `contents`에 할당합니다.

8. `GetUrlContentLengthAsync`에 문자열 결과가 있는 경우 메서드가 문자열 길이를 계산할 수 있습니다. 그런 다음 `GetUrlContentLengthAsync` 작업도 완료되고 대기 이벤트 처리기를 다시 시작할 수 있습니다. 이 항목 뒷부분의 전체 예에서는 이벤트 처리기가 길이 결과 값을 검색하고 출력하는지 확인할 수 있습니다. 비동기 프로그래밍을 처음 접하는 사용자인 경우 동기 동작과 비동기 동작의 차이점을 살펴보세요. 동기 메서드는 작업이 완료될 때 반환되지만(5단계) 비동기 메서드는 작업이 일시 중단될 때 반환됩니다.(3~6단계) 비동기 메서드가 해당 작업을 완료하면 작업이 완료된 것으로 표시되고 결과가 있을 경우 작업에 저장됩니다.

## API 비동기 메서드

`GetStringAsync`와 같이 비동기 프로그래밍을 지원하는 메서드를 어디에서 검색해야 할지 궁금했을 것입니다. .NET Framework 4.5 이상 및 .NET Core에는 `async` 및 `await`에 작동하는 많은 멤버가 포함되어 있습니다. 멤버 이름에 붙는 “`Async`” 접미사와 `Task` 또는 `Task<TResult>`의 반환 형식으로 멤버를 인식할 수 있습니다. 예를 들어, `System.IO.Stream` 클래스는 동기 메서드인 `CopyTo`, `Read` 및 `Write`와 함께 `CopyToAsync`, `ReadAsync` 및 `WriteAsync`와 같은 메서드를 포함합니다.

Windows 런타임에는 Windows 앱에서 `async` 및 `await`와 함께 사용할 수 있는 많은 메서드도 포함되어 있습니다. 자세한 내용은 UWP 개발의 경우 [스레딩 및 비동기 프로그래밍](#)을 참조하세요. Windows 런타임 이전 버전을 사용하는 경우 [비동기 프로그래밍 \(Windows 스토어 앱\)](#) 및 [빠른 시작: C# 또는 Visual Basic](#)에서 비동기 API 호출을 참조하세요.

## 스레드

비동기 메서드는 비차단 작업으로 의도되었습니다. 비동기 메서드의 `await` 식은 대기한 작업이 실행되는 동안 현재 스레드를 차단하지 않습니다. 대신에 이 식은 메서드의 나머지를 연속으로 등록하고 제어 기능을 비동기 메서드 호출자에게 반환합니다.

`async` 및 `await` 키워드로 인해 추가 스레드가 생성되지 않습니다. 비동기 메서드는 자체 스레드에서 실행되지 않으므로 다중 스레드가 필요하지 않습니다. 메서드는 현재 동기화 컨텍스트에서 실행되고 메서드가 활성화된 경우에만 스레드에서 시간을 사용합니다. `Task.Run`을 사용하여 CPU 바인딩 작업을 백그라운드 스레드로 이동할 수 있지만 백그라운드 스레드는 결과를 사용할 수 있을 때까지 기다리는 프로세스를 도와주지 않습니다.

비동기 프로그래밍에 대한 비동기 기반 접근 방법은 거의 모든 경우에 기존 방법보다 선호됩니다. 특히, 이 접근 방식은 코드가 더 간단하고 경합 조건을 방지할 필요가 없기 때문

에 I/O 바인딩 작업의 [BackgroundWorker](#) 클래스보다 효과적입니다. 비동기 프로그래밍은 코드 실행에 대한 조합 세부 정보를 `Task.Run`이 스레드 풀로 변환하는 작업과 구분하기 때문에 `Task.Run` 메서드를 함께 사용하는 비동기 프로그래밍은 CPU 바인딩 작업을 위한 [BackgroundWorker](#)보다 효과가 뛰어납니다.

## Async 및 Await

`async` 한정자를 사용해서 메서드를 비동기 메서드로 지정하면 다음 두 기능이 활성화됩니다.

- 표시된 비동기 메서드는 [Await](#)를 사용하여 일시 중단 지점을 지정할 수 있습니다.  
`await` 연산자는 대기된 비동기 프로세스가 완료될 때까지 비동기 메서드가 해당 지점을 지나 계속할 수 없도록 컴파일러에 지시합니다. 한편, 컨트롤이 비동기 메서드의 호출자로 반환됩니다.  
  
`await` 식에서 비동기 메서드를 일시 중단하더라도 메서드가 종료되지는 않으며 `finally` 블록이 실행되지 않습니다.
- 표시된 비동기 메서드는 이 메서드를 호출한 다른 메서드에 의해 대기할 수 있습니다.

비동기 메서드는 일반적으로 `await` 연산자를 하나 이상 가지고 있지만, `await` 식이 없는 경우 컴파일러 오류가 발생하지는 않습니다. 비동기 메서드에서 `await` 연산자를 사용하여 일시 중단 시점을 표시하지 않는 경우 메서드가 `async` 한정자에 상관없이 동기 메서드가 실행되는 방식으로 실행됩니다. 컴파일러는 해당 메서드에 대해 경고를 표시합니다.

`async` 및 `await`은 상황별 키워드입니다. 자세한 내용과 예제는 다음 항목을 참조하세요.

- [async](#)
- [await](#)

## 반환 형식 및 매개 변수

비동기 메서드는 일반적으로 `Task` 또는 `Task<TResult>`를 반환합니다. 비동기 메서드 내에서 `await` 연산자는 호출에서 다른 비동기 메서드로 전환되는 작업에 적용됩니다.

메서드에 `TResult` 형식의 피연산자를 지정하는 `return` 문이 포함되어 있을 경우 `Task<TResult>`를 반환 형식으로 지정합니다.

메서드에 `return` 문이 없거나 피연산자를 반환하지 않는 `return` 문이 있을 경우 반환 형식으로 `Task`를 사용합니다.

형식에 `GetAwaiter` 메서드가 포함된 경우 다른 반환 형식을 지정할 수도 있습니다.

`ValueTask<TResult>`가 이러한 형식의 예입니다. [System.Threading.Tasks.Extension](#) ↗ NuGet 패키지에서 사용할 수 있습니다.

다음 예제는 `Task<TResult>` 또는 `Task`를 반환하는 메서드를 선언하고 호출하는 방법을 보여줍니다.

C#

```
async Task<int> GetTaskOfTResultAsync()
{
    int hours = 0;
    await Task.Delay(0);

    return hours;
}

Task<int> returnedTaskTResult = GetTaskOfTResultAsync();
int intResult = await returnedTaskTResult;
// Single line
// int intResult = await GetTaskOfTResultAsync();

async Task GetTaskAsync()
{
    await Task.Delay(0);
    // No return statement needed
}

Task returnedTask = GetTaskAsync();
await returnedTask;
// Single line
await GetTaskAsync();
```

반환된 각 작업은 진행 중인 작업을 나타냅니다. 작업은 비동기 프로세스 상태에 대한 정보를 캡슐화하며, 결과적으로 프로세스의 최종 결과 또는 성공하지 못한 경우 프로세스가 발생시키는 예외에 대한 정보를 캡슐화합니다.

비동기 메서드의 반환 형식은 `void`일 수 있습니다. 이 반환 형식은 기본적으로 `void` 반환 형식이 필요할 때 이벤트 처리기를 정의하는 데 사용합니다. 비동기 이벤트 처리기는 비동기 프로그램의 시작점 역할을 하는 경우가 많습니다.

`void` 반환 형식을 가진 비동기 메서드는 대기할 수 없습니다. 또한 `void`를 반환하는 메서드의 호출자는 메서드가 `throw`하는 예외를 `catch`할 수 없습니다.

비동기 메서드는 모든 `in`, `ref` 또는 `out` 매개 변수를 선언할 수 없지만, 이러한 매개 변수가 있는 메서드를 호출할 수는 있습니다. 마찬가지로 비동기 메서드는 참조 반환 값을 사용하여 메서드를 호출할 수 있지만 참조를 통해 값을 반환할 수 없습니다.

자세한 내용과 예제는 [비동기 반환 형식\(C#\)](#)을 참조하세요.

Windows 런타임 프로그래밍의 비동기 API에는 작업과 유사한 다음 반환 형식 중 하나가 있습니다.

- `Task<TResult>`에 해당하는 `IAsyncOperation<TResult>`
- `Task`에 해당하는 `IAsyncAction`
- `IAsyncActionWithProgress<TProgress>`
- `IAsyncOperationWithProgress<TResult,TProgress>`

## 명명 규칙

규칙에 따라 일반적으로 대기 가능한 형식(예: `Task`, `Task<T>`, `ValueTask`, `ValueTask<T>`)을 반환하는 메서드에는 "Async"로 끝나는 이름을 사용해야 합니다. 비동기 작업을 시작 하지만 대기 가능한 형식을 반환하지 않는 메서드는 "Async"로 끝나는 이름을 사용하지 않아야 하지만, "Begin", "Start" 또는 일부 다른 동사로 시작하여 이 메서드가 작업 결과를 반환하거나 예외가 발생하지 않음을 알려야 합니다.

여기서 이벤트, 기본 클래스 또는 인터페이스 계약으로 다른 이름을 제안하는 규칙을 무시할 수 있습니다. 예를 들어, `OnButtonClick`과 같은 공용 이벤트 처리기의 이름을 변경할 수 없습니다.

## 관련 문서(Visual Studio)

제목	Description
<a href="#">async 및 await를 사용하여 병렬로 여러 웹 요청을 만드는 방법(C#)</a>	동시에 여러 작업을 시작하는 방법을 보여줍니다.
<a href="#">비동기 반환 형식(C#)</a>	비동기 메서드에서 반환할 수 있는 형식을 설명하고 각 형식이 언제 적절한가를 설명합니다.
<a href="#">신호 메커니즘으로 취소 토큰이 있는 작업을 취소합니다.</a>	비동기 솔루션에 다음과 같은 기능을 추가하는 방법을 보여줍니다. <ul style="list-style-type: none"><li>- <a href="#">작업 목록 취소(C#)</a></li><li>- <a href="#">일정 기간 이후 작업 취소(C#)</a></li><li>- <a href="#">완료되면 비동기 작업 처리(C#)</a></li></ul>
<a href="#">파일 액세스에 async 사용(C#)</a>	async 및 await를 사용하여 파일에 액세스하는 이점을 나열하고 보여줍니다.
<a href="#">TAP(작업 기반 비동기 패턴)</a>	비동기 패턴에 대해 설명하고 패턴은 <code>Task</code> 및 <code>Task&lt;TResult&gt;</code> 형식을 기반으로 합니다.

제목	Description
비동기 Channel 9 비디오	비동기 프로그래밍에 대한 다양한 비디오로 연결되는 링크를 제공합니다.

## 참조

- [async 및 await를 사용한 비동기 프로그래밍](#)
- [async](#)
- [await](#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# 비동기 반환 형식(C#)

아티클 • 2023. 04. 08.

비동기 메서드의 반환 형식은 다음과 같을 수 있습니다.

- `Task` - 작업을 수행하지만 아무 값도 반환하지 않는 비동기 메서드의 경우
- `Task<TResult>` - 값을 반환하는 비동기 메서드의 경우
- `void` - 이벤트 처리기의 경우
- 액세스 가능한 `GetAwaiter` 메서드가 있는 모든 형식. `GetAwaiter` 메서드에서 반환된 개체는 `System.Runtime.CompilerServices.ICriticalNotifyCompletion` 인터페이스를 구현해야 합니다.
- `IAsyncEnumerable<T>` 비동기 스트림을 반환하는 비동기 메서드의 경우입니다.

비동기 메서드에 관한 자세한 내용은 [async 및 await를 사용한 비동기 프로그래밍\(C#\)](#)을 참조하세요.

Windows 워크로드와 관련된 몇 가지 다른 형식도 있습니다.

- `DispatcherOperation` - Windows로 제한된 비동기 작업에 사용됩니다.
- `IAsyncAction` - 값을 반환하지 않는 UWP의 비동기 작업에 사용됩니다.
- `IAsyncActionWithProgress<TProgress>` - 진행 상황을 보고하지만 값을 반환하지 않는 UWP의 비동기 작업에 사용됩니다.
- `IAsyncOperation<TResult>` - 값을 반환하는 UWP의 비동기 작업에 사용됩니다.
- `IAsyncOperationWithProgress<TResult,TProgress>` - 진행률을 보고하고 값을 반환하는 UWP의 비동기 작업에 사용됩니다.

## Task 반환 형식

`return` 문을 포함하지 않거나 피연산자를 반환하지 않는 `return` 문을 포함하는 비동기 메서드의 반환 형식은 일반적으로 `Task`입니다. 이러한 메서드는 동기적으로 실행될 경우 `void`를 반환합니다. 비동기 메서드에 대해 `Task` 반환 형식을 사용하는 경우 호출된 비동기 메서드가 완료될 때까지 호출 메서드는 `await` 연산자를 사용하여 호출자의 완료를 일시 중단할 수 있습니다.

다음 예제에서 `WaitAndApologizeAsync` 메서드에는 `return` 문이 없으므로 메서드가 `Task` 개체를 반환합니다. `Task`를 반환하면 `WaitAndApologizeAsync`가 대기할 수 있습니다. `Task` 형식에는 반환 값이 없으므로 `Result` 속성이 포함되지 않습니다.

C#

```

public static async Task DisplayCurrentInfoAsync()
{
    await WaitAndApologizeAsync();

    Console.WriteLine($"Today is {DateTime.Now:D}");
    Console.WriteLine($"The current time is {DateTime.Now.TimeOfDay:t}");
    Console.WriteLine("The current temperature is 76 degrees.");
}

static async Task WaitAndApologizeAsync()
{
    await Task.Delay(2000);

    Console.WriteLine("Sorry for the delay...\n");
}

// Example output:
//     Sorry for the delay...
//
// Today is Monday, August 17, 2020
// The current time is 12:59:24.2183304
// The current temperature is 76 degrees.

```

동기 void를 반환하는 메서드에 대한 호출 문과 비슷하게 await 식 대신 await 문을 사용하여 `WaitAndApologizeAsync`가 대기됩니다. 이 경우 await 연산자를 적용하면 값이 산출되지 않습니다. `await`의 오른쪽 피연산자가 `Task<TResult>`인 경우 `await` 식의 결과는 `T`입니다. `await`의 오른쪽 피연산자가 `Task`인 경우 `await` 및 해당 피연산자는 문입니다.

다음 코드와 같이 `WaitAndApologizeAsync` 호출을 await 연산자의 적용과 구분할 수 있습니다. 그러나 `Task`에는 `Result` 속성이 없으므로 await 연산자가 `Task`에 적용될 때 값이 생성되지 않습니다.

다음 코드에서는 `WaitAndApologizeAsync` 메서드 호출과 해당 메서드에서 반환하는 작업 대기를 구분합니다.

C#

```

Task waitAndApologizeTask = WaitAndApologizeAsync();

string output =
    $"Today is {DateTime.Now:D}\n" +
    $"The current time is {DateTime.Now.TimeOfDay:t}\n" +
    "The current temperature is 76 degrees.\n";

await waitAndApologizeTask;
Console.WriteLine(output);

```

## Task<TResult> 반환 형식

`Task<TResult>` 반환 형식은 피연산자가 `TResult` 인 `return` 문이 포함된 비동기 메서드에 사용합니다.

다음 예제에서 `GetLeisureHoursAsync` 메서드는 정수를 반환하는 `return` 문을 포함합니다. 메서드 선언은 `Task<int>`의 반환 형식을 지정해야 합니다. `FromResult` 비동기 메서드는 `DayOfWeek`를 반환하는 작업의 자리 표시자입니다.

C#

```
public static async Task ShowTodaysInfoAsync()
{
    string message =
        $"Today is {DateTime.Today:D}\n" +
        "Today's hours of leisure: " +
        $"{await GetLeisureHoursAsync()}";

    Console.WriteLine(message);
}

static async Task<int> GetLeisureHoursAsync()
{
    DayOfWeek today = await Task.FromResult(DateTime.Now.DayOfWeek);

    int leisureHours =
        today is DayOfWeek.Saturday || today is DayOfWeek.Sunday
        ? 16 : 5;

    return leisureHours;
}
// Example output:
// Today is Wednesday, May 24, 2017
// Today's hours of leisure: 5
```

`ShowTodaysInfo` 메서드의 `await` 식 내에서 `GetLeisureHoursAsync`를 호출하면 `await` 식이 `GetLeisureHours`에서 반환된 작업에 저장된 정수 값(`leisureHours` 값)을 검색합니다. `await` 식에 대한 자세한 내용은 [await](#)를 참조하세요.

다음 코드와 같이 `GetLeisureHoursAsync` 호출을 `await` 적용과 구분하면 `await`가 `Task<T>`에서 결과를 검색하는 방법을 더욱 잘 이해할 수 있습니다. 곧바로 대기 상태가 되지 않는 `GetLeisureHoursAsync` 메서드를 호출하면 메서드 선언에서 예상한 대로 `Task<int>`를 반환합니다. 예제에서 작업이 `getLeisureHoursTask` 변수에 할당됩니다. `getLeisureHoursTask`가 `Task<TResult>`이기 때문에 `TResult` 형식의 `Result` 속성을 포함합니다. 이 경우 `TResult`는 정수 형식을 나타냅니다. `await`가 `getLeisureHoursTask`에 적용되는 경우 `await` 식은 `getLeisureHoursTask`의 `Result` 속성 내용으로 평가됩니다. 값은 `ret` 변수에 할당됩니다.

## ① 중요

**Result** 속성은 차단 속성입니다. 해당 작업이 완료되기 전에 액세스하려고 하면, 작업이 완료되고 값을 사용할 수 있을 때까지 현재 활성화된 스레드가 차단됩니다. 대부분의 경우 속성에 직접 액세스하지 않고 `await`를 사용하여 값에 액세스해야 합니다.

이전 예제에서는 `Main` 메서드가 애플리케이션 종료 전에 `message`를 콘솔에 인쇄할 수 있도록 **Result** 속성의 값을 검색하여 주 스레드를 차단했습니다.

C#

```
var getLeisureHoursTask = GetLeisureHoursAsync();

string message =
    $"Today is {DateTime.Today:D}\n" +
    "Today's hours of leisure: " +
    $"{await getLeisureHoursTask}";

Console.WriteLine(message);
```

## Void 반환 형식

`void` 반환 형식이 필요한 비동기 이벤트 처리기에 `void` 반환 형식을 사용합니다. 값을 반환하지 않는 이벤트 처리기 이외의 메서드의 경우 `void`을 반환하는 비동기 메서드를 대기할 수 없기 때문에 `Task`를 대신 반환해야 합니다. 해당 메서드의 호출자는 호출된 비동기 메서드가 마치는 것을 기다리지 않고 완료될 때까지 계속 진행해야 합니다. 호출자는 비동기 메서드가 생성하는 모든 값 또는 예외와 독립되어 있어야 합니다.

`void`을 반환하는 비동기 메서드의 호출자는 메서드에서 `throw`된 예외를 `catch`할 수 없습니다. 이러한 처리되지 않은 예외로 인해 애플리케이션이 실패할 수 있습니다. `Task` 또는 `Task<TResult>`를 반환하는 메서드가 예외를 `throw`하는 경우 이 예외는 반환된 작업에 저작됩니다. 작업이 대기하는 경우 예외가 다시 `throw`됩니다. 예외를 생성할 수 있는 비동기 메서드의 반환 형식이 `Task` 또는 `Task<TResult>`이고 메서드 호출이 대기 중인지 확인합니다.

다음 예제에서는 비동기 이벤트 처리기의 동작을 보여줍니다. 예제 코드에서 비동기 이벤트 처리기는 주 스레드가 완료되면 이를 알려야 합니다. 그런 다음, 주 스레드는 비동기 이벤트 처리기가 프로그램을 종료하기 전에 완료될 때까지 대기할 수 있습니다.

C#

```
public class NaiveButton
{
    public event EventHandler? Clicked;

    public void Click()
    {
        Console.WriteLine("Somebody has clicked a button. Let's raise the
event...");
        Clicked?.Invoke(this, EventArgs.Empty);
        Console.WriteLine("All listeners are notified.");
    }
}

public class AsyncVoidExample
{
    static readonly TaskCompletionSource<bool> s_tcs = new
TaskCompletionSource<bool>();

    public static async Task MultipleEventHandlersAsync()
    {
        Task<bool> secondHandlerFinished = s_tcs.Task;

        var button = new NaiveButton();

        button.Clicked += OnButtonClicked1;
        button.Clicked += OnButtonClicked2Async;
        button.Clicked += OnButtonClicked3;

        Console.WriteLine("Before button.Click() is called...");
        button.Click();
        Console.WriteLine("After button.Click() is called...");

        await secondHandlerFinished;
    }

    private static void OnButtonClicked1(object? sender, EventArgs e)
    {
        Console.WriteLine("    Handler 1 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("    Handler 1 is done.");
    }

    private static async void OnButtonClicked2Async(object? sender,
EventArgs e)
    {
        Console.WriteLine("    Handler 2 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("    Handler 2 is about to go async...");
        await Task.Delay(500);
        Console.WriteLine("    Handler 2 is done.");
        s_tcs.SetResult(true);
    }

    private static void OnButtonClicked3(object? sender, EventArgs e)
```

```

    {
        Console.WriteLine("    Handler 3 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("    Handler 3 is done.");
    }
}

// Example output:
//
// Before button.Click() is called...
// Somebody has clicked a button. Let's raise the event...
//     Handler 1 is starting...
//     Handler 1 is done.
//     Handler 2 is starting...
//     Handler 2 is about to go async...
//     Handler 3 is starting...
//     Handler 3 is done.
// All listeners are notified.
// After button.Click() is called...
//     Handler 2 is done.

```

## 일반화된 비동기 반환 형식 및 ValueTask<TResult>

비동기 메서드는 `awaiter` 형식의 `instance` 반환하는 액세스 가능한 `GetAwaiter` 메서드가 있는 모든 형식을 반환할 수 있습니다. 또한 `GetAwaiter` 메서드가 반환하는 형식에는 `System.Runtime.CompilerServices.AsyncMethodBuilderAttribute` 특성이 있어야 합니다. 컴파일러에서 읽은 특성 또는 작업 유형 작성기 패턴에 대한 C# 사양에 대한 문서에서 자세히 알아볼 수 있습니다.

이 기능은 `await`의 피연산자에 대한 요구 사항을 설명하는 [대기 가능 식](#)을 보완합니다. 일반화된 비동기 반환 형식을 사용하면 컴파일러가 다른 형식을 반환하는 `async` 메서드를 생성할 수 있습니다. 일반화된 비동기 반환 형식은 .NET 라이브러리의 성능 향상을 지원합니다. `Task` 및 `Task<TResult>`는 참조 형식이므로 특히 타이트 루프에서 할당이 발생하는 경우 성능이 중요한 경로의 메모리 할당으로 인해 성능이 저하될 수 있습니다. 일반화된 반환 형식이 지원되면 추가 메모리 할당을 방지하기 위해 참조 형식 대신 간단한 값 형식을 반환할 수 있습니다.

.NET에서는 `System.Threading.Tasks.ValueTask<TResult>` 구조체를 일반화된 작업 반환 값의 간단한 구현으로 제공합니다. 다음 예제에서는 `ValueTask<TResult>` 구조체를 사용하여 두 주사위 굴리기 값을 검색합니다.

C#

```

class Program
{
    static readonly Random s_rnd = new Random();
}

```

```

static async Task Main() =>
    Console.WriteLine($"You rolled {await GetDiceRollAsync()}");

static async ValueTask<int> GetDiceRollAsync()
{
    Console.WriteLine("Shaking dice...");

    int roll1 = await RollAsync();
    int roll2 = await RollAsync();

    return roll1 + roll2;
}

static async ValueTask<int> RollAsync()
{
    await Task.Delay(500);

    int diceRoll = s_rnd.Next(1, 7);
    return diceRoll;
}
// Example output:
//   Shaking dice...
//   You rolled 8

```

일반화된 비동기 반환 형식 작성은 고급 시나리오이며 특수한 환경에서 사용할 수 있습니다. 비동기 코드에 대한 대부분의 시나리오에 적용되는 `Task`, `Task<T>`, `ValueTask<T>` 형식을 대신 사용하는 것이 좋습니다.

C# 10 이상에서는 비동기 반환 형식 선언 대신 비동기 메서드에 `AsyncMethodBuilder` 특성을 적용하여 해당 형식에 대한 작성기를 재정의할 수 있습니다. 일반적으로 .NET 런타임에 제공된 다른 작성기를 사용하려면 이 특성을 적용합니다.

## IAsyncEnumerable<T>를 사용하는 비동기 스트림

비동기 메서드는 로 표시되는 비동기 스트림을 반환할 `IAsyncEnumerable<T>` 수 있습니다. 비동기 스트림은 반복되는 비동기 호출을 통해 요소가 청크로 생성될 때 스트림에서 읽은 항목을 열거하는 방법을 제공합니다. 다음 예제에서는 비동기 스트림을 생성하는 비동기 메서드를 보여 줍니다.

C#

```

static async IAsyncEnumerable<string> ReadWordsFromStreamAsync()
{
    string data =
        @"This is a line of text.

```

```
        Here is the second line of text.  
        And there is one more for good measure.  
        Wait, that was the penultimate line.";  
  
    using var readStream = new StringReader(data);  
  
    string? line = await readStream.ReadLineAsync();  
    while (line != null)  
    {  
        foreach (string word in line.Split(' ',  
StringSplitOptions.RemoveEmptyEntries))  
        {  
            yield return word;  
        }  
  
        line = await readStream.ReadLineAsync();  
    }  
}
```

앞의 예제에서는 문자열의 줄을 비동기적으로 읽습니다. 각 줄을 읽은 후에는 코드가 문자열에서 각 단어를 열거합니다. 호출자는 `await foreach` 문을 사용하여 각 단어를 열거합니다. 메서드는 소스 문자열에서 다음 줄을 비동기적으로 읽어야 하는 경우 대기합니다.

## 참조

- [FromResult](#)
- [완료되면 비동기 작업 처리](#)
- [async 및 await를 사용한 비동기 프로그래밍\(C#\)](#)
- [async](#)
- [await](#)

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



### .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# 완료되면 비동기 작업 처리(C#)

아티클 • 2023. 05. 23.

[Task.WhenAny](#)를 사용하면 시작된 순서대로 처리하는 대신 동시에 여러 작업을 시작하고 완료 시 하나씩 처리할 수 있습니다.

다음 예제에서는 쿼리를 사용하여 작업 컬렉션을 만듭니다. 각 작업은 지정된 웹 사이트의 콘텐츠를 다운로드합니다. while 루프의 각 반복에서 대기된 [WhenAny](#) 호출은 다운로드를 먼저 완료하는 작업 컬렉션의 작업을 반환합니다. 해당 작업은 컬렉션에서 제거되고 처리됩니다. 컬렉션에 더 이상 작업이 없을 때까지 루프가 반복됩니다.

## 사전 요구 사항

다음 옵션 중 하나를 사용하여 이 자습서를 진행할 수 있습니다.

- **.NET 데스크톱 개발** 워크로드가 설치된 [Visual Studio 2022](#). 이 워크로드를 선택하면 .NET SDK가 자동으로 설치됩니다.
- 선택한 코드 편집기가 있는 [.NET SDK](#)(예: [Visual Studio Code](#)).

## 예제 애플리케이션 만들기

새 .NET Core 콘솔 애플리케이션을 만듭니다. `dotnet new console` 명령 또는 Visual Studio를 사용하여 만들 수 있습니다.

코드 편집기에서 `Program.cs` 파일을 열고 기존 파일을 다음 코드로 바꿉니다.

```
C#  
  
using System.Diagnostics;  
  
namespace ProcessTasksAsTheyFinish;  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        Console.WriteLine("Hello World!");  
    }  
}
```

## 필드 추가하기

`Program` 클래스 정의에 다음 두 개 필드를 추가합니다.

C#

```
static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
    "https://learn.microsoft.com/education",
    "https://learn.microsoft.com/enterprise-mobility-security",
    "https://learn.microsoft.com/gaming",
    "https://learn.microsoft.com/graph",
    "https://learn.microsoft.com/microsoft-365",
    "https://learn.microsoft.com/office",
    "https://learn.microsoft.com/powershell",
    "https://learn.microsoft.com/sql",
    "https://learn.microsoft.com/surface",
    "https://learn.microsoft.com/system-center",
    "https://learn.microsoft.com/visualstudio",
    "https://learn.microsoft.com/windows",
    "https://learn.microsoft.com/xamarin"
};
```

`HttpClient`는 HTTP 요청을 보내고 HTTP 응답을 받는 기능을 공개합니다. `s_urlList`는 애플리케이션이 처리해야 하는 모든 URL을 저장합니다.

## 애플리케이션 진입점 업데이트

콘솔 애플리케이션의 주 진입점은 `Main` 메서드입니다. 기존 메서드를 다음으로 바꿉니다.

C#

```
static Task Main() => SumPageSizesAsync();
```

업데이트된 `Main` 메서드는 이제 `Async main`으로 간주되어 실행 파일에 대한 비동기 진입점을 허용합니다. `SumPageSizesAsync`에 대한 호출로 표현됩니다.

# 비동기 합계 페이지 크기 메서드 만들기

`SumPageSizesAsync` 메서드 아래에 `Main` 메서드를 추가합니다.

C#

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"\\nTotal bytes returned: {total:#,##}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\\n");
}
```

루프는 `while` 각 반복에서 작업 중 하나를 제거합니다. 모든 작업이 완료되면 루프가 종료됩니다. `Stopwatch`를 인스턴스화하고 시작함으로써 메서드가 시작됩니다. 그런 다음, 실행 시 작업 컬렉션을 만드는 쿼리를 포함합니다. 다음 코드에서는 `ProcessUrlAsync`를 호출할 때마다 `Task<TResult>`가 반환됩니다. 여기서 `TResult`는 정수입니다.

C#

```
IEnumerable<Task<int>> downloadTasksQuery =
    from url in s_urlList
    select ProcessUrlAsync(url, s_client);
```

LINQ를 통한 [지연된 실행](#)으로 인해 `Enumerable.ToList`를 호출하여 각 작업을 시작합니다.

C#

```
List<Task<int>> downloadTasks = downloadTasksQuery.ToList();
```

`while` 루프는 컬렉션의 각 작업에서 다음 단계를 수행합니다.

1. 다운로드를 완료한 컬렉션에서 첫 번째 작업을 식별하기 위해 `WhenAny` 호출을 기다립니다.

C#

```
Task<int> finishedTask = await Task.WhenAny(downloadTasks);
```

2. 컬렉션에서 해당 작업을 제거합니다.

C#

```
downloadTasks.Remove(finishedTask);
```

3. `ProcessUrlAsync` 호출에서 반환된 `finishedTask`를 대기합니다. `finishedTask` 변수는 `Task<TResult>`입니다. 여기서 `TResult`은 정수입니다. 작업은 이미 완료되었지만, 다음 예제와 같이 다운로드한 웹 사이트의 길이를 검색하도록 기다립니다. 작업에 오류가 발생하는 경우 `await`는 `AggregateException`을 `throw`하는 `Task<TResult>.Result` 속성을 읽는 것과 달리 `AggregateException`에 저장된 첫 번째 자식 예외를 `throw`합니다.

C#

```
total += await finishedTask;
```

## 프로세스 메서드 추가

`SumPageSizesAsync` 메서드 아래에 다음 `ProcessUrlAsync` 메서드를 추가합니다.

C#

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

지정된 URL에서 메서드는 제공된 `client` 인스턴스를 사용하여 응답을 `byte[]`로 가져옵니다. URL 및 길이가 콘솔에 기록된 후 길이가 반환됩니다.

프로그램을 여러 번 실행하여 다운로드한 길이가 항상 같은 순서로 표시되는지 확인합니다.

### ⊗ 주의

예제에 설명된 대로 루프에서 `WhenAny`를 사용하는 것은 적은 수의 작업이 필요한 문제 해결에 적합합니다. 그러므로 많은 수의 작업을 처리해야 하는 경우에는 다른 접근 방법이 더 효율적입니다. 자세한 내용 및 예제는 [작업이 완료되었을 때 처리 방법](#)을 참조하세요.

## 전체 예제

다음 코드는 예제에 관한 `Program.cs` 파일의 전체 텍스트입니다.

C#

```
using System.Diagnostics;

HttpClient s_client = new()
{
    MaxResponseContentBufferSize = 1_000_000
};

IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
    "https://learn.microsoft.com/education",
    "https://learn.microsoft.com/enterprise-mobility-security",
    "https://learn.microsoft.com/gaming",
    "https://learn.microsoft.com/graph",
    "https://learn.microsoft.com/microsoft-365",
    "https://learn.microsoft.com/office",
    "https://learn.microsoft.com/powershell",
    "https://learn.microsoft.com/sql",
    "https://learn.microsoft.com/surface",
    "https://learn.microsoft.com/system-center",
    "https://learn.microsoft.com/visualstudio",
    "https://learn.microsoft.com/windows",
    "https://learn.microsoft.com/xamarin"
};

await SumPageSizesAsync();
```

```

async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"\\nTotal bytes returned: {total:#,#}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\\n");
}

static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}

// Example output:
// https://learn.microsoft.com                                         132,517
// https://learn.microsoft.com/powershell                         57,375
// https://learn.microsoft.com/gaming                            33,549
// https://learn.microsoft.com/aspnet/core                        88,714
// https://learn.microsoft.com/surface                           39,840
// https://learn.microsoft.com/enterprise-mobility-security   30,903
// https://learn.microsoft.com/microsoft-365                     67,867
// https://learn.microsoft.com/windows                          26,816
// https://learn.microsoft.com/xamarin                          57,958
// https://learn.microsoft.com/dotnet                           78,706
// https://learn.microsoft.com/graph                           48,277
// https://learn.microsoft.com/dynamics365                      49,042
// https://learn.microsoft.com/office                           67,867
// https://learn.microsoft.com/system-center                  42,887
// https://learn.microsoft.com/education                       38,636
// https://learn.microsoft.com/azure                           421,663
// https://learn.microsoft.com/visualstudio                   30,925
// https://learn.microsoft.com/sql                            54,608
// https://learn.microsoft.com/azure/devops                 86,034

// Total bytes returned: 1,454,184
// Elapsed time: 00:00:01.1290403

```

# 참고 항목

- [WhenAny](#)
- [async 및 await를 사용한 비동기 프로그래밍\(C#\)](#)

## Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# 비동기 파일 액세스(C#)

아티클 • 2024. 02. 21.

파일에 액세스하는 비동기 기능을 사용할 수 있습니다. 비동기 기능을 사용하면 콜백을 사용하거나 여러 메서드 또는 람다 식에서 코드를 분할하지 않고도 비동기 메서드를 호출할 수 있습니다. 동기 코드를 비동기로 만들려면 동기 메서드 대신 비동기 메서드를 호출하고 몇 가지 키워드를 코드에 추가하면 됩니다.

파일 액세스 호출에 비동기를 추가하는 이유로 다음을 고려할 수 있습니다.

- ✓ 비동기는 UI 애플리케이션의 응답성을 개선합니다. 작업을 시작하는 UI 스레드가 다른 작업을 수행할 수 있기 때문입니다. UI 스레드가 시간이 오래 걸리는(예: 50밀리초 이상) 코드를 실행해야 하는 경우, I/O가 완료되고 UI 스레드가 키보드와 마우스 입력 및 기타 이벤트를 다시 처리할 수 있을 때까지 UI가 정지할 수 있습니다.
- ✓ 비동기는 스레드의 필요성을 줄임으로써 ASP.NET 및 기타 서버 기반 애플리케이션의 확장성을 개선합니다. 애플리케이션이 각 응답에 전용 스레드를 사용하고 1,000 개의 요청이 동시에 처리되는 경우 수천 개의 스레드가 필요합니다. 비동기 작업은 대기 중에 종종 스레드를 사용할 필요가 없습니다. 끝날 때 기존 I/O 완료 스레드를 잠시 사용합니다.
- ✓ 파일 액세스 작업의 대기 시간은 현재 조건에서 매우 낮을 수 있지만 나중에 대기 시간이 크게 늘어날 수 있습니다. 예를 들어 전 세계에 있는 서버로 파일을 이동할 수 있습니다.
- ✓ 비동기 기능을 사용할 경우에는 추가되는 오버헤드가 적습니다.
- ✓ 비동기 작업은 쉽게 병렬로 실행할 수 있습니다.

## 적절한 클래스 사용

이 항목의 간단한 예제는 `File.WriteAllTextAsync` 및 `File.ReadAllTextAsync`를 보여 줍니다. 파일 I/O 작업에 대한 한정된 제어를 위해 운영 체제 수준에서 비동기 I/O를 일으키는 옵션이 있는 `FileStream` 클래스를 사용합니다. 이 옵션을 사용하면 많은 경우 스레드 풀 스레드가 차단되는 것을 방지할 수 있습니다. 이 옵션을 사용하도록 설정하려면 생성자 호출에서 `useAsync=true` 또는 `options=FileOptions.Asynchronous` 인수를 지정합니다.

파일 경로를 지정하여 직접 여는 경우에는 `StreamReader` 및 `StreamWriter`와 함께 해당 옵션을 사용할 수 없습니다. 그러나 `FileStream` 클래스에서 열린 `Stream`을 제공하면 이 옵션을 사용할 수 있습니다. 대기 중에는 UI 스레드가 차단되지 않으므로 스레드 풀 스레드가 차단된 경우에도 UI 앱에서 비동기 호출이 더 빠릅니다.

## 텍스트 쓰기

다음 예제에서는 파일에 텍스트를 씁니다. 각 await 문에서 메서드가 즉시 종료됩니다. 파일 I/O가 완료되면 await 문 뒤에 오는 문에서 메서드가 다시 시작됩니다. async 한정자는 await 문을 사용하는 메서드의 정의에 있습니다.

## 간단한 예

C#

```
public async Task SimpleWriteAsync()
{
    string filePath = "simple.txt";
    string text = $"Hello World";

    await File.WriteAllTextAsync(filePath, text);
}
```

## 한정된 제어 예

C#

```
public async Task ProcessWriteAsync()
{
    string filePath = "temp.txt";
    string text = $"Hello World{Environment.NewLine}";

    await WriteTextAsync(filePath, text);
}

async Task WriteTextAsync(string filePath, string text)
{
    byte[] encodedText = Encoding.Unicode.GetBytes(text);

    using var sourceStream =
        new FileStream(
            filePath,
            FileMode.Create, FileAccess.Write, FileShare.None,
            bufferSize: 4096, useAsync: true);

    await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
}
```

원래 예제에 있는 `await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);` 문은 다음의 두 문이 축약된 것입니다.

C#

```
Task theTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
await theTask;
```

첫 번째 문은 작업을 반환하여 파일 처리가 시작되도록 합니다. await가 있는 두 번째 문은 메서드를 즉시 종료하고 다른 작업을 반환하도록 합니다. 나중에 파일 처리가 완료되면 await 뒤에 오는 문으로 실행이 반환됩니다.

## 텍스트 읽기

다음 예제에서는 파일에서 텍스트를 읽습니다.

### 간단한 예

C#

```
public async Task SimpleReadAsync()
{
    string filePath = "simple.txt";
    string text = await File.ReadAllTextAsync(filePath);

    Console.WriteLine(text);
}
```

### 한정된 제어 예

텍스트가 버퍼링되고, 이 경우 `StringBuilder`에 배치됩니다. 이전 예제와 달리 await의 계산에서 값이 생성됩니다. `ReadAsync` 메서드는 `Task<Int32>`를 반환하므로 작업이 완료된 후 await 평가에서 `Int32` 값 `numRead`가 생성됩니다. 자세한 내용은 [비동기 반환 형식\(C#\)](#)을 참조하세요.

C#

```
public async Task ProcessReadAsync()
{
    try
    {
        string filePath = "temp.txt";
        if (File.Exists(filePath) != false)
        {
            string text = await ReadTextAsync(filePath);
            Console.WriteLine(text);
        }
        else
        {

```

```

        Console.WriteLine($"file not found: {filePath}");
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
}

async Task<string> ReadTextAsync(string filePath)
{
    using var sourceStream =
        new FileStream(
            filePath,
            FileMode.Open, FileAccess.Read, FileShare.Read,
            bufferSize: 4096, useAsync: true);

    var sb = new StringBuilder();

    byte[] buffer = new byte[0x1000];
    int numRead;
    while ((numRead = await sourceStream.ReadAsync(buffer, 0,
buffer.Length)) != 0)
    {
        string text = Encoding.Unicode.GetString(buffer, 0, numRead);
        sb.Append(text);
    }

    return sb.ToString();
}

```

## 병렬 비동기 I/O

다음 예제에서는 10개의 텍스트 파일을 작성하여 병렬 처리를 보여 줍니다.

### 간단한 예

C#

```

public async Task SimpleParallelWriteAsync()
{
    string folder = Directory.CreateDirectory("tempfolder").Name;
    IList<Task> writeTaskList = new List<Task>();

    for (int index = 11; index <= 20; ++ index)
    {
        string fileName = $"file-{index:00}.txt";
        string filePath = $"{folder}/{fileName}";
        string text = $"In file {index}{Environment.NewLine}";

```

```

        writeTaskList.Add(File.WriteAllTextAsync(filePath, text));
    }

    await Task.WhenAll(writeTaskList);
}

```

## 한정된 제어 예

각 파일에서 `WriteAsync` 메서드는 작업 목록에 추가되는 작업을 반환합니다. `await Task.WhenAll(tasks);` 문은 메서드를 종료하고, 파일 처리가 모든 작업에 대해 완료되면 메서드 내에서 다시 시작됩니다.

이 예제는 작업이 완료된 후 `finally` 블록에서 모든 `FileStream` 인스턴스를 닫습니다. 대신 `using` 문에 각 `FileStream`이 만들어진 경우 작업이 완료되기 전에 `FileStream`이 삭제될 수 있습니다.

성능 향상은 거의 대부분 비동기 처리가 아닌 병렬 처리에서 발생합니다. 비동기의 장점은 다중 스레드를 묶어 두지 않고 사용자 인터페이스 스레드를 묶어 두지 않는다는 것입니다.

C#

```

public async Task ProcessMultipleWritesAsync()
{
    IList<FileStream> sourceStreams = new List<FileStream>();

    try
    {
        string folder = Directory.CreateDirectory("tempfolder").Name;
        IList<Task> writeTaskList = new List<Task>();

        for (int index = 1; index <= 10; ++ index)
        {
            string fileName = $"file-{index:00}.txt";
            string filePath = $"{folder}/{fileName}";

            string text = $"In file {index}{Environment.NewLine}";
            byte[] encodedText = Encoding.Unicode.GetBytes(text);

            var sourceStream =
                new FileStream(
                    filePath,
                    FileMode.Create, FileAccess.Write, FileShare.None,
                    bufferSize: 4096, useAsync: true);

            Task writeTask = sourceStream.WriteAsync(encodedText, 0,
encodedText.Length);
            sourceStreams.Add(sourceStream);
        }

        await Task.WhenAll(writeTaskList);
    }
    finally
    {
        foreach (var stream in sourceStreams)
        {
            stream.Close();
        }
    }
}

```

```
        writeTaskList.Add(writeTask);
    }

    await Task.WhenAll(writeTaskList);
}
finally
{
    foreach (FileStream sourceStream in sourceStreams)
    {
        sourceStream.Close();
    }
}
}
```

WriteAsync 및 ReadAsync 메서드를 사용하는 경우 중간에 작업을 취소하는 데 사용할 수 있는 CancellationToken을 지정할 수 있습니다. 자세한 내용은 [관리형 스레드의 취소](#)를 참조하세요.

## 참고 항목

- [async 및 await를 사용한 비동기 프로그래밍\(C#\)](#)
- [비동기 반환 형식\(C#\)](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 작업 목록 취소

아티클 • 2023. 09. 09.

완료될 때까지 기다리지 않으려는 경우 비동기 콘솔 애플리케이션을 취소할 수 있습니다. 이 항목의 예제에 따라 웹 사이트 목록의 콘텐츠를 다운로드하는 애플리케이션에 취소를 추가할 수 있습니다. `CancellationTokenSource` 인스턴스를 각 작업과 연결하여 많은 작업을 취소할 수 있습니다. `Enter` 키를 선택하면 아직 완료되지 않은 모든 작업이 취소됩니다.

이 자습서에서는 다음 내용을 다룹니다.

- ✓ .NET 콘솔 애플리케이션 만들기
- ✓ 취소를 지원하는 비동기 애플리케이션 작성
- ✓ 취소 신호 보내기 시연

## 필수 조건

이 자습서를 사용하려면 다음이 필요합니다.

- .NET 5 이상 SDK ↗
- IDE(통합 개발 환경)
  - Visual Studio 또는 Visual Studio Code를 사용하는 것이 좋습니다. ↗

## 예제 애플리케이션 만들기

새 .NET Core 콘솔 애플리케이션을 만듭니다. `dotnet new console` 명령 또는 `Visual Studio`를 사용하여 만들 수 있습니다. 선호하는 코드 편집기에서 `Program.cs` 파일을 엽니다.

## using 문 바꾸기

기존 `using` 문을 다음 선언으로 바꿉니다.

C#

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
```

# 필드 추가

Program 클래스 정의에서 다음 세 필드를 추가합니다.

C#

```
static readonly CancellationTokenSource s_cts = new  
CancellationTokenSource();  
  
static readonly HttpClient s_client = new HttpClient  
{  
    MaxResponseContentBufferSize = 1_000_000  
};  
  
static readonly IEnumerable<string> s_urlList = new string[]  
{  
    "https://learn.microsoft.com",  
    "https://learn.microsoft.com/aspnet/core",  
    "https://learn.microsoft.com/azure",  
    "https://learn.microsoft.com/azure/devops",  
    "https://learn.microsoft.com/dotnet",  
    "https://learn.microsoft.com/dynamics365",  
    "https://learn.microsoft.com/education",  
    "https://learn.microsoft.com/enterprise-mobility-security",  
    "https://learn.microsoft.com/gaming",  
    "https://learn.microsoft.com/graph",  
    "https://learn.microsoft.com/microsoft-365",  
    "https://learn.microsoft.com/office",  
    "https://learn.microsoft.com/powershell",  
    "https://learn.microsoft.com/sql",  
    "https://learn.microsoft.com/surface",  
    "https://learn.microsoft.com/system-center",  
    "https://learn.microsoft.com/visualstudio",  
    "https://learn.microsoft.com/windows",  
    "https://learn.microsoft.com/xamarin"  
};
```

CancellationTokenSource는 요청된 취소를 CancellationToken에 신호로 보내는 데 사용됩니다. HttpClient는 HTTP 요청을 보내고 HTTP 응답을 받는 기능을 공개합니다.

s\_urlList는 애플리케이션이 처리해야 하는 모든 URL을 저장합니다.

## 애플리케이션 진입점 업데이트

콘솔 애플리케이션의 주 진입점은 Main 메서드입니다. 기존 메서드를 다음으로 바꿉니다.

C#

```

static async Task Main()
{
    Console.WriteLine("Application started.");
    Console.WriteLine("Press the ENTER key to cancel...\n");

    Task cancelTask = Task.Run(() =>
    {
        while (Console.ReadKey().Key != ConsoleKey.Enter)
        {
            Console.WriteLine("Press the ENTER key to cancel...");
        }

        Console.WriteLine("\nENTER key pressed: cancelling downloads.\n");
        s_cts.Cancel();
    });

    Task sumPageSizesTask = SumPageSizesAsync();

    Task finishedTask = await Task.WhenAny(new[] { cancelTask,
sumPageSizesTask });
    if (finishedTask == cancelTask)
    {
        // wait for the cancellation to take place:
        try
        {
            await sumPageSizesTask;
            Console.WriteLine("Download task completed before cancel request
was processed.");
        }
        catch (TaskCanceledException)
        {
            Console.WriteLine("Download task has been cancelled.");
        }
    }

    Console.WriteLine("Application ending.");
}

```

업데이트된 `Main` 메서드는 이제 `Async main`으로 간주되어 실행 파일에 대한 비동기 진입점을 허용합니다. 콘솔에 몇 가지 지침 메시지를 기록한 다음, `cancelTask`라는 `Task` 인스턴스를 선언합니다. 그러면 콘솔 키 입력이 읽힙니다. `Enter` 키를 누르면 `CancellationTokenSource.Cancel()`이 호출됩니다. 그러면 취소 신호가 전송됩니다. 다음으로, `sumPageSizesTask` 변수가 `SumPageSizesAsync` 메서드에서 할당됩니다. 두 작업은 모두 `Task.WhenAny(Task[])`에 전달되며 해당 항목은 두 작업 중 하나가 완료되면 계속됩니다.

다음 코드 블록은 취소가 처리될 때까지 애플리케이션이 종료되지 않도록 합니다. 완료할 첫 번째 작업이 이 `cancelTask sumPageSizeTask` 작업인 경우 대기합니다. 취소된 경우 대기하면 `System.Threading.Tasks.TaskCanceledException`. 블록은 해당 예외를 `catch`하고 메시지를 출력합니다.

# 비동기 합계 페이지 크기 메서드 만들기

`SumPageSizesAsync` 메서드 아래에 `Main` 메서드를 추가합니다.

C#

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urlList)
    {
        int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
        total += contentLength;
    }

    stopwatch.Stop();

    Console.WriteLine($"\\nTotal bytes returned: {total:#,#}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\\n");
}
```

`Stopwatch`를 인스턴스화하고 시작함으로써 메서드가 시작됩니다. 그런 다음, `s_urlList`의 각 URL을 반복하고 `ProcessUrlAsync`를 호출합니다. 각 반복에서 `s_cts.Token`은 `ProcessUrlAsync` 메서드에 전달되며 코드는 `Task<TResult>`를 반환합니다. 여기서 `TResult`는 정수입니다.

C#

```
int total = 0;
foreach (string url in s_urlList)
{
    int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
    total += contentLength;
}
```

## 프로세스 메서드 추가

`SumPageSizesAsync` 메서드 아래에 다음 `ProcessUrlAsync` 메서드를 추가합니다.

C#

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
{
```

```
    HttpResponseMessage response = await client.GetAsync(url, token);
    byte[] content = await response.Content.ReadAsByteArrayAsync(token);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

지정된 URL에서 메서드는 제공된 `client` 인스턴스를 사용하여 응답을 `byte[]`로 가져옵니다. `CancellationToken` 인스턴스는 `HttpClient.GetAsync(String, CancellationToken)` 및 `HttpContent.ReadAsByteArrayAsync()` 메서드에 전달됩니다. `token`은 요청된 취소를 등록하는 데 사용됩니다. URL 및 길이가 콘솔에 기록된 후 길이가 반환됩니다.

## 예제 애플리케이션 출력

콘솔

```
Application started.
Press the ENTER key to cancel...

https://learn.microsoft.com 37,357
https://learn.microsoft.com/aspnet/core 85,589
https://learn.microsoft.com/azure 398,939
https://learn.microsoft.com/azure/devops 73,663
https://learn.microsoft.com/dotnet 67,452
https://learn.microsoft.com/dynamics365 48,582
https://learn.microsoft.com/education 22,924

ENTER key pressed: cancelling downloads.

Application ending.
```

## 전체 예제

다음 코드는 예제에 관한 `Program.cs` 파일의 전체 텍스트입니다.

C#

```
using System.Diagnostics;

class Program
{
    static readonly CancellationTokenSource s_cts = new
    CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };
}
```

```
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
    "https://learn.microsoft.com/education",
    "https://learn.microsoft.com/enterprise-mobility-security",
    "https://learn.microsoft.com/gaming",
    "https://learn.microsoft.com/graph",
    "https://learn.microsoft.com/microsoft-365",
    "https://learn.microsoft.com/office",
    "https://learn.microsoft.com/powershell",
    "https://learn.microsoft.com/sql",
    "https://learn.microsoft.com/surface",
    "https://learn.microsoft.com/system-center",
    "https://learn.microsoft.com/visualstudio",
    "https://learn.microsoft.com/windows",
    "https://learn.microsoft.com/xamarin"
};

static async Task Main()
{
    Console.WriteLine("Application started.");
    Console.WriteLine("Press the ENTER key to cancel...\n");

    Task cancelTask = Task.Run(() =>
    {
        while (Console.ReadKey().Key != ConsoleKey.Enter)
        {
            Console.WriteLine("Press the ENTER key to cancel...");
        }

        Console.WriteLine("\nENTER key pressed: cancelling
downloads.\n");
        s_cts.Cancel();
    });

    Task sumPageSizesTask = SumPageSizesAsync();

    Task finishedTask = await Task.WhenAny(new[] { cancelTask,
sumPageSizesTask });
    if (finishedTask == cancelTask)
    {
        // wait for the cancellation to take place:
        try
        {
            await sumPageSizesTask;
            Console.WriteLine("Download task completed before cancel
request was processed.");
        }
    }
}
```

```

        catch (TaskCanceledException)
        {
            Console.WriteLine("Download task has been cancelled.");
        }
    }

    Console.WriteLine("Application ending.");
}

static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urlList)
    {
        int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
        total += contentLength;
    }

    stopwatch.Stop();

    Console.WriteLine($"\\nTotal bytes returned: {total:#,#}");
    Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\\n");
}

static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
{
    HttpResponseMessage response = await client.GetAsync(url, token);
    byte[] content = await response.Content.ReadAsByteArrayAsync(token);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
}

```

## 참고 항목

- CancellationToken
- CancellationTokenSource
- async 및 await를 사용한 비동기 프로그래밍(C#)

## 다음 단계

일정 기간 이후 비동기 작업 취소(C#)

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# 일정 기간 이후 비동기 작업 취소

아티클 • 2024. 02. 17.

작업이 완료될 때까지 대기하지 않으려는 경우 일정 기간 후에 `CancellationTokenSource.CancelAfter` 메서드를 사용하여 비동기 작업을 취소할 수 있습니다. 이 메서드는 `CancelAfter` 식으로 지정된 일정 기간 내에 완료되지 않은 연결된 작업의 취소를 예약합니다.

이 예제는 [작업 목록 취소\(C#\)](#)에서 개발된 코드에 추가되어 웹 사이트 목록을 다운로드하고 각 웹 사이트의 콘텐츠 길이를 표시합니다.

이 자습서에서는 다음 내용을 다룹니다.

- ✓ 기존 .NET 콘솔 애플리케이션 업데이트
- ✓ 취소 예약

## 필수 조건

이 자습서를 사용하려면 다음이 필요합니다.

- [작업 목록 취소\(C#\)](#) 자습서에서 애플리케이션을 만들었어야 함
- [.NET 5 이상 SDK](#)
- IDE(통합 개발 환경)
  - [Visual Studio 또는 Visual Studio Code를 권장합니다.](#)

## 애플리케이션 진입점 업데이트

기존 `Main` 메서드를 다음으로 바꿉니다.

C#

```
static async Task Main()
{
    Console.WriteLine("Application started.");

    try
    {
        s_cts.CancelAfter(3500);

        await SumPageSizesAsync();
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("\nTasks cancelled: timed out.\n");
    }
}
```

```

    }
    finally
    {
        s_cts.Dispose();
    }

    Console.WriteLine("Application ending.");
}

```

업데이트된 `Main` 메서드는 몇 가지 지침 메시지를 콘솔에 기록합니다. `try-catch` 내에서 `CancellationTokenSource.CancelAfter(Int32)` 호출은 취소를 예약합니다. 이렇게 하면 일정 기간 후에 취소하라는 신호가 전송됩니다.

다음으로, `SumPageSizesAsync` 메서드는 대기합니다. 예약된 취소보다 모든 URL이 더 빠르게 처리되면 애플리케이션이 종료됩니다. 그러나 모든 URL이 처리되기 전에 예약된 취소가 트리거되면 `OperationCanceledException`이 throw됩니다.

## 예제 애플리케이션 출력

콘솔

```

Application started.

https://learn.microsoft.com                                         37,357
https://learn.microsoft.com/aspnet/core                           85,589
https://learn.microsoft.com/azure                                398,939
https://learn.microsoft.com/azure/devops                         73,663

Tasks cancelled: timed out.

Application ending.

```

## 전체 예제

다음 코드는 예제에 관한 `Program.cs` 파일의 전체 텍스트입니다.

C#

```

using System.Diagnostics;

class Program
{
    static readonly CancellationTokenSource s_cts = new
    CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {

```

```
    MaxResponseContentSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
    "https://learn.microsoft.com/education",
    "https://learn.microsoft.com/enterprise-mobility-security",
    "https://learn.microsoft.com/gaming",
    "https://learn.microsoft.com/graph",
    "https://learn.microsoft.com/microsoft-365",
    "https://learn.microsoft.com/office",
    "https://learn.microsoft.com/powershell",
    "https://learn.microsoft.com/sql",
    "https://learn.microsoft.com/surface",
    "https://learn.microsoft.com/system-center",
    "https://learn.microsoft.com/visualstudio",
    "https://learn.microsoft.com/windows",
    "https://learn.microsoft.com/xamarin"
};

static async Task Main()
{
    Console.WriteLine("Application started.");

    try
    {
        s_cts.CancelAfter(3500);

        await SumPageSizesAsync();
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("\nTasks cancelled: timed out.\n");
    }
    finally
    {
        s_cts.Dispose();
    }

    Console.WriteLine("Application ending.");
}

static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urlList)
    {
```

```

        int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
        total += contentLength;
    }

stopwatch.Stop();

Console.WriteLine($"\\nTotal bytes returned: {total:#,#}");
Console.WriteLine($"Elapsed time: {stopwatch.Elapsed}\\n");
}

static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
{
    HttpResponseMessage response = await client.GetAsync(url, token);
    byte[] content = await response.Content.ReadAsByteArrayAsync(token);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
}

```

## 참고 항목

- CancellationToken
- CancellationTokenSource
- async 및 await를 사용한 비동기 프로그래밍(C#)
- 작업 목록 취소(C#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.



### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

[설명서 문제 열기](#)

[제품 사용자 의견 제공](#)

# 자습서: C# 및 .NET을 사용하여 비동기 스트림 생성 및 사용

아티클 • 2023. 10. 25.

비동기 스트림은 데이터의 스트리밍 원본을 모델링합니다. 데이터 스트림은 종종 요소를 비동기적으로 검색하거나 생성합니다. 비동기 스트리밍 데이터 소스의 자연스러운 프로그래밍 모델을 제공합니다.

이 자습서에서는 다음과 같은 작업을 수행하는 방법을 알아봅니다.

- ✓ 데이터 요소 시퀀스를 비동기적으로 생성하는 데이터 소스를 만듭니다.
- ✓ 데이터 소스를 비동기적으로 사용합니다.
- ✓ 비동기 스트림의 취소 및 캡처된 컨텍스트를 지원합니다.
- ✓ 새 인터페이스 및 데이터 소스가 이전 동기 데이터 시퀀스로 기본 설정되는 경우를 인식합니다.

## 사전 요구 사항

C# 컴파일러를 포함하여 .NET을 실행하도록 머신을 설정해야 합니다. C# 컴파일러는 [Visual Studio 2022](#) 또는 [.NET SDK](#)에서 사용할 수 있습니다.

GitHub GraphQL 엔드포인트에 액세스할 수 있도록 [GitHub 액세스 토큰](#)을 만들어야 합니다. GitHub 액세스 토큰에 사용할 다음 권한을 선택합니다.

- repo:status
- public\_repo

GitHub API 엔드포인트의 액세스 권한을 부여하는 데 사용할 수 있도록 액세스 토큰을 안전한 장소에 보관합니다.

### ⚠ 경고

개인용 액세스 토큰을 안전하게 보관합니다. 개인용 액세스 토큰이 있는 소프트웨어는 액세스 권한을 사용하여 GitHub API 호출을 수행할 수 있습니다.

이 자습서에서는 여러분이 Visual Studio 또는 .NET CLI를 비롯한 C# 및 .NET에 익숙하다고 가정합니다.

## 시작 애플리케이션 실행

비동기 프로그래밍/코드 조각 폴더의 [dotnet/docs](#) 리포지토리에서 이 자습서에서 사용되는 시작 애플리케이션에 대한 코드를 가져올 수 있습니다.

시작 애플리케이션은 [GitHub GraphQL](#) 인터페이스를 사용하여 [dotnet/docs](#) 리포지토리에 기록된 최근 문제를 검색하는 콘솔 애플리케이션입니다. 먼저 시작 앱 `Main` 메서드의 다음 코드를 살펴봅니다.

C#

```
static async Task Main(string[] args)
{
    //Follow these steps to create a GitHub Access Token
    // https://help.github.com/articles/creating-a-personal-access-token-for-the-command-line/#creating-a-token
    //Select the following permissions for your GitHub Access Token:
    // - repo:status
    // - public_repo
    // Replace the 3rd parameter to the following code with your GitHub
    access token.

    var key = GetEnvVariable("GitHubKey",
        "You must store your GitHub key in the 'GitHubKey' environment
variable",
        "");

    var client = new GitHubClient(new
Octokit.ProductHeaderValue("IssueQueryDemo"))
    {
        Credentials = new Octokit.Credentials(key)
    };

    var progressReporter = new progressStatus((num) =>
    {
        Console.WriteLine($"Received {num} issues in total");
    });
    CancellationTokenSource cancellationSource = new
CancellationTokenSource();

    try
    {
        var results = await RunPagedQueryAsync(client, PagedIssueQuery,
"docs",
            cancellationSource.Token, progressReporter);
        foreach(var issue in results)
            Console.WriteLine(issue);
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Work has been cancelled");
    }
}
```

`GitHubKey` 환경 변수를 개인용 액세스 토큰으로 설정하거나, `GetEnvVariable` 호출의 마지막 인수를 개인용 액세스 토큰으로 바꿀 수 있습니다. 소스를 다른 사용자와 공유하는 경우 소스 코드에 액세스 코드를 넣지 마세요. 공유된 소스 리포지토리에 액세스 코드를 업로드하지 마세요.

GitHub 클라이언트를 만든 후 `Main`의 코드는 진행 보고 개체 및 취소 토큰을 만듭니다. 해당 개체가 만들어지면 `Main`이 `RunPagedQueryAsync`를 호출하여 250개의 가장 최근 생성된 문제를 검색합니다. 작업이 완료되면 결과가 표시됩니다.

시작 애플리케이션을 실행할 때 이 애플리케이션의 실행 방식에 대한 몇 가지 중요한 사항을 관찰할 수 있습니다. GitHub에서 반환된 각 페이지에 대해 보고된 진행 상황이 표시됩니다. GitHub가 각 새로운 문제 페이지를 반환하기 전에 분명한 일시 정지를 관찰할 수 있습니다. 마지막으로 이 문제는 10페이지가 GitHub에서 모두 검색된 후에만 표시됩니다.

## 구현 살펴보기

구현은 이전 섹션에서 설명한 동작을 관찰한 이유를 드러냅니다. `RunPagedQueryAsync`에 대한 코드를 검사합니다.

C#

```
private static async Task<JArray> RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName, CancellationToken cancel, IProgress<int>
    progress)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    JArray finalResults = new JArray();
    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results =
        JObject.Parse(response.HttpResponse.Body.ToString()!);
```

```

        int totalCount = (int)issues(results)[ "totalCount" ]!;
        hasMorePages = (bool)pageInfo(results)[ "hasPreviousPage" ]!;
        issueAndPRQuery.Variables[ "start_cursor" ] = pageInfo(results)
        [ "startCursor" ]!.ToString();
        issuesReturned += issues(results)[ "nodes" ]!.Count();
        finalResults.Merge(issues(results)[ "nodes" ]!);
        progress?.Report(issuesReturned);
        cancel.ThrowIfCancellationRequested();
    }
    return finalResults;
}

JObject issues(JObject result) => (JObject)result[ "data" ]!
[ "repository" ]![ "issues" ]!;
JObject pageInfo(JObject result) => (JObject)issues(result)
[ "pageInfo" ]!;
}

```

이 메서드가 가장 먼저 수행하는 작업은 클래스를 사용하여 POST 개체를 `GraphQLRequest` 만드는 것입니다.

C#

```

public class GraphQLRequest
{
    [JsonProperty("query")]
    public string? Query { get; set; }

    [JsonProperty("variables")]
    public IDictionary<string, object> Variables { get; } = new
    Dictionary<string, object>();

    public string ToJsonText() =>
        JsonConvert.SerializeObject(this);
}

```

POST 개체 본문을 형성하고 메서드를 사용하여 단일 문자열로 표시되는 JSON으로 `ToJsonText` 을바르게 변환하는 데 도움이 되며, 요청 본문에서 줄 바꿈 문자를 모두 제거하여 (백슬래시) 이스케이프 문자로 \ 표시합니다.

앞 코드의 페이징 알고리즘 및 비동기 구조를 중점적으로 살펴보겠습니다. (GitHub GraphQL API에 대한 자세한 내용은 [GitHub GraphQL 설명서](#)를 참조하세요.)

`RunPagedQueryAsync` 메서드는 가장 최근에서 가장 오래된 순서로 문제를 열거합니다. 이 메서드는 페이지당 25개 문제를 요청하고 응답의 `pageInfo` 구조체를 검사하여 이전 페이지를 계속 진행합니다. 다중 페이지 응답에 대한 GraphQL의 표준 페이징 지원을 따릅니다. 응답에는 이전 페이지를 요청하는 데 사용되는 `hasPreviousPages` 값과 `startCursor` 값을 포함하는 `pageInfo` 개체가 포함됩니다. 문제는 `nodes` 배열에 있습니다.

`RunPagedQueryAsync` 메서드는 모든 페이지의 모든 결과를 포함하는 배열에 해당 노드를 추가합니다.

결과 페이지를 검색 및 복원한 후에 `RunPagedQueryAsync`가 진행 상황을 보고하고 취소를 확인합니다. 취소가 요청된 경우 `RunPagedQueryAsync`가 `OperationCanceledException`을 throw합니다.

이 코드에서 여러 가지 요소를 개선할 수 있습니다. 가장 중요한 것은 `RunPagedQueryAsync`가 반환된 모든 문제에 대해 스토리지를 할당해야 한다는 것입니다. 모든 미해결 문제를 검색하면 모든 검색된 문제를 저장하는 데 훨씬 더 많은 메모리가 필요하므로 이 샘플은 250개 문제만 검색합니다. 진행률 보고서 및 취소 지원 프로토콜로 인해 알고리즘을 처음 읽을 때 이해하기가 더 어려워집니다. 추가 형식 및 API가 포함됩니다. 취소 요청 위치 및 제공 위치를 파악하려면 `CancellationTokenSource` 및 연결된 `CancellationToken`을 통해 통신을 추적해야 합니다.

## 더 나은 방법을 제공하는 비동기 스트림

비동기 스트림 및 연결된 언어 지원으로 해당 문제가 모두 해결됩니다. 시퀀스를 생성하는 코드에서 이제 `yield return`을 사용하여 `async` 한정자로 선언된 메서드에서 요소를 반환할 수 있습니다. `foreach` 루프를 통해 시퀀스를 사용하는 것처럼 `await foreach` 루프를 통해 비동기 스트림을 사용할 수 있습니다.

이 새로운 언어 기능은 .NET Standard 2.1에 추가되고 .NET Core 3.0에 구현된 세 가지 새 인터페이스를 사용합니다.

- `System.Collections.Generic.IAsyncEnumerable<T>`
- `System.Collections.Generic.IAsyncEnumerator<T>`
- `System.IAsyncDisposable`

이 세 가지 인터페이스는 대부분의 C# 개발자에게 익숙합니다. 이 인터페이스는 동기 인터페이스와 유사한 방식으로 작동합니다.

- `System.Collections.Generic.IEnumerable<T>`
- `System.Collections.Generic.IEnumerator<T>`
- `System.IDisposable`

익숙하지 않을 수도 있는 하나의 형식은 `System.Threading.Tasks.ValueTask`입니다.

`ValueTask` 구조체는 `System.Threading.Tasks.Task` 클래스에 유사한 API를 제공합니다. `ValueTask`는 성능상의 이유로 해당 인터페이스에서 사용됩니다.

## 비동기 스트림으로 변환

그런 다음, `RunPagedQueryAsync` 메서드를 변환하여 비동기 스트림을 생성합니다. 먼저 `RunPagedQueryAsync`의 시그니처를 변경하여 `IEnumerable<JToken>`을 반환하고 다음 코드에 표시된 대로 매개 변수 목록에서 취소 토큰 및 진행 개체를 제거합니다.

C#

```
private static async IEnumerable<JToken>
RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
```

시작 코드는 다음 코드에 표시된 대로 페이지가 검색될 때 각 페이지를 처리합니다.

C#

```
finalResults.Merge(issues(results)[ "nodes" ]!);
progress?.Report(issuesReturned);
cancel.ThrowIfCancellationRequested();
```

해당 세 줄을 다음 코드로 바꿉니다.

C#

```
foreach (JObject issue in issues(results)[ "nodes" ]!)
    yield return issue;
```

이 메서드의 앞부분에 있는 `finalResults` 선언 및 수정한 루프 뒤에 있는 `return` 문을 제거할 수도 있습니다.

비동기 스트림을 생성하기 위한 변경을 완료했습니다. 완료된 메서드는 다음 코드와 유사합니다.

C#

```
private static async IEnumerable<JToken>
RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables[ "repo_name" ] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;
```

```

// Stop with 10 pages, because these are large repos:
while (hasMorePages && (pagesReturned++ < 10))
{
    var postBody = issueAndPRQuery.ToJsonText();
    var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
    postBody, "application/json", "application/json");

    JObject results =
JObject.Parse(response.HttpResponse.Body.ToString()!);

    int totalCount = (int)issues(results)["totalCount"]!;
    hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
    issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
    issuesReturned += issues(results)["nodes"]!.Count();

    foreach (JObject issue in issues(results)["nodes"]!)
        yield return issue;
}

JObject issues(JObject result) => (JObject)result["data"]!
["repository"]![["issues"]];
JObject pageInfo(JObject result) => (JObject)issues(result)
["pageInfo"]!;
}

```

그런 다음, 컬렉션을 사용하는 코드를 변경하여 비동기 스트림을 사용합니다. 문제 컬렉션을 처리하는 `Main`에서 다음 코드를 찾습니다.

C#

```

var progressReporter = new progressStatus((num) =>
{
    Console.WriteLine($"Received {num} issues in total");
});
CancellationTokenSource cancellationSource = new CancellationTokenSource();

try
{
    var results = await RunPagedQueryAsync(client, PagedIssueQuery, "docs",
        cancellationSource.Token, progressReporter);
    foreach(var issue in results)
        Console.WriteLine(issue);
}
catch (OperationCanceledException)
{
    Console.WriteLine("Work has been cancelled");
}

```

해당 코드를 다음 `await foreach` 루프로 바꿉니다.

C#

```
int num = 0;
await foreach (var issue in RunPagedQueryAsync(client, PagedIssueQuery,
"docs"))
{
    Console.WriteLine(issue);
    Console.WriteLine($"Received {++num} issues in total");
}
```

새 인터페이스 `IAsyncEnumerable<T>`는 `IAsyncDisposable`에서 파생됩니다. 즉, 루프가 완료되면 이전 루프가 스트림을 비동기적으로 삭제합니다. 루프는 다음 코드와 같이 표시될 수 있습니다.

C#

```
int num = 0;
var enumerator = RunPagedQueryAsync(client, PagedIssueQuery,
"docs").GetAsyncEnumerator();
try
{
    while (await enumerator.MoveNextAsync())
    {
        var issue = enumerator.Current;
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
} finally
{
    if (enumerator != null)
        await enumerator.DisposeAsync();
}
```

기본적으로 스트림 요소는 캡처된 컨텍스트에서 처리됩니다. 컨텍스트 캡처를 사용하지 않도록 설정하려면 `TaskAsyncEnumerableExtensions.ConfigureAwait` 확장 메서드를 사용합니다. 동기화 컨텍스트 및 현재 컨텍스트 캡처에 대한 자세한 내용은 [작업 기반 비동기 패턴 사용](#)에 대한 문서를 참조하세요.

비동기 스트림은 다른 `async` 메서드와 동일한 프로토콜을 사용하여 취소를 지원합니다. 취소를 지원하기 위해 다음과 같이 비동기 반복기 메서드의 시그니처를 수정합니다.

C#

```
private static async IAsyncEnumerable<JToken>
RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName, [EnumeratorCancellation]
CancellationToken cancellationToken = default)
{
    var issueAndPRQuery = new GraphQLRequest
```

```

{
    Query = queryText
};

issueAndPRQuery.Variables["repo_name"] = repoName;

bool hasMorePages = true;
int pagesReturned = 0;
int issuesReturned = 0;

// Stop with 10 pages, because these are large repos:
while (hasMorePages && (pagesReturned++ < 10))
{
    var postBody = issueAndPRQuery.ToJsonText();
    var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
    postBody, "application/json", "application/json");

    JObject results =
JObject.Parse(response.HttpResponse.Body.ToString()!);

    int totalCount = (int)issues(results)["totalCount"]!;
    hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
    issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
    issuesReturned += issues(results)["nodes"]!.Count();

    foreach (JObject issue in issues(results)["nodes"]!)
        yield return issue;
}

JObject issues(JObject result) => (JObject)result["data"]!
["repository"]![["issues"]];
JObject pageInfo(JObject result) => (JObject)issues(result)
[["pageInfo"]];
}

```

`System.Runtime.CompilerServices.EnumeratorCancellationAttribute` 특성을 사용하면 컴파일러는 비동기 반복기 본문에 표시되는 `GetAsyncEnumerator`에 전달된 토큰을 해당 인수로 만드는 `IAsyncEnumerable<T>`에 관한 코드를 생성합니다. `runQueryAsync` 내에서 토큰 상태를 검사하고 요청 시 추가 작업을 취소할 수 있습니다.

다른 확장 메서드인 `WithCancellation`을 사용하여 취소 토큰을 비동기 스트림에 전달합니다. 다음과 같이 문제를 열거하는 루프를 수정합니다.

C#

```

private static async Task EnumerateWithCancellation(GitHubClient client)
{
    int num = 0;
    var cancellation = new CancellationTokenSource();
    await foreach (var issue in RunPagedQueryAsync(client, PagedIssueQuery,

```

```
"docs")
    .WithCancellation(cancellation.Token))
{
    Console.WriteLine(issue);
    Console.WriteLine($"Received {++num} issues in total");
}
}
```

완료된 자습서에 대한 코드는 비동기 프로그래밍/코드 조각 폴더의 [dotnet/docs](#) 리포지토리에서 가져올 수 있습니다.

## 완료된 애플리케이션 실행

애플리케이션을 다시 실행합니다. 해당 동작을 시작 애플리케이션의 동작과 대조합니다. 결과의 첫 번째 페이지는 사용 가능한 즉시 열거됩니다. 새로운 각 페이지가 요청 및 검색될 때 확인 가능한 일시 정지가 있고 난 뒤 다음 페이지의 결과가 빠르게 열거됩니다. 취소를 처리하는 데는 `try / catch` 블록이 필요하지 않습니다. 호출자가 컬렉션의 열거를 중지할 수 있습니다. 각 페이지가 다운로드될 때 비동기 스트림이 결과를 생성하므로 진행이 명확하게 보고됩니다. 반환된 각 문제에 대한 상태는 `await foreach` 루프에 포함됩니다. 진행 상황을 추적하는 데 콜백 개체가 필요하지 않습니다.

코드를 검사하여 향상된 메모리 사용을 확인할 수 있습니다. 열거되기 전에 모든 결과를 저장하기 위해 더 이상 컬렉션을 할당할 필요가 없습니다. 호출자는 결과를 사용하는 방법 및 스토리지 컬렉션이 필요한지 여부를 결정할 수 있습니다.

시작 및 완료된 애플리케이션을 둘 다 실행하고 직접 구현 간 차이를 관찰할 수 있습니다. 완료한 후 이 자습서를 시작할 때 만든 GitHub 액세스 토큰을 삭제할 수 있습니다. 공격자가 해당 토큰의 액세스 권한을 얻으면 사용자의 자격 증명을 사용하여 GitHub API에 액세스할 수 있습니다.

이 자습서에서는 비동기 스트림을 사용하여 데이터 페이지를 반환하는 네트워크 API에서 개별 항목을 읽었습니다. 비동기 스트림은 주식 시세 또는 센서 디바이스와 같은 "종료되지 않는 스트림"에서 읽을 수도 있습니다. 사용 가능한 즉시 다음 항목을 반환하는 `MoveNextAsync` 호출입니다.

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할



### .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

### 설명서 문제 열기

수도 있습니다. 자세한 내용은  
참여자 가이드를 참조하세요.

☞ 제품 사용자 의견 제공

# nullable 참조 형식

아티클 • 2024. 03. 10.

null 허용 인식 불가능 컨텍스트에서는 모든 참조 형식이 null을 허용합니다. null 허용 참조 형식은 코드로 인해 런타임에서 `System.NullReferenceException`이 throw될 가능성을 최소화하는 null 허용 인식 컨텍스트에서 사용하도록 설정된 기능 그룹을 나타냅니다. `Nullable` 참조 형식에는 참조 형식을 `Nullable`로 명시적으로 표시하는 기능을 포함하여 이러한 예외를 방지하는 데 도움이 되는 세 가지 기능이 포함되어 있습니다.

- 변수를 역참조하기 전에 변수가 `null` 일 수 있는지 여부를 확인하는 정적 흐름 분석이 향상되었습니다.
- 흐름 분석에서 `null-state`를 확인하도록 API에 주석을다는 특성입니다.
- 개발자가 변수에 대해 의도하는 `null-state`를 명시적으로 선언하는 데 사용하는 변수 주석입니다.

컴파일러는 컴파일 시간에 코드에 있는 모든 식의 `null-state`를 추적합니다. `null-state`에는 다음 세 가지 값 중 하나가 있습니다.

- `not-null`: 식이 `not-null`로 알려져 있습니다.
- `maybe-null`: 식은 `null` 일 수 있습니다.
- `인식 불가능`: 컴파일러가 식의 `null-state`를 확인할 수 없습니다.

변수 주석은 참조 형식 변수의 Null 허용 여부를 결정합니다.

- `null`을 허용하지 않음: 변수에 `null` 값이나 `maybe-null` 식을 할당하면 컴파일러가 경고를 표시합니다. `null`을 허용하지 않는 변수의 기본 `null-state`는 `not-null`입니다.
- `null` 허용: 변수에 `null` 값 또는 `maybe-null` 식을 할당할 수 있습니다. 변수의 `null-state`가 `maybe-null`인 경우 변수를 역참조하면 컴파일러는 경고를 표시합니다. 변수의 기본 `null-state`는 `maybe-null`입니다.
- `인식 불가능`: 변수에 `null` 값 또는 `maybe-null` 식을 할당할 수 있습니다. 변수를 역참조하거나 변수에 `maybe-null` 식을 할당할 때 컴파일러는 경고를 발급하지 않습니다.

`인식 불가능` `null-state` 및 `인식 불가능` `null` 허용 여부는 `null` 허용 참조 형식이 도입되기 전의 동작과 일치합니다. 이러한 값은 마이그레이션 중에 또는 앱이 `null` 허용 참조 형식을 사용하도록 설정하지 않은 라이브러리를 사용할 때 유용합니다.

`null-state` 분석 및 변수 주석은 기존 프로젝트에 대해 기본적으로 사용하지 않도록 설정됩니다. 이는 모든 참조 형식이 `null`을 계속 허용함을 의미합니다. .NET 6부터 새 프로젝트에 대해 기본적으로 활성화됩니다. `Nullable` 주석 컨텍스트를 선언하여 이러한 기능을 활성화하는 방법에 대한 자세한 내용은 [Nullable 컨텍스트](#)를 참조하세요.

이 문서의 나머지 부분에서는 코드가 `null` 값을 역참조할 수도 있는 경우 이 세 가지 가능한 영역이 경고를 생성하는 방법을 설명합니다. 변수를 역참조하는 것은 다음 예와 같이 `.(점)` 연산자를 사용하여 해당 멤버 중 하나에 액세스하는 것을 의미합니다.

C#

```
string message = "Hello, World!";
int length = message.Length; // dereferencing "message"
```

값이 `null`인 변수를 역참조하면 런타임에서 `System.NullReferenceException`을 throw합니다.

학습 내용은 다음과 같습니다.

- 컴파일러의 [null-state 분석](#): 컴파일러가 식이 not-null 또는 maybe-null인지 확인하는 방법입니다.
- 컴파일러의 null-state 분석을 위해 더 많은 컨텍스트를 제공하는 API에 적용되는 [특성](#)입니다.
- 변수 의도에 대한 정보를 제공하는 [null 허용 변수 주석](#)입니다. 주석은 필드가 멤버 메서드 시작 부분에 기본 null-state를 설정하는 데 유용합니다.
- [제네릭 형식 인수](#)를 관리하는 규칙입니다. 형식 매개 변수가 참조 형식 또는 값 형식 일 수 있으므로 새로운 제약 조건이 추가되었습니다. ? 접미사는 null 허용 값 형식과 null 허용 참조 형식에 대해 다르게 구현됩니다.
- [null 허용 컨텍스트](#)는 대규모 프로젝트를 마이그레이션하는 데 도움이 됩니다. 마이그레이션할 때 앱의 일부에서 null 허용 컨텍스트 또는 경고를 사용하도록 설정할 수 있습니다. 추가 경고를 해결한 후 전체 프로젝트에 대해 null 허용 참조 형식을 사용하도록 설정할 수 있습니다.

마지막으로 `struct` 형식 및 배열의 null-state 분석에 대해 알려진 문제에 대해 알아봅니다.

C#의 null 허용 안전성에 대한 학습 모듈에서 이러한 개념을 살펴볼 수도 있습니다.

## null-state 분석

null 허용 참조 형식이 사용하도록 설정되면 [null-state 분석](#)은 참조의 null-state를 추적합니다. 식은 *not-null* 또는 *maybe-null*입니다. 컴파일러는 다음 두 가지 방법으로 변수가 *not-null*임을 확인합니다.

1. 변수에 *not-null*로 알려진 값이 할당되었습니다.
2. 변수가 `null`인지 검사되었으며 해당 검사 이후 수정되지 않음.

null 허용 참조 형식이 사용하도록 설정되지 않은 경우 모든 식은 인식 불가능의 null-state를 가집니다. 섹션의 나머지 부분에서는 null 허용 참조 형식이 사용하도록 설정된 경우의 동작을 설명합니다.

컴파일러에서 *not-null*로 확인되지 않은 변수는 *maybe-null*로 간주됩니다. 분석에서는 실수로 `null` 값을 역참조할 수 있는 상황에 대해 경고를 제공합니다. 컴파일러는 *null-state*를 기반으로 경고를 생성합니다.

- 변수가 *not-null*인 경우 해당 변수는 안전하게 역참조될 수 있습니다.
- 변수가 *maybe-null*인 경우 해당 변수를 역참조하기 전에 검사하여 변수가 `null`이 아닌지 확인해야 합니다.

다음 예제를 참조하세요.

C#

```
string message = null;

// warning: dereference null.
Console.WriteLine($"The length of the message is {message.Length}");

var originalMessage = message;
message = "Hello, World!";

// No warning. Analysis determined "message" is not-null.
Console.WriteLine($"The length of the message is {message.Length}");

// warning!
Console.WriteLine(originalMessage.Length);
```

앞의 예제에서 컴파일러는 첫 번째 메시지가 인쇄될 때 `message`가 *maybe-null*임을 확인합니다. 두 번째 메시지에 대한 경고는 없습니다. 마지막 코드 줄은 `originalMessage`가 null일 수 있으므로 경고를 생성합니다. 다음 예는 노드 트리를 루트로 트래버스하고 트래버스 중에 각 노드를 처리하는 더 실용적인 용도를 보여 줍니다.

C#

```
void FindRoot(Node node, Action<Node> processNode)
{
    for (var current = node; current != null; current = current.Parent)
    {
        processNode(current);
    }
}
```

이전 코드는 `current` 변수를 역참조하기 위한 경고를 생성하지 않습니다. 정적 분석에서 `current` 가 *maybe-null*이면 이 변수를 역참조하지 않는 것으로 확인합니다.

`current.Parent`에 액세스하기 전과 `current`를 `ProcessNode` 작업에 전달하기 전에 `current` 변수가 `null`인지 검사됩니다. 이전 예제에서는 지역 변수가 초기화 또는 할당되거나 `null`과 비교될 때 컴파일러가 해당 변수의 *null-state*를 확인하는 방법을 보여 줍니다.

*null-state* 분석은 호출된 메서드를 추적하지 않습니다. 결과적으로 모든 생성자가 호출하는 공통 도우미 메서드에서 초기화된 필드는 다음 템플릿을 사용하여 경고를 생성합니다.

`null`을 허용하지 않는 속성 '`name`'은 생성자를 종료할 때 non-null 값을 포함해야 합니다.

생성자 연결 또는 도우미 메서드의 *null* 허용 특성이라는 두 가지 방법 중 하나로 이러한 경고를 해결할 수 있습니다. 다음 코드에서는 각 예제를 보여 줍니다. `Person` 클래스는 다른 모든 생성자가 호출하는 공통 생성자를 사용합니다. `Student` 클래스에는 `System.Diagnostics.CodeAnalysis.MemberNotNullAttribute` 특성으로 주석이 달린 도우미 메서드가 있습니다.

C#

```
using System.Diagnostics.CodeAnalysis;

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public Person() : this("John", "Doe") { }
}

public class Student : Person
{
    public string Major { get; set; }

    public Student(string firstName, string lastName, string major)
        : base(firstName, lastName)
    {
        SetMajor(major);
    }

    public Student(string firstName, string lastName) :
        base(firstName, lastName)
    {
```

```

        SetMajor();
    }

    public Student()
    {
        SetMajor();
    }

    [MemberNotNull(nameof(Major))]
    private void SetMajor(string? major = default)
    {
        Major = major ?? "Undeclared";
    }
}

```

### ① 참고

C# 10에는 한정된 할당 및 null-state 분석에 몇 가지 개선이 적용되었습니다. C# 10으로 업그레이드하면 가양성인 null 허용 경고가 줄어든 것을 알 수 있습니다. [한정된 할당 개선 기능 사양](#)에서 개선 사항에 대해 자세히 알아보세요.

null 허용 상태 분석 및 컴파일러가 생성하는 경고는 `null`을 역참조하여 프로그램 오류를 방지하는 데 도움이 됩니다. [null 허용 경고 해결](#)에 대한 문서에서는 코드에 표시될 가능성이 가장 높은 경고를 수정하는 기술을 제공합니다.

## API 시그니처의 특성

null-state 분석에는 API의 의미 체계를 이해하기 위한 개발자의 힌트가 필요합니다. 일부 API는 null 검사를 제공하며 변수의 *null-state*를 *maybe-null*에서 *not-null*로 변경합니다. 다른 API는 입력 인수의 *null-state*에 따라 *not-null* 또는 *maybe-null*인 식을 반환합니다. 예를 들어, 메시지를 대문자로 표시하는 다음 코드를 살펴봅니다.

C#

```

void PrintMessageUpper(string? message)
{
    if (!IsNull(message))
    {
        Console.WriteLine($"{DateTime.Now}: {message.ToUpper()}");
    }
}

bool IsNull(string? s) => s == null;

```

검사를 기반으로 개발자는 이 코드를 안전한 것으로 간주하며 코드가 경고를 생성해서는 안 됩니다. 그러나 컴파일러는 `IsNotNull`이 `null` 검사를 제공하고 `message`를 `maybe-null` 변수로 간주하여 `message.ToUpper()` 문에 대해 경고를 발급한다는 사실을 알지 못합니다. 이 경고를 수정하려면 [NotNullWhen](#) 특성을 사용합니다.

C#

```
bool IsNotNull([NotNullWhen(false)] string? s) => s == null;
```

이 특성은 `IsNotNull`이 `false`를 반환하는 경우 매개 변수 `s`가 `null`이 아님을 컴파일러에 알립니다. 컴파일러는 `if (!IsNotNull(message)) {...}` 블록 내에서 `message`의 *null-state*를 *not-null*로 변경합니다. 경고가 발급되지 않습니다.

특성은 멤버를 호출하는 데 사용되는 개체 인스턴스의 인수, 반환 값, 멤버의 *null-state*에 대해 자세한 정보를 제공합니다. 각 특성에 대한 자세한 내용은 [null 허용 참조 특성](#)에 대한 언어 참조 문서에서 찾을 수 있습니다. .NET 5부터는 모든 .NET 런타임 API에 주석이 추가됩니다. 인수 및 반환 값의 *null-state*에 대한 의미 체계 정보를 제공하도록 고유 API에 주석을 달아 정적 분석을 개선합니다.

## null 허용 변수 주석

*null-state* 분석은 지역 변수에 대한 강력한 분석을 제공합니다. 멤버 변수에 대한 더 자세한 정보를 컴파일러에 제공해야 합니다. 멤버의 왼쪽 괄호에 있는 모든 필드의 *null-state*를 설정하려면 컴파일러에 추가 정보가 필요합니다. 액세스 가능한 생성자는 개체를 초기화하는 데 사용할 수 있습니다. 멤버 필드를 `null`로 설정할 수 있는 경우 컴파일러는 각 메서드의 시작 부분에서 *null-state*가 *maybe-null*이라고 가정해야 합니다.

변수가 [null 허용 참조 형식](#) 또는 [null을 허용하지 않는 참조 형식](#)인지 선언할 수 있는 주석을 사용합니다. 이러한 주석은 변수의 *null-state*에 대한 중요한 문을 생성합니다.

- **참조는 `null`이 아니어야 합니다.** `null`을 허용하지 않는 참조 변수의 기본 상태는 *not-null*입니다. 컴파일러는 `null`이 아닌지 먼저 확인하지 않고 이러한 참조를 역참조하기에 안전한지 확인하는 규칙을 적용합니다.
  - 이 변수는 `null`이 아닌 값으로 초기화되어야 합니다.
  - 이 변수에는 `null` 값을 절대로 할당할 수 없습니다. 코드가 `null`이면 안 되는 변수에 *maybe-null* 식을 할당하면 컴파일러가 경고를 발생시킵니다.
- **참조가 `null`일 수 있습니다.** `null` 허용 참조 변수의 기본 상태는 *maybe-null*입니다. 컴파일러는 `null` 참조를 올바르게 확인하도록 규칙을 적용합니다.
  - 변수는 값이 `null`이 아니라는 것을 컴파일러가 보장할 수 있는 경우에만 역참조 될 수 있습니다.

- 이러한 변수는 기본 `null` 값으로 초기화될 수 있으며 다른 코드에서는 `null` 값이 할당될 수 있습니다.
- 코드가 `null`일 수도 있는 변수에 *maybe-null* 식을 할당하면 컴파일러가 경고를 발생시키지 않습니다.

모든 `null`을 허용하지 않는 참조 변수에는 기본 *null-state*인 *not-null*이 있습니다. `null`을 허용하는 모든 참조 변수에는 초기 *null-state*인 *maybe-null*이 있습니다.

**nullable 참조 형식**은 [nullable 값 형식](#)과 동일한 구문을 사용하여 작성합니다. 변수 형식에 `?` 가 추가됩니다. 예를 들어 다음 변수 선언은 nullable 문자열 변수 `name`을 나타냅니다.

C#

```
string? name;
```

`null` 허용 참조 형식이 사용하도록 설정되면 형식 이름에 `?` 이 추가되지 않은 모든 변수는 **null을 허용하지 않는 참조 형식**입니다. 이 기능을 사용하도록 설정하면 기존 코드의 모든 참조 형식 변수가 포함됩니다. 그러나 암시적 형식 지역 변수(`var`을 사용하여 선언됨)는 **null 허용 참조 형식**입니다. 이전 섹션에서 살펴본 것처럼 정적 분석은 지역 변수의 *null-state*를 확인하여 역참조하기 전에 해당 변수가 *maybe-null*인지 확인합니다.

경우에 따라 변수가 `null`이 아님을 알고 있지만 컴파일러에서는 변수의 *null-state*가 *maybe-null*임을 판단하는 경우 경고를 재정의해야 합니다. 변수 이름 뒤에 [null-forgiving 연산자!](#)를 사용하여 *null-state*를 *not-null*로 강제합니다. 예를 들어 `name` 변수가 `null`이 아닌 것으로 알고 있는데 컴파일러 경고가 발생하는 경우 다음 코드를 작성하여 컴파일러 분석을 재정의할 수 있습니다.

C#

```
name!.Length;
```

`null` 허용 참조 형식 및 `null` 허용 값 형식은 유사한 의미 체계 개념을 제공합니다. 즉, 변수는 값이나 개체를 나타낼 수 있거나 해당 변수가 `null`일 수 있습니다. 그러나 `null` 허용 참조 형식과 `null` 허용 값 형식은 서로 다르게 구현됩니다. `null` 허용 값 형식은 [System.Nullable<T>](#)을 사용하여 구현되고, `null` 허용 참조 형식은 컴파일러에서 읽는 특성에 의해 구현됩니다. 예를 들어 `string?` 및 `string`은 둘 다 동일한 형식([System.String](#))으로 표현됩니다. 그러나 `int?` 및 `int`는 각각 `System.Nullable<System.Int32>` 및 [System.Int32](#)로 표현됩니다.

`null` 허용 참조 형식은 컴파일 시간 기능입니다. 즉, 호출자가 경고를 무시하고 의도적으로 `null` 허용이 아닌 참조를 기대하는 메서드에 대한 인수로 `null`을 사용할 수 있음을 의

미합니다. 라이브러리 작성자는 null 인수 값에 대한 런타임 검사를 포함해야 합니다. [ArgumentNullException.ThrowIfNull](#)은 런타임 시 null에 대해 매개 변수를 확인하는 데 기본 설정되는 옵션입니다.

### ① 중요

null 허용 주석을 사용하도록 설정하면 Entity Framework Core가 데이터 멤버가 필요 한지 여부를 결정하는 방법이 변경될 수 있습니다. 자세한 내용은 [Entity Framework Core Fundamentals: null 허용 참조 형식 작업](#) 문서에서 알아볼 수 있습니다.

## 제네릭

제네릭에는 임의 형식 매개 변수 `T`에 대한 `T?`를 처리하는 자세한 규칙이 필요합니다. null 허용 값 형식과 null 허용 참조 형식의 기록과 서로 다른 구현으로 인해 규칙은 자세히 설명되어야 합니다. Null 허용 값 형식은 `System.Nullable<T>` 구조체를 사용하여 구현됩니다. Null 허용 참조 형식은 컴파일러에 의미 체계 규칙을 제공하는 형식 주석으로 구현됩니다.

- `T`의 형식 인수가 참조 형식인 경우 `T?`는 해당 null 허용 참조 형식을 참조합니다. 예를 들어, `T`가 `string` 이면 `T?`는 `string?` 입니다.
- `T`의 형식 인수가 값 형식이면 `T?`는 동일한 값 형식 `T`를 참조합니다. 예를 들어, `T` 가 `int` 이면 `T?`도 `int` 입니다.
- `T`의 형식 인수가 null 허용 참조 형식인 경우 `T?`는 동일한 null 허용 참조 형식을 참조합니다. 예를 들어, `T`가 `string?` 이면 `T?`도 `string?` 입니다.
- `T`의 형식 인수가 null 허용 값 형식인 경우 `T?`는 동일한 null 허용 값 형식을 참조합니다. 예를 들어, `T`가 `int?` 이면 `T?`도 `int?` 입니다.

반환 값의 경우 `T?`는 `[MaybeNull]T`에 해당하고, 인수 값의 경우 `T?`는 `[AllowNull]T`에 해당합니다. 자세한 내용은 언어 참조의 [null-state 분석을 위한 특성](#) 문서를 참조하세요.

[제약 조건](#)을 사용하여 다른 동작을 지정할 수 있습니다.

- `class` 제약 조건은 `T`가 null을 허용하지 않는 참조 형식(예: `string`)이어야 함을 의미합니다. `T`에 대해 `string?` 와 같은 null 허용 참조 형식을 사용하는 경우 컴파일러가 경고를 생성합니다.
- `class?` 제약 조건은 `T`가 null을 허용하지 않는 참조 형식(`string`) 또는 null 허용 참조 형식(예: `string?`)이어야 함을 의미합니다. 형식 매개 변수가 `string?` 와 같은 null 허용 참조 형식인 경우 `T?`의 식은 `string?` 와 같은 동일한 null 허용 참조 형식을 참조합니다.

- `notnull` 제약 조건은 `T`가 `null`을 허용하지 않는 참조 형식 또는 `null`을 허용하지 않는 값 형식이어야 함을 의미합니다. 형식 매개 변수에 `null` 허용 참조 형식 또는 `null` 허용 값 형식을 사용하는 경우 컴파일러가 경고를 생성합니다. 또한 `T`가 값 형식인 경우 반환 값은 해당 `null` 허용 값 형식이 아닌 이 값 형식입니다.

이러한 제약 조건은 `T`가 사용되는 방식에 대한 자세한 정보를 컴파일러에 제공하는데 도움이 됩니다. 이는 개발자가 `T`의 형식을 선택할 때 도움이 되며 제네릭 형식의 인스턴스를 사용할 때 더 나은 *null-state* 분석을 제공합니다.

## Nullable 컨텍스트

소규모 프로젝트의 경우 `null` 허용 참조 형식을 사용하도록 설정하고 경고를 수정한 후 계속할 수 있습니다. 그러나 대규모 프로젝트 및 다중 프로젝트 솔루션의 경우 많은 수의 경고가 발생할 수 있습니다. `null` 허용 참조 형식을 사용하기 시작할 때 `pragma`를 사용하여 파일별로 `null` 허용 참조 형식을 사용하도록 설정할 수 있습니다.

[System.NullReferenceException](#)을 `throw`하는 것을 방지하는 새 기능은 기존 코드베이스에서 사용할 때 중단을 유발할 수 있습니다.

- 명시적 형식의 참조 변수는 모두 `null`을 허용하지 않는 참조 형식으로 해석됩니다.
- 제네릭의 `class` 제약 조건의 의미가 `null`을 허용하지 않는 참조 형식을 의미하도록 변경되었습니다.
- 이러한 새 규칙으로 인해 새로운 경고가 생성됩니다.

`null` 허용 주석 컨텍스트가 컴파일러의 동작을 결정합니다. `null` 허용 주석 컨텍스트에는 4개의 값이 있습니다.

- *disable*: 코드가 `null` 허용 인식 불가능입니다. *Disable*은 새 구문이 오류 대신 경고를 생성한다는 점을 제외하면 `null` 허용 참조 형식이 사용하도록 설정되기 전의 동작과 일치합니다.
  - `null` 허용 경고가 사용되지 않습니다.
  - 모든 참조 형식 변수는 `null` 허용 참조 형식입니다.
  - `?`  접미사를 사용하여 `null` 허용 참조 형식을 선언하면 경고가 생성됩니다.
  - `null forgiving` 연산자인 `!`를 사용할 수 있지만 영향은 없습니다.
- *enable*: 컴파일러는 모든 `null` 참조 분석과 모든 언어 기능을 사용하도록 설정합니다.
  - 모든 새 `null` 허용 경고가 사용됩니다.
  - `?`  접미사를 사용하여 `null` 허용 참조 형식을 선언할 수 있습니다.
  - `?`  접미사가 없는 참조 형식 변수는 `null`을 허용하지 않는 참조 형식입니다.
  - `null forgiving` 연산자는 `null`에 대한 가능한 할당을 알리는 경고를 표시하지 않습니다.

- *warnings*: 컴파일러가 모든 null 분석을 수행하고 코드가 `null`을 역참조할 수 있는 경우 경고를 내보냅니다.
  - 모든 새 null 허용 경고가 사용됩니다.
  - `?` 접미사를 사용하여 null 허용 참조 형식을 선언하면 경고가 생성됩니다.
  - 모든 참조 형식 변수는 null일 수 있습니다. 그러나 멤버는 `?` 접미사로 선언되지 않은 경우 모든 메서드의 여는 괄호에 *not-null*의 *null-state*를 가집니다.
  - null forgiving 연산자인 `!`를 사용할 수 있습니다.
- 주석: 코드가 `null`을 역참조할 때 또는 null을 허용하지 않는 변수에 maybe-null 식을 할당할 때 컴파일러는 경고를 표시하지 않습니다.
  - 새로운 null 허용 경고가 모두 사용 안 함으로 설정됩니다.
  - `?` 접미사를 사용하여 null 허용 참조 형식을 선언할 수 있습니다.
  - `?` 접미사가 없는 참조 형식 변수는 null을 허용하지 않는 참조 형식입니다.
  - null forgiving 연산자인 `!`를 사용할 수 있지만 영향은 없습니다.

.csproj 파일에 [<Nullable> 요소](#)를 사용하여 프로젝트에 대한 nullable 주석 컨텍스트 및 nullable 경고 컨텍스트를 설정할 수 있습니다. 이 요소는 컴파일러가 형식의 null 허용 여부를 해석하는 방법 및 내보내는 경고를 구성합니다. 다음 표에서는 허용 가능한 값을 보여주고 지정한 컨텍스트를 요약합니다.

#### 테이블 확장

Context	Dereference	할당 경고	참조 형식	<code>?</code> 접미사	<code>!</code> 연산자
<code>disable</code>	사용 안 함	사용 안 함	모두 null 허용입니다.	경고 생성	아무런 영향이 없습니다.
<code>enable</code>	사용	사용	<code>?</code> 로 선언하지 않는 한 null을 허용하지 않음	nullable 형식 선언	가능한 <code>null</code> 할당에 대한 경고를 표시하지 않습니다.
<code>warnings</code>	설정됨	해당 없음	모두 null 허용이지만 메서드의 여는 중괄호에서 멤버는 <i>not-null</i> 로 간주 됩니다.	경고 생성	가능한 <code>null</code> 할당에 대한 경고를 표시하지 않습니다.
<code>annotations</code>	사용 안 함	사용 안 함	<code>?</code> 로 선언하지 않는 한 null을 허용하지 않음	nullable 형식 선언	아무런 영향이 없습니다.

*disabled* 컨텍스트에서 컴파일된 코드의 참조 형식 변수는 *null* 허용 인식 불가능입니다. *null* 리터럴이나 *maybe-null* 변수를 *null* 허용 인식 불가능인 변수에 할당할 수 있습니다. 그러나 *nullable-oblivious* 변수의 기본 상태는 *not-null*입니다.

프로젝트에 가장 적합한 설정을 선택할 수 있습니다.

- 진단이나 새로운 기능을 기반으로 업데이트하지 않으려는 레거시 프로젝트에 대해서는 사용 중지를 선택합니다.
- 코드에서 [System.NullReferenceException](#)이 throw될 수 있는 위치를 확인하려면 경고를 선택합니다. *null*을 허용하지 않는 참조 형식을 사용하도록 코드를 수정하기 전에 이러한 경고를 해결할 수 있습니다.
- 경고를 사용하기 전에 설계 의도를 표현하려면 *annotations*를 선택합니다.
- *null* 참조 예외로부터 보호하려는 새 프로젝트와 활성 프로젝트에 대해 사용을 선택합니다.

예제:

XML

```
<Nullable>enable</Nullable>
```

또한 지시문을 사용하여 소스 코드의 아무 곳에나 이러한 동일한 컨텍스트를 설정할 수도 있습니다. 이러한 지시문은 대규모 코드베이스를 마이그레이션할 때 가장 유용합니다.

- `#nullable enable`: *null* 허용 주석 컨텍스트와 *null* 허용 경고 컨텍스트를 **enable**로 설정합니다.
- `#nullable disable`: *null* 허용 주석 컨텍스트와 *null* 허용 경고 컨텍스트를 **사용 중지**로 설정합니다.
- `#nullable restore`: *null* 허용 주석 컨텍스트와 *null* 허용 경고 컨텍스트를 프로젝트 설정으로 복원합니다.
- `#nullable disable warnings`: *null* 허용 경고 컨텍스트를 **disable**로 설정합니다.
- `#nullable enable warnings`: *null* 허용 경고 컨텍스트를 **enable**로 설정합니다.
- `#nullable restore warnings`: *null* 허용 경고 컨텍스트를 프로젝트 설정으로 복원합니다.
- `#nullable disable annotations`: *null* 허용 주석 컨텍스트를 **disable**로 설정합니다.
- `#nullable enable annotations`: *null* 허용 주석 컨텍스트를 **enable**로 설정합니다.
- `#nullable restore annotations`: 주석 경고 컨텍스트를 프로젝트 설정으로 복원합니다.

모든 코드 줄에 대해 다음 조합 중 원하는 것을 설정할 수 있습니다.

경고 컨텍스트	주석 컨텍스트	사용할 용어
프로젝트 기본값	프로젝트 기본값	기본값
enable	disable	분석 경고 수정
enable	프로젝트 기본값	분석 경고 수정
프로젝트 기본값	enable	형식 주석 추가
enable	enable	이미 마이그레이션된 코드
disable	enable	경고를 수정하기 전에 코드에 주석 달기
disable	disable	마이그레이션된 프로젝트에 레거시 코드 추가
프로젝트 기본값	disable	거의 없음
disable	프로젝트 기본값	거의 없음

이러한 9가지 조합은 컴파일러가 코드에 대해 내보내는 진단에 대한 세분화된 제어를 제공합니다. 아직 해결할 준비가 되지 않은 추가 경고를 표시하지 않고도 업데이트 중인 모든 영역에서 더 많은 기능을 사용하도록 설정할 수 있습니다.

### ① 중요

전역 null 허용 컨텍스트는 생성된 코드 파일에 적용되지 않습니다. 두 전략에서 null 허용 컨텍스트는 생성됨으로 표시된 모든 소스 파일에 대해 *disabled*입니다. 즉, 생성된 파일의 API가 주석 처리되지 않습니다. 다음 네 가지 방법으로 파일은 생성됨으로 표시됩니다.

1. .editorconfig에서 해당 파일에 적용되는 섹션에 `generated_code = true`를 지정 합니다.
2. 파일의 맨 위에 있는 주석에 `<auto-generated>` 또는 `<auto-generated/>`를 배치 합니다. 해당 주석의 모든 줄에 넣을 수 있지만 주석 블록은 파일의 첫 번째 요소여야 합니다.
3. 파일 이름을 *TemporaryGeneratedFile\_*로 시작합니다.
4. 파일 이름을 *.designer.cs*, *.generated.cs*, *.g.cs* 또는 *.g.i.cs*로 종료합니다.

생성기는 `#nullable` 전처리기 지시문을 사용하여 옵트인할 수 있습니다.

기본 nullable 주석 및 경고 컨텍스트는 **disabled**입니다. 이는 기존 코드가 변경 없이 새로운 경고를 생성하지 않고 컴파일됨을 의미합니다. .NET 6부터 새 프로젝트에는 모든 프로

젝트 템플릿에 `<Nullable>enable</Nullable>` 요소가 포함됩니다.

이러한 옵션은 null 허용 참조 형식을 사용하도록 [기존 코드베이스를 업데이트](#)하는 두 가지 고유한 전략을 제공합니다.

## 알려진 문제

참조 형식을 포함하는 배열 및 구조체는 null 허용 참조뿐만 아니라 null 안전을 확인하는 정적 분석에서도 알려진 함정입니다. 두 경우 모두 null을 허용하지 않는 참조 형식이 경고를 생성하지 않고 null로 초기화될 수 있습니다.

## 구조체

null을 허용하지 않는 참조 형식을 포함하는 구조에서는 경고 없이 `default`를 할당할 수 있습니다. 다음 예제를 참조하세요.

C#

```
using System;

#nullable enable

public struct Student
{
    public string FirstName;
    public string? MiddleName;
    public string LastName;
}

public static class Program
{
    public static void PrintStudent(Student student)
    {
        Console.WriteLine($"First name: {student.FirstName.ToUpper()}");
        Console.WriteLine($"Middle name: {student.MiddleName?.ToUpper()}");
        Console.WriteLine($"Last name: {student.LastName.ToUpper()}");
    }

    public static void Main() => PrintStudent(default);
}
```

앞의 예제에서 null을 허용하지 않는 참조 형식 `FirstName` 및 `LastName`이 null인 동안 `PrintStudent(default)`에는 경고가 없습니다.

또 다른 일반적인 사례는 일반 구조체를 처리하는 경우입니다. 다음 예제를 참조하세요.

C#

```
#nullable enable

public struct S<T>
{
    public T Prop { get; set; }
}

public static class Program
{
    public static void Main()
    {
        string s = default(S<string>).Prop;
    }
}
```

앞의 예에서 `Prop` 속성은 런타임에 `null`입니다. 경고 없이 `null`을 허용하지 않는 문자열에 할당됩니다.

## 배열

배열은 nullable 참조 형식의 알려진 문제가 되기도 합니다. 경고를 생성하지 않는 다음 예제를 고려하세요.

C#

```
using System;

(nullable enable

public static class Program
{
    public static void Main()
    {
        string[] values = new string[10];
        string s = values[0];
        Console.WriteLine(s.ToUpper());
    }
}
```

앞의 예제에서 배열 선언은 해당 요소가 모두 `null`로 초기화되는 동안 `null`을 허용하지 않는 문자열을 보유함을 나타냅니다. 그런 다음, `s` 변수에는 `null` 값(배열의 첫 번째 요소)이 할당됩니다. 마지막으로 `s` 변수가 역참조되어 런타임 예외가 발생합니다.

## 참고 항목

- Null 허용 참조 형식 제안
- Nullable 참조 형식 사양 초안
- 무제한 형식 매개 변수 주석
- Nullable 참조 소개 자습서
- Null 허용(C# 컴파일러 옵션)

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

ଓ 설명서 문제 열기

ଓ 제품 사용자 의견 제공

# Null 허용 참조 형식으로 코드베이스를 업데이트하여 null 진단 경고 개선

아티클 • 2023. 04. 07.

Null 허용 참조 형식을 사용하면 참조 형식의 변수에 `null` 값을 할당하거나 할당하지 않아야 하는 경우를 선언할 수 있습니다. 코드가 `null`을 역참조하는 경우 컴파일러의 정적 분석과 경고는 이 기능의 가장 중요한 장점입니다. 사용하도록 설정되면 컴파일러는 코드가 실행될 때 `System.NullReferenceException`을 `throw`하지 않는 데 도움이 되는 경고를 생성합니다.

코드베이스가 비교적 작은 경우 [프로젝트에서 기능을 켜고](#), 경고를 처리하고, 향상된 진단을 활용할 수 있습니다. 더 큰 코드베이스에는 시간이 지남에 따라 경고를 처리하기 위해 보다 구조적인 접근 방식이 필요할 수 있으므로 다양한 형식 또는 파일에서 경고를 처리할 때 잠시 동안 기능을 사용할 수 있습니다. 이 문서에서는 이러한 전략과 관련된 코드베이스 및 장단점을 업데이트하는 다양한 전략을 설명합니다. 마이그레이션을 시작하기 전에 [null 허용 참조 형식](#)의 개념적 개요를 읽어보세요. 컴파일러의 정적 분석 *null-state* 값인 *maybe-null* 및 *not-null*과 null 허용 주석을 설명합니다. 해당 개념과 용어를 잘 알고 있으면 코드를 마이그레이션할 준비가 된 것입니다.

## 마이그레이션 계획

코드베이스를 업데이트하는 방법에 관계없이 프로젝트에서 null 허용 경고와 null 허용 주석을 사용하도록 설정하는 것이 목표입니다. 해당 목표에 도달하면 프로젝트에서 `<nullable>Enable</nullable>` 설정을 사용할 수 있습니다. 다른 곳에서 설정을 조정하기 위해 전처리기 지시문이 필요하지 않습니다.

프로젝트의 기본값을 설정하는 것이 가장 좋습니다. 선택 사항은 다음과 같습니다.

- 기본값으로 Null 허용 사용 안 함:** 프로젝트 파일에 요소를 추가하지 않으면 `disable` 가 `Nullable` 기본값입니다. 코드베이스에 새 파일을 능동적으로 추가하지 않는 경우 이 기본값을 사용합니다. 기본 작업은 null 허용 참조 형식을 사용하도록 라이브러리를 업데이트하는 것입니다. 이 기본값을 사용하면 코드를 업데이트할 때 `nullable` 전처리기 지시문을 각 파일에 추가합니다.
- Nullable을 기본값으로 사용하도록 설정:** 새 기능을 적극적으로 개발할 때 이 기본값을 설정합니다. 모든 새 코드가 nullable 참조 형식 및 null 허용 정적 분석의 이점을 활용하려고 합니다. 이 기본값을 사용하면 각 파일의 맨 위에 `#nullable disable` 이 추가해야 합니다. 각 파일의 경고를 처리할 때 이러한 전처리기 지시문을 제거합니다.

3. **기본값으로 Null 허용 경고**: 2단계 마이그레이션에 대해 이 기본값을 선택합니다.  
첫 번째 단계에서 경고를 처리합니다. 두 번째 단계에서는 변수의 예상 *null* 상태를 선언하기 위한 주석을 캡니다. 이 기본값을 사용하면 각 파일의 맨 위에 `#nullable disable`이 추가해야 합니다.
4. **Null 허용 주석을 기본값으로 사용합니다**. 경고를 처리하기 전에 코드에 주석을 담니다.

기본값으로 nullable을 사용하도록 설정하면 모든 파일에 전처리기 지시문을 추가하는 더 많은 실행 작업이 만들어집니다. 장점이라면 프로젝트에 추가되는 모든 새 코드 파일에서 nullable을 사용할 수 있습니다. 모든 새 작업은 nullable 인식이므로 기존 코드만 업데이트만 업데이트하면 됩니다. 기본값으로 null 허용을 사용하지 않도록 설정하는 기능은 라이브러리가 안정적이며 개발에서 주로 nullable 참조 형식을 채택하는 데 중점을 두는 경우에 더 효과적입니다. API에 주석을 달 때 nullable 참조 형식을 설정합니다. 마쳤으면 전체 프로젝트에 대해 nullable 참조 형식을 사용하도록 설정합니다. 새 파일을 만들 때 전처리기 지시문을 추가하고 null을 인식할 수 있도록 해야 합니다. 팀의 개발자가 위 작업을 잊는 경우 새 코드가 이제 작업의 백로그에 있게 되어 모든 코드에서 nullable을 인식하게 됩니다.

어떤 전략을 선택할지는 프로젝트에서 수행 중인 활성 개발이 얼마나 많은지에 따라 달라집니다. 프로젝트의 완성도와 안정도가 높아질수록 두 번째 전략이 더 효과적입니다. 개발 중인 기능이 많을수록 첫 번째 전략이 더 효과적입니다.

### ① 중요

전역 null 허용 컨텍스트는 생성된 코드 파일에 적용되지 않습니다. 두 전략에서 null 허용 컨텍스트는 생성됨으로 표시된 모든 소스 파일에 대해 disabled입니다. 즉, 생성된 파일의 API가 주석 처리되지 않습니다. 다음 네 가지 방법으로 파일은 생성됨으로 표시됩니다.

1. .editorconfig에서 해당 파일에 적용되는 섹션에 `generated_code = true`를 지정합니다.
2. 파일의 맨 위에 있는 주석에 `<auto-generated>` 또는 `<auto-generated/>`를 배치합니다. 해당 주석의 모든 줄에 넣을 수 있지만 주석 블록은 파일의 첫 번째 요소여야 합니다.
3. 파일 이름을 *TemporaryGeneratedFile\_*로 시작합니다.
4. 파일 이름을 *.designer.cs*, *.generated.cs*, *.g.cs* 또는 *.g.i.cs*로 종료합니다.

생성기는 `#nullable` 전처리기 지시문을 사용하여 옵트인할 수 있습니다.

## 컨텍스트 및 경고 이해

경고와 주석을 사용하도록 설정하면 컴파일러가 형식과 null 허용 여부를 참조하는 방법을 제어할 수 있습니다. 모든 형식에는 다음 세 가지 null 허용 여부 중 하나가 포함됩니다.

- *oblivious*: 모든 참조 형식은 주석 컨텍스트가 사용되지 않을 경우 nullable *oblivious*입니다.
- *nonnullable*: 주석이 달리지 않은 참조 형식인 `c`는 주석 컨텍스트가 사용될 경우 *nonnullable*입니다.
- *nullable*: 주석이 달린 참조 형식인 `c?`는 *nullable*이지만 주석 컨텍스트가 사용되지 않을 경우 경고가 실행될 수 있습니다. `var`로 선언된 변수는 주석 컨텍스트가 사용될 경우 *nullable*입니다.

컴파일러는 해당 null 허용 여부에 따라 경고를 생성합니다.

- *nonnullable* 형식은 잠재적 `null` 값이 할당되는 경우 경고를 발생시킵니다.
- *nullable* 형식은 *maybe-null*일 때 역참조되는 경우 경고를 발생시킵니다.
- *oblivious* 형식은 *maybe-null*이고 경고 컨텍스트가 사용될 때 역참조되는 경우 경고를 발생시킵니다.

각 변수에는 null 허용 여부에 따라 달라지는 기본 null 허용 상태가 있습니다.

- Null 허용 변수의 기본 *null-state*는 *maybe-null*입니다.
- Null을 허용하지 않는 변수의 기본 *null-state*는 *not-null*입니다.
- Null을 허용하지 않는 *oblivious* 변수의 기본 *null-state*는 *not-null*입니다.

Null 허용 참조 형식을 사용하도록 설정하기 전에 코드베이스의 모든 선언은 *nullable oblivious*입니다. 이는 모든 참조 형식의 기본 *null-state*가 *not-null*임을 의미하기 때문에 중요합니다.

## 경고 처리

프로젝트에서 Entity Framework Core를 사용하는 경우 [Null 허용 참조 형식 작업](#)에 관한 지침을 참조해야 합니다.

マイグレーション을 시작할 때 먼저 경고만 사용하도록 설정해야 합니다. 모든 선언은 *nullable oblivious*로 유지되지만 *null-state*가 *maybe-null*로 변경된 후 값을 역참조할 때 경고가 표시됩니다. 이러한 경고를 처리할 때 더 많은 위치에서 null과 비교 확인하며 코드베이스의 복원력이 향상됩니다. 다양한 상황을 위한 특정 기술을 알아보려면 [null 허용 경고를 해결하는 기술](#) 문서를 참조하세요.

다른 코드를 계속하기 전에 각 파일이나 클래스에서 경고를 처리하고 주석을 사용하도록 설정할 수 있습니다. 그러나 형식 주석을 사용하도록 설정하기 전에 컨텍스트가 *warnings* 인 동안 생성된 경고를 처리하는 것이 더 효율적일 수도 있습니다. 이렇게 하면 첫 번째 경고 세트를 처리할 때까지 모든 형식은 *oblivious*입니다.

# 형식 주석 사용

첫 번째 경고 세트를 처리한 후 '주석 컨텍스트'를 사용하도록 설정할 수 있습니다. 이렇게 하면 참조 형식이 *oblivious*에서 *nonnullable*로 변경됩니다. `var`로 선언된 모든 변수는 *nullable*입니다. 이 변경으로 인해 새 경고가 발생할 수 있습니다. 컴파일러 경고를 해결하는 첫 번째 단계는 매개 변수와 반환 형식에 `?` 주석을 사용하여 인수나 반환 값이 `null`일 수 있는 경우를 나타내는 것입니다. 이 작업을 수행할 때 목표는 경고를 해결하는 것만이 아닙니다. 무엇보다 컴파일러에서 잠재적 `null` 값을 사용하는 사용자의 의도로 이해하게 만들어야 합니다.

## 특성을 통한 형식 주석 확장

변수의 `null` 상태에 대한 추가 정보를 표현하기 위해 여러 가지 특성이 추가되었습니다. API에 대한 규칙은 모든 매개 변수와 반환 값에 대해 *not-null* 또는 *maybe-null*보다 더 복잡할 수 있습니다. 대부분의 API에는 변수가 `null`일 수 있거나 없는 경우에 대한 더 복잡한 규칙이 있습니다. 이러한 경우에는 특성을 사용하여 해당 규칙을 표현합니다. API의 의미 체계를 설명하는 특성은 [null 허용 분석에 영향을 주는 특성](#)에 관한 문서에서 확인할 수 있습니다.

## 다음 단계

주석을 사용하도록 설정한 후 모든 경고를 처리한 후에는 프로젝트의 기본 컨텍스트를 사용으로 설정할 수 있습니다. 코드에서 `null` 허용 주석 또는 경고 컨텍스트에 대해 `pragma`를 추가한 경우 해당 `pragma`를 제거할 수 있습니다. 시간이 지남에 따라 새 경고가 표시될 수 있습니다. 경고를 발생시키는 코드를 작성할 수 있습니다. *nullable* 참조 형식에 대한 라이브러리 종속성이 업데이트될 수 있습니다. 업데이트하면 해당 라이브러리의 형식이 *nullable oblivious*에서 *nonnullable* 또는 *nullable*로 변경됩니다.

[C#의 Nullable 안전성 학습 모듈](#)에서 이러한 개념을 살펴볼 수도 있습니다.

# C#의 메서드

아티클 • 2024. 05. 17.

메서드는 일련의 문을 포함하는 코드 블록입니다. 프로그램을 통해 메서드를 호출하고 필요한 메서드 인수를 지정하여 문을 실행합니다. C#에서는 실행된 모든 명령이 메서드의 컨텍스트에서 수행됩니다.

## ① 참고

이 항목에서는 명명된 메서드에 대해 설명합니다. 무명 기능에 대한 자세한 내용은 [람다 식](#)을 참조하세요.

## 메서드 시그니처

메서드는 다음을 지정하여 `class`, `record`, `struct`에서 선언됩니다.

- `public` 또는 `private`와 같은 선택적 액세스 수준입니다. 기본값은 `private`입니다.
- `abstract` 또는 `sealed`과 같은 선택적 한정자입니다.
- 반환 값 또는 메서드에 반환 값이 없는 경우 `void`입니다.
- 메서드 이름입니다.
- 메서드 매개 변수입니다. 메서드 매개 변수는 괄호로 묶고 쉼표로 구분합니다. 빈 괄호는 메서드에 매개 변수가 필요하지 않음을 나타냅니다.

이러한 부분이 결합되어 메서드 시그니처를 구성합니다.

## ① 중요

메서드의 반환 값은 메서드 오버로드를 위한 메서드 서명의 파트가 아닙니다. 그러나 대리자와 대리자가 가리키는 메서드 간의 호환성을 결정할 경우에는 메서드 서명의 부분입니다.

다음 예제에서는 다섯 개의 메서드를 포함하는 `Motorcycle`이라는 클래스를 정의합니다.

C#

```
namespace MotorCycleExample
{
    abstract class Motorcycle
    {
        // Anyone can call this.
        public void StartEngine() /* Method statements here */
    }
}
```

```

// Only derived classes can call this.
protected void AddGas(int gallons) { /* Method statements here */ }

// Derived classes can override the base class implementation.
public virtual int Drive(int miles, int speed) { /* Method
statements here */ return 1; }

// Derived classes can override the base class implementation.
public virtual int Drive(TimeSpan time, int speed) { /* Method
statements here */ return 0; }

// Derived classes must implement this.
public abstract double GetTopSpeed();
}

```

`Motorcycle` 클래스에는 오버로드된 메서드 `Drive`(이)가 포함됩니다. 두 메서드는 이름이 같지만 각자의 매개 변수 형식으로 구별됩니다.

## 메서드 호출

메서드는 인스턴스 또는 정적일 수 있습니다. 해당 인스턴스에서 인스턴스 메서드를 호출 하려면 개체를 인스턴스화해야 합니다. 인스턴스 메서드는 해당 인스턴스와 해당 데이터에 대해 작동합니다. 메서드가 속한 형식의 이름을 참조하여 정적 메서드를 호출합니다. 정적 메서드는 인스턴스 데이터에서 작동하지 않습니다. 개체 인스턴스를 통해 정적 메서드를 호출하려고 하면 컴파일러 오류가 생성됩니다.

메서드 호출은 필드 액세스와 비슷합니다. 개체 이름(인스턴스 메서드를 호출하는 경우) 또는 형식 이름(`static` 메서드를 호출하는 경우) 다음 마침표, 메서드 이름 및 괄호를 추가합니다. 인수는 괄호 안에 나열되고 쉼표로 구분됩니다.

메서드 정의는 필요한 모든 매개 변수의 이름 및 형식을 지정합니다. 호출자는 메서드를 호출할 때 각 매개 변수에 대해 인수라는 구체적인 값을 제공합니다. 인수는 매개 변수 형식과 호환되어야 하지만 호출 코드에서 인수 이름을 사용하는 경우 메서드에 정의된 명명된 매개 변수와 같을 필요는 없습니다. 다음 예제에서는 `Square` 메서드에 `i`라는 `int` 형식의 단일 매개 변수가 포함되어 있습니다. 첫 번째 메서드 호출은 `Square` 메서드에 `num`이라는 `int` 형식의 변수를 전달합니다. 두 번째 호출은 숫자 상수, 세 번째 호출은 식을 전달합니다.

C#

```

public static class SquareExample
{
    public static void Main()
    {
        // Call with an int variable.

```

```

        int num = 4;
        int productA = Square(num);

        // Call with an integer literal.
        int productB = Square(12);

        // Call with an expression that evaluates to int.
        int productC = Square(productA * 3);
    }

    static int Square(int i)
    {
        // Store input argument in a local variable.
        int input = i;
        return input * input;
    }
}

```

가장 일반적인 형태의 메서드 호출은 위치 인수를 사용하며, 메서드 매개 변수와 동일한 순서로 인수를 제공합니다. `Motorcycle` 클래스의 메서드는 다음 예제와 같이 호출될 수 있습니다. 예를 들어 `Drive` 메서드 호출에는 메서드 구문의 두 매개 변수에 해당하는 두 개의 인수가 포함되어 있습니다. 첫 번째는 `miles` 매개 변수의 값이 됩니다. 두 번째는 `speed` 매개 변수의 값이 됩니다.

C#

```

class TestMotorcycle : Motorcycle
{
    public override double GetTopSpeed() => 108.4;

    static void Main()
    {
        var moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        _ = moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}

```

메서드를 호출할 때 위치 인수 대신 ‘명명된 인수’를 사용할 수도 있습니다. 명명된 인수를 사용하는 경우 매개 변수 이름 뒤에 콜론(:)과 인수를 지정합니다. 모든 필수 인수가 있으면 하면 메서드의 인수 순서는 중요하지 않습니다. 다음 예제에서는 명명된 인수를 사용하여 `TestMotorcycle.Drive` 메서드를 호출합니다. 이 예제에서는 명명된 인수가 메서드의 매개 변수 목록과 반대 순서로 전달됩니다.

C#

```
namespace NamedMotorCycle;

class TestMotorcycle : Motorcycle
{
    public override int Drive(int miles, int speed) =>
        (int)Math.Round((double)miles / speed, 0);

    public override double GetTopSpeed() => 108.4;

    static void Main()
    {
        var moto = new TestMotorcycle();
        moto.StartEngine();
        moto.AddGas(15);
        int travelTime = moto.Drive(miles: 170, speed: 60);
        Console.WriteLine("Travel time: approx. {0} hours", travelTime);
    }
}

// The example displays the following output:
//      Travel time: approx. 3 hours
```

위치 인수와 명명된 인수 둘 다를 사용하여 메서드를 호출할 수 있습니다. 그러나 명명된 인수가 올바른 위치에 있는 경우에만 명명된 인수 뒤에 위치 인수를 사용할 수 있습니다. 다음 예제에서는 위치 인수 하나와 명명된 인수 하나를 사용하여 이전 예제의 `TestMotorcycle.Drive` 메서드를 호출합니다.

C#

```
int travelTime = moto.Drive(170, speed: 55);
```

## 상속 및 재정의된 메서드

형식은 해당 형식에서 명시적으로 정의된 멤버 외에도 기본 클래스에서 정의된 멤버를 상속합니다. 관리되는 형식 시스템의 모든 형식이 직접 또는 간접적으로 `Object` 클래스에서 상속하므로 모든 형식은 `Equals(Object)`, `GetType()` 및 `ToString()`과 같은 해당 멤버를 상속합니다. 다음 예제에서는 `Person` 클래스를 정의하고, 두 개의 `Person` 객체를 인스턴스화하고, `Person.Equals` 메서드를 호출하여 두 개체가 같은지 여부를 확인합니다. 그러나 `Equals` 메서드는 `Person` 클래스에 정의되어 있지 않습니다. `Object`에서 상속됩니다.

C#

```
public class Person
{
    public string FirstName = default!;
```

```

}

public static class ClassTypeExample
{
    public static void Main()
    {
        Person p1 = new() { FirstName = "John" };
        Person p2 = new() { FirstName = "John" };
        Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
    }
}
// The example displays the following output:
//      p1 = p2: False

```

형식은 `override` 키워드를 사용하고 재정의된 메서드에 대한 구현을 제공하여 상속된 멤버를 재정의할 수 있습니다. 메서드 시그니처는 재정의된 메서드의 시그니처와 같아야 합니다. 다음 예제는 `Equals(Object)` 메서드를 재정의한다는 점을 제외하고 이전 예제와 비슷합니다. 또한 두 메서드가 일치하는 결과를 제공하기 때문에 `GetHashCode()` 메서드를 재정의합니다.

C#

```

namespace methods;

public class Person
{
    public string FirstName = default!;

    public override bool Equals(object? obj) =>
        obj is Person p2 &&
        FirstName.Equals(p2.FirstName);

    public override int GetHashCode() => FirstName.GetHashCode();
}

public static class Example
{
    public static void Main()
    {
        Person p1 = new() { FirstName = "John" };
        Person p2 = new() { FirstName = "John" };
        Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
    }
}
// The example displays the following output:
//      p1 = p2: True

```

## 매개 변수 전달

C#의 형식은 값 형식 또는 참조 형식입니다. 기본 제공 값 형식의 목록은 [형식](#)을 참조하세요. 기본적으로 값 형식과 참조 형식은 둘 다 값으로 메서드에 전달됩니다.

## 값으로 매개 변수 전달

값 형식이 값으로 메서드에 전달되는 경우 개체 자체가 아니라 개체의 복사본이 메서드에 전달됩니다. 따라서 제어가 호출자로 반환될 때 호출된 메서드의 개체 변경 내용은 원래 개체에 영향을 주지 않습니다.

다음 예제에서는 값 형식을 값으로 메서드에 전달하며, 호출된 메서드는 값 형식의 값을 변경하려고 합니다. 값 형식인 `int` 형식의 변수를 정의하고, 해당 값을 20으로 초기화한 다음 `ModifyValue`라는 메서드에 전달하면 이 메서드가 변수의 값을 30으로 변경합니다. 그러나 메서드가 반환될 때는 변수의 값이 변경되지 않습니다.

```
C#  
  
public static class ByValueExample  
{  
    public static void Main()  
    {  
        var value = 20;  
        Console.WriteLine("In Main, value = {0}", value);  
        ModifyValue(value);  
        Console.WriteLine("Back in Main, value = {0}", value);  
    }  
  
    static void ModifyValue(int i)  
    {  
        i = 30;  
        Console.WriteLine("In ModifyValue, parameter value = {0}", i);  
        return;  
    }  
}  
// The example displays the following output:  
//      In Main, value = 20  
//      In ModifyValue, parameter value = 30  
//      Back in Main, value = 20
```

참조 형식의 개체가 메서드에 값으로 전달되는 경우 개체에 대한 참조가 값으로 전달됩니다. 즉, 메서드는 개체 자체가 아니라 개체의 위치를 나타내는 인수를 수신합니다. 이 참조를 사용하여 개체의 멤버를 변경하는 경우 제어가 호출하는 메서드로 반환될 때 변경 내용이 개체에 반영됩니다. 그러나 메서드에 전달되는 개체를 바꾸면 제어가 호출자로 반환될 때 원래 개체에 영향을 주지 않습니다.

다음 예제에서는 `SampleRefType`이라는 클래스(참조 형식)를 정의합니다. `SampleRefType` 개체를 인스턴스화하고, 해당 `value` 필드에 44를 할당한 다음 개체를 `ModifyObject` 메서

드에 전달합니다. 이 예제에서는 기본적으로 메서드에 값으로 인수를 전달하는 이전 예제와 동일한 작업을 수행합니다. 그러나 참조 형식이 사용되므로 결과가 다릅니다. 예제의 출력과 같이 `ModifyObject`에서 `obj.value` 필드에 대해 수정한 내용으로 인해 `Main` 메서드에서 `rt` 인수의 `value` 필드도 33으로 변경됩니다.

C#

```
public class SampleRefType
{
    public int value;
}

public static class ByRefTypeExample
{
    public static void Main()
    {
        var rt = new SampleRefType { value = 44 };
        ModifyObject(rt);
        Console.WriteLine(rt.value);
    }

    static void ModifyObject(SampleRefType obj) => obj.value = 33;
}
```

## 참조로 매개 변수 전달

메서드의 인수 값을 변경하고 제어가 호출하는 메서드로 반환될 때 해당 변경 내용을 반영하려는 경우 참조로 매개 변수를 전달합니다. 참조로 매개 변수를 전달하려면 `ref` 또는 `out` 키워드를 사용합니다. 복사를 방지하지만 여전히 `in` 키워드를 사용하여 수정을 방지하도록 참조로 값을 전달할 수도 있습니다.

다음 예제는 값이 참조로 `ModifyValue` 메서드에 전달된다는 점을 제외하고 이전 예제와 동일합니다. `ModifyValue` 메서드에서 매개 변수의 값을 수정하면 제어가 호출자로 반환될 때 값의 변경 내용이 반영됩니다.

C#

```
public static class ByRefExample
{
    public static void Main()
    {
        var value = 20;
        Console.WriteLine("In Main, value = {0}", value);
        ModifyValue(ref value);
        Console.WriteLine("Back in Main, value = {0}", value);
    }

    private static void ModifyValue(ref int i)
```

```

    {
        i = 30;
        Console.WriteLine("In ModifyValue, parameter value = {0}", i);
        return;
    }
}

// The example displays the following output:
//     In Main, value = 20
//     In ModifyValue, parameter value = 30
//     Back in Main, value = 30

```

by ref 매개 변수를 사용하는 일반적인 패턴은 변수 값의 교환을 포함합니다. 두 개의 변수를 참조로 메서드에 전달하고 메서드가 해당 내용을 바꿉니다. 다음 예제에서는 정수 값을 바꿉니다.

C#

```

public static class RefSwapExample
{
    static void Main()
    {
        int i = 2, j = 3;
        Console.WriteLine("i = {0}  j = {1}", i, j);

        Swap(ref i, ref j);

        Console.WriteLine("i = {0}  j = {1}", i, j);
    }

    static void Swap(ref int x, ref int y) =>
        (y, x) = (x, y);
}

```

// The example displays the following output:  
// i = 2 j = 3  
// i = 3 j = 2

참조 형식 매개 변수를 전달하면 해당 개별 요소 또는 필드의 값이 아니라 참조 자체의 값을 변경할 수 있습니다.

## 매개 변수 컬렉션

경우에 따라 메서드의 인수 개수를 정확하게 지정하라는 요구 사항은 제한적입니다.

`params` 키워드를 사용하여 매개 변수가 매개 변수 컬렉션임을 나타내면 가변 개수의 인수를 사용하여 메서드를 호출할 수 있습니다. `params` 키워드로 태그가 지정된 매개 변수는 컬렉션 형식이어야 하며, 메서드의 매개 변수 목록에서 마지막 매개 변수여야 합니다.

그러면 호출자가 `params` 매개 변수에 대하여 다음 네 가지 방법 중 하나로 메서드를 호출할 수 있습니다.

- 원하는 개수의 요소를 포함하는 적절한 형식의 컬렉션 전달. 이 예제에서는 컴파일러가 적절한 컬렉션 형식을 만들도록 [컬렉션 식](#)을 사용합니다.
- 적절한 형식의 개별 인수가 포함된 쉼표로 구분된 목록을 메서드에 전달 컴파일러가 적절한 컬렉션 형식을 만듭니다.
- `null`을 전달
- 매개 변수 컬렉션에 인수를 제공 안 함.

다음 예제에서는 매개 변수 컬렉션의 모든 모음을 반환하는 `GetVowels` 메서드를 정의합니다. `Main` 메서드는 해당 메서드를 호출하는 네 가지 방법을 모두 보여 줍니다. 호출자는 `params` 한정자를 포함하는 매개 변수에 대한 인수를 제공할 필요가 없습니다. 이 경우 매개 변수는 빈 컬렉션입니다.

C#

```
static class ParamsExample
{
    static void Main()
    {
        string fromArray = GetVowels(["apple", "banana", "pear"]);
        Console.WriteLine($"Vowels from collection expression:
'{fromArray}'");

        string fromMultipleArguments = GetVowels("apple", "banana", "pear");
        Console.WriteLine($"Vowels from multiple arguments:
'{fromMultipleArguments}'");

        string fromNull = GetVowels(null);
        Console.WriteLine($"Vowels from null: '{fromNull}'");

        string fromNoValue = GetVowels();
        Console.WriteLine($"Vowels from no value: '{fromNoValue}'");
    }

    static string GetVowels(params IEnumerable<string>? input)
    {
        if (input == null || !input.Any())
        {
            return string.Empty;
        }

        char[] vowels = ['A', 'E', 'I', 'O', 'U'];
        return string.Concat(
            input.SelectMany(
                word => word.Where(letter =>
vowels.Contains(char.ToUpper(letter)))));
        }
    }
}
```

```
// The example displays the following output:  
//     Vowels from array: 'aeaaaaea'  
//     Vowels from multiple arguments: 'aeaaaaea'  
//     Vowels from null: ''  
//     Vowels from no value: ''
```

C# 13 이전에서는 `params` 한정자를 단일 차원 배열에서만 사용할 수 있습니다.

## 선택적 매개 변수 및 인수

메서드 정의는 해당 매개 변수가 필요하거나 선택 사항임을 지정할 수 있습니다. 기본적으로 매개 변수는 필수입니다. 선택적 매개 변수는 메서드 정의에 매개 변수의 기본값을 포함하여 지정됩니다. 메서드를 호출할 때 선택적 매개 변수에 대한 인수가 제공되지 않은 경우 기본값이 대신 사용됩니다.

다음 종류의 식 중 하나를 사용하여 매개 변수의 기본값을 할당합니다.

- 리터럴 문자열이나 숫자와 같은 상수
- `default(SomeType)` 형식의 식입니다. 여기서 `SomeType`은 값 형식 또는 참조 형식일 수 있습니다. 참조 형식인 경우 사실상 `null`을 지정하는 것과 같습니다. 컴파일러가 매개 변수 선언에서 형식을 유추할 수 있으므로 `default` 리터럴을 사용할 수 있습니다.
- `new ValType()` 형태의 식. 여기서 `ValType`은 값 형식입니다. 이 식은 형식의 실제 멤버가 아닌 값 형식의 암시적 매개 변수 없는 생성자를 호출합니다.

### ① 참고

C# 10 이상에서는 `new ValType()` 형식의 식이 값 형식의 명시적으로 정의된 매개 변수가 없는 생성자를 호출할 경우, 기본 매개 변수 값이 컴파일 시간 상수여야 하므로 컴파일러가 오류를 생성합니다. 기본 매개 변수 값을 제공하려면 `default(ValType)` 식 또는 `default` 리터럴을 사용합니다. 매개 변수가 없는 생성자에 대한 자세한 내용은 [구조체 형식](#) 문서의 [구조체 초기화 및 기본값](#) 섹션을 참조하세요.

메서드에 필수 및 선택적 매개 변수가 둘 다 포함된 경우 선택적 매개 변수는 매개 변수 목록의 끝에서 모든 필수 매개 변수 다음에 정의됩니다.

다음 예제에서는 필수 매개 변수 하나와 선택적 매개 변수 두 개가 있는 `ExampleMethod` 메서드를 정의합니다.

C#

```
public class Options
{
    public void ExampleMethod(int required, int optionalInt = default,
                             string? description = default)
    {
        var msg = $"{description ?? "N/A"}: {required} + {optionalInt} =
{required + optionalInt}";
        Console.WriteLine(msg);
    }
}
```

호출자는 인수가 제공되는 마지막 선택적 매개 변수까지 모든 선택적 매개 변수에 대한 인수를 제공해야 합니다. 예를 들어 `ExampleMethod` 메서드에서 호출자가 `description` 매개 변수에 대한 인수를 제공하는 경우 `optionalInt` 매개 변수에 대한 인수도 제공해야 합니다. `opt.ExampleMethod(2, 2, "Addition of 2 and 2");`는 유효한 메서드 호출이고, `opt.ExampleMethod(2, , "Addition of 2 and 0");`은 "인수가 없습니다." 컴파일러 오류를 생성합니다.

명명된 인수 또는 위치 인수와 명명된 인수의 조합을 사용하여 메서드를 호출하는 경우 호출자는 메서드 호출에서 마지막 위치 인수 뒤에 오는 모든 인수를 생략할 수 있습니다.

다음 예제에서는 `ExampleMethod` 메서드를 세 번 호출합니다. 처음 두 메서드 호출은 위치 인수를 사용합니다. 첫 번째 호출은 선택적 인수를 둘 다 생략하고, 두 번째 호출은 마지막 인수를 생략합니다. 세 번째 메서드 호출은 필수 매개 변수에 대한 위치 인수를 제공하지만 `optionalInt` 인수를 생략하고 명명된 인수를 사용하여 `description` 매개 변수에 값을 제공합니다.

C#

```
public static class OptionsExample
{
    public static void Main()
    {
        var opt = new Options();
        opt.ExampleMethod(10);
        opt.ExampleMethod(10, 2);
        opt.ExampleMethod(12, description: "Addition with zero:");
    }
}
// The example displays the following output:
//      N/A: 10 + 0 = 10
//      N/A: 10 + 2 = 12
//      Addition with zero:: 12 + 0 = 12
```

선택적 매개 변수를 사용하면 다음과 같이 오버로드 확인 또는 C# 컴파일러가 메서드 호출에 대해 호출할 오버로드를 결정하는 방식에 영향을 줍니다.

- 메서드, 인덱서 또는 생성자는 해당 매개 변수가 이름 또는 위치에 따라 단일 인수에 해당하고 이 인수를 매개 변수의 형식으로 변환할 수 있는 경우 실행 후보가 됩니다.
- 둘 이상의 인증서가 있으면 기본 설정 변환에 대한 오버로드 확인 규칙이 명시적으로 지정된 인수에 적용됩니다. 선택적 매개 변수에 대해 생략된 인수는 무시됩니다.
- 두 후보가 똑같이 정상이라고 판단되는 경우 기본적으로 호출에서 인수가 생략된 선택적 매개 변수가 없는 후보가 설정됩니다.

## 반환 값

메서드는 호출자에 값을 반환할 수 있습니다. 반환 형식(메서드 이름 앞에 나열된 형식)이 `void`(이)가 아니면 메서드는 `return` 키워드를 사용하여 값을 반환할 수 있습니다.

`return` 키워드와 그 뒤에 반환 형식과 일치하는 변수, 상수 또는 식이 따르는 문은 메서드 호출자에 해당 값을 반환합니다. `return` 키워드를 사용하여 값을 반환하려면 `void`가 아닌 반환 값을 포함한 메서드가 필요합니다. `return` 키워드는 메서드 실행을 중지합니다.

반환 형식이 `void`이면 값이 없는 `return` 문을 사용하여 메서드 실행을 중지할 수 있습니다. `return` 키워드가 없으면 메서드는 코드 블록 끝에 도달할 때 실행을 중지합니다.

예를 들어 이들 두 메서드에서는 `return` 키워드를 사용하여 정수를 반환합니다.

C#

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2) =>
        number1 + number2;

    public int SquareANumber(int number) =>
        number * number;
}
```

메서드에서 반환된 값을 사용하려면 호출하는 메서드에서 같은 형식의 값으로 충분한 모든 경우에 메서드 호출 자체를 사용하면 됩니다. 반환 값을 변수에 할당할 수도 있습니다. 예를 들어 다음 두 코드 예제에서는 같은 목표를 달성합니다.

C#

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

C#

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

메서드에서 둘 이상의 값을 반환하려는 경우도 있습니다. 튜플 형식과 튜플 리터럴을 사용하여 여러 값을 반환합니다. 튜플 형식은 튜플 요소의 데이터 형식을 정의합니다. 튜플 리터럴은 반환된 튜플의 실제 값을 제공합니다. 다음 예제에서 `(string, string, string, int)` 는 `GetPersonalInfo` 메서드에 의해 반환되는 튜플 형식을 정의합니다.

`(per.FirstName, per.MiddleName, per.LastName, per.Age)` 식은 튜플 리터럴입니다. 메서드는 `PersonInfo` 객체의 나이와 함께 이름, 중간 이름 및 성을 반환합니다.

C#

```
public (string, string, string, int) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

호출자는 다음 코드를 사용하여 반환된 튜플을 사용할 수 있습니다.

C#

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.Item1} {person.Item3}: age = {person.Item4}");
```

튜플 형식 정의의 튜플 요소에 이름을 할당할 수도 있습니다. 다음 예제에서는 명명된 요소를 사용하는 `GetPersonalInfo` 메서드의 대체 버전을 보여 줍니다.

C#

```
public (string FName, string MName, string LName, int Age)
GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

`GetPersonalInfo` 메서드에 대한 이전 호출을 다음과 같이 수정할 수 있습니다.

C#

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.FName} {person.LName}: age = {person.Age}");
```

메서드가 배열을 매개 변수로 사용하고 개별 요소의 값을 수정하는 경우 메서드가 배열을 반환할 필요가 없습니다. C#은 모든 참조 형식을 값으로 전달하고, 배열 참조의 값은 배열에 대한 포인터입니다. 다음 예제에서는 `DoubleValues` 메서드에서 수행한 `values` 배열 내용의 변경 사항을 배열에 대한 참조가 있는 모든 코드에서 관찰할 수 있습니다.

C#

```
public static class ArrayValueExample
{
    static void Main()
    {
        int[] values = [2, 4, 6, 8];
        DoubleValues(values);
        foreach (var value in values)
        {
            Console.Write("{0} ", value);
        }
    }

    public static void DoubleValues(int[] arr)
    {
        for (var ctr = 0; ctr <= arr.GetUpperBound(0); ctr++)
        {
            arr[ctr] *= 2;
        }
    }
}
// The example displays the following output:
//      4 8 12 16
```

## 확장 메서드

일반적으로 기존 형식에 메서드를 추가하는 방법에는 다음 두 가지가 있습니다.

- 해당 형식에 대한 소스 코드를 수정합니다. 원본을 수정하는 경우 메서드를 지원하기 위해 프라이빗 데이터 필드도 추가하면 호환성이 손상되는 변경이 발생합니다.
- 파생 클래스에서 새 메서드를 정의합니다. 구조체 및 열거형과 같은 다른 형식에 상속을 사용하여 이러한 방식으로 메서드를 추가할 수 없습니다. 봉인 클래스에 메서드를 "추가"하는 데 사용할 수도 없습니다.

확장 메서드를 사용하면 형식 자체를 수정하거나 상속된 형식에서 새 메서드를 구현하지 않고 기존 형식에 메서드를 "추가"할 수 있습니다. 또한 확장 메서드는 확장하는 형식과

동일한 어셈블리에 있을 필요가 없습니다. 형식의 정의된 멤버인 것처럼 확장 메서드를 호출합니다.

자세한 내용은 [확장 메서드](#)를 참조하세요.

## 비동기 메서드

비동기 기능을 사용하면 명시적 콜백을 사용하거나 수동으로 여러 메서드 또는 람다식에 코드를 분할하지 않고도 비동기 메서드를 호출할 수 있습니다.

메서드에 `async` 한정자를 표시하면 메서드에서 `await` 연산자를 사용할 수 있습니다. 비동기 메서드에서 컨트롤이 `await` 식에 도달하면 대기 중인 작업이 완료되지 않은 경우 컨트롤이 호출자에게 반환되고 대기 중인 작업이 완료될 때까지 `await` 키워드가 있는 메서드의 진행이 일시 중단됩니다. 작업이 완료되면 메서드가 실행이 다시 시작될 수 있습니다.

### ① 참고

비동기 메서드는 아직 완료되지 않은 첫 번째 대기된 개체를 검색할 때나 비동기 메서드의 끝에 도달할 때 중에서 먼저 발생하는 시점에 호출자에게 반환됩니다.

비동기 메서드는 일반적으로 반환 형식이 `Task<TResult>`, `Task`, `IAsyncEnumerable<T>` 또는 `void`입니다. `void` 반환 형식은 기본적으로 `void` 반환 형식이 필요할 때 이벤트 처리기를 정의하는 데 사용됩니다. `void`를 반환하는 비동기 메서드는 대기할 수 없고 `void`를 반환하는 메서드의 호출자는 메서드가 `throw`하는 예외를 `catch`할 수 없습니다. 비동기 메서드에는 [작업과 유사한 반환 형식](#)이 있을 수 있습니다.

다음 예제에서 `DelayAsync`는 정수를 반환하는 `return` 문을 포함하는 비동기 메서드입니다. 비동기 메서드이므로 메서드 선언에는 반환 형식의 `Task<int>(이)`가 있어야 합니다. 반환 형식이 `Task<int>`이므로 `DoSomethingAsync`의 `await` 식 계산에서 다음 `int result = await delayTask` 문과 같이 정수가 생성됩니다.

C#

```
class Program
{
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
    }
}
```

```

    //int result = await DelayAsync();

    Console.WriteLine($"Result: {result}");
}

static async Task<int> DelayAsync()
{
    await Task.Delay(100);
    return 5;
}
}

// Example output:
//   Result: 5

```

비동기 메서드는 모든 `in`, `ref` 또는 `out` 매개 변수를 선언할 수 없지만, 이러한 매개 변수가 있는 메서드를 호출할 수는 있습니다.

비동기 메서드에 관한 자세한 내용은 [async 및 await를 사용한 비동기 프로그래밍](#) 및 [비동기 반환 형식](#)을 참조하세요.

## 식 본문 멤버

식의 결과와 함께 즉시 반환하거나 메서드의 본문으로 단일 문을 포함하는 메서드 정의가 있는 것이 일반적입니다. `=>`(을)를 사용하여 이러한 메서드를 정의하기 위한 구문 바로 가기가 있습니다.

C#

```

public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);

```

메서드가 `void`를 반환하거나 비동기 메서드이면 메서드 본문은 문 식이어야 합니다(람다에서와 같음). 속성 및 인덱서의 경우 읽기 전용이어야 하며 `get` 접근자 키워드를 사용하지 않습니다.

## Iterators

반복기는 배열 목록과 같은 컬렉션에 대해 사용자 지정 반복을 수행합니다. 반복기는 `yield return` 문을 사용하여 각 요소를 따로따로 반환할 수 있습니다. `yield return` 문에 도달하면 호출자가 시퀀스의 다음 요소를 요청할 수 있도록 현재 위치가 기억됩니다.

반복기의 반환 형식은 `IEnumerable`, `IEnumerable<T>`, `IAsyncEnumerable<T>`, `IEnumerator` 또는 `IEnumerator<T>` 일 수 있습니다.

자세한 내용은 [반복기](#)를 참조하세요.

## 참고 항목

- [액세스 한정자](#)
- [정적 클래스 및 정적 클래스 멤버](#)
- [상속](#)
- [추상/봉인된 클래스 및 클래스 멤버](#)
- [params](#)
- [out](#)
- [ref](#)
- [in](#)
- [매개 변수 전달](#)

### ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

🌟 설명서 문제 열기

↗️ 제품 사용자 의견 제공

# 속성

아티클 • 2024. 04. 13.

속성은 C#의 주요 구성 요소입니다. 언어는 개발자가 디자인 의도를 정확하게 표현하는 코드를 작성할 수 있는 구문을 정의합니다.

속성은 액세스될 때 필드처럼 동작합니다. 그러나 필드와 달리 속성은 속성에 액세스하거나 할당할 때 실행되는 문을 정의하는 접근자로 구현됩니다.

## 속성 구문

속성 구문은 필드에 대한 자연 확장입니다. 필드는 스토리지 위치를 정의합니다.

```
C#  
  
public class Person  
{  
    public string? FirstName;  
  
    // Omitted for brevity.  
}
```

속성 정의에는 해당 속성의 값을 검색하고 할당하는 `get` 및 `set` 접근자에 대한 선언이 포함됩니다.

```
C#  
  
public class Person  
{  
    public string? FirstName { get; set; }  
  
    // Omitted for brevity.  
}
```

위에 표시된 구문은 자동 속성 구문입니다. 컴파일러는 속성을 백업하는 필드의 스토리지 위치를 생성합니다. 또한 컴파일러는 `get` 및 `set` 접근자의 본문을 구현합니다.

때로는 해당 형식의 기본값이 아닌 값으로 속성을 초기화해야 합니다. 이 작업을 위해 C#에서는 닫는 중괄호 뒤에 속성의 값을 설정합니다. `FirstName` 속성의 초기 값을 `null` 대신 빈 문자열로 설정할 수도 있습니다. 아래와 같이 지정하면 됩니다.

```
C#
```

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;

    // Omitted for brevity.
}
```

이 아티클의 뒷부분에서 살펴보겠지만 특정 초기화는 읽기 전용 속성에 가장 유용합니다.

아래 표시된 대로 스토리지를 직접 정의할 수도 있습니다.

C#

```
public class Person
{
    public string? FirstName
    {
        get { return _firstName; }
        set { _firstName = value; }
    }
    private string? _firstName;

    // Omitted for brevity.
}
```

속성 구현이 단일 식인 경우 getter 또는 setter에 대해 식 본문 멤버를 사용할 수 있습니다.

C#

```
public class Person
{
    public string? FirstName
    {
        get => _firstName;
        set => _firstName = value;
    }
    private string? _firstName;

    // Omitted for brevity.
}
```

이 아티클 전체에서 해당하는 경우 이 간소화된 구문이 사용됩니다.

위에 표시된 속성 정의는 읽기/쓰기 속성입니다. set 접근자에서 `value` 키워드를 확인합니다. `set` 접근자에는 항상 `value`라는 단일 매개 변수가 있습니다. `get` 접근자는 속성 형식(이 예제에서는 `string`)으로 변환할 수 있는 값을 반환해야 합니다.

이것이 기본 구문입니다. 다양한 디자인 관용구를 지원하는 다양한 변형이 있습니다. 각 변환에 대한 구문 옵션을 살펴보고 알아봅니다.

## 유효성 검사

위의 예제에서는 가장 간단한 속성 정의 사례 중 하나인 유효성 검사 없는 읽기/쓰기 속성을 보여 주었습니다. `get` 및 `set` 접근자에서 원하는 코드를 작성하여 다양한 시나리오를 만들 수 있습니다.

`set` 접근자에서 코드를 작성하여 속성으로 표현된 값이 항상 유효한지 확인합니다. 예를 들어 이름을 비워 두거나 공백일 수 없다는 `Person` 클래스에 대한 규칙이 있다고 가정합니다. 다음과 같이 작성할 수 있습니다.

C#

```
public class Person
{
    public string? FirstName
    {
        get => _firstName;
        set
        {
            if (string.IsNullOrWhiteSpace(value))
                throw new ArgumentException("First name must not be blank");
            _firstName = value;
        }
    }
    private string? _firstName;

    // Omitted for brevity.
}
```

속성 setter 유효성 검사의 일부인 `throw` 식을 사용하여 앞의 예제를 간소화할 수 있습니다.

C#

```
public class Person
{
    public string? FirstName
    {
        get => _firstName;
        set => _firstName = (!string.IsNullOrWhiteSpace(value)) ? value :
throw new ArgumentException("First name must not be blank");
    }
    private string? _firstName;
```

```
// Omitted for brevity.  
}
```

위의 예제에서는 첫 번째 이름을 비워 두거나 공백일 수 없다는 규칙을 적용합니다. 개발자가 작성하는 경우

C#

```
hero.FirstName = "";
```

해당 할당으로 인해 `ArgumentException`이 throw됩니다. 속성 set 접근자의 반환 형식이 `void`여야 하므로 예외를 throw하여 set 접근자에서 오류를 보고합니다.

이 동일한 구문을 시나리오에서 필요한 항목으로 확장할 수 있습니다. 서로 다른 속성 간의 관계를 확인하거나 모든 외부 조건에 대해 유효성을 검사할 수 있습니다. 유효한 모든 C# 문을 속성 접근자에서 사용할 수 있습니다.

## Access Control

이 시점까지 살펴본 모든 속성 정의는 공용 접근자를 사용한 읽기/쓰기 속성입니다. 속성에 유효한 유일한 액세스 가능성은 아닙니다. 읽기 전용 속성을 만들거나 `set` 및 `get` 접근자에 대해 다른 액세스 가능성을 제공할 수 있습니다. `Person` 클래스가 해당 클래스의 다른 메서드에서만 `FirstName` 속성의 값을 변경할 수 있도록 있다고 가정합니다. `set` 접근자에 `public` 대신 `private` 액세스 가능성을 제공할 수 있습니다.

C#

```
public class Person  
{  
    public string? FirstName { get; private set; }  
  
    // Omitted for brevity.  
}
```

이제 `FirstName` 속성을 모든 코드에서 액세스할 수 있지만 `Person` 클래스의 다른 코드에서만 할당할 수 있습니다.

`set` 또는 `get` 접근자에 제한적인 액세스 한정자를 추가할 수 있습니다. 개별 접근자에 설정하는 액세스 한정자는 속성 정의의 액세스 한정자보다 더 제한적이어야 합니다. 위 내용은 `FirstName` 속성이 `public`이지만 `set` 접근자가 `private`이므로 유효합니다. `public` 접근자를 사용하여 `private` 속성을 선언할 수 없습니다. 속성 선언을 `protected`, `internal`, `protected internal` 또는 `private`로 선언할 수도 있습니다.

`get` 접근자에 더 제한적인 한정자를 설정하는 것도 가능합니다. 예를 들어 `public` 속성이 있지만 `get` 접근자를 `private`로 제한할 수 있습니다. 이 시나리오는 실제로 거의 수행되지 않습니다.

## 읽기 전용

또한 생성자에서만 속성을 설정할 수 있도록 속성 수정을 제한할 수도 있습니다. 다음과 같이 `Person` 클래스를 수정할 수 있습니다.

```
C#  
  
public class Person  
{  
    public Person(string firstName) => FirstName = firstName;  
  
    public string FirstName { get; }  
  
    // Omitted for brevity.  
}
```

## Init-only

앞의 예에서는 호출자가 `FirstName` 매개 변수를 포함하는 생성자를 사용해야 합니다. 호출자는 [개체 이니셜라이저](#)를 사용하여 속성에 값을 할당할 수 없습니다. 이니셜라이저를 지원하려면 다음 코드에 표시된 대로 `set` 접근자를 `init` 접근자로 만들 수 있습니다.

```
C#  
  
public class Person  
{  
    public Person() { }  
    public Person(string firstName) => FirstName = firstName;  
  
    public string? FirstName { get; init; }  
  
    // Omitted for brevity.  
}
```

앞의 예에서는 해당 코드가 `FirstName` 속성을 설정하지 않은 경우에도 호출자가 기본 생성자를 사용하여 `Person`을 만들 수 있습니다. C# 11부터는 호출자에게 해당 속성을 설정하도록 요구할 수 있습니다.

```
C#
```

```
public class Person
{
    public Person() { }

    [SetsRequiredMembers]
    public Person(string firstName) => FirstName = firstName;

    public required string FirstName { get; init; }

    // Omitted for brevity.
}
```

앞의 코드는 `Person` 클래스에 두 가지를 추가합니다. 먼저, `FirstName` 속성 선언에는 `required` 한정자가 포함되어 있습니다. 이는 새 `Person`을 만드는 모든 코드가 이 속성을 설정해야 함을 의미합니다. 둘째, `firstName` 매개 변수를 사용하는 생성자에는 `System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute` 특성이 있습니다. 이 특성은 이 생성자가 모든 `required` 멤버를 설정한다는 것을 컴파일러에 알립니다.

### ① 중요

`required` 와 `null`을 혼동하지 않음을 혼동하지 마세요. `required` 속성을 `null` 또는 `default`로 설정하는 것이 유효합니다. 이 예의 `string`과 같이 형식이 `null`을 허용하지 않는 경우 컴파일러는 경고를 발급합니다.

호출자는 다음 코드에 표시된 대로 `SetsRequiredMembers`와 함께 생성자를 사용하거나 개체 이니셜라이저를 사용하여 `FirstName` 속성을 설정해야 합니다.

C#

```
var person = new VersionNinePoint2.Person("John");
person = new VersionNinePoint2.Person{ FirstName = "John"};
// Error CS9035: Required member `Person.FirstName` must be set:
//person = new VersionNinePoint2.Person();
```

## 계산된 속성

속성은 단순히 멤버 필드의 값을 반환할 필요가 없습니다. 계산된 값을 반환하는 속성을 만들 수 있습니다. `Person` 개체를 확장하여 이름과 성을 연결해서 계산된 전체 이름을 반환하겠습니다.

C#

```
public class Person
{
    public string? FirstName { get; set; }

    public string? LastName { get; set; }

    public string FullName { get { return $"{FirstName} {LastName}"; } }
}
```

위 예에서는 [문자열 보간](#) 기능을 사용하여 전체 이름에 대한 서식이 지정된 문자열을 만듭니다.

계산된 `FullName` 속성을 만드는 보다 간결한 방법을 제공하는 [식 본문 멤버](#)를 사용할 수도 있습니다.

C#

```
public class Person
{
    public string? FirstName { get; set; }

    public string? LastName { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}
```

식 본문 멤버는 람다 식 구문을 사용하여 단일 식이 포함된 메서드를 정의합니다. 여기서 해당 식은 person 개체의 전체 이름을 반환합니다.

## 캐시된 평가 속성

계산된 속성의 개념과 스토리지를 혼합하고 [캐시된 평가](#) 속성을 만들 수 있습니다. 예를 들어 처음 액세스할 때만 문자열 형식이 지정되도록 `FullName` 속성을 업데이트할 수 있습니다.

C#

```
public class Person
{
    public string? FirstName { get; set; }

    public string? LastName { get; set; }

    private string? _fullName;
    public string FullName
    {
        get

```

```

    {
        if (_fullName is null)
            _fullName = $"{FirstName} {LastName}";
        return _fullName;
    }
}

```

하지만 위의 코드에는 버그가 포함되어 있습니다. 코드가 `FirstName` 또는 `LastName` 속성의 값을 업데이트하는 경우 이전에 평가한 `fullName` 필드는 유효하지 않습니다.

`fullName` 필드가 다시 평가되도록 `FirstName` 및 `LastName` 속성의 `set` 접근자를 수정합니다.

C#

```

public class Person
{
    private string? _firstName;
    public string? FirstName
    {
        get => _firstName;
        set
        {
            _firstName = value;
            _fullName = null;
        }
    }

    private string? _lastName;
    public string? LastName
    {
        get => _lastName;
        set
        {
            _lastName = value;
            _fullName = null;
        }
    }

    private string? _fullName;
    public string FullName
    {
        get
        {
            if (_fullName is null)
                _fullName = $"{FirstName} {LastName}";
            return _fullName;
        }
    }
}

```

이 최종 버전은 필요한 경우에만 `FullName` 속성을 평가합니다. 이전에 계산한 버전이 유효한 경우 해당 버전이 사용됩니다. 다른 상태 변경으로 인해 이전에 계산한 버전이 무효화된 경우 다시 계산됩니다. 이 클래스를 사용하는 개발자는 구현의 세부 사항을 알 필요가 없습니다. 이러한 내부 변경 내용은 `Person` 개체의 사용에 영향을 주지 않습니다. 이것이 속성을 사용하여 개체의 데이터 멤버를 노출하는 주요 이유입니다.

## 자동 구현 속성에 특성 연결

자동 구현 속성의 컴파일러 생성 지원 필드에 필드 특성을 연결할 수 있습니다. 예를 들어 고유한 정수 `Id` 속성을 추가하는 `Person` 클래스에 대한 수정 버전을 사용합니다. 자동 구현 속성을 사용하여 `Id` 속성을 작성하지만 디자인은 `Id` 속성을 유지하기 위해 호출하지 않습니다. `NonSerializedAttribute`는 속성이 아니라 필드에만 연결할 수 있습니다. 다음 예제와 같이 특성에 `field:` 지정자를 사용하여 `Id` 속성의 지원 필드에 `NonSerializedAttribute`를 연결할 수 있습니다.

C#

```
public class Person
{
    public string? FirstName { get; set; }

    public string? LastName { get; set; }

    [field:NonSerialized]
    public int Id { get; set; }

    public string FullName => $"{FirstName} {LastName}";
}
```

이 기술은 자동 구현 속성의 지원 필드에 연결하는 모든 특성에서 작동합니다.

## INotifyPropertyChanged 구현

속성 접근자에서 코드를 작성해야 하는 최종 시나리오는 값이 변경되었다고 데이터 바인딩 클라이언트에 알리는 데 사용되는 `INotifyPropertyChanged` 인터페이스를 지원하는 것입니다. 속성의 값이 변경되면 개체가 `INotifyPropertyChanged.PropertyChanged` 이벤트를 발생시켜 변경되었음을 나타냅니다. 데이터 바인딩 라이브러리가 해당 변경 내용에 따라 차례로 표시 요소를 업데이트합니다. 아래 코드는 이 `person` 클래스의 `FirstName` 속성에 대해 `INotifyPropertyChanged`를 구현하는 방법을 보여 줍니다.

C#

```

public class Person : INotifyPropertyChanged
{
    public string? FirstName
    {
        get => _firstName;
        set
        {
            if (string.IsNullOrWhiteSpace(value))
                throw new ArgumentException("First name must not be blank");
            if (value != _firstName)
            {
                _firstName = value;
                PropertyChanged?.Invoke(this,
                    new PropertyChangedEventArgs(nameof(FirstName)));
            }
        }
    }
    private string? _firstName;

    public event PropertyChangedEventHandler? PropertyChanged;
}

```

? . 연산자를 *null* 조건부 연산자라고 합니다. 연산자의 오른쪽을 평가하기 전에 *null* 참조를 확인합니다. 최종 결과로, `PropertyChanged` 이벤트에 대한 구독자가 없는 경우 이벤트를 발생시키는 코드가 실행되지 않습니다. 해당 경우 이 확인을 수행하지 않으면 `NullReferenceException`이 throw됩니다. 자세한 내용은 `events`를 참조하세요. 또한 이 예제에서는 새 `nameof` 연산자를 사용하여 속성 이름 기호에서 해당 텍스트 표현으로 변환합니다. `nameof`을 사용하면 속성 이름을 잘못 입력한 오류를 줄일 수 있습니다.

`INotifyPropertyChanged`를 구현하는 작업은 필요한 시나리오를 지원하기 위해 접근자의 코드를 작성할 수 있는 경우의 예제입니다.

## 요약

속성은 클래스 또는 개체에 있는 스마트 필드의 한 형태입니다. 개체 외부에서는 개체의 필드와 유사하게 나타납니다. 그러나 C# 기능의 전체 팔레트를 사용하여 속성을 구현할 수 있습니다. 유효성 검사, 다른 액세스 가능성, 지역 평가 또는 시나리오에 필요한 모든 요구 사항을 제공할 수 있습니다.

 GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

끌어오기 요청을 만들고 검토할  
수도 있습니다. 자세한 내용은  
[참여자 가이드](#)를 참조하세요.

☞ 설명서 문제 열기

☞ 제품 사용자 의견 제공

# 인덱서

아티클 • 2023. 04. 08.

인덱서는 속성과 비슷합니다. 다양한 방식으로 인덱서는 속성과 동일한 언어 기능을 기반으로 합니다. 인덱서는 인덱싱된 속성, 즉 하나 이상의 인수로 참조된 속성을 사용하도록 설정합니다. 이러한 인수는 일부 값 컬렉션에 인덱스를 제공합니다.

## 인덱서 구문

변수 이름과 대괄호를 통해 인덱서에 액세스합니다. 인덱서 인수를 대괄호 안에 넣습니다.

C#

```
var item = someObject["key"];
someObject["AnotherKey"] = item;
```

`this` 키워드를 속성 이름으로 사용하고 대괄호 내에서 인수를 선언하여 인덱서를 선언합니다. 이 선언은 앞 단락에 표시된 사용법과 일치합니다.

C#

```
public int this[string key]
{
    get { return storage.Find(key); }
    set { storage.SetAt(key, value); }
}
```

이 초기 예제에서 속성 및 인덱서 구문 간의 관계를 확인할 수 있습니다. 이 유사성은 인덱서에 대한 대부분의 구문 규칙에 적용됩니다. 인덱서에는 유효한 모든 액세스 한정자 (public, protected internal, protected, internal, private 또는 private protected)를 사용할 수 있습니다. sealed, virtual 또는 abstract일 수 있습니다. 속성과 마찬가지로, 인덱서의 get 및 set 접근자에 대해 다양한 액세스 한정자를 지정할 수 있습니다. 읽기 전용 인덱서 (set 접근자 생략) 또는 쓰기 전용 인덱서 (get 접근자 생략)를 지정할 수도 있습니다.

속성 작업에서 배운 거의 모든 내용을 인덱서에 적용할 수 있습니다. 해당 규칙의 유일한 예외는 자동 구현 속성입니다. 컴파일러가 항상 인덱서에 올바른 스토리지를 생성할 수 있는 것은 아닙니다.

항목 집합의 항목을 참조하는 인수의 존재 여부로 인덱서와 속성을 구분합니다. 각 인덱서의 인수 목록이 고유하기만 하면 형식에 여러 인덱서를 정의할 수 있습니다. 클래스 정의에 하나 이상의 인덱서를 사용할 수 있는 다양한 시나리오를 살펴보겠습니다.

# 시나리오

API가 해당 컬렉션에 대한 인수가 정의되는 일부 컬렉션을 모델링하는 경우 형식에 인덱서를 정의합니다. 인덱서는 .NET Core Framework의 일부인 컬렉션 형식에 직접 매핑될 수도 있고, 매핑되지 않을 수도 있습니다. 형식에 컬렉션 모델링 이외의 다른 책임이 있을 수도 있습니다. 인덱서를 사용하면 해당 추상화의 값이 저장 또는 계산되는 방법의 내부 세부 정보를 노출하지 않고 형식의 추상화와 일치하는 API를 제공할 수 있습니다.

인덱서를 사용하기 위한 몇 가지 일반적인 시나리오를 살펴보겠습니다. [인덱서에 대한 샘플 풀더](#)에 액세스할 수 있습니다. 다운로드 지침은 [샘플 및 자습서](#)를 참조하세요.

## 배열 및 벡터

인덱서를 만들기 위한 가장 일반적인 시나리오 중 하나는 형식이 배열 또는 벡터를 모델링하는 경우입니다. 인덱서를 만들어 정렬된 데이터 목록을 모델링할 수 있습니다.

사용자 고유의 인덱서를 만드는 경우 해당 컬렉션에 대한 스토리지를 요구 사항에 맞게 정의할 수 있다는 장점이 있습니다. 너무 커서 한 번에 메모리에 로드할 수 없는 기록 데이터를 형식이 모델링하는 시나리오를 가정합니다. 사용량에 따라 컬렉션의 섹션을 로드 및 언로드해야 합니다. 다음 예제에서는 이 동작을 모델링합니다. 존재하는 데이터 요소 수를 보고합니다. 필요에 따라 데이터 섹션이 포함될 페이지를 만듭니다. 최신 요청에 필요한 페이지의 공간을 만들기 위해 메모리에서 페이지를 제거합니다.

C#

```
public class DataSamples
{
    private class Page
    {
        private readonly List<Measurements> pageData = new
List<Measurements>();
        private readonly int startingIndex;
        private readonly int length;
        private bool dirty;
        private DateTime lastAccess;

        public Page(int startingIndex, int length)
        {
            this.startingIndex = startingIndex;
            this.length = length;
            lastAccess = DateTime.Now;

            // This stays as random stuff:
            var generator = new Random();
            for(int i=0; i < length; i++)
            {
                var m = new Measurements
                {
                    Value = generator.Next(100, 1000),
                    Unit = "C"
                };
                pageData.Add(m);
            }
        }

        public void Add(Measurements m)
        {
            pageData.Add(m);
            dirty = true;
        }

        public void Remove(int index)
        {
            pageData.RemoveAt(index);
            dirty = true;
        }

        public void Clear()
        {
            pageData.Clear();
            dirty = true;
        }

        public void Save()
        {
            // ...
        }

        public void Load()
        {
            // ...
        }
    }
}
```

```

        {
            HiTemp = generator.Next(50, 95),
            LoTemp = generator.Next(12, 49),
            AirPressure = 28.0 + generator.NextDouble() * 4
        };
        pageData.Add(m);
    }
}

public bool HasItem(int index) =>
    ((index >= startingIndex) &&
     (index < startingIndex + length));

public Measurements this[int index]
{
    get
    {
        lastAccess = DateTime.Now;
        return pageData[index - startingIndex];
    }
    set
    {
        pageData[index - startingIndex] = value;
        dirty = true;
        lastAccess = DateTime.Now;
    }
}

public bool Dirty => dirty;
public DateTime LastAccess => lastAccess;
}

private readonly int totalSize;
private readonly List<Page> pagesInMemory = new List<Page>();

public DataSamples(int totalSize)
{
    this.totalSize = totalSize;
}

public Measurements this[int index]
{
    get
    {
        if (index < 0)
            throw new IndexOutOfRangeException("Cannot index less than
0");
        if (index >= totalSize)
            throw new IndexOutOfRangeException("Cannot index past the
end of storage");

        var page = updateCachedPagesForAccess(index);
        return page[index];
    }
    set
    {

```

```

        if (index < 0)
            throw new IndexOutOfRangeException("Cannot index less than
0");
        if (index >= totalSize)
            throw new IndexOutOfRangeException("Cannot index past the
end of storage");
        var page = updateCachedPagesForAccess(index);

        page[index] = value;
    }
}

private Page updateCachedPagesForAccess(int index)
{
    foreach (var p in pagesInMemory)
    {
        if (p.HasItem(index))
        {
            return p;
        }
    }
    var startingIndex = (index / 1000) * 1000;
    var nextPage = new Page(startingIndex, 1000);
    addPageToCache(nextPage);
    return nextPage;
}

private void addPageToCache(Page p)
{
    if (pagesInMemory.Count > 4)
    {
        // remove oldest non-dirty page:
        var oldest = pagesInMemory
            .Where(page => !page.Dirty)
            .OrderBy(page => page.LastAccess)
            .FirstOrDefault();
        // Note that this may keep more than 5 pages in memory
        // if too much is dirty
        if (oldest != null)
            pagesInMemory.Remove(oldest);
    }
    pagesInMemory.Add(p);
}
}

```

이 디자인 구문에 따라 전체 데이터 집합을 메모리 내 컬렉션에 로드할 필요가 없는 모든 종류의 컬렉션을 모델링할 수 있습니다. `Page` 클래스는 공용 인터페이스의 일부가 아닌 중첩된 `private` 클래스입니다. 이러한 세부 정보는 이 클래스의 모든 사용자로부터 숨겨집니다.

## 사전

또 다른 일반적인 시나리오는 사전 또는 맵을 모델링해야 하는 경우입니다. 이 시나리오는 형식이 키, 일반적으로 텍스트 키에 따라 값을 저장하는 경우입니다. 이 예제에서는 해당 옵션을 관리하는 [람다 식](#)에 명령줄 인수를 매핑하는 사전을 만듭니다. 다음 예제에서 명령줄 옵션을 `Action` 대리자에 매핑하는 `ArgsActions` 클래스와 해당 옵션을 발견할 경우 `ArgsActions`를 사용하여 각 `Action`을 실행하는 `ArgsProcessor` 클래스 등 두 개의 클래스를 보여 줍니다.

C#

```
public class ArgsProcessor
{
    private readonly ArgsActions actions;

    public ArgsProcessor(ArgsActions actions)
    {
        this.actions = actions;
    }

    public void Process(string[] args)
    {
        foreach(var arg in args)
        {
            actions[arg]?.Invoke();
        }
    }
}

public class ArgsActions
{
    readonly private Dictionary<string, Action> argsActions = new
Dictionary<string, Action>();

    public Action this[string s]
    {
        get
        {
            Action action;
            Action defaultAction = () => {} ;
            return argsActions.TryGetValue(s, out action) ? action :
defaultAction;
        }
    }

    public void SetOption(string s, Action a)
    {
        argsActions[s] = a;
    }
}
```

이 예제에서 `ArgsAction` 컬렉션은 기본 컬렉션과 거의 같도록 매핑됩니다. `get`은 지정된 옵션이 구성되었는지 여부를 확인합니다. 구성된 경우 해당 옵션과 연결된 `Action`을 반환합니다. 구성되지 않은 경우 아무 작업도 수행하지 않는 `Action`을 반환합니다. `public` 접근자는 `set` 접근자를 포함하지 않습니다. 대신, 이 디자인은 옵션 설정에 `public` 메서드를 사용합니다.

## 다차원 맵

여러 인수를 사용하는 인덱서를 만들 수 있습니다. 또한 이러한 인수는 같은 형식으로 제한되지 않습니다. 두 가지 예제를 살펴보겠습니다.

첫 번째 예제는 Mandelbrot 집합의 값을 생성하는 클래스를 보여 줍니다. 집합 뒤의 수학에 대한 자세한 내용은 [이 문서](#)를 참조하세요. 인덱서는 두 개의 `double`을 사용하여 X, Y 평면의 한 지점을 정의합니다. `get` 접근자는 한 지점이 집합에 없는 것으로 확인될 때까지 반복 횟수를 계산합니다. 최대 반복 횟수에 도달하면 지점이 집합에 있고 클래스의 `maxIterations` 값이 반환됩니다. (Mandelbrot 집합에 대해 잘 알려진 컴퓨터 생성 이미지는 한 지점이 집합 외부에 있음을 확인하는 데 필요한 반복 횟수의 색을 정의합니다.)

C#

```
public class Mandelbrot
{
    readonly private int maxIterations;

    public Mandelbrot(int maxIterations)
    {
        this.maxIterations = maxIterations;
    }

    public int this [double x, double y]
    {
        get
        {
            var iterations = 0;
            var x0 = x;
            var y0 = y;

            while ((x*x + y * y < 4) &&
                   (iterations < maxIterations))
            {
                var newX = x * x - y * y + x0;
                y = 2 * x * y + y0;
                x = newX;
                iterations++;
            }
            return iterations;
        }
    }
}
```

```
    }  
}
```

Mandelbrot 집합은 실수 값의 모든 (x, y) 좌표에서 값을 정의합니다. 그러면 무한 개수의 값을 포함할 수 있는 사전이 정의됩니다. 따라서 집합 뒤에는 스토리지가 없습니다. 대신, 이 클래스는 코드에서 `get` 접근자를 호출할 때 각 지점의 값을 계산합니다. 사용되는 기본 스토리지는 없습니다.

인덱서가 서로 다른 형식의 여러 인수를 사용하는 인덱서의 마지막 사용 방법을 살펴보겠습니다. 기록 온도 데이터를 관리하는 프로그램을 가정합니다. 이 인덱서는 도시 및 날짜를 사용하여 해당 위치의 상한 및 하한 온도를 설정하거나 가져옵니다.

C#

```
using DateMeasurements =  
    System.Collections.Generic.Dictionary<System.DateTime,  
IndexersSamples.Common.Measurements>;  
using CityDataMeasurements =  
    System.Collections.Generic.Dictionary<string,  
System.Collections.Generic.Dictionary<System.DateTime,  
IndexersSamples.Common.Measurements>>;  
  
public class HistoricalWeatherData  
{  
    readonly CityDataMeasurements storage = new CityDataMeasurements();  
  
    public Measurements this[string city, DateTime date]  
    {  
        get  
        {  
            var cityData = default(DateMeasurements);  
  
            if (!storage.TryGetValue(city, out cityData))  
                throw new ArgumentOutOfRangeException(nameof(city), "City  
not found");  
  
            // strip out any time portion:  
            var index = date.Date;  
            var measure = default(Measurements);  
            if (cityData.TryGetValue(index, out measure))  
                return measure;  
            throw new ArgumentOutOfRangeException(nameof(date), "Date not  
found");  
        }  
        set  
        {  
            var cityData = default(DateMeasurements);  
  
            if (!storage.TryGetValue(city, out cityData))  
            {  
                cityData = new DateMeasurements();  
                storage[city] = cityData;  
            }  
            cityData[date] = value;  
        }  
    }  
}
```

```

        storage.Add(city, cityData);
    }

    // Strip out any time portion:
    var index = date.Date;
    cityData[index] = value;
}
}
}

```

이 예제에서는 도시(`string`) 및 날짜(`DateTime`)의 두 인수에 대한 날씨 데이터를 매핑하는 인덱서를 만듭니다. 내부 스토리지는 두 개의 `Dictionary` 클래스를 사용하여 2차원 사전을 나타냅니다. 공용 API는 더 이상 기본 스토리지를 나타내지 않습니다. 대신, 기본 스토리지가 다양한 핵심 컬렉션 형식을 사용해야 하는 경우에도 인덱서의 언어 기능을 사용하여 추상화를 나타내는 공용 인터페이스를 만들 수 있습니다.

일부 개발자에게 친숙하지 않을 수 있는 이 코드의 두 부분이 있습니다. 이러한 두 `using` 지시문은 다음과 같습니다.

C#

```

using DateMeasurements =
System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>;
using CityDataMeasurements = System.Collections.Generic.Dictionary<string,
System.Collections.Generic.Dictionary<System.DateTime,
IndexersSamples.Common.Measurements>>;

```

두 문은 생성된 제네릭 형식의 별칭을 만듭니다. 이러한 문을 통해 나중에 코드에서 `Dictionary<DateTime, Measurements>` 및 `Dictionary<string, Dictionary<DateTime, Measurements>>`의 제네릭 구문이 아니라 더 설명적인 `DateMeasurements` 및 `CityDataMeasurements` 이름을 사용할 수 있습니다. 이 구문의 경우 `=` 기호의 오른쪽에 정규화된 형식 이름을 사용해야 합니다.

두 번째 방법은 컬렉션에 인덱싱하는 데 사용되는 `DateTime` 개체의 시간 부분을 제거하는 것입니다. .NET에는 날짜 전용 형식이 포함되어 있지 않습니다. 개발자는 `DateTime` 형식을 사용하지만 `Date` 속성을 사용하여 해당 날짜의 `DateTime` 개체가 모두 같도록 합니다.

## 요약

해당 속성이 단일 값이 아니라 각 개별 항목이 인수 집합으로 식별되는 값 컬렉션을 나타내는, 속성과 유사한 요소가 클래스에 있을 경우 항상 인덱서를 만들어야 합니다. 이러한 인수는 참조해야 하는 컬렉션의 항목을 고유하게 식별할 수 있습니다. 인덱서는 [속성](#) 개

념을 확장하며, 이 경우 멤버가 클래스 외부의 데이터 항목처럼 처리되지만 부가적으로 내부의 메서드처럼 처리됩니다. 인덱서를 사용하면 인수가 항목 집합을 나타내는 속성에서 단일 항목을 찾을 수 있습니다.

# 반복기

아티클 • 2023. 04. 08.

작성하는 거의 모든 프로그램에서 컬렉션을 반복해야 하는 경우가 있습니다. 컬렉션에 있는 모든 항목을 조사하는 코드를 작성합니다.

또한 해당 클래스의 요소에 대해 반복기를 생성하는 메서드인 반복기 메서드를 만듭니다. 반복기는 컨테이너, 특히 목록을 트래버스하는 개체입니다. 반복기는 다음과 같은 경우에 사용할 수 있습니다.

- 컬렉션의 각 항목에 대한 작업 수행.
- 사용자 지정 컬렉션 열거.
- [LINQ](#) 또는 다른 라이브러리 확장.
- 데이터가 반복기 메서드를 통해 효율적으로 흐르는 데이터 파이프라인 만들기.

C# 언어는 시퀀스를 생성하고 사용하는 기능을 제공합니다. 이러한 시퀀스는 동기적 또는 비동기적으로 생성 및 사용될 수 있습니다. 이 문서에서는 해당 기능에 대한 개요를 제공합니다.

## foreach로 반복 처리

컬렉션 열거는 간단합니다. `foreach` 키워드는 컬렉션을 열거하여 컬렉션의 각 요소에 대해 포함된 문을 한 번 실행합니다.

C#

```
foreach (var item in collection)
{
    Console.WriteLine(item?.ToString());
}
```

그게 다입니다. 컬렉션의 모든 내용을 반복하려면 `foreach` 문만 있으면 됩니다. 하지만 `foreach` 문이 마법은 아닙니다. 이 명령문은 컬렉션을 반복하는 데 필요한 코드를 생성하기 위해 .NET core 라이브러리에 정의된 두 개의 제네릭 인터페이스인 `IEnumerable<T>` 및 `IEnumerator<T>`를 사용합니다. 이 메커니즘은 아래에 더 자세히 설명되어 있습니다.

이러한 인터페이스 둘 다에는 제네릭이 아닌 인터페이스 `IEnumerable` 및 `IEnumerator`도 있습니다. 최신 코드에는 [제네릭](#) 버전이 기본적으로 사용됩니다.

시퀀스를 비동기적으로 생성하는 경우 `await foreach` 문을 사용하여 비동기적으로 시퀀스를 사용할 수 있습니다.

C#

```
await foreach (var item in asyncSequence)
{
    Console.WriteLine(item?.ToString());
}
```

시퀀스가 `System.Collections.Generic.IEnumerable<T>` 인 경우 `foreach`를 사용합니다. 시퀀스가 `System.Collections.Generic.IAsyncEnumerable<T>` 인 경우 `await foreach`를 사용합니다. 후자의 경우 시퀀스는 비동기식으로 생성됩니다.

## 반복기 메서드를 사용하는 열거형 소스

C# 언어의 또 다른 유용한 기능을 통해 열거형 소스를 만드는 메서드를 작성할 수 있습니다. 이러한 메서드를 **반복기 메서드**라고 합니다. 반복기 메서드는 요청될 때 시퀀스에서 개체를 생성하는 방법을 정의합니다. `yield return` 상황별 키워드를 사용하여 반복기 메서드를 정의합니다.

이 메서드를 작성하여 0에서 9 사이의 정수 시퀀스를 생성할 수 있습니다.

C#

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    yield return 0;
    yield return 1;
    yield return 2;
    yield return 3;
    yield return 4;
    yield return 5;
    yield return 6;
    yield return 7;
    yield return 8;
    yield return 9;
}
```

위의 코드에서는 반복기 메서드에서 여러 개의 고유 `yield return` 문을 사용할 수 있다는 사실을 강조하기 위해 고유 `yield return` 문을 보여 줍니다. 다른 언어 구문을 사용하여 반복기 메서드의 코드를 단순화할 수 있으며 종종 그렇게 합니다. 아래의 메서드 정의는 정확히 동일한 시퀀스의 숫자를 생성합니다.

C#

```
public IEnumerable<int> GetSingleDigitNumbersLoop()
{
    int index = 0;
```

```
    while (index < 10)
        yield return index++;
}
```

둘 중 하나를 결정할 필요가 없습니다. 메서드의 요구를 충족하는 데 필요한 만큼 `yield return` 문을 사용할 수 있습니다.

C#

```
public IEnumerable<int> GetSetsOfNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    index = 100;
    while (index < 110)
        yield return index++;
}
```

위의 모든 예제에는 비동기 대응 항목이 있습니다. 각 경우에서 `IEnumerable<T>`의 반환 형식을 `IAsyncEnumerable<T>`로 바꿉니다. 예를 들어 앞의 예제에는 다음과 같은 비동기 버전이 있습니다.

C#

```
public async IAsyncEnumerable<int> GetSetsOfNumbersAsync()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    await Task.Delay(500);

    yield return 50;

    await Task.Delay(500);

    index = 100;
    while (index < 110)
        yield return index++;
}
```

이는 동기 및 비동기 반복기 모두에 대한 구문입니다. 실제 예제를 생각해 보겠습니다. IoT 프로젝트를 진행하고 있고 디바이스 센서는 매우 큰 데이터 스트림을 생성한다고 가

정합니다. 데이터를 파악하려면 N번째 데이터 요소마다 샘플링하는 메서드를 작성할 수 있습니다. 이 작은 반복기 메서드면 충분합니다.

C#

```
public static IEnumerable<T> Sample<T>(this IEnumerable<T> sourceSequence,
int interval)
{
    int index = 0;
    foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}
```

IoT 디바이스에서 읽기가 비동기 시퀀스를 생성하는 경우 다음 메서드가 보여 주는 것처럼 메서드를 수정합니다.

C#

```
public static async IAsyncEnumerable<T> Sample<T>(this IAsyncEnumerable<T>
sourceSequence, int interval)
{
    int index = 0;
    await foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}
```

반복기 메서드에는 한 가지 중요한 제한이 있습니다. `return` 문과 `yield return` 문 둘 다를 동일한 메서드에서 사용할 수 없습니다. 다음 코드는 컴파일되지 않습니다.

C#

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    // generates a compile time error:
    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109};
    return items;
}
```

일반적으로 이 제한은 문제가 되지 않습니다. 메서드 전체에서 `yield return`을 사용하거나, 원래 메서드를 여러 메서드로 분리하여 일부는 `return`을 사용하고 일부는 `yield return`을 사용할 수 있습니다.

모든 위치에서 `yield return`을 사용하기 위해 마지막 메서드를 약간 수정할 수 있습니다.

C#

```
public IEnumerable<int> GetFirstDecile()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109};
    foreach (var item in items)
        yield return item;
}
```

경우에 따라 반복기 메서드를 두 개의 다른 메서드로 분할하는 것이 정답일 수 있습니다. 하나는 `return`을 사용하고 다른 하나는 `yield return`을 사용합니다. 부울 인수에 따라 빈 컬렉션 또는 처음 5개의 홀수를 반환하려는 상황을 가정해 보세요. 다음과 같은 두 메서드로 작성할 수 있습니다.

C#

```
public IEnumerable<int> GetSingleDigitOddNumbers(bool getCollection)
{
    if (getCollection == false)
        return new int[0];
    else
        return IteratorMethod();
}

private IEnumerable<int> IteratorMethod()
{
    int index = 0;
    while (index < 10)
    {
        if (index % 2 == 1)
            yield return index;
        index++;
    }
}
```

위의 메서드를 살펴보세요. 첫 번째 메서드는 표준 `return` 문을 사용하여 빈 컬렉션 또는 두 번째 메서드에서 만든 반복기를 반환합니다. 두 번째 메서드는 `yield return` 문을 사용하여 요청된 시퀀스를 만듭니다.

## foreach 심층 분석

`foreach` 문은 `IEnumerable<T>` 및 `IEnumerator<T>` 인터페이스를 사용하여 컬렉션의 모든 요소에서 반복하는 표준 관용구로 확장됩니다. 또한 개발자가 리소스를 제대로 관리하지 못해 발생하는 오류를 최소화합니다.

컴파일러는 첫 번째 예제에 표시된 `foreach` 루프를 다음과 유사한 구문으로 변환합니다.

C#

```
IEnumerator<int> enumerator = collection.GetEnumerator();
while (enumerator.MoveNext())
{
    var item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

컴파일러에서 생성되는 정확한 코드는 더 복잡하며 `GetEnumerator()`에서 반환된 객체가 `IDisposable` 인터페이스를 구현하는 상황을 처리합니다. 전체 확장에서는 다음과 더 유사한 코드를 생성합니다.

C#

```
{
    var enumerator = collection.GetEnumerator();
    try
    {
        while (enumerator.MoveNext())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    }
    finally
    {
        // dispose of enumerator.
    }
}
```

컴파일러는 첫 번째 비동기 샘플을 다음과 유사한 구문으로 변환합니다.

C#

```
{  
    var enumerator = collection.GetAsyncEnumerator();  
    try  
    {  
        while (await enumerator.MoveNextAsync())  
        {  
            var item = enumerator.Current;  
            Console.WriteLine(item.ToString());  
        }  
    }  
    finally  
    {  
        // dispose of async enumerator.  
    }  
}
```

열거자가 삭제되는 방식의 `enumerator` 형식의 특성에 따라 달라집니다. 일반 동기 사례에서 `finally` 절은 다음과 같이 확장됩니다.

C#

```
finally  
{  
    (enumerator as IDisposable)?.Dispose();  
}
```

일반 비동기 사례는 다음과 같이 확장됩니다.

C#

```
finally  
{  
    if (enumerator is IAsyncDisposable asyncDisposable)  
        await asyncDisposable.DisposeAsync();  
}
```

그러나 `enumerator`의 형식이 sealed 형식이고 `enumerator`의 형식에서 `IDisposable` 또는 `IAsyncDisposable`로의 암시적 변환이 없는 경우 `finally` 절은 빈 블록으로 확장됩니다.

C#

```
finally  
{  
}
```

`enumerator`의 형식에서 `IDisposable`로의 암시적 변환이 있고 `enumerator` 형식이 nullable이 아닌 값 형식인 경우 `finally` 절은 다음과 같이 확장됩니다.

C#

```
finally
{
    ((IDisposable)enumerator).Dispose();
}
```

다행히도 이러한 세부 정보를 모두 기억할 필요가 없습니다. `foreach` 문에서 이러한 차이를 모두 처리합니다. 컴파일러는 이러한 구문에 대한 올바른 코드를 생성합니다.

# C#의 대리자 및 이벤트 소개

아티클 • 2024. 02. 19.

대리자는 .NET에서 런타임에 바인딩 메커니즘을 제공합니다. 런타임에 바인딩은 호출자가 알고리즘의 일부를 구현하는 하나 이상의 메서드도 제공하는 알고리즘을 만든다는 의미입니다.

예를 들어 천문학 애플리케이션에서 별 목록을 정렬한다는 가정해 보세요. 이러한 별은 지구로부터의 거리, 별의 크기, 인식된 밝기 등에 따라 정렬할 수 있습니다.

모든 경우에 Sort() 메서드는 기본적으로 동일한 작업을 수행합니다. 즉, 몇 가지 비교를 통해 목록의 항목을 정렬합니다. 별 두 개를 비교하는 코드는 각 정렬 순서마다 다릅니다.

이러한 종류의 솔루션이 반세기 동안 소프트웨어에서 사용되었습니다. C# 언어 대리자 개념은 최고 수준의 언어 지원 및 해당 개념 관련 형식 안정성을 제공합니다.

이 시리즈의 뒷부분에서 볼 수 있듯이 이와 같은 알고리즘에 대해 작성하는 C# 코드는 형식이 안전합니다. 컴파일러는 형식이 인수 및 반환 형식과 일치하는지 확인합니다.

[함수 포인터](#)는 호출 규칙에 대한 더 많은 제어가 필요한 유사한 시나리오를 지원합니다. 대리자와 연결된 코드는 대리자 형식에 추가된 가상 메서드를 사용하여 호출됩니다. 함수 포인터를 사용하여 다른 규칙을 지정할 수 있습니다.

## 대리자의 언어 디자인 목표

언어 디자이너는 결국 대리자가 된 기능에 대해 여러 가지 목표를 열거했습니다.

팀은 런타임에 바인딩 알고리즘에 사용할 수 있는 공용 언어 구문을 원했습니다. 대리자를 통해 개발자는 하나의 개념을 익히고 여러 가지 다양한 소프트웨어 문제에서 동일한 개념을 사용할 수 있습니다.

두 번째로 팀은 단일 및 멀티캐스트 메서드 호출을 모두 지원하기를 원했습니다. (멀티캐스트 대리자는 여러 메서드 호출을 함께 연결하는 대리자입니다. 예제는 [이 시리즈의 뒷부분](#)에서 확인할 수 있습니다.)

팀은 개발자가 모든 C# 구문에서 기대하는 동일한 형식 안전성을 대리자가 지원하기를 원했습니다.

마지막으로 팀은 이벤트 패턴이 대리자 또는 지역 바인딩 알고리즘이 유용한 하나의 특정 패턴임을 인식했습니다. 팀은 대리자에 대한 코드가 .NET 이벤트 패턴의 기반을 제공할 수 있도록 하려고 했습니다.

이러한 모든 작업의 결과가 C# 및 .NET의 대리자와 이벤트 지원이었습니다.

이 시리즈의 나머지 문서에서는 대리자 및 이벤트로 작업할 때 사용되는 언어 기능, 라이브러리 지원 및 일반적인 관용구를 다룹니다. 다음에 대해 알아봅니다.

- `delegate` 키워드 및 해당 키워드에서 생성하는 코드입니다.
- `System.Delegate` 클래스의 기능 및 해당 기능이 사용되는 방법입니다.
- 형식이 안전한 대리자를 만드는 방법입니다.
- 대리자를 통해 호출할 수 있는 메서드를 만드는 방법입니다.
- 람다 식을 사용하여 대리자 및 이벤트로 작업하는 방법입니다.
- 대리자가 LINQ의 구성 요소 중 하나가 되는 방법입니다.
- 대리자가 .NET 이벤트 패턴의 기초가 되는 방식과 대리자의 차이점입니다.

시작하겠습니다.

다음

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# System.Delegate 및 delegate 키워드

아티클 • 2023. 04. 08.

## 이전

이 문서에서는 .NET에서 대리자를 지원하는 클래스와 해당 클래스가 `delegate` 키워드에 매핑되는 방법을 다룹니다.

## 대리자 유형 정의

먼저 'delegate' 키워드를 살펴보겠습니다. 대리자 작업을 수행할 때 기본적으로 이 키워드를 사용하기 때문입니다. `delegate` 키워드를 사용할 때 컴파일러가 생성하는 코드는 `Delegate` 및 `MulticastDelegate` 클래스의 멤버를 호출하는 메서드 호출에 매핑됩니다.

메서드 시그니처를 정의하는 것과 비슷한 구문을 사용하여 대리자 형식을 정의합니다. `delegate` 키워드를 정의에 추가하면 됩니다.

계속해서 `List.Sort()` 메서드를 예제로 사용합니다. 첫 번째 단계는 비교 대리자에 대한 형식을 만드는 것입니다.

C#

```
// From the .NET Core library  
  
// Define the delegate type:  
public delegate int Comparison<in T>(T left, T right);
```

컴파일러에서는 사용된 시그니처와 일치하는 `System.Delegate`에서 파생된 클래스를 생성합니다(이 경우 정수를 반환하고 두 개의 인수가 포함된 메서드). 해당 대리자의 형식은 `Comparison`입니다. `Comparison` 대리자 형식은 제네릭 형식입니다. 제네릭에 대한 자세한 내용은 [여기를 참조하세요](#).

구문이 변수를 선언하는 것처럼 보일 수 있지만 실제로는 형식을 선언합니다. 클래스 내부, 직접 네임스페이스 내부 또는 전역 네임스페이스에 대리자 형식을 정의할 수 있습니다.

### ① 참고

전역 네임스페이스에 직접 대리자 형식(또는 기타 형식)을 선언하는 것은 권장하지 않습니다.

이 클래스의 클라이언트가 인스턴스의 호출 목록에서 메서드를 추가 및 제거할 수 있도록 컴파일러에서는 이 새로운 형식에 대한 추가 및 제거 처리기를 생성합니다. 컴파일러는 추가되거나 제거되는 메서드의 시그니처가 메서드를 선언할 때 사용된 시그니처와 일치하도록 지정합니다.

## 대리자 인스턴스 선언

대리자를 정의한 후 해당 형식의 인스턴스를 만들 수 있습니다. C#의 모든 변수처럼 네임스페이스에서 직접 또는 전역 네임스페이스에서 대리자 인스턴스를 선언할 수 없습니다.

C#

```
// inside a class definition:  
  
// Declare an instance of that type:  
public Comparison<T> comparator;
```

변수 형식은 앞에서 정의한 대리자 형식인 `Comparison<T>`입니다. 변수 이름은 `comparator`입니다.

위의 코드 조각에서는 클래스 내부에 멤버 변수를 선언했습니다. 메서드에 대한 인수 또는 지역 변수인 대리자 변수를 선언할 수도 있습니다.

## 대리자 호출

대리자를 호출하여 대리자 호출 목록에 있는 메서드를 호출합니다. `Sort()` 메서드 내부에서 코드는 비교 메서드를 호출하여 개체를 배치할 순서를 결정합니다.

C#

```
int result = comparator(left, right);
```

위 줄에서 코드는 대리자에 연결된 메서드를 호출합니다. 변수를 메서드 이름으로 처리하고 일반 메서드 호출 구문을 사용하여 변수를 호출합니다.

해당 코드 줄은 안전하지 않은 가정을 생성합니다. 대상이 대리자에 추가되었다는 보장이 없습니다. 대상이 연결되지 않은 경우 위 줄은 `NullReferenceException`을 throw합니다. 이 문제를 해결하는 데 사용된 관용구는 간단한 null 검사보다 더 복잡하고 이 [시리즈](#)의 뒷부분에서 설명합니다.

## 호출 대상 할당, 추가 및 제거

이 방법으로 대리자 형식을 정의하고 대리자 인스턴스를 선언 및 호출합니다.

`List.Sort()` 메서드를 사용하려는 개발자는 시그니처가 대리자 형식 정의와 일치하는 메서드를 정의하고 정렬 메서드에서 사용된 대리자에 할당해야 합니다. 이 할당으로 해당 대리자 객체의 호출 목록에 메서드를 추가합니다.

길이별로 문자열 목록을 정렬한다고 가정합니다. 비교 함수는 다음과 같을 수 있습니다.

C#

```
private static int CompareLength(string left, string right) =>
    left.Length.CompareTo(right.Length);
```

메서드는 `private` 메서드로 선언됩니다. 괜찮습니다. 이 메서드를 `public` 인터페이스에 포함하지 않으려고 할 수 있습니다. 대리자에 연결될 경우 비교 메서드로 계속 사용할 수 있습니다. 호출 코드에서는 이 메서드를 대리자 객체의 대상 목록에 연결하고 해당 대리자를 통해 메서드에 액세스할 수 있습니다.

해당 메서드를 `List.Sort()` 메서드에 전달하여 관계를 만듭니다.

C#

```
phrases.Sort(CompareLength);
```

메서드 이름은 괄호 없이 사용됩니다. 메서드를 인수로 사용하면 메서드 참조를 대리자 호출 대상으로 사용될 수 있는 참조로 변환하고 해당 메서드를 호출 대상으로 연결하도록 컴파일러에 알립니다.

`Comparison<string>` 형식의 변수를 선언하고 할당을 수행하여 명시적 상태일 수도 있습니다.

C#

```
Comparison<string> comparer = CompareLength;
phrases.Sort(comparer);
```

대리자 대상으로 사용되는 메서드가 작은 메서드인 경우에는 일반적으로 [람다 식](#) 구문을 사용하여 할당을 수행합니다.

C#

```
Comparison<string> comparer = (left, right) =>
    left.Length.CompareTo(right.Length);
phrases.Sort(comparer);
```

대리자 대상에 람다 식을 사용하는 방법은 [이후 섹션](#)에서 자세히 설명합니다.

Sort() 예제에서는 일반적으로 단일 대상 메서드를 대리자에 연결합니다. 그러나 대리자 개체는 여러 대상 메서드가 대리자 개체에 연결되어 있는 호출 목록을 지원합니다.

## Delegate 및 MulticastDelegate 클래스

위에 설명된 언어 지원은 일반적으로 대리자를 사용할 때 필요한 기능과 지원을 제공합니다. 이러한 기능은 .NET Core Framework의 두 가지 클래스 [Delegate](#) 및 [MulticastDelegate](#)를 기반으로 빌드됩니다.

`System.Delegate` 클래스와 단일 직접 하위 클래스 `System.MulticastDelegate`는 대리자를 만들고, 메서드를 대리자 대상으로 등록하고, 대리자 대상으로 등록된 모든 메서드를 호출하기 위한 프레임워크 지원을 제공합니다.

흥미롭게도 `System.Delegate` 및 `System.MulticastDelegate` 클래스는 자체가 대리자 형식이 아닙니다. 모든 특정 대리자 형식에 대한 기초를 제공합니다. 동일한 언어 디자인 프로세스에서는 `Delegate` 또는 `MulticastDelegate`에서 파생되는 클래스를 선언할 수 없도록 요구했습니다. C# 언어 규칙은 이러한 선언을 금지합니다.

대신에 C# 컴파일러는 C# 언어 키워드를 사용하여 대리자 형식을 선언할 경우 `MulticastDelegate`에서 파생된 클래스의 인스턴스를 만듭니다.

이 디자인은 C# 및 .NET의 첫 번째 릴리스를 기반으로 합니다. 디자인 팀의 한 가지 목적은 언어에서 대리자를 사용할 때 형식 안전성을 적용하는지 확인하는 것입니다. 이는 대리자가 인수의 올바른 형식 및 개수를 사용하여 호출되는지 확인함을 의미합니다. 또한 컴파일 시간에 반환 형식이 제대로 표시되었는지 확인함을 의미합니다. 대리자는 이전에 제네릭이었던 1.0 .NET 릴리스의 일부였습니다.

이 형식 안전성을 적용하는 가장 좋은 방법은 컴파일러에서 사용되는 메서드 시그니처를 표현한 구체적인 대리자 클래스를 만드는 것입니다.

파생 클래스를 만들 수 없더라도 이러한 클래스에서 정의된 메서드를 사용하게 됩니다. 대리자 관련 작업을 수행할 때 사용할 가장 일반적인 메서드를 살펴보겠습니다.

기억해야 하는 가장 중요한 첫 번째 사실은 사용하는 모든 대리자는 `MulticastDelegate`에서 파생된다는 것입니다. 멀티캐스트 대리자는 대리자를 통해 호출할 경우 두 개 이상의 메서드 대상이 호출될 수 있음을 의미합니다. 원래 디자인에서는 하나의 대상 메서드만 연결 및 호출할 수 대리자와 여러 대상 메서드를 연결 및 호출할 수 있는 대리자를 구분하도록 고려했습니다. 이 구분은 원래 고려한 것보다 실제로 그다지 유용하지 않았습니다. 두 가지 클래스가 이미 만들어졌고 초기 공식 릴리스 이후 프레임워크에 포함되었습니다.

대리자와 함께 가장 많이 사용할 메서드는 `Invoke()` 및 `BeginInvoke()` / `EndInvoke()`입니다. `Invoke()`는 특정 대리자 인스턴스에 연결된 모든 메서드를 호출합니다. 위에서 확인한 대로, 일반적으로 대리자 변수에서 메서드 호출 스택을 사용하여 대리자를 호출합니다. 이 시리즈의 뒷부분에서 살펴보겠지만 이러한 메서드에서 직접 사용되는 패턴이 있습니다.

이제 언어 구문 및 대리자 지원 클래스를 확인했으므로 강력한 형식의 대리자를 사용하고, 만들고, 호출하는 방법을 살펴보겠습니다.

[다음](#)

# 강력한 형식의 대리자

아티클 • 2023. 04. 08.

## 이전

이전 문서에서는 `delegate` 키워드를 사용하여 특정 대리자 형식을 만드는 것을 확인했습니다.

추상 대리자 클래스는 느슨한 결합 및 호출을 위한 인프라를 제공합니다. 대리자 개체에 대한 호출 목록에 추가되는 메서드에 대해 형식 안정성을 도입하고 적용하면 구체적인 대리자 형식이 훨씬 더 유용해집니다. `delegate` 키워드를 사용하고 구체적인 대리자 형식을 정의하면 컴파일러에서 해당 메서드를 생성합니다.

실제로 이렇게 하면 다른 메서드 시그니처가 필요할 때마다 새로운 대리자 형식이 생성됩니다. 일정 시간이 지나면 이 작업은 지루해질 수 있습니다. 모든 새 기능에는 새 대리자 형식이 필요합니다.

다행히도 반드시 필요하지는 않습니다. .NET Core 프레임워크에는 대리자 형식이 필요할 때마다 재사용할 수 있는 여러 가지 형식이 포함되어 있습니다. 이러한 형식은 [제네릭](#) 정의이므로 새 메서드 선언이 필요할 때 사용자 지정을 선언할 수 있습니다.

이러한 형식 중 첫 번째는 [Action](#) 형식이며 여러 가지 변형이 있습니다.

C#

```
public delegate void Action();
public delegate void Action<in T>(T arg);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
// Other variations removed for brevity.
```

제네릭 형식 인수에 대한 `in` 한정자는 공변성(Covariance)에 대한 문서에서 설명합니다.

`Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16>` 같이 최대 16개의 인수가 포함된 `Action` 대리자의 변형이 있습니다. 이러한 정의에서는 각 대리자 인수에 대해 서로 다른 제네릭 인수를 사용하여 최대한의 유연성을 제공합니다. 메서드 인수는 같은 형식일 필요가 없지만 같은 형식일 수 있습니다.

`void` 반환 형식을 갖는 대리자 형식에 대해 `Action` 형식 중 하나를 사용합니다.

또한 프레임워크에는 값을 반환하는 대리자 형식에 사용할 수 있는 여러 가지 제네릭 대리자 형식이 포함됩니다.

C#

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T1, out TResult>(T1 arg);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
// Other variations removed for brevity
```

결과 제네릭 형식 인수에 대한 `out` 한정자는 공변성(Covariance)에 대한 문서에서 설명합니다.

`Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>` 같이 최대 16개의 입력 인수가 포함된 `Func` 대리자의 변형이 있습니다. 규칙에 따라 결과의 형식은 모든 `Func` 선언에서 항상 마지막 형식 매개 변수입니다.

값을 반환하는 모든 대리자 형식에 대해 `Func` 형식 중 하나를 사용합니다.

또한 단일 값에 대한 테스트를 반환하는 대리자에 대해 특수화된 `Predicate<T>` 형식이 있습니다.

C#

```
public delegate bool Predicate<in T>(T obj);
```

모든 `Predicate` 형식에 대해 구조적으로 동일한 `Func` 형식이 있다는 것을 알 수 있습니다. 예를 들면 다음과 같습니다.

C#

```
Func<string, bool> TestForString;
Predicate<string> AnotherTestForString;
```

이러한 두 형식이 동일하다고 생각할 수 있습니다. 그러나 동일하지 않습니다. 이러한 두 변수는 서로 교환해서 사용할 수 없습니다. 한 형식의 변수에 다른 형식을 할당할 수 없습니다. C# 형식 시스템에서는 구조체가 아니라 정의된 형식의 이름을 사용합니다.

.NET Core 라이브러리의 이러한 모든 대리자 형식 정의는 대리자가 필요한 새 기능에 대해 새 대리자 형식을 정의할 필요가 없음을 의미해야 합니다. 이러한 제네릭 정의는 대부분의 상황에서 필요한 모든 대리자 형식을 제공해야 합니다. 필수 형식 매개 변수를 사용하여 이러한 형식 중 하나를 인스턴스화할 수 있습니다. 제네릭으로 만들 수 있는 알고리즘의 경우 이러한 대리자를 제네릭 형식으로 사용할 수 있습니다.

그러면 시간이 절약되고 대리자로 작업하기 위해 만들어야 하는 새로운 형식 수가 최소화됩니다.

다음 문서에서는 실제로 대리자로 작업하기 위한 몇 가지 일반적인 패턴이 표시됩니다.

다음

# 대리자에 대한 일반적인 패턴

아티클 • 2023. 04. 07.

## 이전

대리자는 구성 요소 간 결합이 최소화된 소프트웨어 디자인을 사용하는 메커니즘을 제공합니다.

이러한 디자인의 가장 좋은 예는 LINQ입니다. LINQ 쿼리 식 패턴은 모든 기능에 대리자를 사용합니다. 간단한 다음 예제를 살펴보세요.

C#

```
var smallNumbers = numbers.Where(n => n < 10);
```

이 예제에서는 숫자 시퀀스를 값 10보다 작은 숫자로만 필터링합니다. `Where` 메서드는 필터를 통과하는 시퀀스의 요소를 결정하는 대리자를 사용합니다. LINQ 쿼리를 만들 때 이러한 특정 용도를 위해 대리자의 구현을 제공합니다.

`Where` 메서드의 프로토타입은 다음과 같습니다.

C#

```
public static IEnumerable<TSource> Where<TSource> (this IEnumerable<TSource> source, Func<TSource, bool> predicate);
```

이 예제에서는 LINQ의 일부인 모든 메서드가 반복됩니다. 이러한 메서드는 모두 특정 쿼리를 관리하는 코드에 대해 대리자를 사용합니다. 이 API 디자인 패턴은 간단하게 배우고 이해할 수 있습니다.

이 간단한 예제에서는 어떻게 대리자에 구성 요소 간 결합이 거의 필요하지 않은지를 보여 줍니다. 특정 기본 클래스에서 파생되는 클래스를 만들 필요가 없습니다. 특정 인터페이스를 구현하지 않아도 됩니다. 현재 작업에 기본적으로 필요한 하나의 메서드만 구현하면 됩니다.

## 대리자를 사용하여 고유한 구성 요소 빌드

대리자를 사용하는 디자인으로 구성 요소를 만들어 해당 예제에서 빌드해 보겠습니다.

큰 시스템의 로그 메시지에 사용할 수 있는 구성 요소를 정의해 보겠습니다. 라이브러리 구성 요소는 여러 가지 다른 플랫폼의 다양한 환경에서 사용할 수 있습니다. 로그를 관리하는 구성 요소에는 공통된 기능이 많이 있습니다. 시스템의 모든 구성 요소에서 전달된

메시지를 수락해야 합니다. 이러한 메시지는 핵심 구성 요소에서 관리할 수 있는 우선 순위가 다릅니다. 메시지는 최종 보관된 양식에 타임스탬프가 있어야 합니다. 고급 시나리오의 경우 소스 구성 요소에 따라 메시지를 필터링할 수 있습니다.

메시지가 기록되는 위치와 같이 자주 변경되는 기능이 있습니다. 일부 환경에서는 메시지가 오류 콘솔에 기록되고, 다른 환경에서는 파일에 기록될 수 있습니다. 데이터베이스 스토리지, OS 이벤트 로그, 기타 문서 스토리지 등에 기록될 수도 있습니다.

또한 다양한 시나리오에서 사용될 수 있는 출력의 조합이 있습니다. 메시지를 콘솔 및 파일에 기록하려고 할 수 있습니다.

대리자 기반 디자인은 뛰어난 유연성을 제공하며 나중에 추가할 수 있는 스토리지 메커니즘을 지원하기가 쉽습니다.

이 디자인에서 기본 로그 구성 요소는 비가상이며 sealed 클래스일 수도 있습니다. 모든 대리자 집합을 연결하여 다양한 스토리지 미디어에 메시지를 기록할 수 있습니다. 멀티캐스트 대리자에 대한 기본 제공 지원을 통해 메시지가 여러 위치(파일 및 콘솔)에 기록되어야 하는 시나리오를 쉽게 지원할 수 있습니다.

## 첫 번째 구현

작은 규모로 시작해 보겠습니다. 초기 구현에서는 새 메시지를 수락하고 연결된 대리자를 사용하여 메시지를 기록합니다. 메시지를 콘솔에 기록하는 대리자에서 시작할 수 있습니다.

```
C#  
  
public static class Logger  
{  
    public static Action<string>? WriteMessage;  
  
    public static void LogMessage(string msg)  
    {  
        if (WriteMessage is not null)  
            WriteMessage(msg);  
    }  
}
```

위의 정적 클래스는 사용할 수 있는 가장 간단한 클래스입니다. 메시지를 콘솔에 기록하는 메서드에 대한 단일 구현을 작성해야 합니다.

```
C#  
  
public static class LoggingMethods  
{  
    public static void LogToConsole(string message)
```

```
{  
    Console.Error.WriteLine(message);  
}  
}
```

마지막으로 로거에 선언된 WriteMessage 대리자에 대리자를 연결함으로써 대리자를 연결해야 합니다.

C#

```
Logger.WriteMessage += LoggingMethods.LogToConsole;
```

## 사례

지금까지 샘플은 매우 간단하지만 대리자 관련 디자인에 대한 몇 가지 중요한 지침을 보여 줍니다.

Core Framework에 정의된 대리자 형식을 사용하면 사용자가 대리자를 사용하기가 더 쉽습니다. 새 형식을 정의할 필요가 없으며, 라이브러리를 사용하는 개발자는 특수화된 새 대리자 형식을 배울 필요가 없습니다.

사용되는 인터페이스는 최대한 최소화되고 유연합니다. 새 출력 로거를 만들려면 메서드 하나를 만들어야 합니다. 이 메서드는 정적 메서드이거나 인스턴스 메서드일 수 있습니다. 모든 액세스 권한이 있을 수 있습니다.

## 출력 형식 지정

이 첫 번째 버전을 약간 더 강력하게 만든 후 다른 로깅 메커니즘을 만들어 보겠습니다.

다음으로 로그 클래스에서 보다 구조적인 메시지를 만들도록 LogMessage() 메서드에 몇 가지 인수를 추가해 보겠습니다.

C#

```
public enum Severity  
{  
    Verbose,  
    Trace,  
    Information,  
    Warning,  
    Error,  
    Critical  
}
```

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;

    public static void LogMessage(Severity s, string component, string msg)
    {
        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        if (WriteMessage is not null)
            WriteMessage(outputMsg);
    }
}
```

그런 다음 해당 `Severity` 인수를 사용하여 로그의 출력으로 전송되는 메시지를 필터링해 보겠습니다.

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;

    public static Severity LogLevel { get; set; } = Severity.Warning;

    public static void LogMessage(Severity s, string component, string msg)
    {
        if (s < LogLevel)
            return;

        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        if (WriteMessage is not null)
            WriteMessage(outputMsg);
    }
}
```

## 사례

로깅 인프라에 새 기능을 추가했습니다. 로거 구성 요소는 모든 출력 메커니즘에 매우 느슨하게 결합되므로 로거 대리자를 구현하는 코드에 영향을 주지 않고 이러한 새 기능을 추가할 수 있습니다.

계속 구축하면 이 느슨한 결합을 통해 다른 위치를 변경하지 않고 사이트의 일부를 업데이트할 때 유연성을 향상시킬 수 있는 방법에 대한 예제가 더 많이 표시됩니다. 실제로 더 큰 애플리케이션에서는 로거 출력 클래스가 다른 어셈블리에 있을 수 있으며 심지어 다시 작성할 필요도 없습니다.

## 두 번째 출력 엔진 구축

로그 구성 요소가 함께 제공됩니다. 메시지를 파일에 기록하는 출력 엔진을 하나 더 추가해 보겠습니다. 그러면 약간 더 복잡한 출력 엔진이 됩니다. 이 엔진은 파일 작업을 캡슐화하는 클래스가 되고 기록할 때마다 파일이 항상 닫히도록 합니다. 또한 각 메시지가 생성된 후 모든 데이터가 디스크로 플러시되도록 합니다.

파일 기반 로거입니다.

C#

```
public class FileLogger
{
    private readonly string logPath;
    public FileLogger(string path)
    {
        logPath = path;
        Logger.WriteMessage += LogMessage;
    }

    public void DetachLog() => Logger.WriteMessage -= LogMessage;
    // make sure this can't throw.
    private void LogMessage(string msg)
    {
        try
        {
            using (var log = File.AppendText(logPath))
            {
                log.WriteLine(msg);
                log.Flush();
            }
        }
        catch (Exception)
        {
            // Hmm. We caught an exception while
            // logging. We can't really log the
            // problem (since it's the log that's failing).
            // So, while normally, catching an exception
            // and doing nothing isn't wise, it's really the
            // only reasonable option here.
        }
    }
}
```

이 클래스를 만들었으면 인스턴스화하고 해당 LogMessage 메서드를 로거 구성 요소에 연결합니다.

C#

```
var file = new FileLogger("log.txt");
```

이러한 두 메서드는 함께 사용할 수 있습니다. 두 로그 메서드를 연결하고 메시지를 콘솔 및 파일에 생성할 수 있습니다.

C#

```
var fileOutput = new FileLogger("log.txt");
Logger.WriteMessage += LoggingMethods.LogToConsole; // LoggingMethods is the
static class we utilized earlier
```

나중에 시스템에 문제를 발생시키지 않고 동일한 애플리케이션에서도 대리자 중 하나를 제거할 수 있습니다.

C#

```
Logger.WriteMessage -= LoggingMethods.LogToConsole;
```

## 사례

이제 로깅 하위 시스템에 대한 두 번째 출력 처리기를 추가했습니다. 파일 시스템을 올바르게 지원하려면 조금 더 많은 인프라가 필요합니다. 대리자는 인스턴스 메서드입니다. 전용 메서드이기도 합니다. 대리자 인프라는 대리자를 연결할 수 있기 때문에 더 큰 액세스 가능성이 필요하지 않습니다.

두 번째로 대리자 기반 디자인에서는 코드를 추가하지 않고도 여러 출력 메서드를 사용합니다. 따라서 여러 출력 메서드를 지원하기 위해 추가 인프라를 구축하지 않아도 됩니다. 이러한 메서드는 호출 목록에서 또 다른 메서드가 됩니다.

파일 로깅 출력 메서드의 코드에 특별히 주의해야 합니다. 예외를 throw하지 않도록 이 코드를 코딩해야 합니다. 이는 항상 엄격한 요건은 아니지만 종종 좋은 사례가 됩니다. 대리자 메서드 중 하나가 예외를 throw하는 경우 호출 목록에 있는 나머지 대리자가 호출되지 않습니다.

마지막으로 파일 로거는 각 로그 메시지에서 파일을 열고 닫아 해당 리소스를 관리해야 합니다. 파일을 열어 두고 작업이 완료되면 `IDisposable`을 구현하여 파일을 닫도록 선택할 수 있습니다. 두 메서드에는 장점과 단점이 있습니다. 둘 다 클래스 간 결합을 약간 더 많이 만듭니다.

두 시나리오를 지원하기 위해 `Logger` 클래스의 코드를 업데이트할 필요가 없습니다.

## Null 대리자 처리

마지막으로 출력 메커니즘을 선택하지 않은 경우에 보다 강력하도록 LogMessage 메서드를 업데이트해 보겠습니다. WriteMessage 대리자에 연결된 호출 목록이 없는 경우 현재 구현에서는 NullReferenceException을 throw합니다. 메서드가 연결되지 않은 경우에도 자동으로 계속되는 디자인을 원할 수 있습니다. Delegate.Invoke() 메서드와 결합된 null 조건부 연산자를 사용하면 간단합니다.

C#

```
public static void LogMessage(string msg)
{
    WriteMessage?.Invoke(msg);
}
```

왼쪽 피연산자(이 경우 WriteMessage)가 null이면 null 조건부 연산자(?.)가 단락됩니다. 즉, 메시지를 기록하려고 시도하지 않는다는 의미입니다.

System.Delegate 또는 System.MulticastDelegate에 대한 설명서에 나열된 Invoke() 메서드를 찾을 수 없습니다. 컴파일러는 선언된 모든 대리자 형식에 대해 형식이 안전한 Invoke 메서드를 생성합니다. 따라서 이 예제에서 Invoke는 단일 string 인수를 사용하고 void 반환 형식을 갖습니다.

## 사례의 요약

다른 기록기 및 다른 기능으로 확장할 수 있는 로그 구성 요소의 시작 부분을 살펴보았습니다. 디자인에서 대리자를 사용하면 서로 다른 구성 요소가 느슨하게 결합됩니다. 그러면 여러 가지 장점을 제공합니다. 새 출력 메커니즘을 만들어 시스템에 연결하기가 쉽습니다. 이러한 다른 메커니즘에는 로그 메시지를 기록하는 메서드 하나만 있으면 됩니다. 이 디자인은 새 기능을 추가할 때 복원력이 있습니다. 모든 기록기에 필요한 계약은 하나의 메서드를 구현하는 것입니다. 해당 메서드는 정적 메서드이거나 인스턴스 메서드일 수 있습니다. 공용, 개인 또는 기타 법적 액세스할 수 있습니다.

로거 클래스는 새로운 변경 사항 없이 원하는 대로 기능 향상 또는 변경을 수행할 수 있습니다. 모든 클래스와 마찬가지로 새로운 변경 위험이 없으면 공용 API를 수정할 수 없습니다. 그러나 로거와 모든 출력 엔진 간 결합은 대리자를 통해서만 가능하므로 다른 형식 (예: 인터페이스 또는 기본 클래스)이 관련되지 않습니다. 결합은 최대한 작아야 합니다.

[다음](#)

# 이벤트 소개

아티클 • 2023. 04. 08.

## 이전

대리자와 같은 이벤트는 런타임에 바인딩 메커니즘입니다. 실제로 이벤트는 대리자에 대한 언어 지원을 기반으로 작성됩니다.

이벤트는 어떠한 문제가 발생했다는 사실을 개체가 시스템의 모든 관련 구성 요소에 브로드캐스트하는 방법입니다. 다른 모든 구성 요소는 이벤트를 구독하고 이벤트가 발생할 때 알림을 받을 수 있습니다.

일부 프로그래밍에서 이벤트를 사용했을 수 있습니다. 많은 그래픽 시스템에는 사용자 조작을 보고하는 이벤트 모델이 있습니다. 이러한 이벤트는 마우스 이동, 단추 누름 등의 조작을 보고합니다. 가장 일반적인 시나리오이기는 하지만 이벤트가 사용되는 유일한 시나리오는 아닙니다.

클래스에 대해 발생해야 하는 이벤트를 정의할 수 있습니다. 이벤트로 작업할 때 한 가지 중요한 고려 사항은 특정 이벤트에 대해 등록된 개체가 없을 수 있다는 점입니다. 구성된 수신기가 없는 경우 이벤트를 발생시키지 않도록 코드를 작성해야 합니다.

이벤트를 구독하면 두 개체(이벤트 소스와 이벤트 싱크) 간 결합도 생성됩니다. 더 이상 이벤트에 관심이 없는 경우 이벤트 싱크가 이벤트 소스를 구독 취소하도록 해야 합니다.

## 이벤트 지원의 디자인 목표

이벤트에 대한 언어 디자인은 이러한 목표를 대상으로 합니다.

- 이벤트 소스와 이벤트 싱크 간에 최소 결합을 사용하도록 설정합니다. 이러한 두 구성 요소는 동일한 조직에서 작성할 수 없으며 완전히 다른 일정으로 업데이트할 수도 있습니다.
- 이벤트를 구독하고 동일한 이벤트를 구독 취소하는 작업은 매우 간단해야 합니다.
- 이벤트 소스는 여러 이벤트 구독자를 지원해야 합니다. 또한 연결된 이벤트 구독자가 없는 경우를 지원해야 합니다.

이벤트의 목표가 대리자의 목표와 매우 유사하다는 것을 확인할 수 있습니다. 따라서 이벤트 언어 지원은 대리자 언어 지원을 기반으로 합니다.

## 이벤트의 언어 지원

이벤트를 정의하고 이벤트를 구독 또는 구독 취소하는 구문은 대리자에 대한 구문의 확장입니다.

`event` 키워드를 사용하는 이벤트를 정의하려면

C#

```
public event EventHandler<FileListArgs> Progress;
```

이벤트(이 예제의 경우 `EventHandler<FileListArgs>`)의 형식은 대리자 형식이어야 합니다. 이벤트를 선언할 때 따라야 할 여러 가지 규칙이 있습니다. 일반적으로 이벤트 대리자 형식에는 `void` 반환이 있습니다. 이벤트 선언은 동사 또는 동사 구여야 합니다. 이벤트가 발생한 문제를 보고할 때는 과거 시제를 사용합니다. 발생하려고 하는 어떤 문제를 보고 하려면 현재 시제 동사(예: `closing`)를 사용합니다. 종종 현재 시제를 사용하여 클래스가 몇 가지 사용자 지정 동작을 지원함을 나타내기도 합니다. 가장 일반적인 시나리오 중 하나는 취소를 지원하는 것입니다. 예를 들어 `Closing` 이벤트에는 닫기 작업이 계속되어야 하는지 여부를 나타내는 인수가 포함될 수 있습니다. 다른 시나리오를 통해 호출자는 이벤트 인수의 속성을 업데이트하여 동작을 수정할 수 있습니다. 알고리즘에서 수행하도록 제안된 다음 작업을 나타내는 이벤트를 발생시킬 수 있습니다. 이벤트 처리기는 이벤트 인수의 속성을 수정하여 다른 작업을 위임할 수 있습니다.

이벤트를 발생시키려면 대리자 호출 구문을 사용하여 이벤트 처리기를 호출합니다.

C#

```
Progress?.Invoke(this, new FileListArgs(file));
```

대리자에 대한 섹션에서 설명한 대로 `?.` 연산자를 사용하면 해당 이벤트에 대한 구독자가 없을 때 이벤트를 발생시키지 않도록 하기가 쉽습니다.

`+=` 연산자를 사용하여 이벤트를 구독합니다.

C#

```
EventHandler<FileListArgs> onProgress = (sender, eventArgs) =>
    Console.WriteLine(eventArgs.FoundFile);

fileLister.Progress += onProgress;
```

위에 표시된 대로 처리기 메서드는 일반적으로 접두사 'On'과 그다음에 오는 이벤트 이름입니다.

`-=` 연산자를 사용하여 구독 취소합니다.

C#

```
fileLister.Progress -= onProgress;
```

이벤트 처리기를 나타내는 식에 대해 지역 변수를 선언하는 것이 중요합니다. 따라서 구독 취소하면 처리기가 제거됩니다. 대신 람다 식의 본문을 사용한 경우 연결되지 않아 아무 작업도 수행하지 않는 처리기를 제거하려고 합니다.

다음 문서에서는 일반적인 이벤트 패턴 및 이 예제의 다양한 변형에 대해 자세히 알아봅니다.

[다음](#)

# 표준 .NET 이벤트 패턴

아티클 • 2023. 04. 08.

## 이전

.NET 이벤트는 일반적으로 몇 가지 알려진 패턴을 따릅니다. 이러한 패턴의 표준화는 개발자가 해당 표준 패턴에 대한 지식을 활용하여 모든 .NET 이벤트 프로그램에 적용할 수 있다는 의미입니다.

표준 이벤트 소스를 만들고 코드에서 표준 이벤트를 구독 및 처리하는 데 필요한 모든 정보를 얻을 수 있도록 이러한 표준 패턴을 살펴보겠습니다.

## 이벤트 대리자 시그니처

.NET 이벤트 대리자에 대한 표준 시그니처는 다음과 같습니다.

C#

```
void EventRaised(object sender, EventArgs args);
```

반환 형식은 void입니다. 이벤트는 대리자를 기반으로 하며 멀티캐스트 대리자입니다. 또한 모든 이벤트 소스에 대해 여러 구독자를 지원합니다. 메서드의 단일 반환 값은 여러 이벤트 구독자로 확장되지 않습니다. 이벤트 발생 후 이벤트 소스에 표시되는 반환 값은 무엇인가요? 이 문서의 뒷부분에서 이벤트 소스에 정보를 보고하는 이벤트 구독자를 지원하는 이벤트 프로토콜을 만드는 방법에 대해 살펴보겠습니다.

인수 목록에는 보낸 사람과 이벤트 인수의 두 인수가 포함됩니다. `sender`의 컴파일 시간 형식은 `System.Object`이지만 항상 올바르면서도 더 많이 파생된 형식을 알고 있을 수도 있습니다. 규칙에 따라 `object`를 사용합니다.

두 번째 인수는 일반적으로 `System.EventArgs`에서 파생된 형식이었습니다. ([다음 섹션](#)에서 이 규칙이 더 이상 적용되지 않는 것을 확인할 수 있습니다.) 이벤트 유형에 추가 인수가 필요하지 않은 경우에도 두 인수를 모두 제공합니다. 이벤트에 추가 정보가 포함되어 있지 않음을 나타낼 때 사용해야 하는 특수 값 `EventArgs.Empty`가 있습니다.

디렉터리 또는 패턴을 따르는 모든 하위 디렉터리의 파일을 나열하는 클래스를 만들어 보겠습니다. 이 구성 요소는 검색된 각 파일에 대해 패턴과 일치하는 이벤트를 발생시킵니다.

이벤트 모델을 사용하면 몇 가지 디자인 장점이 있습니다. 검색된 파일을 찾으면 다른 작업을 수행하는 여러 이벤트 수신기를 만들 수 있습니다. 서로 다른 수신기를 결합하면 더

강력한 알고리즘을 만들 수 있습니다.

검색된 파일을 찾기 위한 초기 이벤트 인수 선언은 다음과 같습니다.

C#

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }

    public FileFoundArgs(string fileName) => FoundFile = fileName;
}
```

이 형식은 작은 데이터 전용 형식처럼 보이지만 규칙을 따르고 참조(`class`) 형식으로 만들어야 합니다. 즉, 인수 개체가 참조로 전달되며 모든 구독자가 데이터에 대한 모든 업데이트를 볼 수 있습니다. 첫 번째 버전은 변경할 수 없는 개체입니다. 이벤트 인수 형식에서 속성을 변경할 수 없도록 하는 것이 좋습니다. 이런 방식으로 다른 구독자에게 값이 표시되기 전에는 한 구독자가 값을 변경할 수 없습니다. 여기에는 아래와 같은 예외가 있습니다.

다음으로 FileSearcher 클래스에서 이벤트 선언을 만들어야 합니다. `EventHandler<T>` 형식을 활용하면 또 다른 형식 정의를 만들 필요가 없습니다. 제네릭 특수화를 사용하면 됩니다.

FileSearcher 클래스를 입력하여 패턴과 일치하는 파일을 검색하고 일치하는 항목이 검색되면 올바른 이벤트를 발생시켜 보겠습니다.

C#

```
public class FileSearcher
{
    public event EventHandler<FileFoundArgs>? FileFound;

    public void Search(string directory, string searchPattern)
    {
        foreach (var file in Directory.EnumerateFiles(directory,
searchPattern))
        {
            RaiseFileFound(file);
        }
    }

    private void RaiseFileFound(string file) =>
        FileFound?.Invoke(this, new FileFoundArgs(file));
}
```

# 필드와 유사한 이벤트 정의 및 발생

이벤트를 클래스에 추가하는 가장 간단한 방법은 이전 예제와 같이 해당 이벤트를 public 필드로 선언하는 것입니다.

C#

```
public event EventHandler<FileEventArgs>? FileFound;
```

이 예제는 public 필드를 선언하는 것처럼 보이며, 잘못된 개체 지향 사례인 것 같습니다. 속성 또는 메서드를 통해 데이터 액세스를 보호하려고 합니다. 이렇게 하면 잘못된 사례처럼 보일 수 있지만 안전한 방식으로 이벤트 개체에만 액세스할 수 있도록 컴파일러에 의해 생성된 코드에서 래퍼를 만듭니다. 필드와 유사한 이벤트에서 사용 가능한 유일한 작업은 처리기 추가입니다.

C#

```
var fileLister = new FileSearcher();
int filesFound = 0;

EventHandler<FileEventArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    filesFound++;
};

fileLister.FileFound += onFileFound;
```

또한 처리기 제거도 가능합니다.

C#

```
fileLister.FileFound -= onFileFound;
```

처리기에 대한 지역 변수가 있어야 합니다. 람다 식의 본문을 사용한 경우 제거가 올바르게 작동하지 않습니다. 대리자의 다른 인스턴스가 되고 자동으로 아무 작업도 수행하지 않습니다.

클래스 외부의 코드는 이벤트를 발생시킬 수 없으며 다른 작업도 수행할 수 없습니다.

## 이벤트 구독자에서 값 반환

간단한 버전은 제대로 작동하고 있습니다. 이제 또 다른 기능인 취소를 추가해 보겠습니다.

찾은 이벤트를 발생시킬 때 이 파일이 마지막으로 검색된 파일인 경우 수신기에서 추가 처리를 중지할 수 있어야 합니다.

이벤트 처리기는 값을 반환하지 않으므로 다른 방식으로 값을 전달해야 합니다. 표준 이벤트 패턴은 `EventArgs` 개체를 사용하여 이벤트 구독자가 취소를 전달하는 데 사용할 수 있는 필드를 포함합니다.

취소 계약의 의미 체계에 따라 두 가지 다른 패턴을 사용할 수 있습니다. 두 경우 모두 찾은 파일 이벤트에 대한 `EventArgs`에 부울 필드를 추가합니다.

한 가지 패턴에서는 임의의 구독자 한 명이 작업을 취소할 수 있습니다. 이 패턴에서는 새 필드가 `false`로 초기화됩니다. 임의의 구독자가 이 값을 `true`로 변경할 수 있습니다. 모든 구독자가 발생된 이벤트를 확인하면 FileSearcher 구성 요소에서 부울 값을 검사하고 작업을 수행합니다.

두 번째 패턴에서는 모든 구독자가 작업 취소를 원하는 경우에만 작업을 취소합니다. 이 패턴에서는 작업이 취소되어야 함을 나타내도록 새 필드가 초기화되고 임의의 구독자는 작업이 계속되어야 함을 나타내도록 이 필드를 변경할 수 있습니다. 모든 구독자가 발생된 이벤트를 확인하면 FileSearcher 구성 요소에서 부울 값을 검사하고 작업을 수행합니다. 이 패턴에는 하나의 추가 단계가 있습니다. 구독자가 이벤트를 확인했는지 여부를 구성 요소에서 알고 있어야 합니다. 구독자가 없으면 필드는 취소를 잘못 나타내게 됩니다.

이 샘플에 대한 첫 번째 버전을 구현해 보겠습니다. `CancelRequested`라는 부울 필드를 `FileFoundArgs` 형식에 추가해야 합니다.

C#

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }
    public bool CancelRequested { get; set; }

    public FileFoundArgs(string fileName) => FoundFile = fileName;
}
```

이 새 필드는 자동으로 `Boolean` 필드의 기본값인 `false`로 초기화되므로 실수로 취소될 가능성이 없습니다. 구성 요소에서 유일한 다른 변경 사항은 이벤트를 발생시킨 후 플래그를 확인하여 구독자가 취소를 요청했는지를 확인하는 것입니다.

C#

```
private void SearchDirectory(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        FileFoundArgs args = RaiseFileFound(file);
```

```

        if (args.CancelRequested)
    {
        break;
    }
}

private FileFoundArgs RaiseFileFound(string file)
{
    var args = new FileFoundArgs(file);
    FileFound?.Invoke(this, args);
    return args;
}

```

이 패턴의 한 가지 장점은 새로운 변경 사항이 아니라는 점입니다. 이전에 취소를 요청한 구독자가 없으며 여전히 요청하지 않습니다. 새로운 취소 프로토콜을 지원하려는 경우가 아니라면 구독자 코드를 업데이트할 필요가 없습니다. 이 경우 매우 느슨하게 결합되어 있습니다.

첫 번째 실행 파일을 찾으면 취소를 요청하도록 구독자를 업데이트해 보겠습니다.

C#

```

EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    eventArgs.CancelRequested = true;
};

```

## 다른 이벤트 선언 추가

기능을 하나 더 추가하고 이벤트에 대한 다른 언어 관용구를 보여 드리겠습니다. 파일 검색에서 모든 하위 디렉터리를 트래버스하는 `Search` 메서드의 오버로드를 추가해 보겠습니다.

하위 디렉터리가 많은 디렉터리에서 이 작업은 시간이 오래 걸릴 수 있습니다. 각각의 새 디렉터리 검색이 시작될 때 발생되는 이벤트를 추가해 보겠습니다. 이렇게 하면 구독자가 진행률을 추적하고 진행률에 대해 사용자에게 업데이트할 수 있습니다. 지금까지 만든 모든 샘플은 `public`입니다. 이제 내부 이벤트로 만들어 보겠습니다. 즉, 인수에 사용된 형식을 내부 형식으로 설정할 수도 있습니다.

먼저 새 디렉터리 및 진행률을 보고하는 새 `EventArgs` 파생 클래스를 만듭니다.

C#

```
internal class SearchDirectoryArgs : EventArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int completedDirs)
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

다시 권장 사항에 따라 이벤트 인수에 대해 변경할 수 없는 참조 형식을 만들 수 있습니다.

다음으로 이벤트를 정의합니다. 이번에는 다른 구문을 사용합니다. 필드 구문을 사용할 뿐만 아니라 추가 및 제거 처리기와 함께 속성을 명시적으로 만들 수도 있습니다. 이 샘플에서 해당 처리기에는 추가 코드가 필요하지 않지만 여기서는 만드는 방법을 보여 줍니다.

C#

```
internal event EventHandler<SearchDirectoryArgs> DirectoryChanged
{
    add { _directoryChanged += value; }
    remove { _directoryChanged -= value; }
}
private EventHandler<SearchDirectoryArgs>? _directoryChanged;
```

여러 측면에서, 여기에서 작성하는 코드는 앞에서 살펴본 필드 이벤트 정의에 대해 컴파일러에서 생성하는 코드를 미러링합니다. 속성에 사용된 것과 매우 유사한 구문을 사용하여 이벤트를 만듭니다. 처리기의 이름은 `add`와 `remove`로 다릅니다. 이러한 처리기를 호출하여 이벤트를 구독하거나 이벤트에서 구독을 취소합니다. 또한 이벤트 변수를 저장하려면 `private` 지원 필드를 선언해야 합니다. 이 필드는 `null`로 초기화됩니다.

다음으로 하위 딕터리를 트래버스하고 두 이벤트를 발생시키는 `Search` 메서드의 오버로드를 추가해 보겠습니다. 이 작업을 수행하는 가장 쉬운 방법은 기본 인수를 사용하여 모든 딕터리를 검색하도록 지정하는 것입니다.

C#

```
public void Search(string directory, string searchPattern, bool searchSubDirs = false)
{
```

```

    if (searchSubDirs)
    {
        var allDirectories = Directory.GetDirectories(directory, "*.*",
SearchOption.AllDirectories);
        var completedDirs = 0;
        var totalDirs = allDirectories.Length + 1;
        foreach (var dir in allDirectories)
        {
            RaiseSearchDirectoryChanged(dir, totalDirs, completedDirs++);
            // Search 'dir' and its subdirectories for files that match the
search pattern:
            SearchDirectory(dir, searchPattern);
        }
        // Include the Current Directory:
        RaiseSearchDirectoryChanged(directory, totalDirs, completedDirs++);

        SearchDirectory(directory, searchPattern);
    }
    else
    {
        SearchDirectory(directory, searchPattern);
    }
}

private void SearchDirectory(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        FileFoundArgs args = RaiseFileFound(file);
        if (args.CancelRequested)
        {
            break;
        }
    }
}

private void RaiseSearchDirectoryChanged(
    string directory, int totalDirs, int completedDirs) =>
    _directoryChanged?.Invoke(
        this,
        new SearchDirectoryArgs(directory, totalDirs, completedDirs));

private FileFoundArgs RaiseFileFound(string file)
{
    var args = new FileFoundArgs(file);
    FileFound?.Invoke(this, args);
    return args;
}

```

이제 모든 하위 디렉터리를 검색하기 위해 오버로드를 호출하는 애플리케이션을 실행할 수 있습니다. 새 `DirectoryChanged` 이벤트에는 구독자가 없지만 `??.Invoke()` 관용구를 사용하여 이 작업이 제대로 작동하도록 합니다.

콘솔 창에 진행률을 표시하는 줄을 작성하는 처리기를 추가해 보겠습니다.

C#

```
fileLister.DirectoryChanged += (sender, eventArgs) =>
{
    Console.Write($"Entering '{eventArgs.CurrentSearchDirectory}'.");
    Console.WriteLine($" {eventArgs.CompletedDirs} of {eventArgs.TotalDirs}
completed...");
};
```

.NET 에코시스템 전체에 적용되는 패턴을 살펴보았습니다. 이러한 패턴 및 규칙을 학습하면 자연스러운 C# 및 .NET을 신속하게 작성할 수 있습니다.

## 추가 정보

- 이벤트 소개
- 이벤트 디자인
- 이벤트 처리 및 발생

다음으로 .NET의 최신 릴리스에서 이러한 패턴의 일부 변경 내용을 확인합니다.

다음

# 업데이트된 .NET Core 이벤트 패턴

아티클 • 2023. 04. 08.

## 이전

이전 문서에서는 가장 일반적인 이벤트 패턴을 설명했습니다. .NET Core에는 보다 완화된 패턴이 있습니다. 이 버전에서는 `EventHandler<TEventArgs>` 정의에 `TEventArgs`가 `System.EventArgs`에서 파생된 클래스여야 한다는 제약 조건이 더 이상 없습니다.

이 때문에 유연성이 증가하고 이전 버전과 호환됩니다. 유연성부터 살펴보겠습니다. `System.EventArgs` 클래스는 개체의 부분 복사본을 만드는 `MemberwiseClone()` 메서드 하나를 소개합니다. 이 메서드는 `EventArgs`에서 파생된 클래스에 대해 해당 기능을 구현하기 위해 리플렉션을 사용해야 합니다. 특정 파생 클래스에서는 해당 기능을 더 쉽게 만들 수 있습니다. 이는 `System.EventArgs`에서 파생되는 것이 디자인을 제한하는 제약 조건이지만 추가적인 혜택은 없음을 의미합니다. 실제로 `EventArgs`에서 파생되지 않도록 `FileEventArgs` 및 `SearchDirectoryEventArgs`의 정의를 변경할 수 있습니다. 프로그램은 동일하게 작동합니다.

한 가지 더 변경하는 경우 `SearchDirectoryEventArgs`를 구조체로 변경할 수 있습니다.

C#

```
internal struct SearchDirectoryArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int completedDirs) : this()
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

추가 변경은 모든 필드를 초기화하는 생성자를 입력하기 전에 매개 변수 없는 생성자를 호출하는 것입니다. 해당 코드를 추가하지 않으면 C#의 규칙에서 속성이 할당되기 전에 액세스된다고 보고합니다.

`FileEventArgs`를 클래스(참조 형식)에서 구조체(값 형식)로 변경하면 안 됩니다. 이는 취소를 처리하기 위한 프로토콜에서 이벤트 인수가 참조로 전달되도록 요구하기 때문입니다. 동일한 변경을 수행하면 파일 검색 클래스가 이벤트 구독자의 변경 내용을 관찰할 수

없습니다. 구조체의 새 복사본이 각 구독자에 사용되며, 해당 복사본은 파일 검색 개체에 표시되는 것과는 다른 복사본입니다.

다음으로, 이러한 변경 내용이 이전 버전과 호환될 수 방법을 살펴보겠습니다. 제약 조건을 제거해도 기존 코드에는 영향을 주지 않습니다. 기존 이벤트 인수 형식은 여전히 `System.EventArgs`에서 파생됩니다. 이전 버전과의 호환성은 `System.EventArgs`에서 계속 파생되는 한 가지 주요 이유입니다. 기존 이벤트 구독자는 클래식 패턴을 따르는 이벤트의 구독자가 됩니다.

유사한 논리에 따라 이제 생성되는 이벤트 인수 형식은 기존 코드베이스에 구독자가 없습니다. `System.EventArgs`에서 파생되지 않는 새 이벤트 유형은 이러한 코드베이스를 중단하지 않습니다.

## 비동기 구독자가 포함된 이벤트

알아볼 한 가지 최종 패턴은 비동기 코드를 호출하는 이벤트 구독자를 올바르게 작성하는 방법입니다. 이 과제는 `async` 및 `await`에 대한 문서에서 설명합니다. 비동기 메서드의 반환 형식이 `void`일 수도 있지만 권장되지는 않습니다. 이벤트 구독자 코드에서 비동기 메서드를 호출하는 경우 `async void` 메서드를 만들 수밖에 없습니다. 이벤트 처리기 시그니처에 이 메서드가 필요합니다.

이 반대 지침을 조정해야 합니다. 어떻게 해서든 안전한 `async void` 메서드를 만들어야 합니다. 구현해야 하는 패턴의 기본 사항은 아래와 같습니다.

C#

```
worker.StartWorking += async (sender, eventArgs) =>
{
    try
    {
        await DoWorkAsync();
    }
    catch (Exception e)
    {
        //Some form of logging.
        Console.WriteLine($"Async task failure: {e.ToString()}");
        // Consider gracefully, and quickly exiting.
    }
};
```

첫째, 처리기는 비동기 처리기로 표시됩니다. 이벤트 처리기 대리자 형식에 할당되므로 `void` 반환 형식을 갖습니다. 즉, 처리기에 표시된 패턴을 따르고 비동기 처리기의 컨텍스트 외부에서 예외가 `throw`되지 않도록 해야 합니다. 작업을 반환하지 않으므로 오류 상태를 입력하여 오류를 보고할 수 있는 작업이 없습니다. 메서드가 비동기이므로 메서드에서

단순히 예외를 throw할 수 없습니다. 호출하는 메서드는 `async` 이므로 실행을 계속합니다. 실제 런타임 동작은 환경에 따라 다르게 정의됩니다. 스레드 또는 스레드를 소유한 프로세스를 종료하거나, 프로세스를 확정되지 않은 상태로 둘 수 있습니다. 이러한 모든 잠재적 결과는 바람직하지 않습니다.

이 때문에 비동기 작업에 대한 `await` 문을 고유한 `try` 블록에 래핑해야 합니다. 오류 작업이 발생하는 경우 오류를 기록할 수 있습니다. 애플리케이션이 복구할 수 없는 오류인 경우 빠르고 정상적으로 프로그램을 종료할 수 있습니다.

이러한 기능은 .NET 이벤트 패턴에 대한 주요 업데이트입니다. 작업할 라이브러리에 이전 버전의 예제가 많이 표시됩니다. 그러나 최신 패턴이 무엇인지도 이해해야 합니다.

이 시리즈의 다음 문서는 디자인에 `delegates` 를 사용하는 경우와 `events` 를 사용하는 경우를 구분하는 데 도움이 됩니다. 비슷한 개념이며 해당 문서를 통해 프로그램에 대한 최상의 결정을 내릴 수 있습니다.

[다음](#)

# 대리자 및 이벤트를 구별

아티클 • 2024. 02. 13.

## 이전

.NET Core 플랫폼을 처음 사용하는 개발자는 `delegates` 기반 디자인과 `events` 기반 디자인 중에서 결정할 때 종종 어려움을 겪습니다. 두 언어 기능이 유사하므로 대리자 또는 이벤트를 선택하는 것이 어렵습니다. 이벤트는 대리자에 대한 언어 지원을 사용하여 작성되기도 합니다.

둘 다 런타임 바인딩 시나리오를 제공합니다. 즉, 런타임에만 알려지는 메서드를 호출하여 구성 요소가 통신하는 시나리오를 지원합니다. 모두 단일 및 다중 구독자 메서드를 지원하는데, 이를 단일 캐스트 및 멀티캐스트 지원이라고 할 수 있습니다. 둘 다 처리기 추가 및 제거에 대해 유사한 구문을 지원합니다. 마지막으로 이벤트를 발생시키고 대리자를 호출하는 작업에서 정확히 동일한 메서드 호출 구문을 사용합니다. 또한 `?.` 연산자와 함께 사용하도록 동일한 `Invoke()` 메서드 구문을 지원합니다.

이러한 모든 유사성으로 인해 언제 어떤 언어 기능을 사용할지를 결정하기가 어려울 수 있습니다.

## 이벤트 수신은 선택 사항임

사용할 언어 기능을 결정할 때 가장 중요하게 고려할 사항은 연결된 구독자가 있어야 하는지 여부입니다. 코드에서 구독자가 제공하는 코드를 호출해야 하는 경우에는 콜백을 구현해야 할 때 대리자 기반 디자인을 사용해야 합니다. 코드에서 구독자를 호출하지 않고 모든 작업을 완료할 수 있는 경우에는 이벤트 기반 디자인을 사용해야 합니다.

이 섹션 중 작성된 예제를 살펴보겠습니다. `List.Sort()`를 사용하여 작성한 코드에서 요소를 제대로 정렬하려면 비교자 함수가 제공되어야 합니다. 반환할 요소를 결정하려면 대리자와 함께 LINQ 쿼리를 제공해야 합니다. 둘 다 대리자로 작성된 디자인을 사용했습니다.

`Progress` 이벤트를 살펴보겠습니다. 이 이벤트는 작업의 진행률을 보고합니다. 수신기가 있는지 여부에 관계없이 작업이 계속 진행됩니다. `FileSearcher`는 또 다른 예제입니다. 연결된 이벤트 구독자가 없는 경우에도 검색된 모든 파일을 계속 검색하고 찾습니다. UX 컨트롤은 이벤트를 수신하는 구독자가 없는 경우에도 여전히 올바르게 작동합니다. 둘 다 이벤트 기반 디자인을 사용합니다.

## 반환 값에 대리자 필요

또 다른 고려 사항은 대리자 메서드에 필요한 메서드 프로토타입입니다. 지금까지 살펴본 대로 이벤트에 사용된 대리자는 모두 void 반환 형식을 갖습니다. 또한 이벤트 인수 개체의 속성을 수정하여 이벤트 소스에 다시 정보를 전달하는 이벤트 처리기를 만드는 관용 구가 있음을 확인했습니다. 이러한 관용구도 작업을 수행하기는 하지만 메서드에서 값을 반환하는 것만큼 자연스럽지 않습니다.

이러한 두 추론은 종종 둘 다 제공될 수 있습니다. 대리자 메서드가 값을 반환하는 경우 어떤 방식으로든 알고리즘에 영향을 줄 가능성이 있습니다.

## 이벤트에 프라이빗 호출이 있음

이벤트가 포함된 클래스가 아닌 다른 클래스는 이벤트 수신기를 추가하고 제거할 수만 있습니다. 이벤트가 포함된 클래스만 이벤트를 호출할 수 있습니다. 이벤트는 일반적으로 공용 클래스 멤버입니다. 반면 대리자는 종종 매개 변수로 전달되고 프라이빗 클래스 멤버로 저장됩니다(저장되는 경우).

## 종종 이벤트 수신기의 수명이 길어짐

이 이벤트 수신기는 수명이 길수록 근거가 약간 약해집니다. 그러나 이벤트 소스가 오랜 시간 동안 이벤트를 발생시킬 경우에는 이벤트 기반 디자인이 더 자연스러울 수 있습니다. 많은 시스템에서 UX 컨트롤의 이벤트 기반 디자인 예제를 확인할 수 있습니다. 이벤트를 구독하면 이벤트 소스가 프로그램의 수명 주기 전체에 걸쳐 이벤트를 발생시킬 수 있습니다. 이벤트가 더 이상 필요하지 않은 경우 이벤트 구독을 취소할 수 있습니다.

대리자가 메서드의 인수로 사용되고 해당 메서드가 반환된 후에는 대리자가 사용되지 않는 많은 대리자 기반 디자인과 비교해 보세요.

## 신중하게 평가

위의 고려 사항은 엄격한 규칙이 아닙니다. 대신 특정 용도에 가장 적합한 선택 항목을 결정하는 데 도움이 되는 지침을 나타냅니다. 유사하기 때문에 둘 다를 프로토타입화할 수도 있고 작업에 더 자연스러운 항목을 고려할 수 있습니다. 둘 다 런타임에 바인딩 시나리오도 처리합니다. 최상의 디자인을 전달하는 기능을 사용하세요.

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할

### .NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

수도 있습니다. 자세한 내용은  
참여자 가이드를 참조하세요.

☞ 설명서 문제 열기

☞ 제품 사용자 의견 제공

# C#으로 버전 관리

아티클 • 2023. 04. 08.

이 자습서에서는 .NET에서 버전 관리가 어떤 의미인지에 대해 배웁니다. 또한 라이브러리의 버전을 관리할 때 및 라이브러리의 새 버전으로 업그레이드할 때 고려해야 할 요소에 대해 배웁니다.

## 라이브러리 작성

공용 .NET 라이브러리를 만든 개발자라면 새로운 업데이트를 출시해야 하는 상황을 겪은 적 있을 것입니다. 이 프로세스의 진행 방법은 기존 코드를 새 버전의 라이브러리로 원활하게 전환해야 하는 경우 매우 중요합니다. 새 릴리스를 만들 때 다음과 같은 몇 가지 사항을 고려해야 합니다.

## 유의적 버전

유의적 버전 [\(줄여서 SemVer\)](#)은 특정 중요 시점 이벤트를 나타내기 위해 라이브러리의 버전에 적용되는 명명 규칙입니다. 라이브러리에 제공하는 버전 정보를 이용해 개발자가 동일한 라이브러리의 이전 버전을 사용하는 프로젝트와의 호환성을 확인하는 것이 가장 바람직합니다.

SemVer에 대한 가장 기본적인 접근법은 3개 구성 요소 형식 `MAJOR.MINOR.PATCH`입니다. 여기서:

- `MAJOR`은 호환되지 않는 API 변경 사항이 있을 때 증가합니다.
- `MINOR`는 이전 버전과 호환성 방식으로 기능을 추가할 때 증가합니다.
- `PATCH`는 이전 버전과 호환성 버그 수정을 만들 때 증가합니다.

.NET 라이브러리에 버전 정보를 적용할 때 시험판 버전과 같은 다른 시나리오를 지정하는 방법도 있습니다.

## 이전 버전과의 호환성

라이브러리의 새 버전을 릴리스하면 이전 버전과의 호환성이 주요 관심사 중 하나가 될 것입니다. 이전 버전에 종속된 코드가 다시 컴파일할 때 새 버전과 작동하는 경우, 라이브러리의 새 버전은 이전 버전과 소스가 호환됩니다. 이전 버전을 사용하는 애플리케이션이 다시 컴파일하지 않고도 새 버전과 작동하는 경우 라이브러리의 새 버전은 이진 호환됩니다.

라이브러리 이전 버전과의 호환성을 유지하고자 할 경우 다음 사항을 고려해야 합니다.

- 가상 메서드: 새 버전에서 가상 메서드를 가상이 아닌 상태로 만들 경우, 해당 메서드를 재정의하는 프로젝트를 업데이트해야 합니다. 여기에는 엄청난 변화가 따르므로 권장하지 않습니다.
- 메서드 시그니처: 메서드 동작 업데이트 시 시그니처도 변경해야 하는 경우, 해당 메서드에 대한 코드 호출이 계속 작동하도록 오버로드를 대신 만들어야 합니다. 구현의 일관성이 유지되도록 항상 이전 메서드 시그니처를 조작하여 새 메서드 시그니처를 호출할 수 있습니다.
- **Obsolete 특성**: 사용되지 않는 클래스 또는 클래스 멤버를 지정하고 이후 버전에서 제거되도록 하려면 코드에 이 특성을 사용할 수 있습니다. 이렇게 하면 라이브러리를 활용하는 개발자가 큰 변화에 더 잘 대비할 수 있습니다.
- 선택적 메서드 인수: 전에는 선택 사항이었던 메서드 인수를 필수로 만들거나 기본 값을 변경하는 경우, 해당 인수를 제공하지 않는 모든 코드를 업데이트해야 합니다.

### ① 참고

필수 인수를 선택 사항으로 만드는 것은 특히 메서드의 동작을 변경하지 않을 경우 거의 영향을 미치지 않습니다.

라이브러리의 새 버전으로 업그레이드하는 방법이 쉬울수록 사용자가 더 빨리 업그레이드할 가능성이 높아집니다.

## 애플리케이션 구성 파일

.NET 개발자는 대부분의 프로젝트 형식에서 [app.config 파일](#)을 발견할 가능성이 매우 높습니다. 이 간단한 구성 파일은 새로운 업데이트 출시를 개선하는 데 큰 도움이 될 수 있습니다. 일반적으로 라이브러리를 설계할 때 정기적으로 변경될 가능성이 있는 정보를 `app.config` 파일에 저장하도록 해야 합니다. 이렇게 하면 해당 정보가 업데이트될 때 라이브러리를 다시 컴파일하지 않고 이전 버전의 구성 파일을 새로운 파일로 바꾸기만 하면 됩니다.

## 라이브러리 사용

다른 개발자가 만든 .NET 라이브러리를 사용하는 개발자는 라이브러리의 새 버전이 자신의 프로젝트와 완전히 호환되지 않을 수 있으며 그러한 변경 사항에 적응하기 위해 자신의 코드를 업데이트해야 상황에 종종 처하게 된다는 사실을 알고 있을 것입니다.

다행히 C# 및 .NET 에코시스템에는 큰 변화가 생긴 새로운 라이브러리 버전과 작동하도록 앱을 손쉽게 업데이트할 수 있는 기능과 기술이 마련되어 있습니다.

## 어셈블리 바인딩 리디렉션

앱이 사용하는 라이브러리의 버전을 업데이트하기 위해 `app.config` 파일을 사용할 수 있습니다. [바인딩 리디렉션](#)을 추가하면 앱을 다시 컴파일하지 않고도 새 라이브러리 버전을 사용할 수 있습니다. 다음 예제에서는 원래 컴파일된 `1.0.0` 버전 대신 `ReferencedLibrary`의 `1.0.1` 패치 버전을 사용하도록 앱의 `app.config` 파일을 업데이트하는 방법을 보여 줍니다.

#### XML

```
<dependentAssembly>
    <assemblyIdentity name="ReferencedLibrary"
publicKeyToken="32ab4ba45e0a69a1" culture="en-us" />
    <bindingRedirect oldVersion="1.0.0" newVersion="1.0.1" />
</dependentAssembly>
```

#### ① 참고

이 방법은 `ReferencedLibrary`의 새 버전이 앱과 이진 호환되는 경우에만 적용됩니다. 호환성을 결정할 때 확인해야 할 변경 사항은 위의 [이전 버전과 호환성](#) 섹션을 참조하세요.

## new

상속된 기본 클래스의 멤버를 숨기려면 `new` 한정자를 사용합니다. 이것은 파생 클래스가 기본 클래스의 업데이트에 응답할 수 있는 한 방법입니다.

다음 예제를 참조하세요.

#### C#

```
public class BaseClass
{
    public void MyMethod()
    {
        Console.WriteLine("A base method");
    }
}

public class DerivedClass : BaseClass
{
    public new void MyMethod()
    {
        Console.WriteLine("A derived method");
    }
}

public static void Main()
```

```
{  
    BaseClass b = new BaseClass();  
    DerivedClass d = new DerivedClass();  
  
    b.MyMethod();  
    d.MyMethod();  
}
```

## 출력

콘솔

```
A base method  
A derived method
```

위의 예제에서는 `DerivedClass` 가 `BaseClass` 에 있는 `MyMethod` 메서드를 숨기는 방법을 확인할 수 있습니다. 즉, 라이브러리의 새 버전에 있는 기본 클래스가 파생 클래스에 이미 있는 멤버를 추가하면 파생 클래스 멤버에 `new` 한정자를 사용하여 기본 클래스 멤버를 숨길 수 있습니다.

`new` 한정자를 지정하지 않으면 파생 클래스는 기본 클래스에서 충돌하는 멤버를 기본적으로 숨깁니다. 컴파일러 경고가 생성되지만 코드는 여전히 컴파일됩니다. 즉, 기존 클래스에 새 멤버를 추가하기만 하면 라이브러리의 새 버전이 소스와 이진 모두 여기에 종속된 코드와 호환됩니다.

## override

`override` 한정자를 사용하면 파생된 구현은 기본 클래스 멤버의 구현을 숨기는 대신 확장합니다. 기본 클래스 멤버에 `virtual` 한정자를 적용해야 합니다.

C#

```
public class MyBaseClass  
{  
    public virtual string MethodOne()  
    {  
        return "Method One";  
    }  
}  
  
public class MyDerivedClass : MyBaseClass  
{  
    public override string MethodOne()  
    {  
        return "Derived Method One";  
    }  
}
```

```
public static void Main()
{
    MyBaseClass b = new MyBaseClass();
    MyDerivedClass d = new MyDerivedClass();

    Console.WriteLine("Base Method One: {0}", b.MethodOne());
    Console.WriteLine("Derived Method One: {0}", d.MethodOne());
}
```

## 출력

### 콘솔

```
Base Method One: Method One
Derived Method One: Derived Method One
```

`override` 한정자는 컴파일 시간에 평가되며, 재정의할 가상 멤버를 찾지 못하면 컴파일러에서 오류가 발생합니다.

라이브러리 버전 간을 더욱 간편하게 전환하려면 여기서 설명한 방법을 익히고 어떤 상황에서 이를 사용해야 할지를 이해해야 합니다.

# 방법(C#)

아티클 • 2024. 02. 14.

C# 가이드의 방법 섹션에서 일반적인 질문에 대한 빠른 답변을 찾을 수 있습니다. 경우에 따라 문서는 여러 섹션에 나타날 수 있습니다. 여러 검색 경로를 쉽게 찾을 수 있도록 했습니다.

## 일반 C# 개념

일반적인 C# 개발자 사례인 몇 가지 팁과 요령이 있습니다.

- 개체 이니셜라이저를 사용하여 개체를 초기화합니다.
- 연산자 오버로드를 사용합니다.
- 사용자 지정 확장 메서드를 구현하고 호출합니다.
- 확장 메서드를 사용하여 enum 형식에 대해 새 메서드를 만듭니다.

## 클래스, 레코드, 구조체 멤버

클래스, 레코드, 구조체를 만들어 프로그램을 구현합니다. 이러한 기술은 클래스, 레코드 또는 구조체를 작성할 때 자주 사용됩니다.

- 자동 구현 속성을 선언합니다.
- 읽기/쓰기 속성을 선언하고 사용합니다.
- 상수를 정의합니다.
- ToString 메서드를 재정의하여 문자열 출력을 제공합니다.
- 추상 속성을 정의합니다.
- XML 문서 기능을 사용하여 코드를 문서화합니다.
- 인터페이스 멤버를 명시적으로 구현하여 공용 인터페이스 간소화를 유지합니다.
- 두 인터페이스의 멤버를 명시적으로 구현합니다.

## 컬렉션으로 작업

이러한 문서를 통해 데이터의 컬렉션으로 작업할 수 있습니다.

- 컬렉션 이니셜라이저를 사용하여 사전을 초기화합니다.

## 문자열 사용

문자열은 텍스트를 표시하거나 조작하는 데 사용되는 기본 데이터 형식입니다. 이러한 문서는 문자열이 포함된 일반적인 사례를 보여줍니다.

- 문자열을 비교합니다.
- 문자열의 내용을 수정합니다.
- 문자열이 숫자를 나타내는지 여부를 확인합니다.
- String.Split를 사용하여 문자열을 구분합니다.
- 여러 문자열을 하나로 결합합니다.
- 문자열 내에서 텍스트를 검색합니다.

## 형식 변환

한 개체를 다른 형식으로 변환해야 하는 경우가 있습니다.

- 문자열이 숫자를 나타내는지 여부를 확인합니다.
- 16진수를 나타내는 문자열과 숫자 사이를 변환합니다.
- 문자열을 DateTime로 변환합니다.
- 바이트 배열을 정수로 변환합니다.
- 문자열을 숫자로 변환합니다.
- 패턴 일치, as 및 is 연산자를 사용하여 안전하게 다른 형식으로 캐스팅합니다.
- 사용자 지정 형식 변환을 정의합니다.
- 형식이 nullable 값 형식인지 여부를 확인합니다.
- nullable과 비 nullable 값 형식 사이를 변환합니다.

## 같음 및 순서 비교

같음에 대한 자체 규칙을 정의하거나 해당 형식의 개체 간의 자연 정렬을 정의하는 형식을 만들 수 있습니다.

- 참조 기반 같음을 테스트합니다.
- 형식에 대해 값 기반 같음을 정의합니다.

## 예외 처리

.NET 프로그램은 메서드가 예외를 throw하여 작업을 성공적으로 완료되지 않았음을 보고합니다. 이 문서에서는 예외를 사용하는 방법에 대해 살펴보겠습니다.

- try 및 catch를 사용하여 예외를 처리합니다.
- finally 절을 사용하여 리소스를 정리합니다.
- 비 CLS(공용 언어 사양) 예외에서 복구합니다.

## 대리자 및 이벤트

대리인과 이벤트는 느슨하게 결합된 코드 블록을 포함하는 전략에 대한 기능을 제공합니다.

- 대리자를 선언하고, 인스턴스화하고, 사용합니다.
- 멀티캐스트 대리자를 결합합니다.

이벤트는 알림을 구독하거나 게시하는 메커니즘을 제공합니다.

- 이벤트를 구독하거나 구독 취소합니다.
- 인터페이스에서 선언된 이벤트를 구현합니다.
- 코드가 이벤트를 게시할 때 .NET 지침을 준수합니다.
- 파생된 클래스로부터 기본 클래스에서 정의된 이벤트를 발생시킵니다.
- 사용자 지정 이벤트 접근자를 구현합니다.

## LINQ 사례

LINQ를 사용하면 LINQ 쿼리 식 패턴을 지원하는 데이터 소스를 쿼리하는 코드를 작성할 수 있습니다. 이러한 문서는 패턴을 이해하고 다른 데이터 원본으로 작업하는 데 도움이 됩니다.

- 컬렉션을 쿼리합니다.
- 쿼리 식에서 var를 사용합니다.
- 쿼리에서 요소 속성의 하위 집합을 반환합니다.
- 복합 필터링으로 쿼리를 작성합니다.
- 데이터 원본의 요소를 정렬합니다.
- 여러 키로 요소를 정렬합니다.
- 프로젝션 형식을 제어합니다.
- 소스 시퀀스에서 값의 발생 수를 카운트합니다.
- 중간 값을 계산합니다.
- 여러 원본의 데이터를 병합합니다.
- 두 시퀀스 간의 차집합을 반환합니다.
- 빈 쿼리 결과를 디버깅합니다.
- 사용자 지정 메서드를 LINQ 쿼리에 추가합니다.

## 여러 스레드 및 비동기 처리

최신 프로그램은 종종 비동기 작업을 사용합니다. 이러한 문서를 통해 이러한 기법을 사용하는 방법을 배울 수 있습니다.

- System.Threading.Tasks.Task.WhenAll를 사용하여 비동기 성능을 개선합니다.
- async 및 await를 사용하여 여러 웹을 동시에 요청합니다.
- 스레드 풀을 사용합니다.

# 프로그램에 대한 명령줄 인수

일반적으로 C# 프로그램에는 명령줄 인수가 있습니다. 이 문서에서는 이러한 명령줄 인수를 액세스하고 처리하는 방법을 배울 수 있습니다.

- `for`가 포함된 모든 명령줄 인수를 검색합니다.

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# C#에서 String.Split을 사용하여 문자열을 분리하는 방법

아티클 • 2024. 05. 25.

`String.Split` 메서드는 하나 이상의 구분 기호를 기준으로 입력 문자열을 분할하여 부분 문자열 배열을 만듭니다. 이 메서드는 종종 단어 경계에서 문자열을 분리하는 가장 쉬운 방법입니다. 다른 특정 문자 또는 문자열에서 문자열을 분할하는 데도 사용됩니다.

## ① 참고

이 문서의 C# 예제는 [Try.NET](#) 인라인 코드 러너 및 놀이터에서 실행됩니다. 대화형 창에서 예제를 실행하려면 **실행** 버튼을 선택합니다. 코드를 실행하면 **실행**을 다시 선택하여 코드를 수정하고 수정된 코드를 실행할 수 있습니다. 수정된 코드는 대화형 창에서 실행되거나, 컴파일이 실패하면 대화형 창에 모든 C# 컴파일러 오류 메시지가 표시됩니다.

다음 코드는 공통 어구를 각 단어에 대한 문자열의 배열로 나눕니다.

C#

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

구분 문자의 모든 인스턴스는 반환된 배열에 값을 생성합니다. C#의 배열은 제로 인덱싱 되므로 배열의 각 문자열은 0에서 `Array.Length` 속성에서 반환된 값에서 1을 뺀 값으로 인덱싱됩니다.

C#

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' ');

for (int i = 0; i < words.Length; i++)
{
    System.Console.WriteLine($"Index {i}: <{words[i]}>");
}
```

연속된 구분 문자는 반환된 배열의 값으로 빈 문자열을 생성합니다. 공백 문자를 구분 기호로 사용하는 다음 예제에서 빈 문자열을 만드는 방법을 확인할 수 있습니다.

C#

```
string phrase = "The quick brown    fox    jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

이 동작은 표 형식 데이터를 나타내는 쉼표로 구분된 값(CSV) 파일과 같은 형식을 더 쉽게 만듭니다. 연속된 쉼표는 빈 열을 나타냅니다.

반환된 배열에 빈 문자열을 제외하기 위해 선택적인

[StringSplitOptions.RemoveEmptyEntries](#) 매개 변수를 전달할 수 있습니다. 반환된 컬렉션의 더 복잡한 처리를 위해 [LINQ](#)를 사용하여 결과 시퀀스를 조작할 수 있습니다.

[String.Split](#)은 다중 구분 문자를 사용할 수 있습니다. 다음 예제에서는 공백, 쉼표, 마침표, 콜론 및 탭을 구분 문자로 사용하며, 해당 문자는 [Split](#)의 배열로 전달됩니다. 코드 맨 아래의 루프는 반환된 배열의 각 단어를 표시합니다.

C#

```
char[] delimiterChars = { ' ', ',' , '.', ':' , '\t' };

string text = "one\ttwo three:four,five six seven";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
System.Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

연속되는 모든 구분 기호의 인스턴스는 출력 배열에 빈 문자열을 생성합니다.

C#

```
char[] delimiterChars = { ' ', ',' , '.', ':' , '\t' };

string text = "one\ttwo :,five six seven";
System.Console.WriteLine($"Original text: '{text}'");
```

```
string[] words = text.Split(delimiterChars);
System.Console.WriteLine(${words.Length} words in text:");

foreach (var word in words)
{
    System.Console.WriteLine($"<{word}>");
}
```

`String.Split`은 문자열 배열(단일 문자 대신 대상 문자열을 구문 분석하는 구분 기호 역할을 하는 문자 시퀀스)을 사용할 수 있습니다.

C#

```
string[] separatingStrings = { "<<", "..." };

string text = "one<<two.....three<four";
System.Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(separatingStrings,
System.StringSplitOptions.RemoveEmptyEntries);
System.Console.WriteLine(${words.Length} substrings in text:");

foreach (var word in words)
{
    System.Console.WriteLine(word);
}
```

## 참고 항목

- 문자열에서 요소 추출
- 문자열
- .NET 정규식

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 여러 문자열 연결 방법(C# 가이드)

아티클 • 2024. 03. 15.

연결은 한 문자열을 다른 문자열의 끝에 추가하는 프로세스입니다. `+` 연산자를 사용하여 문자열을 연결합니다. 문자열 리터럴 및 문자열 상수의 경우 연결 시 컴파일이 발생하며, 런타임 연결은 발생하지 않습니다. 문자열 변수의 경우 연결은 런타임에서만 발생합니다.

## ① 참고

이 문서의 C# 예제는 [Try.NET](#) 인라인 코드 러너 및 놀이터에서 실행됩니다. 대화형 창에서 예제를 실행하려면 **실행** 버튼을 선택합니다. 코드를 실행하면 **실행**을 다시 선택하여 코드를 수정하고 수정된 코드를 실행할 수 있습니다. 수정된 코드는 대화형 창에서 실행되거나, 컴파일이 실패하면 대화형 창에 모든 C# 컴파일러 오류 메시지가 표시됩니다.

## 문자열 리터럴

다음 예제에서는 소스 코드의 가독성을 향상하기 위해 긴 문자열 리터럴을 작은 문자열로 분할합니다. 이 코드에서는 작은 문자열을 연결하여 긴 문자열 리터럴을 만듭니다. 컴파일 시간에 부분이 연결되어 단일 문자열이 됩니다. 이 경우 처리되는 문자열 수와 관계 없이 런타임 성능에 미치는 영향은 없습니다.

C#

```
// Concatenation of literals is performed at compile time, not run time.
string text = "Historically, the world of data and the world of objects " +
    "have not been well integrated. Programmers work in C# or Visual Basic " +
    "and also in SQL or XQuery. On the one side are concepts such as classes,
    +
    "objects, fields, inheritance, and .NET Framework APIs. On the other side "
    +
    "are tables, columns, rows, nodes, and separate languages for dealing with "
    +
    "them. Data types often require translation between the two worlds; there
    are " +
    "different standard functions. Because the object world has no notion of
    query, a " +
    "query can only be represented as a string without compile-time type
    checking or " +
    "IntelliSense support in the IDE. Transferring data from SQL tables or XML
    trees to " +
    "objects in memory is often tedious and error-prone.;"
```

```
System.Console.WriteLine(text);
```

## + 및 += 연산자

문자열 변수를 연결하려면 `+` 또는 `+=` 연산자, [문자열 보간](#) 또는 `String.Format`, `String.Concat`, `String.Join` 또는 `StringBuilder.Append` 메서드를 사용할 수 있습니다. `+` 연산자는 사용하기 쉽고 직관적인 코드를 만들니다. 하나의 문에서 여러 `+` 연산자를 사용해도 문자열 콘텐츠는 한 번만 복사됩니다. 다음 코드는 `+` 및 `+=` 연산자를 사용하여 문자열을 연결하는 예제를 보여줍니다.

C#

```
string userName = "<Type your name here>";
string dateString = DateTime.Today.ToShortDateString();

// Use the + and += operators for one-time concatenations.
string str = "Hello " + userName + ". Today is " + dateString + ".";
System.Console.WriteLine(str);

str += " How are you today?";
System.Console.WriteLine(str);
```

## 문자열 보간

일부 식에서는 다음 코드와 같이 문자열 보간을 사용하여 문자열을 연결하는 것이 더 쉽습니다.

C#

```
string userName = "<Type your name here>";
string date = DateTime.Today.ToShortDateString();

// Use string interpolation to concatenate strings.
string str = $"Hello {userName}. Today is {date}.";
System.Console.WriteLine(str);

str = $"{str} How are you today?";
System.Console.WriteLine(str);
```

① 참고

문자열 연결 연산에서 C# 컴파일러는 null 문자열을 빈 문자열과 동일하게 처리합니다.

C# 10부터 자리 표시자에 사용되는 모든 식이 상수 문자열인 경우 문자열 보간을 사용하여 상수 문자열을 초기화할 수 있습니다.

## String.Format

문자열을 연결하는 다른 메서드는 [String.Format](#)입니다. 이 메서드는 작은 수의 구성 요소 문자열에서 문자열을 빌드할 때 잘 작동합니다.

## StringBuilder

다른 경우는 결합하는 소스 문자열의 개수를 모르는 루프에서 문자열을 결합할 수 있으며 소스 문자열의 실제 개수는 클 수 있습니다. [StringBuilder](#) 클래스는 이러한 시나리오를 위해 설계되었습니다. 다음 코드는 [StringBuilder](#) 클래스의 [Append](#) 메서드를 사용하여 문자열을 연결합니다.

C#

```
// Use StringBuilder for concatenation in tight loops.  
var sb = new System.Text.StringBuilder();  
for (int i = 0; i < 20; i++)  
{  
    sb.AppendLine(i.ToString());  
}  
System.Console.WriteLine(sb.ToString());
```

문자열 연결 또는 [StringBuilder](#) 클래스를 선택하는 이유에 대해 자세히 읽을 수 있습니다.

## String.Concat 또는 String.Join

컬렉션의 문자열을 조인하는 또 다른 옵션은 [String.Concat](#) 메서드를 사용하는 것입니다. 구분 기호가 소스 문자열을 구분해야 하는 경우 메서드를 사용합니다 [String.Join](#). 다음 코드는 두 메서드 모두를 사용하여 단어 배열을 결합합니다.

C#

```
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the",  
    "lazy", "dog." };  
  
var unreadablePhrase = string.Concat(words);  
System.Console.WriteLine(unreadablePhrase);
```

```
var readablePhrase = string.Join(" ", words);
System.Console.WriteLine(readablePhrase);
```

## LINQ 및 Enumerable.Aggreagte

마지막으로 LINQ와 Enumerable.Aggreagte 메서드를 사용하여 컬렉션의 문자열을 조인할 수 있습니다. 이 메서드는 람다 식을 사용하여 소스 문자열을 결합합니다. 람다 식은 각 문자열을 기준의 누적 문자열에 추가하는 작업을 수행합니다. 다음 예제에서는 배열의 각 단어 사이에 공백을 추가하여 단어 배열을 결합합니다.

C#

```
string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the",
"lazy", "dog." };

var phrase = words.Aggregate((partialPhrase, word) => $"{partialPhrase}
{word}");
System.Console.WriteLine(phrase);
```

이 옵션을 사용하면 반복할 때마다 중간 문자열을 만들므로 컬렉션을 연결하는 다른 메서드보다 할당이 늘어날 수 있습니다. 성능 최적화가 중요한 경우 Enumerable.Aggreagte 대신 StringBuilder 클래스나 String.Concat 또는 String.Join 메서드를 사용하여 컬렉션을 연결하는 것이 좋습니다.

## 참고 항목

- [String](#)
- [StringBuilder](#)
- [문자열](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# 문자열 검색 방법

아티클 • 2024. 02. 14.

문자열에서 텍스트를 검색하는 두 가지 기본 전략을 사용할 수 있습니다. [String](#) 클래스의 메서드는 특정 텍스트를 검색합니다. 정규식은 텍스트에서 패턴을 검색합니다.

## ① 참고

이 문서의 C# 예제는 [Try.NET](#) 인라인 코드 러너 및 놀이터에서 실행됩니다. 대화형 창에서 예제를 실행하려면 **실행** 버튼을 선택합니다. 코드를 실행하면 **실행**을 다시 선택하여 코드를 수정하고 수정된 코드를 실행할 수 있습니다. 수정된 코드는 대화형 창에서 실행되거나, 컴파일이 실패하면 대화형 창에 모든 C# 컴파일러 오류 메시지가 표시됩니다.

[System.String](#) 클래스의 별칭인 [string](#) 형식은 문자열 내용을 검색하기 위한 여러 가지 유용한 메서드를 제공합니다. 그중에 [Contains](#), [StartsWith](#), [EndsWith](#), [IndexOf](#), [LastIndexOf](#)입니다. [System.Text.RegularExpressions.Regex](#) 클래스는 텍스트에서 패턴을 검색하는 풍부한 어휘를 제공합니다. 이 문서에서는 이러한 기술 및 요구 사항에 대해 최상의 메서드를 선택하는 방법을 알아봅니다.

## 문자열에 텍스트가 포함되어 있나요?

[String.Contains](#), [String.StartsWith](#) 및 [String.EndsWith](#) 메서드는 문자열에서 특정 텍스트를 검색합니다. 다음 예에서는 이러한 메서드 각각 및 대/소문자 구분 검색을 사용하는 변형을 보여 줍니다.

C#

```
string factMessage = "Extension methods have all the capabilities of regular
static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\\"");

// Simple comparisons are always case sensitive!
bool containsSearchResult = factMessage.Contains("extension");
Console.WriteLine($"Contains \"extension\"? {containsSearchResult}");

// For user input and strings that will be displayed to the end user,
// use the StringComparison parameter on methods that have it to specify how
// to match strings.
bool ignoreCaseSearchResult = factMessage.StartsWith("extension",
System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Starts with \"extension\"? {ignoreCaseSearchResult}")
```

```
(ignoring case"));

bool endsWithSearchResult = factMessage.EndsWith(".",  
System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Ends with '.'? {endsWithSearchResult}");
```

앞의 예제에서는 이러한 메서드를 사용하는 경우 중요한 사항을 보여줍니다. 검색은 기본적으로 대/소문자를 구분합니다. `StringComparison.CurrentCultureIgnoreCase` 열거형 값을 사용하여 대/소문자 구분 검색을 지정합니다.

## 검색된 텍스트가 있는 문자열의 위치는 어디인가요?

`IndexOf` 및 `LastIndexOf` 메서드도 문자열에서 텍스트를 검색합니다. 이러한 메서드는 검색되는 텍스트의 위치를 반환합니다. 텍스트를 찾을 수 없으면 `-1`을 반환합니다. 다음 예제에서는 "메서드"라는 단어의 첫 번째 및 마지막 항목에 대한 검색을 보여주고, 그 사이에 텍스트를 표시합니다.

C#

```
string factMessage = "Extension methods have all the capabilities of regular  
static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\\"");

// This search returns the substring between two strings, so
// the first index is moved to the character just after the first string.
int first = factMessage.IndexOf("methods") + "methods".Length;
int last = factMessage.LastIndexOf("methods");
string str2 = factMessage.Substring(first, last - first);
Console.WriteLine($"Substring between \"methods\" and \"methods\":  
'{str2}'");
```

## 정규식을 사용하여 특정 텍스트 찾기

`System.Text.RegularExpressions.Regex` 클래스를 사용하여 문자열을 검색할 수 있습니다. 이 검색은 단순한 텍스트 패턴에서 복잡한 텍스트 패턴까지 복잡성의 범위를 지정할 수 있습니다.

다음 코드 예제에서는 문장에서 "the" 또는 "their"라는 문자를 검색하고 대/소문자를 무시합니다. 고정 메서드 `Regex.IsMatch`은 검색을 수행합니다. 검색할 문자열 및 검색 패턴을 입력합니다. 이 경우에 세 번째 인수는 대/소문자 구분 검색을 지정합니다. 자세한 내용은 `System.Text.RegularExpressions.RegexOptions`를 참조하세요.

검색 패턴은 검색할 텍스트를 설명합니다. 다음 표에서는 검색 패턴의 각 요소에 대해 설명합니다. (아래 표에서는 C# 문자열에서 \\로 이스케이프되어야 하는 단일 \를 사용합니다.)

### 테이블 확장

패턴	의미
the	텍스트 "the" 일치
(eir)?	"eir"과 0 또는 1개 항목 일치
\s	공백 문자 찾기

C#

```
string[] sentences =
{
    "Put the water over there.",
    "They're quite thirsty.",
    "Their water bottles broke."
};

string sPattern = "the(ir)?\\s";

foreach (string s in sentences)
{
    Console.WriteLine($"{s,24}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern,
System.Text.RegularExpressions.RegexOptions.IgnoreCase))
    {
        Console.WriteLine($"  (match for '{sPattern}' found)");
    }
    else
    {
        Console.WriteLine();
    }
}
```

### 팁

정확한 문자열을 검색하는 경우 일반적으로 `string` 메서드를 사용하는 것이 좋습니다. 원본 문자열에서 몇 가지 패턴을 검색하는 경우 정규식을 사용하는 것이 좋습니다.

# 문자열은 패턴을 따르나요?

다음 코드는 정규식을 사용하여 배열에서 각 문자열 형식의 유효성을 검사합니다. 유효성 검사에서는 각 문자열이 전화 번호의 형식을 사용해야 합니다. 이 경우 3개의 숫자 그룹이 대시로 구분되고, 처음 두 그룹에는 세 자리 숫자가 포함되며, 세 번째 그룹에는 네 자리 숫자가 포함됩니다. 검색 패턴은 정규식 `^\d{3}-\d{3}-\d{4}$` 을 사용합니다. 자세한 내용은 [정규식 언어 - 빠른 참조](#)를 참조하세요.

[+] 테이블 확장

패턴	의미
<code>^</code>	문자열의 시작 부분 일치
<code>\d{3}</code>	정확히 3자리 문자 일치
<code>-</code>	'-' 문자 일치
<code>\d{4}</code>	정확히 4자리 문자 일치
<code>\$</code>	문자열의 끝 일치

C#

```
string[] numbers =
{
    "123-555-0190",
    "444-234-22450",
    "690-555-0178",
    "146-893-232",
    "146-555-0122",
    "4007-555-0111",
    "407-555-0111",
    "407-2-5555",
    "407-555-8974",
    "407-2ab-5555",
    "690-555-8148",
    "146-893-232-"
};

string sPattern = "^\d{3}-\d{3}-\d{4}$";

foreach (string s in numbers)
{
    Console.WriteLine($"{s,14}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern))
    {
        Console.WriteLine(" - valid");
    }
}
```

```
    else
    {
        Console.WriteLine(" - invalid");
    }
}
```

이 단일 검색 패턴은 많은 유효한 문자열과 일치합니다. 단일 텍스트 문자열이 아닌 패턴을 검색하거나 유효성을 확인하기 위해 정규식을 사용하는 것이 좋습니다.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [문자열](#)
- [System.Text.RegularExpressions.Regex](#)
- [.NET 정규식](#)
- [정규식 언어 - 빠른 참조](#)
- [.NET에서 문자열 사용에 대한 모범 사례](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# C에서 문자열 콘텐츠를 수정하는 방법 #

아티클 • 2023. 04. 08.

이 문서에서는 기존 `string`을 수정하여 `string`을 생성하는 다양한 기술을 보여 줍니다. 설명된 모든 기술은 새 `string` 개체로 수정의 결과를 반환합니다. 원래 문자열과 수정된 문자열이 고유 인스턴스임을 보여 주기 위해 이 예제에서는 결과를 새 변수에 저장합니다. 각 예제를 실행할 때 원래 `string`과 수정된 새 `string`을 확인할 수 있습니다.

## ① 참고

이 문서의 C# 예제는 Try.NET<sup>↗</sup> 인라인 코드 러너 및 놀이터에서 실행됩니다. 대화형 창에서 예제를 실행하려면 **실행** 버튼을 선택합니다. 코드를 실행하면 **실행**을 다시 선택하여 코드를 수정하고 수정된 코드를 실행할 수 있습니다. 수정된 코드는 대화형 창에서 실행되거나, 컴파일이 실패하면 대화형 창에 모든 C# 컴파일러 오류 메시지가 표시됩니다.

이 문서에서 설명되는 여러 가지 기술이 있습니다. 기존 텍스트를 바꿀 수 있습니다. 패턴을 검색하고 다른 텍스트와 일치하는 텍스트를 바꿀 수 있습니다. 일련의 문자로 문자열을 처리할 수 있습니다. 공백을 제거하는 편리한 메서드를 사용할 수도 있습니다. 시나리오에 가장 일치하는 기술을 선택합니다.

## 텍스트 바꾸기

다음 코드는 기존 텍스트를 대체로 바꿔 새 문자열을 만듭니다.

C#

```
string source = "The mountains are behind the clouds today.";

// Replace one substring with another with String.Replace.
// Only exact matches are supported.
var replacement = source.Replace("mountains", "peaks");
Console.WriteLine($"The source string is <{source}>");
Console.WriteLine($"The updated string is <{replacement}>");
```

위의 코드는 문자열의 이러한 **변경할 수 없는 속성**을 보여 줍니다. 앞의 예제에서 원래 문자열, `source`가 수정되지 않는 것을 볼 수 있습니다. `String.Replace` 메서드는 수정 내용을 포함하는 새 `string`을 만듭니다.

`Replace` 메서드는 문자열 또는 단일 문자를 바꿀 수 있습니다. 두 경우 모두에서 검색된 텍스트의 모든 항목이 대체됩니다. 다음 예제에서는 모든 '' 문자를 '\_'로 바꿉니다.

C#

```
string source = "The mountains are behind the clouds today.";

// Replace all occurrences of one char with another.
var replacement = source.Replace(' ', '_');
Console.WriteLine(source);
Console.WriteLine(replacement);
```

원본 문자열은 변경되지 않으며, 새 문자열은 교체와 함께 반환됩니다.

## 공백 제거

[String.Trim](#), [String.TrimStart](#) 및 [String.TrimEnd](#) 메서드를 사용하여 선행 또는 후행 공백을 제거할 수 있습니다. 다음 코드에서는 각 예제를 보여 줍니다. 원본 문자열 변경되지 않습니다. 이러한 메서드는 수정된 내용이 포함된 새 문자열을 반환합니다.

C#

```
// Remove trailing and leading white space.
string source = "    I'm wider than I need to be.    ";
// Store the results in a new string variable.
var trimmedResult = source.Trim();
var trimLeading = source.TrimStart();
var trimTrailing = source.TrimEnd();
Console.WriteLine($"<{source}>");
Console.WriteLine($"<{trimmedResult}>");
Console.WriteLine($"<{trimLeading}>");
Console.WriteLine($"<{trimTrailing}>");
```

## 텍스트 제거

[String.Remove](#) 메서드를 사용하여 문자열에서 텍스트를 제거할 수 있습니다. 이 메서드는 특정 인덱스에서 시작하는 문자 수를 제거합니다. 다음 예제에서는 [Remove](#)가 뒤에 오는 [String.IndexOf](#)를 사용하여 문자열에서 텍스트를 제거하는 방법을 보여 줍니다.

C#

```
string source = "Many mountains are behind many clouds today.";
// Remove a substring from the middle of the string.
string toRemove = "many ";
string result = string.Empty;
int i = source.IndexOf(toRemove);
if (i >= 0)
{
    result = source.Remove(i, toRemove.Length);
```

```
        }
        Console.WriteLine(source);
        Console.WriteLine(result);
```

## 일치 패턴 바꾸기

정규식을 사용하여 패턴에 의해 정의된 새 텍스트로 텍스트 일치 패턴을 바꿀 수 있습니다. 다음 예제에서는 `System.Text.RegularExpressions.Regex` 클래스를 사용하여 원본 문자열의 패턴을 찾고 적절한 대/소문자로 대체합니다. `Regex.Replace(String, String, MatchEvaluator, RegexOptions)` 메서드는 해당 인수 중 하나로 대체 논리를 제공하는 함수를 사용합니다. 이 예제에서 해당 함수, `LocalReplaceMatchCase`는 샘플 메서드 내에서 선언된 **로컬 함수**입니다. `LocalReplaceMatchCase`는 `System.Text.StringBuilder` 클래스를 사용하여 적절한 대/소문자로 대체 문자열을 작성합니다.

정규식은 알려진 텍스트가 아닌 패턴을 따르는 텍스트를 검색하고 대체하는 데 가장 유용합니다. 자세한 내용은 [문자열 검색 방법](#)을 참조하세요. 검색 패턴, "the\s"는 공백 문자가 뒤에 오는 문자 "the"를 검색합니다. 패턴의 해당 부분을 사용하면 원본 문자열의 "there"와 일치하지 않습니다. 정규식 언어 요소에 대한 자세한 내용은 [정규식 언어 - 빠른 참조](#)를 참조하세요.

C#

```
string source = "The mountains are still there behind the clouds today.";

// Use Regex.Replace for more flexibility.
// Replace "the" or "The" with "many" or "Many".
// using System.Text.RegularExpressions
string replaceWith = "many ";
source = System.Text.RegularExpressions.Regex.Replace(source, "the\\s",
LocalReplaceMatchCase,
    System.Text.RegularExpressions.RegexOptions.IgnoreCase);
Console.WriteLine(source);

string LocalReplaceMatchCase(System.Text.RegularExpressions.Match
matchExpression)
{
    // Test whether the match is capitalized
    if (Char.ToUpper(matchExpression.Value[0]))
    {
        // Capitalize the replacement string
        System.Text.StringBuilder replacementBuilder = new
System.Text.StringBuilder(replaceWith);
        replacementBuilder[0] = Char.ToUpper(replacementBuilder[0]);
        return replacementBuilder.ToString();
    }
    else
    {
        return replaceWith;
    }
}
```

```
    }  
}
```

`StringBuilder.ToString` 메서드는 `StringBuilder` 객체의 내용으로 변경할 수 없는 문자열을 반환합니다.

## 개별 문자 수정

문자열에서 문자 배열을 생성하고, 배열의 내용을 수정한 다음, 수정된 배열의 내용에서 새 문자열을 만들 수 있습니다.

다음 예제에서는 문자열에서 문자 집합을 대체하는 방법을 보여 줍니다. 먼저 `String.ToCharArray()` 메서드를 사용하여 문자의 배열을 만듭니다. 메서드를 `IndexOf` 사용하여 "fox"라는 단어의 시작 인덱스를 찾습니다. 다음 세 문자는 다른 단어로 대체됩니다. 마지막으로 새 문자열은 업데이트된 문자 배열에서 생성됩니다.

C#

```
string phrase = "The quick brown fox jumps over the fence";  
Console.WriteLine(phrase);  
  
char[] phraseAsChars = phrase.ToCharArray();  
int animalIndex = phrase.IndexOf("fox");  
if (animalIndex != -1)  
{  
    phraseAsChars[animalIndex++] = 'c';  
    phraseAsChars[animalIndex++] = 'a';  
    phraseAsChars[animalIndex] = 't';  
}  
  
string updatedPhrase = new string(phraseAsChars);  
Console.WriteLine(updatedPhrase);
```

## 프로그래밍 방식으로 문자열 콘텐츠 작성

문자열은 변경할 수 없으므로 이전 예제에서는 모두 임시 문자열 또는 문자 배열을 만듭니다. 고성능 시나리오에서는 이 힙 할당을 방지하는 것이 좋습니다. .NET Core는 중간 임시 문자열 할당을 방지하면서 콜백을 통해 문자열의 문자 콘텐츠를 프로그래밍 방식으로 채울 수 있는 `String.Create` 메서드를 제공합니다.

C#

```
// constructing a string from a char array, prefix it with some additional  
// characters  
char[] chars = { 'a', 'b', 'c', 'd', '\0' };
```

```
int length = chars.Length + 2;
string result = string.Create(length, chars, (Span<char> strContent, char[]
charArray) =>
{
    strContent[0] = '0';
    strContent[1] = '1';
    for (int i = 0; i < charArray.Length; i++)
    {
        strContent[i + 2] = charArray[i];
    }
});

Console.WriteLine(result);
```

안전하지 않은 코드를 사용하여 고정 블록의 문자열을 수정할 수 있지만 문자열을 만든 후에는 문자열 콘텐츠를 수정하지 않는 것이 좋습니다. 그렇게 하면 예측할 수 없는 방식으로 작업이 중단되기 때문입니다. 예를 들어 콘텐츠가 동일한 문자열을 다른 사용자가 인턴(intern)하는 경우 해당 사용자는 복사본을 얻게 되며 문자열을 수정해도 해당 사용자의 문자열은 수정되지 않습니다.

## 참조

- [.NET 정규식](#)
- [정규식 언어 - 빠른 참조](#)

# C#에서 문자열을 비교하는 방법

아티클 • 2024. 03. 20.

문자열을 비교하여 "이 두 문자열이 같은가요?" 또는 "정렬할 때 이러한 문자열을 어떤 순서로 배치해야 하나요?"라는 두 가지 질문 중 하나에 대답합니다.

이러한 두 가지 질문은 문자열 비교에 영향을 주는 요소에 의해 복잡해집니다.

- 서수 또는 언어 비교를 선택할 수 있습니다.
- 대/소문자를 구분할지 여부를 선택할 수 있습니다.
- 문화권별 비교를 선택할 수 있습니다.
- 언어적 비교는 문화권 및 플랫폼에 따라 다릅니다.

`System.StringComparison` 열거형 필드는 다음 선택 항목을 나타냅니다.

- **CurrentCulture**: 문화권 구분 정렬 규칙과 현재 문화권을 사용하여 문자열을 비교합니다.
- **CurrentCultureIgnoreCase**: 문화권 구분 정렬 규칙, 현재 문화권을 사용하여 문자열을 비교하고 비교되는 문자열의 경우를 무시합니다.
- **InvariantCulture**: 문화권 구분 정렬 규칙과 고정 문화권을 사용하여 문자열을 비교합니다.
- **InvariantCultureIgnoreCase**: 문화권 구분 정렬 규칙, 고정 문화권을 사용하여 문자열을 비교하고 비교되는 문자열의 경우를 무시합니다.
- **Ordinal**: 서수(이진) 정렬 규칙을 사용하여 문자열을 비교합니다.
- **OrdinalIgnoreCase**: 서수(이진) 정렬 규칙을 사용하여 문자열을 비교하고 비교되는 문자열의 경우를 무시합니다.

## ① 참고

이 문서의 C# 예제는 [Try.NET](#) 인라인 코드 러너 및 놀이터에서 실행됩니다. 대화형 창에서 예제를 실행하려면 **실행** 버튼을 선택합니다. 코드를 실행하면 **실행**을 다시 선택하여 코드를 수정하고 수정된 코드를 실행할 수 있습니다. 수정된 코드는 대화형 창에서 실행되거나, 컴파일이 실패하면 대화형 창에 모든 C# 컴파일러 오류 메시지가 표시됩니다.

문자열을 비교할 때 순서를 정의합니다. 비교는 문자열 시퀀스를 정렬하는 데 사용됩니다. 시퀀스가 알려진 순서대로 되면 소프트웨어와 사람 모두 검색하기가 더 쉽습니다. 문자열이 같으면 다른 비교가 검사 수 있습니다. 이러한 동일성 검사 같음과 유사하지만 대/소문자 차이와 같은 일부 차이점은 무시될 수 있습니다.

# 기본 서수 비교

기본적으로 가장 일반적인 작업:

- `String.Equals`
- `String.Equality` 및 `String.Inequality`, 즉, 같음 연산 `==` 자와 `!=` 각각 대/소문자를 구분하는 서수 비교를 수행합니다. `String.Equals`에는 정렬 규칙을 변경하기 위해 인수를 제공할 수 있는  `StringComparison` 오버로드가 있습니다. 다음은 해당 예입니다.

C#

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result ? "equal." : "not equal.")}");

result = root.Equals(root2, StringComparison.OrdinalIgnoreCase);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result ? "equal." : "not equal.")}");

Console.WriteLine($"Using == says that <{root}> and <{root2}> are {(root == root2 ? "equal" : "not equal")});
```

기본 서수 비교는 문자열을 비교할 때 언어 규칙을 고려하지 않습니다. 두 문자열에서 각 `Char` 개체의 이진값을 비교합니다. 결과적으로, 기본 서수 비교도 대/소문자를 구분합니다.

`String.Equals`, `==` 및 `!=` 연산자를 사용한 같음 테스트는 `String.CompareTo` 및 `Compare(String, String)` 메서드를 사용한 문자열 비교와 다릅니다. 모두 대/소문자를 구분하여 비교합니다. 그러나 같음 테스트는 서수 비교를 수행하는 반면 `CompareTo` 및 `Compare` 메서드는 현재 문화권을 사용하여 문화권 인식 언어 비교를 수행합니다. 수행할 비교 유형을 명시적으로 지정하는 오버로드를 호출하여 코드의 의도를 명확하게 합니다.

## 대/소문자를 구분하지 않는 서수 비교

`String.Equals(String, StringComparison)` 메서드를 사용하면 대/소문자를 구분하지 않는 서수 비교를 위해  `StringComparison.OrdinalIgnoreCase`의  `StringComparison` 값을 지정할 수 있습니다. 인수의 값을  `StringComparison.OrdinalIgnoreCase` 지정하는 경우 대/소문자를 구분하지 않는 서수 비교를 수행하는 정적 `String.Compare(String, String, StringComparison)` 메서드도 있습니다. 이러한 비교는 다음 코드에 나와 있습니다.

C#

```

string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2, StringComparison.OrdinalIgnoreCase);
bool areEqual = String.Equals(root, root2,
StringComparison.OrdinalIgnoreCase);
int comparison = String.Compare(root, root2, comparisonType:
StringComparison.OrdinalIgnoreCase);

Console.WriteLine($"Ordinal ignore case: <{root}> and <{root2}> are {(result
? "equal." : "not equal.")}");
Console.WriteLine($"Ordinal static ignore case: <{root}> and <{root2}> are
{({areEqual ? "equal." : "not equal."})}");
if (comparison < 0)
    Console.WriteLine($"<{root}> is less than <{root2}>");
else if (comparison > 0)
    Console.WriteLine($"<{root}> is greater than <{root2}>");
else
    Console.WriteLine($"<{root}> and <{root2}> are equivalent in order");

```

이러한 메서드는 대/소문자를 구분하지 않는 서수 비교를 수행할 때 고정 문화권의 대/소문자 구분 규칙을 사용합니다.

## 언어 비교

많은 문자열 비교 메서드(예: `String.StartsWith`)는 기본적으로 현재 문화권 언어 규칙을 사용하여 입력 순서를 지정합니다. 이 언어 비교를 "단어 정렬 순서"라고도 합니다. 언어 비교를 수행할 때 일부 무수 유니코드 문자에는 특수 가중치가 할당될 수 있습니다. 예를 들어 하이픈 "-"에는 작은 가중치가 할당되어 "co-op" 및 "coop"이 정렬 순서대로 나란히 표시될 수 있습니다. 일부 인쇄되지 않는 컨트롤 문자는 무시될 수 있습니다. 또한 일부 유니코드 문자는 인스턴스 시 `Char` 퀸스에 해당할 수 있습니다. 다음 예제에서는 "거리에서 춤을 춘다"라는 문구를 독일어로 사용하고 한 문자열에 "ss"(U+0073 U+0073)를 사용하고 다른 문자열에는 'ß'(U+00DF)를 사용합니다. 언어적으로(Windows의 경우) "ss"는 "en-US" 및 "de-DE" 문화권에서 독일어 Esszet: 'ß' 문자와 같습니다.

C#

```

string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

bool equal = String.Equals(first, second,
StringComparison.InvariantCulture);
Console.WriteLine($"The two strings {(equal == true ? "are" : "are not")}
equal.");

```

```

showComparison(first, second);

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words);
showComparison(word, other);
showComparison(words, other);
void showComparison(string one, string two)
{
    int compareLinguistic = String.Compare(one, two,
StringComparison.InvariantCulture);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using invariant
culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using invariant
culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using invariant culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal
comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal
comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using ordinal comparison");
}

```

Windows에서는 .NET 5 이전에는 언어 비교에서 서수 비교로 변경하면 "cop", "coop" 및 "co-op"의 정렬 순서가 변경됩니다. 두 개의 독일어 문장도 서로 다른 비교 형식을 사용하여 다르게 비교됩니다. .NET 5 이전에는 .NET 세계화 API에서 NLS(국가 언어 지원) [라이브러리를 사용했습니다](#). .NET 5 이상 버전에서 .NET 세계화 API는 지원되는 모든 운영 체제에서 .NET의 세계화 동작을 통합하는 [ICU\(유니코드용 국제 구성 요소\)](#) 라이브러리를 사용합니다.

## 특정 문화권을 사용한 비교

다음 예제에서는 en-US 및 de-DE 문화권에 대한 개체를 저장 [CultureInfo](#) 합니다. 비교는 문화권별 비교를 보장하기 위해 [CultureInfo](#) 개체를 사용하여 수행됩니다. 사용된 문화권은 언어 비교에 영향을 줍니다. 다음 예제에서는 "en-US" 문화권과 "de-DE" 문화권을 사용하여 두 개의 독일어 문장을 비교한 결과를 보여줍니다.

```

string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

var en = new System.Globalization.CultureInfo("en-US");

// For culture-sensitive comparisons, use the String.Compare
// overload that takes a StringComparison value.
int i = String.Compare(first, second, en,
System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {en.Name} returns {i}.");

var de = new System.Globalization.CultureInfo("de-DE");
i = String.Compare(first, second, de,
System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {de.Name} returns {i}.");

bool b = String.Equals(first, second, StringComparison.CurrentCulture);
Console.WriteLine($"The two strings {(b ? "are" : "are not")}) equal.");

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words, en);
showComparison(word, other, en);
showComparison(words, other, en);
void showComparison(string one, string two, System.Globalization.CultureInfo
culture)
{
    int compareLinguistic = String.Compare(one, two, en,
System.Globalization.CompareOptions.None);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using en-US
culture");
    else if (compareLinguistic > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using en-US
culture");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using en-US culture");
    if (compareOrdinal < 0)
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal
comparison");
    else if (compareOrdinal > 0)
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal
comparison");
    else
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using en-US culture");
}

```

```
using ordinal comparison");
}
```

문화권별 비교는 일반적으로 사용자가 입력한 문자열을 사용자가 입력한 다른 문자열과 비교하여 정렬하는데 사용됩니다. 이러한 문자열의 문자 및 정렬 규칙은 사용자 컴퓨터의 로캘에 따라 달라질 수 있습니다. 똑같은 문자가 포함된 문자열이라도 현재 스레드의 문화권에 따라 다르게 정렬될 수 있습니다.

## 배열의 언어적 정렬 및 문자열 검색

다음 예에서는 현재 문화권에 따라 언어 비교를 사용하여 배열에서 문자열을 정렬하고 검색하는 방법을 보여줍니다. [System.StringComparer](#) 매개 변수를 사용하는 정적 [Array](#) 메서드를 사용합니다.

다음 예제에서는 현재 문화권을 사용하여 문자열 배열을 정렬하는 방법을 보여 줍니다.

C#

```
string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"    {s}");
}

Console.WriteLine("\n\nSorted order:");

// Specify Ordinal to demonstrate the different behavior.
Array.Sort(lines, StringComparer.CurrentCulture);

foreach (string s in lines)
{
    Console.WriteLine($"    {s}");
}
```

배열이 정렬되면 이진 검색을 사용하여 항목을 검색할 수 있습니다. 이진 검색은 컬렉션의 중간에서 시작하여 컬렉션의 절반에서 검색 문자열이 포함되어 있는지 확인합니다. 이후의 각 비교는 컬렉션의 나머지 부분을 절반으로 세분합니다. 배열은 [StringComparer.CurrentCulture](#)을 사용하여 정렬됩니다. 로컬 함수 [ShowWhere](#)는 문자열

이 발견된 위치에 대한 정보를 표시합니다. 문자열을 찾을 수 없는 경우 반환된 값은 발견된 그 위치를 나타냅니다.

C#

```
string[] lines = new string[]
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Array.Sort(lines, StringComparer.CurrentCulture);

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = Array.BinarySearch(lines, searchString,
StringComparer.CurrentCulture);
ShowWhere<string>(lines, result);

Console.WriteLine($"{(result > 0 ? "Found" : "Did not find")}{searchString}");

void ShowWhere<T>(T[] array, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
            Console.Write("beginning of sequence and ");
        else
            Console.Write($"{array[index - 1]} and ");

        if (index == array.Length)
            Console.WriteLine("end of sequence.");
        else
            Console.WriteLine($"{array[index]}.");
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}
```

## 컬렉션의 서수 정렬 및 검색

다음 코드에서는 `System.Collections.Generic.List<T>` 컬렉션 클래스를 사용하여 문자열을 저장합니다. 문자열은 `List<T>.Sort` 메서드를 사용하여 정렬됩니다. 이 메서드에는 두 문자열을 비교하고 정렬하는 대리자가 필요합니다. `String.CompareTo` 메서드는 비교 함수를 제공합니다. 이 샘플을 실행하고 순서를 관찰합니다. 이 정렬 작업은 서수 대/소문자 구분 정렬을 사용합니다. 정적 `String.Compare` 메서드를 사용하여 여러 비교 규칙을 지정합니다.

C#

```
List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"    {s}");
}

Console.WriteLine("\n\rSorted order:");

lines.Sort((left, right) => left.CompareTo(right));
foreach (string s in lines)
{
    Console.WriteLine($"    {s}");
}
```

정렬된 후에는 이진 검색을 사용하여 문자열 목록을 검색할 수 있습니다. 다음 샘플에서는 동일한 비교 함수를 사용하여 정렬된 목록을 검색하는 방법을 보여줍니다. 로컬 함수 `ShowWhere`는 검색된 텍스트가 어디에 있는지를 보여줍니다.

C#

```
List<string> lines = new List<string>
{
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
};
lines.Sort((left, right) => left.CompareTo(right));

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = lines.BinarySearch(searchString);
ShowWhere<string>(lines, result);
```

```

Console.WriteLine($"{{(result > 0 ? "Found" : "Did not find")}}
{searchString}");
```

```

void ShowWhere<T>(IList<T> collection, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
            Console.Write("beginning of sequence and ");
        else
            Console.WriteLine($"{collection[index - 1]} and ");

        if (index == collection.Count)
            Console.WriteLine("end of sequence.");
        else
            Console.WriteLine($"{collection[index]}.");
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}

```

정렬 및 검색 시 항상 동일한 형식의 비교를 사용하도록 하세요. 정렬 및 검색에 대해 서로 다른 비교 형식을 사용하면 예기치 않은 결과가 발생합니다.

[System.Collections.Hashtable](#), [System.Collections.Generic.Dictionary<TKey, TValue>](#), [System.Collections.Generic.List<T>](#) 등의 컬렉션 클래스는 요소 또는 키의 형식이 `string`인 경우 [System.StringComparer](#) 매개 변수를 사용하는 생성자를 포함합니다. 일반적으로 가능하면 이러한 생성자를 사용하고 [StringComparer.Ordinal](#) 또는 [StringComparer.OrdinalIgnoreCase](#)를 지정해야 합니다.

## 참고 항목

- [System.Globalization.CultureInfo](#)
- [System.StringComparer](#)
- 문자열
- 문자열 비교
- 애플리케이션 전역화 및 지역화

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# CLS 규격이 아닌 예외를 catch하는 방법

아티클 • 2024. 02. 14.

C++/CLI를 포함한 일부 .NET 언어에서는 개체가 `Exception`에서 파생되지 않은 예외를 `throw`할 수 있습니다. 이러한 예외를 *CLS 규격이 아닌 예외* 또는 *예외가 아닌 항목*이라고 합니다. C#에서는 CLS 규격이 아닌 예외를 `throw`할 수 없지만 다음 두 가지 방법으로 해당 예외를 `catch`할 수 있습니다.

- `catch (RuntimeWrappedException e)` 블록 내에서.

기본적으로 Visual C# 어셈블리는 CLS 규격이 아닌 예외를 래핑된 예외로 `catch`합니다. `RuntimeWrappedException.WrappedException` 속성을 통해 액세스할 수 있는 원래 예외에 액세스해야 할 경우 이 메서드를 사용합니다. 이 항목의 뒤에 나오는 절차에서는 이 방식으로 예외를 `catch`하는 방법을 설명합니다.

- 일반 `catch` 블록(예외 형식이 지정되지 않은 `catch` 블록) 내에서 예외는 다른 모든 `catch` 블록 뒤에 배치됩니다.

CLS 규격이 아닌 예외에 대한 응답으로 일부 작업(예: 로그 파일에 쓰기)을 수행하고자 하고 예외 정보에 액세스할 필요가 없는 경우 이 방법을 사용합니다. 기본적으로 공용 언어 런타임은 모든 예외를 래핑합니다. 이 동작을 사용하지 않으려면 일반적으로 `AssemblyInfo.cs` 파일에 있는 어셈블리 수준 특성 `[assembly:`

`RuntimeCompatibilityAttribute(WrapNonExceptionThrows = false)]`를 코드에 추가합니다.

## CLS 규격이 아닌 예외를 catch하려면

`catch(RuntimeWrappedException e)` 블록 내에서

`RuntimeWrappedException.WrappedException` 속성을 통해 원래 예외에 액세스합니다.

## 예시

다음 예제에서는 C++/CLI로 작성된 클래스 라이브러리에서 `throw`된 CLS 규격이 아닌 예외를 `catch`하는 방법을 보여 줍니다. 이 예제에서 C# 클라이언트 코드는 `throw`되는 예외 형식이 `System.String`이라는 것을 사전에 알고 있습니다. 코드에서 해당 형식에 액세스할 수 있으면 `RuntimeWrappedException.WrappedException` 속성을 다시 원래 형식으로 캐스팅할 수 있습니다.

C#

```
// Class library written in C++/CLI.  
var myClass = new ThrowNonCLS.Class1();  
  
try  
{  
    // throws gcnew System::String(  
    // "I do not derive from System.Exception!");  
    myClass.TestThrow();  
}  
catch (RuntimeWrappedException e)  
{  
    String s = e.WrappedException as String;  
    if (s != null)  
    {  
        Console.WriteLine(s);  
    }  
}
```

## 참고 항목

- [RuntimeWrappedException](#)
- [예외 및 예외 처리](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 특성

아티클 • 2024. 02. 18.

특성은 메타데이터 또는 선언적 정보를 코드(어셈블리, 형식, 메서드, 속성 등)에 연결하는 강력한 방법을 제공합니다. 특성이 프로그램 엔터티와 연결되면 리플렉션이라는 기법을 사용하여 런타임에 특성이 쿼리될 수 있습니다.

특성에는 다음과 같은 속성이 있습니다.

- 특성은 프로그램에 메타데이터를 추가합니다. *메타데이터*는 프로그램에 정의된 형식에 대한 정보를 의미합니다. 모든 .NET 어셈블리에는 어셈블리에 정의된 형식 및 형식 멤버를 설명하는 지정된 메타데이터 집합이 포함됩니다. 필요한 추가 정보를 지정하는 사용자 지정 특성을 추가할 수 있습니다.
- 전체 어셈블리, 모듈 또는 좀 더 작은 프로그램 요소(예: 클래스 및 속성)에 하나 이상의 특성을 적용할 수 있습니다.
- 메서드 및 속성의 경우와 같은 방식으로 특성은 인수를 수락할 수 있습니다.
- 프로그램은 리플렉션을 사용하여 자체 메타데이터 또는 다른 프로그램의 메타데이터를 검사할 수 있습니다.

리플렉션은 어셈블리, 모듈 및 형식을 설명하는 개체(*Type* 형식)를 제공합니다. 리플렉션을 사용하면 동적으로 형식 인스턴스를 만들거나, 형식을 기준 개체에 바인딩하거나, 기준 개체에서 형식을 가져와 해당 메서드를 호출하거나, 필드 및 속성에 액세스할 수 있습니다. 코드에서 특성을 사용하는 경우 리플렉션은 특성에 대한 액세스를 제공합니다. 자세한 내용은 [특성](#)을 참조하세요.

다음은 `GetType()` 메서드(`Object` 기본 클래스의 모든 형식에 상속됨)를 사용하여 변수 형식을 가져오는 간단한 리플렉션 예제입니다.

## ① 참고

`using System;` 및 `using System.Reflection;` 을 .cs 파일의 맨 위에 추가해야 합니다.

C#

```
// Using GetType to obtain type information:  
int i = 42;  
Type type = i.GetType();  
Console.WriteLine(type);
```

출력은 `System.Int32`입니다.

다음 예제에서는 리플렉션을 사용하여 로드된 어셈블리의 전체 이름을 가져옵니다.

C#

```
// Using Reflection to get information of an Assembly:  
Assembly info = typeof(int).Assembly;  
Console.WriteLine(info);
```

출력은 다음과 같습니다. System.Private.CoreLib, Version=7.0.0.0, Culture=neutral,  
PublicKeyToken=7cec85d7bea7798e

### ① 참고

C# 키워드 `protected` 및 `internal`은 IL(중간 언어)에서 아무런 의미가 없으며 리플렉션 API에서 사용되지 않습니다. IL의 해당 용어는 *Family* 및 *Assembly*입니다. 리플렉션을 사용하는 `internal` 메서드를 식별하려면 `IsAssembly` 속성을 사용합니다. `protected internal` 메서드를 식별하려면 `IsFamilyOrAssembly`를 사용합니다.

## 특성 사용

특정 특성은 유효한 선언 형식이 제한적일 수 있지만 대부분의 선언에 특성을 사용할 수 있습니다. C#에서는 특성 이름을 대괄호([])로 묶어 적용하고자 하는 엔터티의 선언 위에 배치하여 특성을 지정합니다.

이 예제에서 `SerializableAttribute` 특성은 클래스에 특정 특성을 적용하는 데 사용됩니다.

C#

```
[Serializable]  
public class SampleClass  
{  
    // Objects of this type can be serialized.  
}
```

`DllImportAttribute` 특성을 사용하는 메서드는 다음 예제와 같이 선언됩니다.

C#

```
[System.Runtime.InteropServices.DllImport("user32.dll")]  
extern static void SampleMethod();
```

다음 예제와 같이 둘 이상의 특성을 하나의 선언에 추가할 수 있습니다.

C#

```
void MethodA([In][Out] ref double x) { }
void MethodB([Out][In] ref double x) { }
void MethodC([In, Out] ref double x) { }
```

지정된 엔터티에 대해 일부 특성을 두 번 이상 지정할 수 있습니다. 이러한 다용도 특성의 예로 [ConditionalAttribute](#)가 있습니다.

C#

```
[Conditional("DEBUG"), Conditional("TEST1")]
void TraceMethod()
{
    // ...
}
```

### ① 참고

규칙에 따라 모든 특성 이름은 .NET 라이브러리의 다른 항목과 구분하기 위해 "Attribute" 단어로 끝납니다. 그러나 코드에서 특성을 사용하는 경우 특성 접미사를 지정할 필요가 없습니다. 예를 들어 `[DllImport]` 는 `[DllImportAttribute]` 와 같지만, `DllImportAttribute` 는 .NET 클래스 라이브러리에서 특성의 실제 이름입니다.

## 특성 매개 변수

많은 특성에는 매개 변수를 위치, 명명되지 않은 또는 명명된 상태의 매개 변수가 있을 수 있습니다. 위치 매개 변수를 특정 순서로 지정해야 하며 생략할 수는 없습니다. 명명된 매개 변수는 선택 사항이며 순서에 관계 없이 지정할 수 있습니다. 위치 매개 변수가 가장 먼저 지정됩니다. 예를 들어 다음 세 가지 특성은 동급입니다.

C#

```
[DllImport("user32.dll")]
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

첫 번째 매개 변수인 DLL 이름은 위치 매개 변수이므로 항상 맨 먼저 오고 나머지 매개 변수가 지정됩니다. 이 경우 명명된 두 매개 변수는 기본적으로 `false`이므로 생략할 수 있습니다. 위치 매개 변수는 특성 생성자의 매개 변수에 해당합니다. 명명된 또는 선택적 매개 변수는 특성의 속성 또는 필드에 해당합니다. 기본 매개 변수 값에 대한 자세한 내용은 개별 특성의 설명서를 참조하세요.

허용되는 매개 변수 형식에 대한 자세한 내용은 [C# 언어 사양의 특성 섹션](#)을 참조하세요.

# 특성 대상

특성의 대상은 특성이 적용되는 엔터티입니다. 예를 들어 특성은 클래스, 특정 메서드 또는 전체 어셈블리에 적용될 수 있습니다. 기본적으로 특성은 그 뒤에 오는 요소에 적용됩니다. 하지만 특성이 메서드, 해당 매개 변수 또는 해당 반환 값 중 어디에 적용될지를 명시적으로 지정할 수 있습니다.

특성 대상을 명시적으로 식별하려면 다음 구문을 사용합니다.

C#

```
[target : attribute-list]
```

가능한 `target` 값 목록은 다음 표에 나와 있습니다.

 테이블 확장

대상 값	적용 대상
<code>assembly</code>	전체 어셈블리
<code>module</code>	현재 어셈블리 모듈
<code>field</code>	클래스 또는 구조체의 필드
<code>event</code>	이벤트
<code>method</code>	메서드 또는 <code>get</code> 및 <code>set</code> 속성 접근자
<code>param</code>	메서드 매개 변수 또는 <code>set</code> 속성 접근자 매개 변수
<code>property</code>	속성
<code>return</code>	메서드, 속성 인덱서 또는 <code>get</code> 속성 접근자의 반환 값
<code>type</code>	구조체, 클래스, 인터페이스, 열거형 또는 대리자

[자동 구현 속성](#)에 대해 생성된 지원 필드에 특성을 적용하기 위해 `field` 대상 값을 지정합니다.

다음 예제에서는 어셈블리와 모듈에 특성을 적용하는 방법을 보여 줍니다. 자세한 내용은 [공통 특성\(C#\)](#)을 참조하세요.

C#

```
using System;
using System.Reflection;
```

```
[assembly: AssemblyTitle("Production assembly 4")]
[module: CLSCompliant(true)]
```

다음 예제에서는 C#에서 메서드, 메서드 매개 변수 및 메서드 반환 값에 특성을 적용하는 방법을 보여 줍니다.

C#

```
// default: applies to method
[ValidatedContract]
int Method1() { return 0; }

// applies to method
[method: ValidatedContract]
int Method2() { return 0; }

// applies to parameter
int Method3([ValidatedContract] string contract) { return 0; }

// applies to return value
[return: ValidatedContract]
int Method4() { return 0; }
```

### ① 참고

`ValidatedContract`이 정의되는 대상이 유효한지에 상관없이, 반환 값에만 적용하도록 `ValidatedContract`이 정의된 경우에도 `return` 대상을 지정해야 합니다. 즉, 컴파일러는 모호한 특성 대상을 확인하기 위해 `AttributeUsage` 정보를 사용하지 않습니다. 자세한 내용은 `AttributeUsage`를 참조하세요.

## 특성의 일반적인 용도

다음 목록에는 코드에서 특성이 사용되는 일반적인 경우가 나와 있습니다.

- SOAP 프로토콜을 통해 메서드를 호출할 수 있음을 나타내기 위해 웹 서비스에서 `WebMethod` 특성을 사용하여 메서드에 표시. 자세한 내용은 `WebMethodAttribute`를 참조하세요.
- 네이티브 코드와 상호 운용될 경우 메서드 매개 변수를 마샬링하는 방법 설명. 자세한 내용은 `MarshalAsAttribute`를 참조하세요.
- 클래스, 메서드 및 인터페이스에 대한 COM 속성 설명
- `DllImportAttribute` 클래스를 사용하는 비관리 코드 호출
- 제목, 버전, 설명 또는 상표를 기준으로 어셈블리 설명
- 지속성을 위해 `serialize`할 클래스의 멤버 설명

- XML serialization를 위해 클래스 멤버 및 XML 노드 간을 매핑하는 방법 설명
- 메서드에 대한 보안 요구 사항 설명
- 보안을 적용하는 데 사용되는 특징 지정
- 코드를 쉽게 디버그할 수 있도록 하기 위해 JIT(Just-In-Time) 컴파일러를 통해 최적화 제어
- 메서드 호출자에 대한 정보 얻기

## 리플렉션 개요

리플렉션은 다음과 같은 상황에서 유용합니다.

- 프로그램 메타데이터의 특성에 액세스해야 하는 경우. 자세한 내용은 [특성에 저장된 정보 검색](#)을 참조하세요.
- 어셈블리에서 형식을 검사하고 인스턴스화하려는 경우.
- 런타임에 새 형식을 빌드하려는 경우. [System.Reflection.Emit](#)의 클래스를 사용합니다.
- 런타임에 바인딩을 수행하고 런타임에 생성된 형식의 메서드에 액세스하려는 경우. [동적으로 형식 로드 및 사용](#) 문서를 참조하세요.

## 관련 단원

자세한 내용은 다음에서 확인합니다.

- [공통 특성\(C#\)](#)
- [호출자 정보\(C#\)](#)
- [특성](#)
- [리플렉션](#)
- [형식 정보 보기](#)
- [리플렉션 및 제네릭 형식](#)
- [System.Reflection.Emit](#)
- [특성에 저장된 정보 검색](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

☞ 설명서 문제 열기

☞ 제품 사용자 의견 제공

# 사용자 지정 특성 만들기

아티클 • 2024. 02. 13.

메타데이터를 통해 특성의 정의를 빠르고 쉽게 식별할 수 있도록 해주는 `Attribute`로부터 직접적으로 또는 간접적으로 상속한 특성 클래스를 정의하여 사용자 지정 특성을 만들 수 있습니다. 형식을 작성한 프로그래머의 이름을 형식에 태그로 지정한다고 가정해봅시다. 사용자 지정 `Author` 특성 클래스를 아래와 같이 정의할 수 있습니다.

C#

```
[System.AttributeUsage(System.AttributeTargets.Class |  
    System.AttributeTargets.Struct)  
]  
public class AuthorAttribute : System.Attribute  
{  
    private string Name;  
    public double Version;  
  
    public AuthorAttribute(string name)  
    {  
        Name = name;  
        Version = 1.0;  
    }  
}
```

클래스 이름 `AuthorAttribute`는 특성의 이름인 `Author`와 `Attribute` 접미사입니다.

`System.Attribute`를 상속하므로 사용자 지정 특성 클래스입니다. 생성자의 매개 변수는 사용자 지정 특성의 위치 매개 변수입니다. 이 예제에서는 `name`이 위치 매개 변수입니다. 모든 `public` 읽기-쓰기 필드 또는 속성은 명명된 매개 변수입니다. 이 경우에는 `version`이 유일한 명명된 매개 변수입니다. 클래스 및 `struct` 선언에서만 `Author` 특성을 유효하게 설정하려면 `AttributeUsage` 특성을 사용해야 합니다.

이 새로운 특성은 다음과 같이 사용할 수 있습니다.

C#

```
[Author("P. Ackerman", Version = 1.1)]  
class SampleClass  
{  
    // P. Ackerman's code goes here...  
}
```

`AttributeUsage`에는 사용자 지정 특성을 한 번 또는 여러 번 사용하도록 설정하기 위해 사용하는 명명된 매개 변수인 `AllowMultiple`이 있습니다. 다음 코드 예제에서는 다중 사

용 특성을 만듭니다.

C#

```
[System.AttributeUsage(System.AttributeTargets.Class |  
                      System.AttributeTargets.Struct,  
                      AllowMultiple = true) // Multiuse attribute.  
]  
public class AuthorAttribute : System.Attribute  
{  
    string Name;  
    public double Version;  
  
    public AuthorAttribute(string name)  
    {  
        Name = name;  
  
        // Default value.  
        Version = 1.0;  
    }  
  
    public string GetName() => Name;  
}
```

다음 코드 예제에서는 같은 형식의 여러 특성이 한 클래스에 적용됩니다.

C#

```
[Author("P. Ackerman"), Author("R. Koch", Version = 2.0)]  
public class ThirdClass  
{  
    // ...  
}
```

## 참고 항목

- [System.Reflection](#)
- [사용자 지정 특성 작성](#)
- [AttributeUsage\(C#\)](#)

 GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

수도 있습니다. 자세한 내용은  
참여자 가이드를 참조하세요.

☞ 설명서 문제 열기

☞ 제품 사용자 의견 제공

# 리플렉션을 사용하여 특성 액세스

아티클 • 2024. 02. 17.

어느 정도 해당 정보를 검색하고 이에 따라 작업을 수행하지 않는다면 사용자 지정 특성을 정의하고 소스 코드에 배치할 수 있다는 사실은 별로 중요하지 않습니다. 리플렉션을 통해 사용자 지정 특성을 사용하여 정의된 정보를 검색할 수 있습니다. 핵심 메서드는 소스 코드 특성에 해당하는 런타임 항목인 개체의 배열을 반환하는 `GetCustomAttributes`입니다. 이 메서드에는 오버로드된 버전이 많이 있습니다. 자세한 내용은 [Attribute](#)를 참조하세요.

다음과 같은 특성 사양은

C#

```
[Author("P. Ackerman", Version = 1.1)]
class SampleClass { }
```

(은)는 개념적으로 다음 코드와 동일합니다.

C#

```
var anonymousAuthorObject = new Author("P. Ackerman")
{
    Version = 1.1
};
```

그러나 `SampleClass`(이)가 특성에 대해 쿼리될 때까지 코드가 실행되지 않습니다.

`SampleClass`에서 `GetCustomAttributes`(을)를 호출하면 `Author` 개체가 생성되고 초기화됩니다. 클래스에 다른 특성이 있으면 다른 특성 개체가 비슷하게 구성됩니다. 그런 다음 `GetCustomAttributes`는 `Author` 개체 및 기타 특성 개체를 배열로 반환합니다. 이 배열을 반복하고, 각 배열 요소의 형식에 따라 적용된 특성을 확인하고, 특성 개체에서 정보를 추출할 수 있습니다.

다음은 전체 예제입니다. 사용자 지정 특성이 정의되어 있고 여러 엔터티에 적용되었으며 리플렉션을 통해 검색합니다.

C#

```
// Multiuse attribute.
[System.AttributeUsage(System.AttributeTargets.Class |
    System.AttributeTargets.Struct,
    AllowMultiple = true) // Multiuse attribute.
]
public class AuthorAttribute : System.Attribute
```

```
{  
    string Name;  
    public double Version;  
  
    public AuthorAttribute(string name)  
    {  
        Name = name;  
  
        // Default value.  
        Version = 1.0;  
    }  
  
    public string GetName() => Name;  
}  
  
// Class with the Author attribute.  
[Author("P. Ackerman")]  
public class FirstClass  
{  
    // ...  
}  
  
// Class without the Author attribute.  
public class SecondClass  
{  
    // ...  
}  
  
// Class with multiple Author attributes.  
[Author("P. Ackerman"), Author("R. Koch", Version = 2.0)]  
public class ThirdClass  
{  
    // ...  
}  
  
class TestAuthorAttribute  
{  
    public static void Test()  
    {  
        PrintAuthorInfo(typeof(FirstClass));  
        PrintAuthorInfo(typeof(SecondClass));  
        PrintAuthorInfo(typeof(ThirdClass));  
    }  
  
    private static void PrintAuthorInfo(System.Type t)  
    {  
        System.Console.WriteLine($"Author information for {t}");  
  
        // Using reflection.  
        System.Attribute[] attrs = System.Attribute.GetCustomAttributes(t);  
        // Reflection.  
  
        // Displaying output.  
        foreach (System.Attribute attr in attrs)  
        {
```

```
        if (attr is AuthorAttribute a)
    {
        System.Console.WriteLine($"    {a.GetName()}, version
{a.Version:f}");
    }
}
*/
/* Output:
Author information for FirstClass
P. Ackerman, version 1.00
Author information for SecondClass
Author information for ThirdClass
R. Koch, version 2.00
P. Ackerman, version 1.00
*/
```

## 참고 항목

- [System.Reflection](#)
- [Attribute](#)
- [특성에 저장된 정보 검색](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# C#에서 특성을 사용하여 C/C++ 공용 구조체를 만드는 방법

아티클 • 2024. 02. 17.

특성을 사용하여 메모리에서 구조체가 레이아웃되는 방식을 필요에 맞게 변경할 수 있습니다. 예를 들어 `StructLayout(LayoutKind.Explicit)` 및 `FieldOffset` 특성을 사용하여 C/C++에서 공용 구조체로 알려진 항목을 만들 수 있습니다.

이 코드 세그먼트에서 `TestUnion`의 모든 필드는 메모리의 같은 위치에서 시작합니다.

C#

```
[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestUnion
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public double d;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public char c;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public byte b;
}
```

다음 코드는 명시적으로 설정된 다른 위치에서 필드가 시작하는 또 다른 예제입니다.

C#

```
[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestExplicit
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public long lg;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i1;

    [System.Runtime.InteropServices.FieldOffset(4)]
    public int i2;

    [System.Runtime.InteropServices.FieldOffset(8)]
    public double d;

    [System.Runtime.InteropServices.FieldOffset(12)]
}
```

```
public char c;

[System.Runtime.InteropServices.FieldOffset(14)]
public byte b;
}
```

두 개의 정수 필드 `i1` 및 `i2`는 함께 `lg`와 동일한 메모리 위치를 공유합니다. `lg`가 처음 8 바이트를 사용하거나, `i1`이 처음 4바이트를 사용하고 `i2`가 다음 4바이트를 사용합니다. 구조체 레이아웃에 대한 이러한 종류의 제어는 플랫폼 호출을 사용할 때 유용합니다.

## 참고 항목

- [System.Reflection](#)
- [Attribute](#)
- [특성](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# 제네릭 및 특성

아티클 • 2024. 02. 18.

특성은 제네릭이 아닌 형식과 동일한 방식으로 제네릭 형식에 적용할 수 있습니다. 그러나 부분적으로 생성된 제네릭 형식이 아닌 개방형 제네릭 형식 및 닫힌 생성된 제네릭 형식에만 특성을 적용할 수 있습니다. 개방형 제네릭 형식은 형식 인수가 지정되지 않는 형식입니다(예: `Dictionary< TKey, TValue >` 하나의 닫힌 생성된 제네릭 형식이 `Dictionary< string, object >` 와(과) 같은 모든 형식 인수를 지정) 부분적으로 생성된 제네릭 형식은 일부 형식 인수를 지정하지만 전부는 아닙니다. 예제는 `Dictionary< string, TValue >` 입니다.

다음 예에서는 이러한 사용자 지정 특성을 사용합니다.

C#

```
class CustomAttribute : Attribute
{
    public object? info;
}
```

특성은 개방형 제네릭 형식을 참조할 수 있습니다.

C#

```
public class GenericClass1<T> { }

[CustomAttribute(info = typeof(GenericClass1<>))]
class ClassA { }
```

적절한 수의 쉼표를 사용하여 여러 형식 매개 변수를 지정합니다. 이 예제에서 `GenericClass2` 는 두 개의 형식 매개 변수를 가집니다.

C#

```
public class GenericClass2<T, U> { }

[CustomAttribute(info = typeof(GenericClass2<, >))]
class ClassB { }
```

특성은 폐쇄형으로 생성된 제네릭 형식을 참조할 수 있습니다.

C#

```
public class GenericClass3<T, U, V> { }

[CustomAttribute(info = typeof(GenericClass3<int, double, string>))]
class ClassC { }
```

제네릭 형식 매개 변수를 참조하는 특성으로 인해 컴파일 시간 오류가 발생합니다.

C#

```
[CustomAttribute(info = typeof(GenericClass3<int, T, string>))] //Error
CS0416
class ClassD<T> { }
```

C# 11부터 제네릭 형식은 [Attribute](#)에서 상속할 수 있습니다.

C#

```
public class CustomGenericAttribute<T> : Attribute { } //Requires C# 11
```

런타임에 제네릭 형식 또는 형식 매개 변수에 대한 정보를 얻으려면 [System.Reflection](#)의 메서드를 사용합니다. 자세한 내용은 [제네릭 및 리플렉션](#)을 참조하세요.

## 참고 항목

- [제네릭](#)
- [특성](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# 리플렉션을 사용하여 어셈블리의 메타데이터를 쿼리하는 방법(LINQ)

아티클 • 2024. 02. 21.

.NET 리플렉션 API를 사용하여 .NET 어셈블리의 메타데이터를 검사하고 해당 어셈블리에 있는 형식, 형식 멤버 및 매개 변수의 컬렉션을 만듭니다. 이러한 컬렉션은 제네릭 `IEnumerable<T>` 인터페이스를 지원하므로 LINQ를 사용하여 쿼리할 수 있습니다.

다음 예제에서는 리플렉션과 함께 LINQ를 사용하여 지정된 검색 조건과 일치하는 메서드에 대한 특정 메타데이터를 검색하는 방법을 보여 줍니다. 이 경우 쿼리는 배열과 같은 열거형 형식을 반환하는 모든 메서드의 이름을 어셈블리에서 검색합니다.

C#

```
Assembly assembly = Assembly.Load("System.Private.CoreLib, Version=7.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e");
var pubTypesQuery = from type in assembly.GetTypes()
                    where type.IsPublic
                    from method in type.GetMethods()
                    where method.ReturnType.isArray == true
                        || (method.ReturnType.GetInterface(
                            typeof(System.Collections.Generic.IEnumerable<>)).FullName!) != null
                            && method.ReturnType.FullName != "System.String")
                    group method.ToString() by type.ToString();

foreach (var groupOfMethods in pubTypesQuery)
{
    Console.WriteLine("Type: {0}", groupOfMethods.Key);
    foreach (var method in groupOfMethods)
    {
        Console.WriteLine("  {0}", method);
    }
}
```

이 예제에서는 `Assembly.GetTypes` 메서드를 사용하여 지정된 어셈블리의 형식 배열을 반환합니다. `public` 형식만 반환되도록 `where` 필터가 적용됩니다. 각 `public` 형식에 대해 `Type.GetMethods` 호출에서 반환된 `MethodInfo` 배열을 사용하여 하위 쿼리가 생성됩니다. 이러한 결과는 해당 반환 형식이 배열이거나 `IEnumerable<T>`을 구현하는 형식인 메서드만 반환하도록 필터링됩니다. 마지막으로, 이러한 결과는 형식 이름을 키로 사용하여 그룹화됩니다.

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 제네릭 및 리플렉션

아티클 • 2024. 02. 17.

CLR(공용 언어 런타임)은 런타임에 제네릭 형식 정보에 액세스할 수 있으므로 제네릭이 아닌 형식에 대한 방법과 동일한 방법으로 리플렉션을 사용하여 제네릭 형식에 대한 정보를 가져올 수 있습니다. 자세한 내용은 [런타임의 제네릭](#)을 참조하세요.

`System.Reflection.Emit` 네임스페이스에도 제네릭을 지원하는 새 멤버가 포함되어 있습니다. 방법: 리플렉션 내보내기를 사용하여 제네릭 형식 정의를 참조하세요.

제네릭 리플렉션에 사용되는 용어의 고정 조건 목록은 [IsGenericType](#) 속성 설명을 참조하세요.

- [IsGenericType](#): 형식이 제네릭이면 `true`를 반환합니다.
- [GetGenericArguments](#): 생성된 형식에 제공된 형식 인수 또는 제네릭 형식 정의의 형식 매개 변수를 나타내는 `Type` 개체의 배열을 반환합니다.
- [GetGenericTypeDefinition](#): 현재 생성된 유형에 대한 기본 제네릭 형식 정의를 반환합니다.
- [GetGenericParameterConstraints](#): 현재 제네릭 형식 매개 변수에 대한 제약 조건을 나타내는 `Type` 개체의 배열을 반환합니다.
- [ContainsGenericParameters](#): 형식 또는 바깥쪽 형식이나 메서드에 제공되지 않은 특정 형식에 대한 형식 매개 변수가 포함된 경우 `true`를 반환합니다.
- [GenericParameterAttributes](#): 현재 제네릭 형식 매개 변수의 특수 제약 조건을 설명하는 `GenericParameterAttributes` 플래그의 조합을 가져옵니다.
- [GenericParameterPosition](#): `Type` 개체가 형식 매개 변수를 나타내는 경우 형식 매개 변수를 선언한 제네릭 형식 정의 또는 제네릭 메서드 정의의 형식 매개 변수 목록에서 해당 형식 매개 변수의 위치를 가져옵니다.
- [IsGenericParameter](#): 현재 `Type`이(가) 제네릭 형식 또는 메서드 정의의 형식 매개 변수를 나타내는지를 나타내는 값을 가져옵니다.
- [IsGenericTypeDefinition](#): 현재 `Type`이(가) 다른 제네릭 형식을 생성하는 데 사용될 수 있는 제네릭 형식 정의를 나타내는지 여부를 가리키는 값을 가져옵니다. 형식이 제네릭 형식의 정의를 나타내는 경우 `true`를 반환합니다.
- [DeclaringMethod](#): 현재 제네릭 형식 매개 변수를 정의한 제네릭 메서드를 반환하거나 형식 매개 변수가 제네릭 메서드에 의해 정의되지 않은 경우 `null`을 반환합니다.
- [MakeGenericType](#): 형식 배열의 요소를 현재 제네릭 형식 정의의 형식 매개 변수로 대체하며 생성된 형식을 나타내는 `Type` 개체를 반환합니다.

또한 `MethodInfo` 클래스에 새 멤버는 제네릭 메서드에 대한 런타임 정보를 사용하도록 설정합니다. 제네릭 메서드를 반영하는 데 사용되는 용어에 대한 고정 조건 목록은 [IsGenericMethod](#) 속성 설명을 참조하세요.

- [IsGenericMethod](#): 메서드가 제네릭인 경우 true를 반환합니다.
- [GetGenericArguments](#): 생성된 제네릭 메서드의 형식 인수나 제네릭 메서드 정의의 형식 매개 변수를 나타내는 Type 개체의 배열을 반환합니다.
- [GetGenericMethodDefinition](#): 현재 생성된 메서드에 대한 기본 제네릭 메서드 정의를 반환합니다.
- [ContainsGenericParameters](#): 메서드 또는 바깥쪽 형식에 제공되지 않은 특정 형식에 대한 형식 매개 변수가 포함된 경우 true를 반환합니다.
- [IsGenericMethodDefinition](#): 현재 MethodInfo이(가) 제네릭 메서드의 정의를 나타내는 경우 true를 반환합니다.
- [MakeGenericMethod](#): 현재 제네릭 메서드 정의의 형식 매개 변수를 형식 배열의 요소로 대체하고, 결과로 생성된 메서드를 나타내는 MethodInfo 개체를 반환합니다.

## 참고 항목

- [제네릭](#)
- [리플렉션 및 제네릭 형식](#)
- [제네릭](#)

### ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

💡 설명서 문제 열기

✍️ 제품 사용자 의견 제공

# 사용자 지정 특성 정의 및 읽기

아티클 • 2024. 02. 17.

특성은 선언적으로 정보를 코드와 연결하는 방법을 제공합니다. 또한 다양한 대상에 적용할 수 있는 재사용 가능한 요소를 제공할 수도 있습니다. [ObsoleteAttribute](#)를 고려합니다. 이 특성은 클래스, 구조체, 메서드, 생성자 등에 적용될 수 있으며 요소가 더 이상 필요하지 않다는 사실을 선언합니다. 이 특성을 찾고 응답으로 특정 작업을 수행하는 것은 모두 C# 컴파일러가 진행합니다.

이 자습서에서는 코드에 특성을 추가하는 방법, 고유한 특성을 만들고 사용하는 방법, .NET에 기본 제공되는 일부 특성을 사용하는 방법을 알아봅니다.

## 필수 조건

.NET을 실행하려면 컴퓨터를 설정해야 합니다. [.NET 다운로드](#) 페이지에서 설치 지침을 찾을 수 있습니다. Windows, Ubuntu Linux, macOS 또는 Docker 컨테이너에서 이 애플리케이션을 실행할 수 있습니다. 원하는 코드 편집기를 설치해야 합니다. 다음 설명에서는 오픈 소스 플랫폼 간 편집기인 [Visual Studio Code](#)를 사용합니다. 그러나 자신에게 편리한 도구를 사용할 수 있습니다.

## 앱 만들기

이제 모든 도구를 설치했으므로 새 .NET 콘솔 앱을 만듭니다. 명령줄 생성기를 사용하려면 즐겨 사용하는 셀에서 다음 명령을 실행합니다.

.NET CLI

```
dotnet new console
```

이 명령은 기본 .NET 프로젝트 파일을 만듭니다. `dotnet restore`를 실행하여 이 프로젝트를 컴파일하는 데 필요한 종속성을 복원합니다.

`dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish` 및 `dotnet pack` 등 복원이 필요한 모든 명령에 의해 암시적으로 실행되므로 `dotnet restore`를 실행할 필요가 없습니다. 암시적 복원을 사용하지 않으려면 `--no-restore` 옵션을 사용합니다.

`dotnet restore` 명령은 [Azure DevOps Services](#)의 연속 통합 빌드 또는 복원 발생 시점을 명시적으로 제어해야 하는 빌드 시스템과 같이 명시적으로 복원이 가능한 특정 시나리오에서 여전히 유용합니다.

NuGet 피드를 관리하는 방법에 대한 자세한 내용은 [dotnet restore 설명서](#)를 참조하세요.

이 프로그램을 실행하려면 `dotnet run`을 사용합니다. 콘솔에 "Hello, World" 출력이 표시됩니다.

## 코드에 특성 추가

C#에서 특성은 `Attribute` 기본 클래스에서 상속되는 클래스입니다. `Attribute`에서 상속되는 모든 클래스는 코드의 다른 부분에서 일종의 "태그"로 사용될 수 있습니다. 예를 들어, `ObsoleteAttribute`라는 특성이 있습니다. 이 특성은 코드가 더 이상 사용되지 않으며 더 이상 사용되어서는 안 된다는 신호입니다. 예를 들어 대괄호를 사용하여 클래스에 이 특성을 배치합니다.

C#

```
[Obsolete]
public class MyClass
{}
```

클래스 이름은 `ObsoleteAttribute`이지만 코드에서는 `[Obsolete]`만 사용하면 됩니다. 대부분의 C# 코드는 이 규칙을 따릅니다. 원활 경우 전체 이름 `[ObsoleteAttribute]`를 사용할 수 있습니다.

클래스를 더 이상 사용되지 않는 것으로 표시할 경우 더 이상 사용되지 않는 이유 및/또는 대신 사용할 항목에 대한 정보를 제공하는 것이 좋습니다. 이 설명을 제공하려면 `Obsolete` 특성에 문자열 매개 변수를 포함합니다.

C#

```
[Obsolete("ThisClass is obsolete. Use ThisClass2 instead.")]
public class ThisClass
{}
```

문자열은 마치 `var attr = new ObsoleteAttribute("some string")`을 작성하는 것처럼 `ObsoleteAttribute` 생성자에 인수로 전달됩니다.

특성 생성자에 대한 매개 변수는 단순 형식/리터럴인 `bool, int, double, string, Type, enums, etc` 및 해당 형식의 배열로 제한됩니다. 식이나 변수를 사용할 수 없습니다. 위치 매개 변수나 명명된 매개 변수를 자유롭게 사용할 수 있습니다.

# 고유한 특성 만들기

Attribute 기본 클래스에서 상속되는 새 클래스를 정의하여 특성을 만듭니다.

C#

```
public class MySpecialAttribute : Attribute
{
}
```

앞의 코드를 사용하면 코드베이스의 다른 곳에서 [MySpecial] (또는 [MySpecialAttribute])를 특성으로 사용할 수 있습니다.

C#

```
[MySpecial]
public class SomeOtherClass
{}
```

ObsoleteAttribute 트리거 같은 .NET 기본 클래스 라이브러리의 특성은 컴파일러 내에 특정 동작을 포함합니다. 그러나 만드는 특성은 메타데이터의 역할만 수행하며 특성 클래스 내의 코드는 실행되지 않습니다. 코드의 다른 곳에서 해당 메타데이터에 대해 조치를 취하는 것은 사용자에게 달려 있습니다.

여기서 주의해야 할 '문제'가 있습니다. 앞서 언급했듯이 특성을 사용할 때 특정 형식만 인수로 전달될 수 있습니다. 그러나 특성 유형을 만들 때 C# 컴파일러는 사용자가 해당 매개 변수를 만들지 못하게 하지 않습니다. 다음 예에서는 올바르게 컴파일되는 생성자를 사용하여 특성을 만들었습니다.

C#

```
public class GotchaAttribute : Attribute
{
    public GotchaAttribute(Foo myClass, string str)
    {
    }
}
```

그러나 이 생성자를 특성 구문과 함께 사용할 수는 없습니다.

C#

```
[Gotcha(new Foo(), "test")] // does not compile
public class AttributeFail
```

```
{  
}
```

앞의 코드는 `Attribute constructor parameter 'myClass' has type 'Foo', which is not a valid attribute parameter type`과 같은 컴파일러 오류를 발생시킵니다.

## 특성 사용을 제한하는 방법

특성은 다음 "대상"에 사용할 수 있습니다. 앞의 예에서는 클래스에 대해 보여 주지만 다음에서도 사용할 수 있습니다.

- 어셈블리
- 클래스
- 생성자
- 대리인
- 열거형
- 이벤트
- 필드
- GenericParameter
- 인터페이스
- 메서드
- 모듈
- 매개 변수
- 속성
- ReturnValue
- 구조체

특성 클래스를 만들 때 기본적으로 C#을 사용하면 가능한 모든 특성 대상에서 해당 특성을 사용할 수 있습니다. 특성을 특정 대상으로 제한하려는 경우 특성 클래스에 대해 `AttributeUsageAttribute`를 사용하면 됩니다. 맞습니다. 특성에 대한 특성입니다.

C#

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]  
public class MyAttributeForClassAndStructOnly : Attribute  
{  
}
```

클래스 또는 구조체 이외의 항목에 대해 위 특성을 적용하려고 하면 `Attribute 'MyAttributeForClassAndStructOnly' is not valid on this declaration type. It is only valid on 'class, struct' declarations`와 같은 컴파일러 오류가 발생합니다.

C#

```
public class Foo
{
    // if the below attribute was uncommented, it would cause a compiler
    // error
    // [MyAttributeForClassAndStructOnly]
    public Foo()
    {
    }
}
```

## 코드 요소에 연결된 특성을 사용하는 방법

특성은 메타데이터로 작동합니다. 외부의 힘이 없으면 실제로 아무것도 하지 않습니다.

특성을 찾아서 작업하려면 리플렉션이 필요합니다. 리플렉션을 사용하면 다른 코드를 검사하는 코드를 C#으로 작성할 수 있습니다. 예를 들어 리플렉션을 사용하여 클래스에 대한 정보(코드 헤드에 `using System.Reflection;` 추가)를 가져올 수 있습니다.

C#

```
TypeInfo typeInfo = typeof(MyClass).GetTypeInfo();
Console.WriteLine("The assembly qualified name of MyClass is " +
typeInfo.AssemblyQualifiedName);
```

다음과 같이 인쇄됩니다. `The assembly qualified name of MyClass is`

```
ConsoleApplication.MyClass, attributes, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null
```

`TypeInfo` 객체(또는 `MethodInfo`, `FieldInfo` 또는 기타 객체)가 있으면 `GetCustomAttributes` 메서드를 사용할 수 있습니다. 이 메서드는 `Attribute` 객체의 컬렉션을 반환합니다. `GetCustomAttribute`를 사용하고 특성 유형을 지정할 수도 있습니다.

`MyClass`의 `MethodInfo` 인스턴스에 대해 `GetCustomAttributes`를 사용하는 예제는 다음과 같습니다(앞에서 살펴본 `[Obsolete]` 특성을 포함하는 예제).

C#

```
var attrs = typeInfo.GetCustomAttributes();
foreach(var attr in attrs)
    Console.WriteLine("Attribute on MyClass: " + attr.GetType().Name);
```

콘솔에 인쇄됩니다. `Attribute on MyClass: ObsoleteAttribute`. `MyClass`에 다른 특성을 추가해 보세요.

이러한 `Attribute` 개체가 지연되어 인스턴스화되는 것에 유의해야 합니다. 즉, `GetCustomAttribute` 또는 `GetCustomAttributes`를 사용할 때까지 인스턴스화되지 않습니다. 또한 매번 인스턴스화됩니다. `GetCustomAttributes`를 연속으로 두 번 호출하면 `ObsoleteAttribute`의 서로 다른 두 인스턴스가 반환됩니다.

## 런타임의 공통 특성

특성은 많은 도구 및 프레임워크에서 사용됩니다. NUnit은 NUnit Test Runner에서 사용되는 `[Test]` 및 `[TestFixture]` 같은 특성을 사용합니다. ASP.NET MVC는 `[Authorize]`와 같은 특성을 사용하고 MVC 작업에 대해 크로스 커팅(Cross-Cutting) 문제를 해결하기 위한 작업 필터 프레임워크를 제공합니다. PostSharp<sup>↗</sup>은 특성 구문을 사용하여 C#을 사용한 AOP(Aspect-Oriented Programming)를 허용합니다.

.NET Core 기본 클래스 라이브러리에 기본 제공되는 몇 가지 유의할 만한 특성은 다음과 같습니다.

- `[Obsolete]`. 이 특성은 위 예제에서 사용되었으며 `System` 네임스페이스에 있습니다. 기본 코드를 변경하는 방법에 대해 선언적 설명서를 제공하는 것이 유용합니다. 메시지는 문자열의 형태로 제공될 수 있으며 다른 부울 매개 변수가 컴파일러 경고를 컴파일러 오류로 에스컬레이션하는 데 사용될 수 있습니다.
- `[Conditional]`. 이 특성은 `System.Diagnostics` 네임스페이스에 있습니다. 이 특성은 메서드(또는 특성 클래스)에 적용할 수 있습니다. 생성자에는 문자열을 전달해야 합니다. 해당 문자열이 `#define` 지시문과 일치하지 않으면 C# 컴파일러는 해당 메서드에 대한 모든 호출을 제거합니다(메서드 자체는 제외). 일반적으로 디버깅(진단) 목적으로 이 기술을 사용합니다.
- `[CallerMemberName]`. 이 특성은 매개 변수에 사용될 수 있으며 `System.Runtime.CompilerServices` 네임스페이스에 있습니다. `CallerMemberName`은 다른 메서드를 호출하는 메서드의 이름을 삽입하는 데 사용되는 특성입니다. 다양한 UI 프레임워크에서 `INotifyPropertyChanged`를 구현할 때 '매직의 문자열'을 제거하는 방법입니다. 예를 들어

C#

```
public class MyUIClass : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler? PropertyChanged;

    public void RaisePropertyChanged([CallerMemberName] string propertyName =
= default!)
    {
        PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
    }
}
```

```
private string? _name;
public string? Name
{
    get { return _name; }
    set
    {
        if (value != _name)
        {
            _name = value;
            RaisePropertyChanged(); // notice that "Name" is not
needed here explicitly
        }
    }
}
```

위의 코드에서는 리터럴 "Name" 문자열이 없어도 됩니다. `CallerMemberName`을 사용하면 오타 관련 버그를 방지하고 리팩터링/이름 바꾸기를 더 원활하게 할 수 있습니다. 특성은 C#에 선언적 기능을 제공하지만 코드의 메타데이터 형식이며 단독으로 동작하지 않습니다.

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

ଓ 설명서 문제 열기

ଓ 제품 사용자 의견 제공

# 자습서: 기본 인터페이스 메서드로 인터페이스를 업데이트

아티클 • 2023. 03. 18.

인터페이스 멤버 선언 시 구현을 정의할 수 있습니다. 가장 일반적인 시나리오는 이미 릴리스되어 수많은 클라이언트가 사용하는 인터페이스에 멤버를 안전하게 추가하는 것입니다.

이 자습서에서는 다음과 같은 작업을 수행하는 방법을 알아봅니다.

- ✓ 구현으로 메서드를 추가하여 안전하게 인터페이스를 확장합니다.
- ✓ 매개 변수가 있는 구현을 생성하여 향상된 유연성을 제공합니다.
- ✓ 구현자가 재정의 형식으로 더 구체적인 구현을 제공하도록 지원합니다.

## 사전 요구 사항

C# 컴파일러를 포함하여 .NET을 실행하도록 컴퓨터를 설정해야 합니다. C# 컴파일러는 [Visual Studio 2022](#) 또는 [.NET SDK](#)에서 사용할 수 있습니다.

## 시나리오 개요

이 자습서는 고객 관계 라이브러리의 버전 1부터 시작합니다. [GitHub의 샘플 리포지토리](#)에서 시작 애플리케이션을 다운로드할 수 있습니다. 이 라이브러리를 구축한 회사는 기존 애플리케이션이 있는 고객이 자사의 라이브러리를 채택할 것을 의도했으며, 구현할 라이브러리의 사용자에게 최소의 인터페이스 정의를 제공했습니다. 다음은 고객에 대한 인터페이스 정의입니다.

C#

```
public interface ICustomer
{
    IEnumerable<IOrder> PreviousOrders { get; }

    DateTime DateJoined { get; }
    DateTime? LastOrder { get; }
    string Name { get; }
    IDictionary<DateTime, string> Reminders { get; }
}
```

이들은 주문을 나타내는 두 번째 인터페이스를 정의했습니다.

C#

```
public interface IOrder
{
    DateTime Purchased { get; }
    decimal Cost { get; }
}
```

이러한 인터페이스에서, 팀은 사용자가 고객을 위해 더 나은 경험을 만들 수 있는 라이브러리를 빌드할 수 있었습니다. 이들의 목표는 기존 고객과 더 깊은 관계를 형성하고 신규 고객과의 관계를 개선하는 것이었습니다.

이제, 다음 릴리스를 위해 라이브러리를 업그레이드할 시간입니다. 요청된 기능 중 하나는 주문건이 많은 고객을 위해 충성도 할인을 지원하는 것입니다. 고객이 주문할 때마다 이 새로운 충성도 할인이 적용됩니다. 특정 할인은 각 개인 고객의 속성입니다. 구현된 각 `ICustomer`마다 고객 할인에 대해 다른 규칙을 설정할 수 있습니다.

이 기능을 추가하는 가장 일반적인 방법은 충성도 할인을 적용할 메서드로 `ICustomer` 인터페이스를 개선하는 것입니다. 이러한 설계 제안은 숙련된 개발자 사이에서 우려를 일으켰습니다. "인터페이스는 릴리스된 후에는 변경이 불가합니다! 호환성이 손상되는 변경을 하지 마세요!" 인터페이스 업그레이드를 위한 기본 인터페이스 구현입니다. 라이브러리 작성자가 인터페이스에 새 멤버를 추가하고 해당 멤버에 대한 기본 구현을 제공할 수 있습니다.

기본 인터페이스 구현을 통해 구현자는 해당 구현을 재지정하고 개발자는 인터페이스를 업그레이드할 수 있습니다. 라이브러리의 사용자는 기본 구현을 일반적인 변경으로 받아들일 수 있습니다. 비즈니스 규칙이 다른 경우 재지정할 수 있습니다.

## 기본 인터페이스 메서드를 사용하여 업그레이드

팀은 가장 가능성 있는 기본 구현, 즉 고객을 위한 충성도 할인에 합의했습니다.

업그레이드는 할인 자격을 갖추기 위해 필요한 주문 수, 할인율, 이 두 가지 속성에 기능을 제공해야 합니다. 이러한 기능을 사용하면 기본 인터페이스 메서드에 대한 완벽한 시나리오가 됩니다. `ICustomer` 인터페이스에 메서드를 추가하고 가장 가능성 있는 구현을 제공할 수 있습니다. 모든 기존, 그리고 새 구현은 기본 구현을 사용하거나 자체 구현을 제공할 수 있습니다.

먼저 새 메서드를 메서드의 본문을 포함하여 인터페이스에 추가합니다.

C#

```
// Version 1:
public decimal ComputeLoyaltyDiscount()
{
```

```

        DateTime TwoYearsAgo = DateTime.Now.AddYears(-2);
        if ((DateJoined < TwoYearsAgo) && (PreviousOrders.Count() > 10))
        {
            return 0.10m;
        }
        return 0;
    }

```

라이브러리 작성자는 구현을 확인하기 위해 첫 번째 테스트를 작성했습니다.

C#

```

SampleCustomer c = new SampleCustomer("customer one", new DateTime(2010, 5,
31))
{
    Reminders =
    {
        { new DateTime(2010, 08, 12), "child's birthday" },
        { new DateTime(2012, 11, 15), "anniversary" }
    }
};

SampleOrder o = new SampleOrder(new DateTime(2012, 6, 1), 5m);
c.AddOrder(o);

o = new SampleOrder(new DateTime(2013, 7, 4), 25m);
c.AddOrder(o);

// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount:
{theCustomer.ComputeLoyaltyDiscount()}");

```

테스트의 다음 부분을 확인합니다.

C#

```

// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount:
{theCustomer.ComputeLoyaltyDiscount()}");

```

`SampleCustomer`에서 `ICustomer`로의 캐스팅이 필요합니다. `SampleCustomer` 클래스는 `ComputeLoyaltyDiscount`에 대한 구현을 제공할 필요가 없으며, `ICustomer` 인터페이스에서 제공합니다. 그러나 `SampleCustomer` 클래스는 인터페이스에서 멤버를 상속하지 않습니다. 이 규칙은 바뀌지 않았습니다. 인터페이스에서 선언 및 구현된 메서드를 호출하려면 변수는 인터페이스 유형(이 예에서는 `ICustomer`)이어야 합니다.

# 매개 변수화 제공

기본 구현이 너무 제한적입니다. 이 시스템의 많은 소비자가 구매 건수에 다른 임계값, 다른 멤버십 기간 또는 다른 할인율을 선택할 수 있습니다. 이러한 매개 변수를 설정하는 방법을 제공하여 더 많은 고객에게 향상된 업그레이드 환경을 제공할 수 있습니다. 기본 구현을 제어하는 세 개의 매개 변수를 설정하는 정적 메서드를 추가해 보겠습니다.

C#

```
// Version 2:  
public static void SetLoyaltyThresholds(  
    TimeSpan ago,  
    int minimumOrders = 10,  
    decimal percentageDiscount = 0.10m)  
{  
    length = ago;  
    orderCount = minimumOrders;  
    discountPercent = percentageDiscount;  
}  
private static TimeSpan length = new TimeSpan(365 * 2, 0, 0, 0); // two years  
private static int orderCount = 10;  
private static decimal discountPercent = 0.10m;  
  
public decimal ComputeLoyaltyDiscount()  
{  
    DateTime start = DateTime.Now - length;  
  
    if ((DateJoined < start) && (PreviousOrders.Count() > orderCount))  
    {  
        return discountPercent;  
    }  
    return 0;  
}
```

이 작은 코드 조각에 표시되는 많은 새 언어 기능이 있습니다. 이제 인터페이스는 필드 및 메서드를 포함한 정적 멤버를 포함할 수 있습니다. 서로 다른 액세스 한정자도 사용할 수 있습니다. 다른 필드는 `private`이고, 새 메서드는 `public`입니다. 어떠한 한정자도 인터페이스 멤버에서 허용됩니다.

충성도 할인을 계산하기 위해 일반 공식을 사용하지만 매개 변수는 다른 애플리케이션은 사용자 지정 구현을 제공할 필요가 없지만, 정적 메서드를 통해 인수를 설정할 수 있습니다. 예를 들어, 다음 코드는 멤버십이 1개월 이상인 고객에게 보답하는 “고객 감사”를 설정합니다.

C#

```
ICustomer.SetLoyaltyThresholds(new TimeSpan(30, 0, 0, 0), 1, 0.25m);  
Console.WriteLine($"Current discount:
```

```
{theCustomer.ComputeLoyaltyDiscount()}");
```

## 기본 구현 확장

지금까지 추가한 코드는 사용자가 기본 구현과 같은 것을 원하거나, 관련 없는 규칙 세트를 제공하는 시나리오에 편리한 구현을 제공했습니다. 최종 기능을 위해, 코드를 약간 리팩터링하여 사용자가 기본 구현을 기반으로 구축하려는 시나리오를 구현해 보겠습니다.

신규 고객을 유치하고 싶은 스타트업 회사가 있다고 해보겠습니다. 이 회사는 신규 고객의 첫 주문에 50% 할인을 제공합니다. 한편, 기존 고객에게는 표준 할인이 적용됩니다. 라이브러리 작성자는 이 인터페이스를 구현하는 클래스가 해당 구현에서 코드를 재사용할 수 있도록 기본 구현을 `protected static` 메서드로 이동해야 합니다. 인터페이스 멤버의 기본 구현은 이 공유 메서드로 호출합니다.

C#

```
public decimal ComputeLoyaltyDiscount() => DefaultLoyaltyDiscount(this);
protected static decimal DefaultLoyaltyDiscount(ICustomer c)
{
    DateTime start = DateTime.Now - length;

    if ((c.DateJoined < start) && (c.PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

이 인터페이스를 구현하는 클래스의 구현에서 재지정은 정적 도우미 메서드를 호출하며, 이 논리를 확장하여 “신규 고객” 할인을 제공할 수 있습니다.

C#

```
public decimal ComputeLoyaltyDiscount()
{
    if (PreviousOrders.Any() == false)
        return 0.50m;
    else
        return ICustomer.DefaultLoyaltyDiscount(this);
}
```

GitHub의 샘플 리포지토리 [↗](#)에서 완성된 전체 코드를 볼 수 있습니다. GitHub의 샘플 리포지토리 [↗](#)에서 시작 애플리케이션을 다운로드할 수 있습니다.

이러한 새 기능은 신규 멤버에 대한 합리적인 기본 구현이 있는 경우 인터페이스를 안전하게 업데이트할 수 있음을 의미합니다. 인터페이스를 신중하게 디자인하여 여러 클래스에서 구현한 단일 기능 아이디어를 표현합니다. 이를 통해 동일한 기능 아이디어에 새로운 요구 사항이 발견될 경우 해당 인터페이스 정의를 훨씬 쉽게 업그레이드할 수 있습니다.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

**.NET feedback**

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# 자습서: 기본 인터페이스 메서드를 사용하는 인터페이스를 통해 클래스를 만드는 경우의 기능 혼합

아티클 • 2023. 03. 18.

인터페이스 멤버 선언 시 구현을 정의할 수 있습니다. 이 기능은 인터페이스에 선언된 기능에 대한 기본 구현을 정의할 수 있는 새로운 기능을 제공합니다. 클래스는 기능을 재정의할 시기, 기본 기능을 사용할 시기 및 불연속 기능에 대한 지원을 선언하지 않을 시기를 선택할 수 있습니다.

이 자습서에서 학습할 방법은 다음과 같습니다.

- ✓ 불연속 기능을 설명하는 구현을 사용하여 인터페이스를 만듭니다.
- ✓ 기본 구현을 사용하는 클래스를 만듭니다.
- ✓ 기본 구현의 일부 또는 전체를 재정의하는 클래스를 만듭니다.

## 사전 요구 사항

C# 컴파일러를 포함하여 .NET을 실행하도록 컴퓨터를 설정해야 합니다. C# 컴파일러는 [Visual Studio 2022](#) 또는 [.NET SDK](#)에서 사용할 수 있습니다.

## 확장 메서드의 제한 사항

인터페이스의 일부로 나타나는 동작을 구현할 수 있는 한 가지 방법은 기본 동작을 제공하는 [확장 메서드](#)를 정의하는 것입니다. 인터페이스는 해당 멤버를 구현하는 클래스에 더 큰 노출 영역을 제공하는 동시에 최소 멤버 집합을 선언합니다. 예를 들어 [Enumerable](#)의 확장 메서드는 모든 시퀀스가 LINQ 쿼리의 원본이 되도록 구현합니다.

확장 메서드는 선언된 변수 형식을 사용하여 컴파일 시간에 확인됩니다. 인터페이스를 구현하는 클래스는 모든 확장 메서드에 대해 더 나은 구현을 제공할 수 있습니다. 컴파일러가 해당 구현을 선택할 수 있도록 변수 선언이 구현 형식과 일치해야 합니다. 컴파일 시간 형식이 인터페이스와 일치하는 경우 메서드 호출은 확장 메서드를 확인합니다. 확장 메서드의 또 다른 문제는 확장 메서드를 포함하는 클래스에 액세스할 수 있는 모든 위치에서 메서드에 액세스할 수 있다는 것입니다. 클래스는 확장 메서드에 선언된 기능을 제공해야 하거나 제공하지 않아야 하는 경우 선언할 수 없습니다.

기본 구현을 인터페이스 메서드로 선언할 수 있습니다. 그러면 모든 클래스에서 자동으로 기본 구현을 사용합니다. 더 나은 구현을 제공할 수 있는 모든 클래스는 더 나은 알고리즘

으로 인터페이스 메서드를 재정의할 수 있습니다. 어떤 의미에서 이 기술은 확장 메서드를 사용하는 방법과 비슷합니다.

이 문서에서는 기본 인터페이스 구현에서 새 시나리오를 사용하도록 설정하는 방법을 알아봅니다.

## 애플리케이션 설계

홈 자동화 애플리케이션을 고려합니다. 집 전체에서 사용할 수 있는 다양한 광원 및 표시등이 있을 수 있습니다. 모든 광원은 켜고 끄고 현재 상태를 보고할 수 있도록 API가 지원되어야 합니다. 일부 광원 및 표시등은 다음과 같은 다른 기능을 지원할 수 있습니다.

- 광원을 켜고 타이머에 맞춰 끕니다.
- 일정 시간 동안 광원이 깜박입니다.

이러한 확장 기능 중 일부는 최소 설정을 지원하는 디바이스에서 에뮬레이트될 수 있습니다. 이는 기본 구현을 제공한다는 의미입니다. 더 많은 기능이 내장된 디바이스의 경우 디바이스 소프트웨어는 기본 기능을 사용합니다. 다른 광원의 경우 인터페이스를 구현하고 기본 구현을 사용하도록 선택할 수 있습니다.

기본 인터페이스 멤버는 확장 메서드보다 이 시나리오에 더 나은 솔루션을 제공합니다. 클래스 작성자는 구현할 인터페이스를 제어할 수 있습니다. 선택한 인터페이스는 메서드로 사용할 수 있습니다. 또한 기본 인터페이스 메서드는 기본적으로 가상이기 때문에 메서드 디스패치는 항상 클래스에서 구현을 선택합니다.

이러한 차이점을 보여주는 코드를 만들어 보겠습니다.

## 인터페이스 만들기

모든 광원의 동작을 정의하는 인터페이스를 만드는 것부터 시작합니다.

```
C#  
  
public interface ILight  
{  
    void SwitchOn();  
    void SwitchOff();  
    bool IsOn();  
}
```

기본 오버헤드 광원 픽스쳐는 다음 코드와 같이 이 인터페이스를 구현할 수 있습니다.

```
C#
```

```
public class OverheadLight : ILight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" :
"off")}";
}
```

이 자습서에서 코드는 IoT 디바이스를 구동하지 않지만 콘솔에 메시지를 작성하여 해당 활동을 에뮬레이트합니다. 집을 자동화하지 않고 코드를 탐색할 수 있습니다.

다음으로 시간 제한 후 자동으로 꺼질 수 있는 광원의 인터페이스를 정의해 보겠습니다.

C#

```
public interface ITimerLight : ILight
{
    Task TurnOnFor(int duration);
}
```

기본 구현을 오버헤드 광원에 추가할 수 있지만, 이 인터페이스 정의를 수정하여 `virtual` 기본 구현을 제공하는 것이 더 좋습니다.

C#

```
public interface ITimerLight : ILight
{
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Using the default interface method for the
ITimerLight.TurnOnFor.");
        SwitchOn();
        await Task.Delay(duration);
        SwitchOff();
        Console.WriteLine("Completed ITimerLight.TurnOnFor sequence.");
    }
}
```

클래스는 `OverheadLight` 인터페이스에 대한 지원을 선언하여 타이머 함수를 구현할 수 있습니다.

C#

```
public class OverheadLight : ITimerLight { }
```

다른 광원 형식은 보다 정교한 프로토콜을 지원할 수 있습니다. 다음 코드와 같이 `TurnOnFor`에 대한 자체 구현을 제공할 수 있습니다.

C#

```
public class HalogenLight : ITimerLight
{
    private enum HalogenLightState
    {
        Off,
        On,
        TimerModeOn
    }

    private HalogenLightState state;
    public void SwitchOn() => state = HalogenLightState.On;
    public void SwitchOff() => state = HalogenLightState.Off;
    public bool IsOn() => state != HalogenLightState.Off;
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Halogen light starting timer function.");
        state = HalogenLightState.TimerModeOn;
        await Task.Delay(duration);
        state = HalogenLightState.Off;
        Console.WriteLine("Halogen light finished custom timer function");
    }

    public override string ToString() => $"The light is {state}";
}
```

재정의 가상 클래스 메서드와 달리 클래스의 선언 `TurnOnFor` 은 `HalogenLight` 키워드 (keyword) 사용하지 `override` 않습니다.

## 조합 및 일치 기능

고급 기능을 도입할수록 기본 인터페이스 방법의 장점이 더 명확해집니다. 인터페이스를 사용하면 기능을 조합하고 일치시킬 수 있습니다. 또한 각 클래스 작성자가 기본 구현과 사용자 지정 구현 중에서 선택할 수 있습니다. 깜박이는 광원에 대한 기본 구현으로 인터페이스를 추가해 보겠습니다.

C#

```
public interface IBlinkingLight : ILight
{
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Using the default interface method for
IBlinkingLight.Blink.");
        for (int count = 0; count < repeatCount; count++)
    }
```

```

    {
        SwitchOn();
        await Task.Delay(duration);
        SwitchOff();
        await Task.Delay(duration);
    }
    Console.WriteLine("Done with the default interface method for
IBlinkingLight.Blink.");
}
}

```

기본 구현을 사용하면 모든 광원이 깜박일 수 있습니다. 오버헤드 광원은 기본 구현을 사용하여 타이머 및 깜박임 기능을 모두 추가할 수 있습니다.

C#

```

public class OverheadLight : ILight, ITimerLight, IBlinkingLight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" :
"off")}";
}

```

새로운 광원 형식인 `LEDLight`는 timer 함수와 blink 함수를 직접 지원합니다. 이 광원 스타일은 `ITimerLight` 및 `IBlinkingLight` 인터페이스를 모두 구현하고 `Blink` 메서드를 재정의합니다.

C#

```

public class LEDLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("LED Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("LED Light has finished the Blink function.");
    }

    public override string ToString() => $"The light is {(isOn ? "on" :
"off")}";
}

```

ExtraFancyLight 는 blink 및 timer 함수를 직접 지원할 수 있습니다.

C#

```
public class ExtraFancyLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Extra Fancy Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("Extra Fancy Light has finished the Blink
function.");
    }
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Extra Fancy light starting timer function.");
        await Task.Delay(duration);
        Console.WriteLine("Extra Fancy light finished custom timer
function");
    }

    public override string ToString() => $"The light is {isOn ? "on" :
"off")}";
}
```

이전에 만든 HalogenLight 는 깜박임을 지원하지 않습니다. 따라서 지원되는 인터페이스 목록에 IBlinkingLight 를 추가하지 마세요.

## 패턴 일치를 사용하여 광원 형식 검색

다음으로 몇 가지 테스트 코드를 작성해 보겠습니다. C#의 패턴 일치 기능을 사용하여 지원되는 인터페이스를 검사하고 광원의 기능을 결정할 수 있습니다. 다음 메서드는 각 광원의 지원되는 기능을 연습합니다.

C#

```
private static async Task TestLightCapabilities(ILight light)
{
    // Perform basic tests:
    light.SwitchOn();
    Console.WriteLine($"\\tAfter switching on, the light is {(light.IsOn() ?
"on" : "off")}");
    light.SwitchOff();
    Console.WriteLine($"\\tAfter switching off, the light is {(light.IsOn() ?
"on" : "off")}");
}
```

```

if (light is ITimerLight timer)
{
    Console.WriteLine("\tTesting timer function");
    await timer.TurnOnFor(1000);
    Console.WriteLine("\tTimer function completed");
}
else
{
    Console.WriteLine("\tTimer function not supported.");
}

if (light is IBlinkingLight blinker)
{
    Console.WriteLine("\tTesting blinking function");
    await blinker.Blink(500, 5);
    Console.WriteLine("\tBlink function completed");
}
else
{
    Console.WriteLine("\tBlink function not supported.");
}
}

```

Main 메서드의 다음 코드는 각 광원 형식을 차례로 만들고 해당 광원을 테스트합니다.

C#

```

static async Task Main(string[] args)
{
    Console.WriteLine("Testing the overhead light");
    var overhead = new OverheadLight();
    await TestLightCapabilities(overhead);
    Console.WriteLine();

    Console.WriteLine("Testing the halogen light");
    var halogen = new HalogenLight();
    await TestLightCapabilities(halogen);
    Console.WriteLine();

    Console.WriteLine("Testing the LED light");
    var led = new LEDLight();
    await TestLightCapabilities(led);
    Console.WriteLine();

    Console.WriteLine("Testing the fancy light");
    var fancy = new ExtraFancyLight();
    await TestLightCapabilities(fancy);
    Console.WriteLine();
}

```

# 컴파일러가 최선의 구현을 결정하는 방법

이 시나리오에서는 구현이 없는 기본 인터페이스를 보여 줍니다. `ILight` 인터페이스에 메서드를 추가하면 새로운 복잡성이 발생합니다. 기본 인터페이스 메서드를 제어하는 언어 규칙은 여러 파생 인터페이스를 구현하는 구체적인 클래스에 대한 영향을 최소화합니다. 새 메서드로 원래 인터페이스를 개선하여 사용 방법을 변경할 수 있는 방법을 확인해보겠습니다. 모든 표시등 광원은 해당 전원 상태를 열거형 값으로 보고할 수 있습니다.

C#

```
public enum PowerStatus
{
    NoPower,
    ACPower,
    FullBattery,
    MidBattery,
    LowBattery
}
```

기본 구현에서는 전원이 없는 것으로 가정합니다.

C#

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
    public PowerStatus Power() => PowerStatus.NoPower;
}
```

`ExtraFancyLight`에서 `ILight` 인터페이스 및 파생된 인터페이스, `ITimerLight` 및 `IBlinkingLight`에 대한 지원을 선언하더라도 이러한 변경 내용은 완전히 컴파일됩니다. `ILight` 인터페이스에 선언된 "가장 가까운" 구현은 하나뿐입니다. 재정의를 선언한 모든 클래스는 하나의 "가장 가까운" 구현이 됩니다. 이전 클래스에서 다른 파생 인터페이스의 멤버를 재정의하는 예를 살펴보았습니다.

여러 파생 인터페이스에서 동일한 메서드를 재정의하지 마세요. 이렇게 하면 클래스가 파생된 두 인터페이스를 모두 구현할 때마다 모호한 메서드 호출을 만듭니다. 컴파일러는 더 나은 단일 메서드를 선택할 수 없으므로 오류가 발생합니다. 예를 들어 `IBlinkingLight` 및 `ITimerLight` 모두 `PowerStatus` 재정의를 구현하는 경우 `OverheadLight`는 보다 구체적인 재정의를 제공해야 합니다. 그렇지 않으면 컴파일러는 두 파생 인터페이스의 구현 사이에서 선택할 수 없습니다. 일반적으로 인터페이스 정의를

작게 유지하고 하나의 기능에 집중하여 이러한 상황을 방지할 수 있습니다. 이 시나리오에서 조명의 각 기능은 자체 인터페이스입니다. 클래스만 여러 인터페이스를 상속합니다.

이 샘플은 클래스에 결합될 수 있는 불연속 기능을 정의할 수 있는 시나리오를 보여 줍니다. 클래스가 지원하는 인터페이스를 선언하여 지원되는 기능 집합을 선언합니다. 가상 기본 인터페이스 메서드를 사용하면 클래스가 일부 또는 모든 인터페이스 메서드에 대해 다른 구현을 사용하거나 정의할 수 있습니다. 이 언어 기능은 빌드 중인 실제 시스템을 모델링하는 새로운 방법을 제공합니다. 기본 인터페이스 메서드는 해당 기능의 가상 구현을 사용하여 다른 기능을 조합하고 일치시킬 수 있는 관련 클래스를 더 명확하게 표현하는 방법을 제공합니다.

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

 .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

# 식 트리

아티클 • 2024. 02. 17.

식 트리는 트리와 유사한 데이터 구조의 코드를 나타낼 수 있습니다. 여기서 각 노드는 식(예: 메서드 호출 또는 `x < y` 같은 이진 연산)입니다.

LINQ를 사용했다면 `Func` 형식이 API 집합의 일부인 풍부한 라이브러리 경험이 있는 것입니다. LINQ에 익숙하지 않은 경우 [LINQ 자습서](#) 및 [람다 식](#)에 대한 문서를 먼저 읽어보는 것이 좋습니다. 식 트리에서는 함수인 인수를 사용하는 보다 풍부한 조작을 제공합니다.

LINQ 쿼리를 만들 때 일반적으로 람다 식을 사용하여 함수 인수를 작성합니다. 일반적인 LINQ 쿼리에서 이러한 함수 인수는 컴파일러가 만드는 대리자로 변환됩니다.

식 트리를 사용하는 코드를 이미 작성했을 수 있습니다. Entity Framework의 LINQ API는 LINQ 쿼리 식 패턴에 대한 인수로 식 트리를 허용합니다. 따라서 [Entity Framework](#)는 C#에서 작성된 쿼리를 데이터베이스 엔진에서 실행되는 SQL로 변환할 수 있습니다. 또 다른 예로 널리 사용되는 .NET 모의 프레임워크인 [Moq](#) 가 있습니다.

보다 풍부한 조작을 원하는 경우 식 트리를 사용해야 합니다. 식 트리는 검사, 수정 또는 실행할 수 있는 구조로 코드를 나타냅니다. 이러한 도구는 런타임에 코드를 조작할 수 있는 강력한 기능을 제공합니다. 실행 중인 알고리즘을 검사하고 새 기능을 삽입하는 코드를 작성합니다. 보다 고급 시나리오에서는 실행 중인 알고리즘을 수정하고 C# 식을 다른 형태로 변환하여 다른 환경에서 실행합니다.

식 트리로 표시되는 코드를 컴파일하고 실행합니다. 식 트리를 빌드하고 실행하면 실행 가능한 코드를 동적으로 수정하고, 다양한 데이터베이스에서 LINQ 쿼리를 실행하고, 동적 쿼리를 만들 수 있습니다. LINQ의 식 트리에 대한 자세한 내용은 [식 트리를 사용하여 동적 쿼리 빌드 방법](#)을 참조하세요.

식 트리는 동적 언어와 .NET 간에 상호 운용성을 제공하고 컴파일러 작성기가 MSIL(Microsoft Intermediate Language) 대신 식 트리를 내보낼 수 있도록 DLR(동적 언어 런타임)에서도 사용됩니다. DLR에 대한 자세한 내용은 [동적 언어 런타임 개요](#)를 참조하세요.

익명 람다 식을 기준으로 C# 또는 Visual Basic 컴파일러가 식 트리를 자동으로 만들도록 할 수도 있고 [System.Linq.Expressions](#) 네임스페이스를 사용하여 식 트리를 수동으로 만들 수도 있습니다.

람다 식을 `Expression<TDelegate>` 형식 변수에 할당하면 컴파일러가 해당 람다 식을 나타내는 식 트리를 작성하기 위해 코드를 내보냅니다.

C# 컴파일러는 람다 식(한 줄 람다)에서만 식 트리를 생성합니다. 문 람다(혹은 여러 줄 람다)는 구문 분석할 수 없습니다. C#의 람다 식에 대한 자세한 내용은 [람다 식](#)을 참조하세요.

요.

다음 코드 예제에서는 C# 컴파일러에서 람다 식 `num => num < 5`를 나타내는 식 트리를 만드는 방법을 보여 줍니다.

C#

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

코드에서 식 트리를 만듭니다. 각 노드를 만들고 노드를 트리 구조에 연결하여 트리를 빌드합니다. [식 트리 작성](#)에 대한 문서에서 식을 만드는 방법을 알아봅니다.

식 트리는 변경할 수 없습니다. 식 트리를 수정하려면 기존 식 트리를 복사한 다음 트리 내의 노드를 바꾸는 방법으로 새 식 트리를 생성해야 합니다. 식 트리 방문자를 사용하면 기존 식 트리를 트래버스 합니다. 자세한 내용은 [식 트리 번역](#)에 대한 문서를 참조하세요.

식 트리를 빌드한 후에는 [식 트리가 나타내는 코드를 실행](#)합니다.

## 제한 사항

식 트리로 잘 변환되지 않는 최신 C# 언어 요소가 몇 가지 있습니다. 식 트리에는 `await` 식이나 `async` 람다 식이 포함될 수 없습니다. C# 6 및 이후 릴리스에서 추가된 기능 중 상당수는 식 트리에 작성된 그대로 표시되지 않습니다. 대신, 가능한 경우 최신 기능이 식 트리에 해당하는 이전 구문으로 노출됩니다. 다른 구문을 사용할 수 없습니다. 식 트리를 해석하는 코드는 새 언어 기능이 도입될 때 동일하게 작동한다는 의미입니다. 이러한 제한 사항에도 불구하고 식 트리를 사용하면 데이터 구조로 표시되는 코드를 해석하고 수정하는 동적 알고리즘을 만들 수 있습니다. Entity Framework와 같은 풍부한 라이브러리가 수행하는 작업을 수행할 수 있습니다.

식 트리는 새 식 노드 형식을 지원하지 않습니다. 식 트리를 해석하는 모든 라이브러리에서 새 노드 형식을 도입하는 것은 호환성이 손상되는 변경입니다. 다음 목록에는 사용할 수 없는 대부분의 C# 언어 요소가 포함되어 있습니다.

- 제거된 [조건부 메서드](#)
- [base 액세스](#)
- 메서드 그룹 [주소\(&\)](#)와 익명 메서드 식을 포함한 메서드 그룹 식
- [로컬 함수](#)에 대한 참조
- 대입(`=`) 및 문 본문 식을 포함한 문
- 정의 선언만 있는 [부분 메서드](#)
- [안전하지 않은 포인터](#) 작업
- [dynamic](#)작업

- 연산자를 null 또는 default 리터럴 왼쪽으로 병합, null 병합 할당 및 null 전파 연산자(?)
- 다차원 배열 이니셜라이저, 인덱싱된 속성 및 사전 이니셜라이저
- throw표현식
- static virtual 또는 abstract 인터페이스 멤버 액세스
- 특성이 있는 람다 식
- 보간된 문자열
- UTF-8 문자열 변환 또는 UTF-8 문자열 리터럴
- 변수 인수, 명명된 인수 또는 선택적 인수를 사용하는 메서드 호출
- System.Index 또는 System.Range, 인덱스 "from end"(^) 연산자 또는 범위 식(..)을 사용한 식
- async 람다 식 또는 await 식, await foreach 및 await using 포함
- 튜플 리터럴, 튜플 변환, 튜플 == 또는 != 또는 with 식
- 무시(), 분해 할당, 패턴 일치 is 연산자 또는 패턴 일치 switch 식
- 인수에서 ref 생략된 COM 호출
- ref, in 또는 out 매개 변수, ref 반환 값, out 인수 또는 ref struct 형식의 모든 값

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 식 트리 - 코드를 정의하는 데이터

아티클 • 2024. 02. 17.

식 트리는 코드를 정의하는 데이터 구조이며, 식 트리는 컴파일러가 코드를 분석하고 컴파일된 출력을 생성하는 데 사용하는 것과 동일한 구조를 기반으로 합니다. 이 문서를 읽으면서 식 트리와 Roslyn API에서 [분석기 및 CodeFixes](#) 빌드하는 데 사용되는 형식 간에 상당한 유사성을 알 수 있습니다. (분석기 및 CodeFixes는 코드에 대한 정적 분석을 수행하고 개발자를 위한 잠재적 수정 사항을 제안하는 NuGet 패키지입니다.) 개념이 유사하며 최종 결과는 의미 있는 방식으로 소스 코드를 검사할 수 있는 데이터 구조입니다. 그러나 식 트리는 Roslyn API와는 다른 클래스 및 API 집합을 기반으로 합니다. 코드 줄은 다음과 같습니다.

C#

```
var sum = 1 + 2;
```

위의 코드를 식 트리로 분석하면 트리에 여러 노드가 포함됩니다. 가장 바깥쪽 노드는 할당을 사용하는 변수 선언문(`var sum = 1 + 2;`)입니다. 이 가장 바깥쪽 노드에는 변수 선언, 대입 연산자 및 등호의 오른쪽을 나타내는 식과 같은 여러 개의 자식 노드가 포함됩니다. 이 식은 더하기 연산을 나타내는 식, 더하기의 왼쪽과 오른쪽 피연산자로 다시 구분됩니다.

등호의 오른쪽을 구성하는 식을 좀더 자세히 살펴보겠습니다. 식은 이진 식 `1 + 2`입니다. 보다 구체적으로 이진 더하기 식입니다. 이진 더하기 식에는 더하기 식의 왼쪽과 오른쪽 노드를 나타내는 두 개의 자식이 있습니다. 여기에서 두 노드는 상수 식입니다. 왼쪽 피연산자는 `1` 값이고, 오른쪽 피연산자는 `2` 값입니다.

시각적으로 전체 문은 트리입니다. 루트 노드에서 시작하여 트리의 각 노드로 이동하면서 문을 구성하는 코드를 확인할 수 있습니다.

- 할당을 사용하는 변수 선언문(`var sum = 1 + 2;`)
  - 암시적 변수 형식 선언(`var sum`)
  - 암시적 var 키워드(`var`)
  - 변수 이름 선언(`sum`)
  - 대입 연산자(`=`)
  - 이진 더하기 식(`1 + 2`)
    - 왼쪽 피연산자(`1`)
    - 더하기 연산자(`+`)
    - 오른쪽 피연산자(`2`)

위의 트리는 복잡해 보일 수 있지만 매우 강력합니다. 동일한 프로세스에 따라 훨씬 더 복잡한 식을 분해합니다. 다음 식을 살펴보세요.

C#

```
var finalAnswer = this.SecretSauceFunction(  
    currentState.createInterimResult(), currentState.createSecondValue(1,  
    2),  
    decisionServer.considerFinalOptions("hello")) +  
    MoreSecretSauce('A', DateTime.Now, true);
```

앞의 식은 할당이 있는 변수 선언이기도 합니다. 이 경우 할당의 오른쪽이 훨씬 더 복잡한 트리입니다. 이 식을 분해하지 않고 다른 노드가 무엇인지 고려합니다. 현재 개체를 수신 기로 사용하는 메서드 호출, 명시적 `this` 수신기가 있는 메서드 호출, 그렇지 않은 메서드 호출이 있습니다. 다른 수신기 개체를 사용하는 메서드 호출이 있고 다양한 형식의 상수 인수가 있습니다. 마지막으로 이진 추가 연산자가 있습니다. `SecretSauceFunction()` 또는 `MoreSecretSauce()`의 반환 형식에 따라 해당 이진 더하기 연산자는 재정의된 더하기 연산자에 대한 메서드 호출이 되어 클래스에 대해 정의된 이진 더하기 연산자에 대한 정적 메서드 호출로 확인될 수 있습니다.

이러한 인식된 복잡성에도 불구하고 앞의 식은 첫 번째 샘플처럼 쉽게 탐색되는 트리 구조를 만듭니다. 자식 노드를 계속 트래버스하여 식에서 리프 노드를 찾습니다. 부모 노드에는 자식에 대한 참조가 있으며 각 노드에는 노드의 종류를 설명하는 속성이 있습니다.

식 트리의 구조는 거의 일치합니다. 기본 사항을 학습한 후에는 식 트리로 표현될 때 가장 복잡한 코드도 이해합니다. 데이터 구조의 우아함은 C# 컴파일러가 가장 복잡한 C# 프로그램을 분석하고 복잡한 소스 코드에서 적절한 출력을 만드는 방법을 설명합니다.

식 트리의 구조에 익숙해지면 얻은 지식을 통해 더 많은 고급 시나리오를 사용할 수 있습니다. 식 트리에는 정말 강력한 기능이 있습니다.

다른 환경에서 실행할 알고리즘을 변환하는 것 외에도 식 트리를 사용하면 코드를 실행하기 전에 코드를 검사하는 알고리즘을 더 쉽게 작성할 수 있습니다. 인수가 식인 메서드를 작성한 다음 코드를 실행하기 전에 해당 식을 검사합니다. 식 트리는 코드의 전체 표현이며, 모든 하위 식의 값을 확인할 수 있습니다. 메서드 및 속성 이름을 확인할 수 있습니다. 모든 상수 식의 값을 확인할 수 있습니다. 식 트리를 실행 대리자로 변환하고 코드를 실행할 수 있습니다.

식 트리에 대한 API에서는 거의 모든 유효한 코드 구문을 나타내는 트리를 만들 수 있습니다. 그러나 최대한 간단하게 유지하기 위해 식 트리에는 일부 C# 관용구를 만들 수 없습니다. 한 가지 예는 비동기 식(`async` 및 `await` 키워드 사용)입니다. 비동기 알고리즘이 필요한 경우 컴파일러 지원을 사용하기보다는 `Task` 개체를 직접 조작해야 합니다. 또 다른 예로 루프 만들기가 있습니다. 일반적으로 `for`, `foreach`, `while` 또는 `do` 루프를 사용하여

이러한 루프를 만듭니다. [이 시리즈의 뒷부분](#)에서 살펴보겠지만 식 트리에 대한 API는 루프 반복을 제어하는 `break` 및 `continue` 식과 함께 단일 루프 식을 지원합니다.

단, 식 트리는 수정할 수 없습니다. 식 트리는 변경할 수 없는 데이터 구조입니다. 식 트리를 변경하려면 원하는 변경 사항을 포함하여 원본의 복사본인 새 트리를 만들어야 합니다.

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# 식 트리에 대한 .NET 런타임 지원

아티클 • 2024. 02. 19.

.NET 런타임에는 식 트리에서 작동하는 많은 클래스 목록이 있습니다. 전체 목록은 [System.Linq.Expressions](#)에서 확인할 수 있습니다. 전체 목록을 열거하는 대신 런타임 클래스가 디자인된 방식을 이해해 보겠습니다.

언어 디자인에서 식이란 값을 계산하고 반환하는 코드의 본문입니다. 식은 간단할 수 있습니다. 상수 식 1은(는) 1의 상수 값을 반환합니다. 식은 더 복잡할 수도 있습니다. (-B + Math.Sqrt(B\*B - 4 \* A \* C)) / (2 \* A) 식은 2차 방정식의 근을 반환합니다(방정식에 솔루션이 있는 경우).

## System.Linq.Expression 및 파생 형식

식 트리 사용의 복잡성 중 하나는 많은 종류의 식이 프로그램의 여러 위치에서 유효하다는 점입니다. 대입 식을 살펴보세요. 대입 식의 오른쪽은 상수 값, 변수, 메서드 호출 식 등이 될 수 있습니다. 해당 언어 유연성은 식 트리를 트래버스할 때 트리 노드의 모든 위치에서 여러 가지 다른 식 형식이 표시될 수 있음을 의미합니다. 따라서 기본 식 형식으로 작업하는 경우 가장 간단한 작업 방법입니다. 그러나 더 많은 것을 알아야 하는 경우가 있습니다. 기본 식 클래스에는 `NodeType` 속성이 이러한 용도로 포함되어 있으며, 가능한 식 형식의 열거형인 `ExpressionType` 을(를) 반환합니다. 노드의 형식을 알고 나면 해당 형식으로 캐스팅하고 식 노드의 형식을 알고 특정 작업을 수행합니다. 특정 노드 유형을 검색한 다음 이러한 식의 특정 속성을 사용할 수 있습니다.

예를 들어 이 코드는 변수 액세스 식에 대한 변수의 이름을 출력합니다. 다음 코드에서는 노드 형식을 확인한 다음 변수 액세스 식으로 캐스팅한 다음 특정 식 형식의 속성을 확인하는 방법을 보여 줍니다.

C#

```
Expression<Func<int, int>> addFive = (num) => num + 5;

if (addFive is LambdaExpression lambdaExp)
{
    var parameter = lambdaExp.Parameters[0]; -- first

    Console.WriteLine(parameter.Name);
    Console.WriteLine(parameter.Type);
}
```

## 식 트리 만들기

`System.Linq.Expression` 클래스에는 식을 만드는 많은 정적 메서드도 포함되어 있습니다. 이러한 메서드는 자식에 대해 제공된 인수를 사용하여 식 노드를 만듭니다. 이러한 방식으로 리프 노드에서 식을 빌드합니다. 예를 들어 다음 코드는 더하기 식을 작성합니다.

C#

```
// Addition is an add expression for "1 + 2"
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
```

이 간단한 예제를 통해 식 트리를 만들고 사용하는 데 많은 형식이 관련됨을 확인할 수 있습니다. 이러한 복잡성은 C# 언어에서 제공하는 풍부한 어휘의 기능을 제공하는 데 필요합니다.

## API 탐색

C# 언어의 거의 모든 구문 요소에 매핑되는 식 노드 유형이 있습니다. 각 형식에는 해당 언어 요소 형식에 대한 특정 메서드가 있습니다. 한 번에 기억해야 할 사항이 많습니다. 모든 것을 암기하려고 하는 대신 식 트리를 사용하는 데 사용하는 기술은 다음과 같습니다.

1. `ExpressionType` 열거형의 멤버를 확인하여 검색할 수 있는 노드를 결정합니다. 이 목록은 식 트리를 트래버스하고 이해하려는 경우에 도움이 됩니다.
2. `Expression` 클래스의 정적 멤버를 확인하여 식을 작성합니다. 이러한 메서드는 자식 노드 집합에서 모든 식 형식을 작성할 수 있습니다.
3. `ExpressionVisitor` 클래스를 확인하여 수정된 식 트리를 작성합니다.

이 세 영역을 살펴보면 더 많은 것을 찾을 수 있습니다. 이러한 세 단계 중 하나로 시작할 때 필요한 항목을 항상 찾을 수 있습니다.

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 식 트리 실행

아티클 • 2024. 02. 17.

식 트리는 일부 코드를 나타내는 데이터 구조입니다. 컴파일되고 실행 가능한 코드가 아닙니다. 식 트리로 표시되는 .NET 코드를 실행하려면 실행 가능한 IL 명령으로 변환해야 합니다. 식 트리를 실행할 때 값이 반환될 수 있거나, 메서드 호출 등의 작업만 수행할 수도 있습니다.

람다 식을 나타내는 식 트리만 실행할 수 있습니다. 람다 식을 나타내는 식 트리는 [LambdaExpression](#) 또는 [Expression<TDelegate>](#) 형식입니다. 이러한 식 트리를 실행하려면 [Compile](#) 메서드를 호출하여 실행 가능한 대리자를 만든 후 대리자를 호출합니다.

## ① 참고

대리자의 형식을 알 수 없는 경우, 즉 람다 식이 [Expression<TDelegate>](#) 형식이 아니라 [LambdaExpression](#) 형식인 경우 대리자를 직접 호출하는 대신 대리자의 [DynamicInvoke](#) 메서드를 호출합니다.

식 트리가 람다 식을 나타내지 않는 경우 [Lambda<TDelegate>\(Expression, IEnumerable<ParameterExpression>\)](#) 메서드를 호출하여 원래 식 트리가 본문으로 포함된 새 람다 식을 만들 수 있습니다. 그런 다음 이 섹션의 앞부분에서 설명한 대로 람다 식을 실행할 수 있습니다.

## 람다 식을 함수로 변환

모든 [LambdaExpression](#) 또는 [LambdaExpression](#)에서 파생된 모든 형식을 실행 가능한 IL로 변환할 수 있습니다. 다른 식 형식은 코드로 직접 변환할 수 없습니다. 실제로 이 제한은 거의 효과가 없습니다. 람다 식은 실행 가능한 IL(중간 언어)로 변환하여 실행하려는 식의 유일한 형식입니다. [System.Linq.Expressions.ConstantExpression](#)을 직접 실행하는 것의 의미를 생각해 보세요. 유용한 의미가 있나요?

[System.Linq.Expressions.LambdaExpression](#)이거나 [LambdaExpression](#)에서 파생된 형식인 모든 식 트리는 IL로 변환할 수 있습니다. 식 형식

[System.Linq.Expressions.Expression<TDelegate>](#) 는 .NET Core 라이브러리에서 유일하게 구체적인 예제입니다. 이 형식은 모든 대리자 형식에 매핑되는 식을 나타내는 데 사용됩니다. 이 형식은 대리자 형식에 매핑되므로 .NET에서 식을 검사하고 람다 식의 시그니처와 일치하는 적절한 대리자에 대해 IL을 생성할 수 있습니다. 대리자 형식은 식 형식을 기반으로 합니다. 강력한 형식의 방식으로 대리자 개체를 사용하려면 반환 형식 및 인수 목록을 알고 있어야 합니다. [LambdaExpression.Compile\(\)](#) 메서드는 [Delegate](#) 형식을 반환합

니다. 컴파일 시간 도구에서 인수 목록 또는 반환 형식을 확인할 수 있도록 하려면 올바른 대리자 형식으로 캐스팅해야 합니다.

대부분의 경우 식과 해당 대리자 간의 간단한 매핑이 존재합니다. 예를 들어, `Expression<Func<int>>`로 표시되는 식 트리는 `Func<int>` 형식의 대리자로 변환됩니다. 반환 형식 및 인수 목록을 사용하는 람다 식의 경우 람다 식으로 표시된 실행 코드의 대상 형식인 대리자 형식이 있습니다.

`System.Linq.Expressions.LambdaExpression` 형식에는 식 트리를 실행 코드로 변환하는 데 사용되는 `LambdaExpression.Compile` 및 `LambdaExpression.CompileToMethod` 멤버가 포함됩니다. `Compile` 메서드는 대리자를 만듭니다. `CompileToMethod` 메서드는 식 트리의 컴파일된 출력을 나타내는 IL로 `System.Reflection.Emit.MethodBuilder` 개체를 업데이트 합니다.

### ① 중요

`CompileToMethod` 는 .NET Core 또는 .NET 5 이상에서는 사용할 수 없고 .NET Framework에서만 사용할 수 있습니다.

선택적으로 생성된 대리자 개체에 대한 기호 디버깅 정보를 수신하는 `System.Runtime.CompilerServices.DebugInfoGenerator`를 제공할 수도 있습니다. `DebugInfoGenerator`는 생성된 대리자에 대한 전체 디버깅 정보를 제공합니다.

다음 코드를 사용하여 식을 대리자로 변환합니다.

C#

```
Expression<Func<int>> add = () => 1 + 2;
var func = add.Compile(); // Create Delegate
var answer = func(); // Invoke Delegate
Console.WriteLine(answer);
```

다음 코드 예에서는 식 트리를 컴파일하고 실행할 때 사용되는 구체적인 형식을 보여 줍니다.

C#

```
Expression<Func<int, bool>> expr = num => num < 5;

// Compiling the expression tree into a delegate.
Func<int, bool> result = expr.Compile();

// Invoking the delegate and writing the result to the console.
Console.WriteLine(result(4));
```

```
// Prints True.

// You can also use simplified syntax
// to compile and run an expression tree.
// The following line can replace two previous statements.
Console.WriteLine(expr.Compile()(4));

// Also prints True.
```

다음 코드 예제에서는 람다 식을 만들고 실행하여 숫자의 거듭제곱을 나타내는 식 트리를 실행하는 방법을 보여 줍니다. 숫자의 거듭제곱을 나타내는 결과가 표시됩니다.

C#

```
// The expression tree to execute.
BinaryExpression be = Expression.Power(Expression.Constant(2d),
                                         Expression.Constant(3d));

// Create a lambda expression.
Expression<Func<double>> le = Expression.Lambda<Func<double>>(be);

// Compile the lambda expression.
Func<double> compiledExpression = le.Compile();

// Execute the lambda expression.
double result = compiledExpression();

// Display the result.
Console.WriteLine(result);

// This code produces the following output:
// 8
```

## 실행 및 수명

`LambdaExpression.Compile()` 을 호출할 때 만든 대리자를 호출하여 코드를 실행합니다. 앞의 코드 `add.Compile()` 은 대리자를 반환합니다. 코드를 실행하는 `func()` 를 호출하여 해당 대리자를 호출합니다.

이 대리자는 식 트리의 코드를 나타냅니다. 해당 대리자에 대한 핸들을 유지하고 나중에 호출할 수 있습니다. 식 트리가 나타내는 코드를 실행할 때마다 식 트리를 컴파일할 필요는 없습니다. (식 트리는 변경할 수 없으며 나중에 동일한 식 트리를 컴파일하면 동일한 코드를 실행하는 대리자가 만들어집니다.)

### ⊗ 주의

불필요한 컴파일 호출을 방지하여 성능을 향상시키기 위해 더 정교한 캐싱 메커니즘을 만들지 마세요. 두 개의 임의 식 트리를 비교하여 동일한 알고리즘을 나타내는지 확인하는 작업에는 시간이 많이 걸립니다. `LambdaExpression.Compile()`에 대한 추가 호출을 방지하기 위해 절약한 컴퓨팅 시간은 두 개의 서로 다른 식 트리가 동일한 실행 코드를 생성하는지 확인하는 코드를 실행하는 데 소요되는 시간보다 길 가능성이 높습니다.

## 제한 사항

람다식을 대리자로 컴파일하고 해당 대리자를 호출하는 것은 식 트리로 수행할 수 있는 가장 간단한 작업 중 하나입니다. 그러나 이 간단한 작업에서도 주의해야 할 사항이 있습니다.

람다식은 식에서 참조되는 모든 지역 변수에 대해 클로저를 만듭니다. 대리자의 일부가 되는 모든 변수는 `Compile`을 호출하는 위치 및 결과 대리자를 실행할 때 사용할 수 있도록 보장해야 합니다. 컴파일러는 변수가 범위 내에 있는지 확인합니다. 그러나 식이 `IDisposable`을 구현하는 변수에 액세스하는 경우 코드는 식 트리에서 보유한 객체를 삭제할 수 있습니다.

예를 들어, 다음 코드는 `int`이 `IDisposable`을 구현하지 않기 때문에 제대로 작동합니다.

C#

```
private static Func<int, int> CreateBoundFunc()
{
    var constant = 5; // constant is captured by the expression tree
    Expression<Func<int, int>> expression = (b) => constant + b;
    var rVal = expression.Compile();
    return rVal;
}
```

대리자가 지역 변수 `constant`에 대한 참조를 캡처했습니다. 해당 변수는 나중에 `CreateBoundFunc`에서 반환한 함수가 실행될 때 언제든지 액세스할 수 있습니다.

그러나 `System.IDisposable`을 구현하는 다음(다소 인위적인) 클래스를 고려해 보세요.

C#

```
public class Resource : IDisposable
{
    private bool _isDisposed = false;
    public int Argument
    {
        get
```

```

    {
        if (!_disposed)
            return 5;
        else throw new ObjectDisposedException("Resource");
    }
}

public void Dispose()
{
    _disposed = true;
}
}

```

다음 코드에 표시된 대로 식에서 이를 사용하면 `Resource.Argument` 속성에서 참조하는 코드를 실행할 때 `System.ObjectDisposedException`을 가져옵니다.

C#

```

private static Func<int, int> CreateBoundResource()
{
    using (var constant = new Resource()) // constant is captured by the
    expression tree
    {
        Expression<Func<int, int>> expression = (b) => constant.Argument +
        b;
        var rVal = expression.Compile();
        return rVal;
    }
}

```

이 메서드에서 반환된 대리자는 `constant`를 통해 닫히고 삭제되었습니다. 이 대리자는 `using` 문에서 선언되었기 때문에 삭제되었습니다.

이제 이 메서드에서 반환된 대리자를 실행하면 실행 시점에 `ObjectDisposedException`이 `throw`됩니다.

컴파일 시간 구문을 나타내는 런타임 오류가 발생하면 이상하게 보일 수 있지만 식 트리를 사용하면 이런 환경이 시작됩니다.

이 문제에는 다양한 변형이 있으므로 이를 방지하기 위한 일반적인 지침을 제공하기는 어렵습니다. 식을 정의할 때 지역 변수에 액세스할 때 주의하고, 공용 API를 통해 반환된 식 트리를 만들 때 현재 개체(`this`로 표시됨)의 상태에 액세스할 때 주의해야 합니다.

식의 코드는 다른 어셈블리의 메서드나 속성을 참조할 수 있습니다. 해당 어셈블리는 식이 정의될 때, 컴파일될 때 및 결과 대리자가 호출될 때 액세스할 수 있어야 합니다. 존재하지 않는 경우에는 `ReferencedAssemblyNotFoundException`이 표시됩니다.

# 요약

람다 식을 나타내는 식 트리를 컴파일하면 실행할 수 있는 대리자를 만들 수 있습니다. 식 트리는 식 트리로 표시되는 코드를 실행하는 하나의 메커니즘을 제공합니다.

식 트리는 생성되는 특정 구문에 대해 실행되는 코드를 나타냅니다. 코드를 컴파일하고 실행하는 환경이 식을 만드는 환경과 일치하는 경우 모든 작업이 예상대로 작동합니다. 그렇지 않은 경우 오류는 예측 가능하며 식 트리를 사용하는 코드의 첫 번째 테스트에서 발견됩니다.

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 식 해석

아티클 • 2024. 02. 13.

다음 코드 예제에서는 람다 식 `num => num < 5`를 나타내는 식 트리를 개별 구성 요소로 구성 해제하는 방법을 보여 줍니다.

C#

```
// Add the following using directive to your code file:  
// using System.Linq.Expressions;  
  
// Create an expression tree.  
Expression<Func<int, bool>> exprTree = num => num < 5;  
  
// Decompose the expression tree.  
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];  
BinaryExpression operation = (BinaryExpression)exprTree.Body;  
ParameterExpression left = (ParameterExpression)operation.Left;  
ConstantExpression right = (ConstantExpression)operation.Right;  
  
Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",  
    param.Name, left.Name, operation.NodeType, right.Value);  
  
// This code produces the following output:  
  
// Decomposed expression: num => num LessThan 5
```

이제 식 트리의 구조를 검사하는 몇 가지 코드를 작성해 보겠습니다. 식 트리의 모든 노드는 `Expression`에서 파생된 클래스의 개체입니다.

이러한 설계는 식 트리의 모든 노드 방문 작업을 비교적 간단한 재귀 작업으로 만듭니다. 일반적인 전략은 루트 노드에서 시작하고 노드의 종류를 확인하는 것입니다.

노드 형식에 자식이 있는 경우 자식을 재귀적으로 방문합니다. 각 자식 노드에서 루트 노드에 사용된 프로세스를 반복합니다. 즉, 형식을 확인하고 형식에 자식이 있는 경우 각 자식을 방문합니다.

## 자식이 없는 식 검사

먼저 간단한 식 트리의 각 노드를 방문해 보겠습니다. 다음은 상수 식을 만든 다음 해당 속성을 검사하는 코드입니다.

C#

```
var constant = Expression.Constant(24, typeof(int));
```

```
Console.WriteLine($"This is a/an {constant.NodeType} expression type");
Console.WriteLine($"The type of the constant value is {constant.Type}");
Console.WriteLine($"The value of the constant value is {constant.Value}");
```

위에 있는 코드의 결과는 다음과 같습니다.

출력

```
This is a/an Constant expression type
The type of the constant value is System.Int32
The value of the constant value is 24
```

이제 이 식을 검색하고 식에 대해 몇 가지 중요한 속성을 작성하는 코드를 작성해 보겠습니다.

## 더하기 식

이 섹션의 소개 부분에 있는 더하기 샘플로 시작해 보겠습니다.

C#

```
Expression<Func<int>> sum = () => 1 + 2;
```

### ① 참고

대리자의 자연 형식이 `Expression<Func<int>>(0)` 아니라 `Func<int>`이기 때문에 `var(을)`를 사용하여 이 식 트리를 선언하지 마세요.

루트 노드는 `LambdaExpression`입니다. => 연산자의 오른쪽에서 흥미로운 코드를 가져오려면 `LambdaExpression`의 자식 중 하나를 찾아야 합니다. 이 섹션의 모든 식을 사용하여 이 작업을 수행합니다. 부모 노드는 `LambdaExpression`의 반환 형식을 찾는데 도움이 됩니다.

이 식의 각 노드를 검사하려면 여러 노드를 재귀적으로 방문해야 합니다. 다음은 간단한 첫 번째 구현입니다.

C#

```
Expression<Func<int, int, int>> addition = (a, b) => a + b;

Console.WriteLine($"This expression is a {addition.NodeType} expression
type");
Console.WriteLine($"The name of the lambda is {((addition.Name == null) ? "
```

```

<null>" : addition.Name}"));
Console.WriteLine($"The return type is {addition.ReturnType.ToString()}");
Console.WriteLine($"The expression has {addition.Parameters.Count}
arguments. They are:");
foreach (var argumentExpression in addition.Parameters)
{
    Console.WriteLine($"{"\tParameter Type:
{argumentExpression.Type.ToString()}, Name: {argumentExpression.Name}"});
}

var additionBody = (BinaryExpression)addition.Body;
Console.WriteLine($"The body is a {additionBody.NodeType} expression");
Console.WriteLine($"The left side is a {additionBody.Left.NodeType}
expression");
var left = (ParameterExpression)additionBody.Left;
Console.WriteLine($"{"\tParameter Type: {left.Type.ToString()}, Name:
{left.Name}"});
Console.WriteLine($"The right side is a {additionBody.Right.NodeType}
expression");
var right = (ParameterExpression)additionBody.Right;
Console.WriteLine($"{"\tParameter Type: {right.Type.ToString()}, Name:
{right.Name}"});

```

이 샘플은 다음과 같이 출력됩니다.

출력

```

This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 arguments. They are:
    Parameter Type: System.Int32, Name: a
    Parameter Type: System.Int32, Name: b
The body is a/an Add expression
The left side is a Parameter expression
    Parameter Type: System.Int32, Name: a
The right side is a Parameter expression
    Parameter Type: System.Int32, Name: b

```

이전 코드 샘플에서 많은 반복을 확인할 수 있습니다. 이러한 반복을 정리하고 보다 일반적인 용도의 식 노드 방문자를 빌드해 보겠습니다. 그러려면 재귀 알고리즘을 작성해야 합니다. 모든 노드는 자식이 있는 형식일 수 있습니다. 자식이 있는 모든 노드에서는 해당 자식을 방문하여 해당 노드의 종류를 확인해야 합니다. 다음은 재귀를 활용하여 더하기 연산을 방문하는 정리된 버전입니다.

C#

```

using System.Linq.Expressions;
namespace Visitors;

```

```

// Base Visitor class:
public abstract class Visitor
{
    private readonly Expression node;

    protected Visitor(Expression node) => this.node = node;

    public abstract void Visit(string prefix);

    public ExpressionType NodeType => node.NodeType;
    public static Visitor CreateFromExpression(Expression node) =>
        node.NodeType switch
        {
            ExpressionType.Constant => new
ConstantVisitor((ConstantExpression)node),
            ExpressionType.Lambda => new
LambdaVisitor((LambdaExpression)node),
            ExpressionType.Parameter => new
ParameterVisitor((ParameterExpression)node),
            ExpressionType.Add => new BinaryVisitor((BinaryExpression)node),
            _ => throw new NotImplementedException($"Node not processed yet:
{node.NodeType}"),
        };
}

// Lambda Visitor
public class LambdaVisitor : Visitor
{
    private readonly LambdaExpression node;
    public LambdaVisitor(LambdaExpression node) : base(node) => this.node =
node;

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType}
expression type");
        Console.WriteLine($"{prefix}The name of the lambda is {{({node.Name
== null) ? "<null>" : node.Name})}");
        Console.WriteLine($"{prefix}The return type is {node.ReturnType}");
        Console.WriteLine($"{prefix}The expression has
{node.Parameters.Count} argument(s). They are:");
        // Visit each parameter:
        foreach (var argumentExpression in node.Parameters)
        {
            var argumentVisitor = CreateFromExpression(argumentExpression);
            argumentVisitor.Visit(prefix + "\t");
        }
        Console.WriteLine($"{prefix}The expression body is:");
        // Visit the body:
        var bodyVisitor = CreateFromExpression(node.Body);
        bodyVisitor.Visit(prefix + "\t");
    }
}

// Binary Expression Visitor:

```

```

public class BinaryVisitor : Visitor
{
    private readonly BinaryExpression node;
    public BinaryVisitor(BinaryExpression node) : base(node) => this.node =
node;

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This binary expression is a {NodeType} expression");
        var left = CreateFromExpression(node.Left);
        Console.WriteLine($"{prefix}The Left argument is:");
        left.Visit(prefix + "\t");
        var right = CreateFromExpression(node.Right);
        Console.WriteLine($"{prefix}The Right argument is:");
        right.Visit(prefix + "\t");
    }
}

// Parameter visitor:
public class ParameterVisitor : Visitor
{
    private readonly ParameterExpression node;
    public ParameterVisitor(ParameterExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}Type: {node.Type}, Name: {node.Name},
ByRef: {node.IsByRef}");
    }
}

// Constant visitor:
public class ConstantVisitor : Visitor
{
    private readonly ConstantExpression node;
    public ConstantVisitor(ConstantExpression node) : base(node) =>
this.node = node;

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}The type of the constant value is
{node.Type}");
        Console.WriteLine($"{prefix}The value of the constant value is
{node.Value}");
    }
}

```

이 알고리즘은 임의의 `LambdaExpression`(을)를 방문하는 알고리즘의 기초입니다. 만든 코드는 발생할 수 있는 식 트리 노드 집합의 작은 샘플만 찾습니다. 그러나 생성되는 항목에서 많은 부분을 익힐 수 있습니다. `visitor.CreateFromExpression` 메서드의 기본 사례에서는 새 노드 형식이 나타나면 오류 콘솔에 메시지를 출력합니다. 이런 방식으로 새로운 식 형식을 추가합니다.

이전 추가 식에서 이 방문자를 실행하면 다음 출력이 표시됩니다.

```
출력

This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
    This is an Parameter expression type
    Type: System.Int32, Name: a, ByRef: False
    This is an Parameter expression type
    Type: System.Int32, Name: b, ByRef: False
The expression body is:
    This binary expression is a Add expression
    The Left argument is:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
    The Right argument is:
        This is an Parameter expression type
        Type: System.Int32, Name: b, ByRef: False
```

보다 일반적인 방문자 구현을 구축했으므로 훨씬 더 다양한 형식을 식을 방문하여 처리할 수 있습니다.

## 피연산자를 더 많이 사용하는 추가 식

더 복잡한 예제를 시도해 보겠지만 노드 형식은 여전히 더하기로만 제한합니다.

```
C#

Expression<Func<int>> sum = () => 1 + 2 + 3 + 4;
```

방문자 알고리즘에서 이러한 예제를 실행하기 전에 생각 연습을 수행하여 출력이 무엇인지 알아보세요. + 연산자는 이진 연산자임을 기억하세요. 이 연산자에는 왼쪽과 오른쪽 피연산자를 나타내는 두 개의 자식이 있어야 합니다. 여러 가지 방법으로 올바른 트리를 생성할 수 있습니다.

```
C#
```

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
Expression<Func<int>> sum2 = () => ((1 + 2) + 3) + 4;

Expression<Func<int>> sum3 = () => (1 + 2) + (3 + 4);
Expression<Func<int>> sum4 = () => 1 + ((2 + 3) + 4);
Expression<Func<int>> sum5 = () => (1 + (2 + 3)) + 4;
```

가장 유망한 해답을 강조하기 위해 두 가지 가능한 해답으로 분리된 것을 볼 수 있습니다. 첫 번째는 오른쪽 결합성 식을 나타내고, 두 번째는 왼쪽 결합형 식을 나타냅니다. 이러한 두 형식의 장점은 형식이 임의 개수의 더하기 식으로 확장된다는 점입니다.

방문자를 통해 이 식을 실행하면 단순 추가 식이 왼쪽 결합인지 확인하여 이 출력이 표시됩니다.

이 샘플을 실행하고 전체 식 트리를 보려면 원본 식 트리를 한 번 변경합니다. 식 트리에 모든 상수가 포함되어 있으면 결과 트리에는 상수 값 10만 포함됩니다. 컴파일러는 모든 더하기를 수행하고 가장 간단한 형태로 식을 줄입니다. 식에서 하나의 변수를 추가하기만 하면 원래 트리를 확인하는 데 충분합니다.

C#

```
Expression<Func<int, int>> sum = (a) => 1 + a + 3 + 4;
```

이 합계에 대한 방문자를 만들고 이 출력이 표시되는 방문자를 실행합니다.

출력

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
    This is an Parameter expression type
    Type: System.Int32, Name: a, ByRef: False
The expression body is:
    This binary expression is a Add expression
    The Left argument is:
        This binary expression is a Add expression
        The Left argument is:
            This binary expression is a Add expression
            The Left argument is:
                This is an Constant expression type
                The type of the constant value is
System.Int32
                The value of the constant value is 1
    The Right argument is:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
    The Right argument is:
```

```
This is an Constant expression type  
The type of the constant value is System.Int32  
The value of the constant value is 3  
The Right argument is:  
This is an Constant expression type  
The type of the constant value is System.Int32  
The value of the constant value is 4
```

방문자 코드를 통해 다른 샘플을 실행하고 해당 샘플이 나타내는 트리를 확인할 수 있습니다. 다음은 이전 `sum3` 식의 예입니다(컴파일러가 상수를 계산하지 못하도록 하는 추가 매개 변수 포함).

C#

```
Expression<Func<int, int, int>> sum3 = (a, b) => (1 + a) + (3 + b);
```

방문자의 출력은 다음과 같습니다.

출력

```
This expression is a/an Lambda expression type  
The name of the lambda is <null>  
The return type is System.Int32  
The expression has 2 argument(s). They are:  
    This is an Parameter expression type  
    Type: System.Int32, Name: a, ByRef: False  
    This is an Parameter expression type  
    Type: System.Int32, Name: b, ByRef: False  
The expression body is:  
    This binary expression is a Add expression  
    The Left argument is:  
        This binary expression is a Add expression  
        The Left argument is:  
            This is an Constant expression type  
            The type of the constant value is System.Int32  
            The value of the constant value is 1  
    The Right argument is:  
        This is an Parameter expression type  
        Type: System.Int32, Name: a, ByRef: False  
The Right argument is:  
    This binary expression is a Add expression  
    The Left argument is:  
        This is an Constant expression type  
        The type of the constant value is System.Int32  
        The value of the constant value is 3  
    The Right argument is:  
        This is an Parameter expression type  
        Type: System.Int32, Name: b, ByRef: False
```

괄호는 출력의 일부가 아닙니다. 식 트리에는 입력 식의 괄호를 나타내는 노드가 없습니다. 식 트리의 구조에는 우선 순위를 전달하는 데 필요한 모든 정보가 포함됩니다.

## 이 샘플 확장

이 샘플은 가장 기본적인 식 트리만 처리합니다. 이 섹션에서 살펴본 코드는 상수 정수와 이진 + 연산자만 처리합니다. 최종 샘플로 방문자를 업데이트하여 더 복잡한 식을 처리해 보겠습니다. 다음 팩터리 식에 대해 작동하도록 하겠습니다.

C#

```
Expression<Func<int, int>> factorial = (n) =>
    n == 0 ?
    1 :
    Enumerable.Range(1, n).Aggregate((product, factor) => product * factor);
```

이 코드는 수학 계승 함수에 가능한 한 가지 구현을 나타냅니다. 코드를 이렇게 작성하면 식에 람다 식을 할당하여 식 트리를 빌드하는데 두 가지 제한 사항을 강조합니다. 첫째, 문 람다는 허용되지 않습니다. 즉, C#에서 공통된 루프, 블록, if/else 문 및 기타 제어 구조를 사용할 수 없습니다. 식 사용으로 제한됩니다. 둘째, 동일한 식을 재귀적으로 호출할 수 없습니다. 이미 대리자인 경우 가능하지만 식 트리 양식에서 호출할 수 없습니다. [식 트리 빌드](#) 섹션에서는 이러한 제한 사항을 극복하는 기술을 알아봅니다.

이 식에서는 다음과 같은 형식의 노드가 모두 나타납니다.

1. 같음(이진 식)
2. 곱하기(이진 식)
3. 조건부( ? : 식)
4. 메서드 호출 식(Range() 및 Aggregate() 호출)

방문자 알고리즘을 수정하는 한 가지 방법은 해당 알고리즘을 계속 실행하고 `default` 절에 도달할 때마다 노드 형식을 기록하는 것입니다. 몇 번 반복 후에는 각 잠재적 노드가 표시됩니다. 그러면 다 끝났습니다. 결과는 다음과 같이 나타납니다.

C#

```
public static Visitor CreateFromExpression(Expression node) =>
    node.NodeType switch
    {
        ExpressionType.Constant => new ConstantVisitor((ConstantExpression)node),
        ExpressionType.Lambda     => new LambdaVisitor((LambdaExpression)node),
        ExpressionType.Parameter  => new ParameterVisitor((ParameterExpression)node),
```

```

        ExpressionType.Add      => new
BinaryVisitor((BinaryExpression)node),
        ExpressionType.Equal    => new
BinaryVisitor((BinaryExpression)node),
        ExpressionType.Multiply   => new BinaryVisitor((BinaryExpression)
node),
        ExpressionType.Conditional => new
ConditionalVisitor((ConditionalExpression) node),
        ExpressionType.Call      => new
MethodCallVisitor((MethodCallExpression) node),
        _ => throw new NotImplementedException($"Node not processed yet:
{node.NodeType}"),
    };

```

ConditionalVisitor 및 MethodCallVisitor 두 노드를 처리합니다.

C#

```

public class ConditionalVisitor : Visitor
{
    private readonly ConditionalExpression node;
    public ConditionalVisitor(ConditionalExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType}
expression");
        var testVisitor = Visitor.CreateFromExpression(node.Test);
        Console.WriteLine($"{prefix}The Test for this expression is:");
        testVisitor.Visit(prefix + "\t");
        var trueVisitor = Visitor.CreateFromExpression(node.IfTrue);
        Console.WriteLine($"{prefix}The True clause for this expression
is:");
        trueVisitor.Visit(prefix + "\t");
        var falseVisitor = Visitor.CreateFromExpression(node.IfFalse);
        Console.WriteLine($"{prefix}The False clause for this expression
is:");
        falseVisitor.Visit(prefix + "\t");
    }
}

public class MethodCallVisitor : Visitor
{
    private readonly MethodCallExpression node;
    public MethodCallVisitor(MethodCallExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)

```

```

{
    Console.WriteLine($"{prefix}This expression is a {NodeType}
expression");
    if (node.Object == null)
        Console.WriteLine($"{prefix}This is a static method call");
    else
    {
        Console.WriteLine($"{prefix}The receiver (this) is:");
        var receiverVisitor = Visitor.CreateFromExpression(node.Object);
        receiverVisitor.Visit(prefix + "\t");

        var MethodInfo = node.Method;
        Console.WriteLine($"{prefix}The method name is
{MethodInfo.DeclaringType}.{MethodInfo.Name}");
        // There is more here, like generic arguments, and so on.
        Console.WriteLine($"{prefix}The Arguments are:");
        foreach (var arg in node.Arguments)
        {
            var argVisitor = Visitor.CreateFromExpression(arg);
            argVisitor.Visit(prefix + "\t");
        }
    }
}

```

식 트리에 대한 출력은 다음과 같습니다.

#### 출력

```

This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
    This is an Parameter expression type
    Type: System.Int32, Name: n, ByRef: False
The expression body is:
    This expression is a Conditional expression
    The Test for this expression is:
        This binary expression is a Equal expression
        The Left argument is:
            This is an Parameter expression type
            Type: System.Int32, Name: n, ByRef: False
        The Right argument is:
            This is an Constant expression type
            The type of the constant value is System.Int32
            The value of the constant value is 0
    The True clause for this expression is:
        This is an Constant expression type
        The type of the constant value is System.Int32
        The value of the constant value is 1
    The False clause for this expression is:
        This expression is a Call expression
        This is a static method call

```

```

The method name is System.Linq.Enumerable.Aggregate
The Arguments are:
    This expression is a Call expression
    This is a static method call
    The method name is System.Linq.Enumerable.Range
    The Arguments are:
        This is an Constant expression type
        The type of the constant value is
System.Int32
        The value of the constant value is 1
        This is an Parameter expression type
        Type: System.Int32, Name: n, ByRef: False
This expression is a Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 arguments. They are:
    This is an Parameter expression type
    Type: System.Int32, Name: product, ByRef:
False
    This is an Parameter expression type
    Type: System.Int32, Name: factor, ByRef:
False
The expression body is:
    This binary expression is a Multiply
expression
    The Left argument is:
        This is an Parameter expression type
        Type: System.Int32, Name: product,
ByRef: False
    The Right argument is:
        This is an Parameter expression type
        Type: System.Int32, Name: factor,
ByRef: False

```

## 샘플 라이브러리 확장

이 섹션의 샘플은 식 트리의 노드를 방문하여 검사하는 핵심 기술을 보여 줍니다. 식 트리에서 노드를 방문하고 액세스하는 핵심 작업에 집중하기 위해 발생할 노드 유형을 간소화했습니다.

첫째, 방문자는 정수인 상수만 처리합니다. 상수 값은 다른 모든 숫자 형식이 될 수 있으며 C# 언어는 해당 형식 간의 변환 및 승격을 지원합니다. 이 코드의 보다 강력한 버전은 이러한 모든 기능을 미러링합니다.

마지막 예제도 가능한 노드 형식의 하위 집합을 인식합니다. 여전히 실패하게 만드는 많은 식을 공급할 수 있습니다. 전체 구현은 [ExpressionVisitor](#)라는 이름으로 .NET Standard에 포함되며 가능한 노드 형식을 모두 처리할 수 있습니다.

마지막으로 이 문서에 사용된 라이브러리는 데모 및 학습을 위해 빌드되었습니다. 최적화되지 않았습니다. 구조를 명확히 하고 노드를 방문하고 해당 노드를 분석하는 데 사용되는 기술을 강조 표시합니다.

이러한 제한 사항에도 불구하고 식 트리를 읽고 이해하는 알고리즘 작성을 완료해야 합니다.

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

💬 설명서 문제 열기

↗️ 제품 사용자 의견 제공

# 식 트리 빌드

아티클 • 2024. 02. 17.

C# 컴파일러는 지금까지 본 모든 식 트리를 만들었습니다. `Expression<Func<T>>` 또는 유사한 형식으로 형식이 할당된 변수에 할당된 람다 식을 만들었습니다. 많은 시나리오에서 런타임 시 메모리에 식을 빌드합니다.

식 트리는 변경할 수 없습니다. 변경할 수 없다는 것은 리프에서 루트까지 위로 트리를 작성해야 한다는 의미입니다. 식 트리를 작성하는 데 사용할 API에 이 사실이 반영됩니다. 즉, 노드를 작성하는 데 사용되는 메서드는 모든 자식을 인수로 사용합니다. 이 기술을 보여 주는 몇 가지 예제를 살펴보겠습니다.

## 노드 만들기

다음 섹션 전체에서 작업해 온 덧셈 식으로 시작합니다.

C#

```
Expression<Func<int>> sum = () => 1 + 2;
```

해당 식 트리를 구성하려면 먼저 리프 노드를 구성합니다. 리프 노드는 상수입니다.

`Constant` 메서드를 사용하여 노드를 만듭니다.

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
```

다음으로 추가 식을 빌드합니다.

C#

```
var addition = Expression.Add(one, two);
```

덧셈 식을 빌드한 후에는 람다 식을 만듭니다.

C#

```
var lambda = Expression.Lambda(addition);
```

이 람다 식에는 인수가 없습니다. 이 섹션의 뒷부분에서는 인수를 매개 변수에 매핑하고 더 복잡한 식을 작성하는 방법을 살펴봅니다.

이와 같은 식의 경우 모든 호출을 단일 문으로 결합할 수 있습니다.

C#

```
var lambda2 = Expression.Lambda(
    Expression.Add(
        Expression.Constant(1, typeof(int)),
        Expression.Constant(2, typeof(int))
    )
);
```

## 트리 빌드

이전 섹션에서는 메모리에 식 트리를 빌드하는 기본 사항을 보여 주었습니다. 좀 더 복잡한 트리는 일반적으로 노드 유형과 트리의 노드가 더 많음을 의미합니다. 예제를 하나 더 실행하고 식 트리를 만들 때 일반적으로 작성하는 인수 노드와 메서드 호출 노드라는 노드 유형을 두 개 더 살펴보겠습니다. 식 트리를 작성하여 다음 식을 만들어 보겠습니다.

C#

```
Expression<Func<double, double, double>> distanceCalc =
    (x, y) => Math.Sqrt(x * x + y * y);
```

x 및 y에 대한 매개 변수 식을 만드는 것부터 시작합니다.

C#

```
var xParameter = Expression.Parameter(typeof(double), "x");
var yParameter = Expression.Parameter(typeof(double), "y");
```

곱하기와 더하기 식을 만들 때 이미 살펴본 패턴을 따릅니다.

C#

```
var xSquared = Expression.Multiply(xParameter, xParameter);
var ySquared = Expression.Multiply(yParameter, yParameter);
var sum = Expression.Add(xSquared, ySquared);
```

다음으로 Math.Sqrt를 호출하기 위한 메서드 호출 식을 만들어야 합니다.

C#

```
var sqrtMethod = typeof(Math).GetMethod("Sqrt", new[] { typeof(double) }) ??
    throw new InvalidOperationException("Math.Sqrt not found!");
var distance = Expression.Call(sqrtMethod, sum);
```

메서드를 찾을 수 없는 경우 `GetMethod` 호출은 `null`을 반환할 수 있습니다. 메서드 이름의 철자를 잘못 입력했기 때문일 가능성이 높습니다. 그렇지 않으면 필요한 어셈블리가 로드되지 않았음을 의미할 수 있습니다. 마지막으로 메서드 호출을 람다 식에 넣고 람다 식에 대한 인수를 정의해야 합니다.

C#

```
var distanceLambda = Expression.Lambda(
    distance,
    xParameter,
    yParameter);
```

더 복잡한 이 예제에서는 식 트리를 만드는 데 자주 필요한 기술을 몇 가지 더 확인할 수 있습니다.

먼저 매개 변수 또는 지역 변수를 나타내는 개체를 만든 후에 사용해야 합니다. 이러한 개체를 만들었으면 필요할 때마다 식 트리에서 사용할 수 있습니다.

두 번째로 해당 메서드에 액세스하는 식 트리를 만들 수 있도록 리플렉션 API의 하위 집합을 사용하여 `System.Reflection.MethodInfo` 개체를 만들어야 합니다. .NET Core 플랫폼에서 사용할 수 있는 리플렉션 API의 하위 집합으로 제한해야 합니다. 다시 말하지만 이러한 기술은 다른 식 트리로 확장됩니다.

## 심층적인 코드 빌드

이러한 API를 사용하여 빌드할 수 있는 항목으로 제한되지 않습니다. 그러나 작성하려는 식 트리가 복잡할수록 코드를 관리하고 읽기가 더 어려워집니다.

다음 코드에 해당하는 식 트리를 작성해 보겠습니다.

C#

```
Func<int, int> factorialFunc = (n) =>
{
    var res = 1;
    while (n > 1)
    {
        res = res * n;
        n--;
    }
}
```

```
    return res;  
};
```

앞의 코드는 식 트리를 빌드하지 않고 단순히 대리자를 빌드했습니다. `Expression` 클래스를 사용하여 문 람다를 빌드할 수 없습니다. 다음은 동일한 기능을 빌드하는 데 필요한 코드입니다. `while` 루프를 빌드하기 위한 API는 없습니다. 대신 조건부 테스트가 포함된 루프와 루프에서 벗어날 레이블 대상을 빌드해야 합니다.

C#

```
var nArgument = Expression.Parameter(typeof(int), "n");  
var result = Expression.Variable(typeof(int), "result");  
  
// Creating a label that represents the return value  
LabelTarget label = Expression.Label(typeof(int));  
  
var initializeResult = Expression.Assign(result, Expression.Constant(1));  
  
// This is the inner block that performs the multiplication,  
// and decrements the value of 'n'  
var block = Expression.Block(  
    Expression.Assign(result,  
        Expression.Multiply(result, nArgument)),  
    Expression.PostDecrementAssign(nArgument)  
);  
  
// Creating a method body.  
BlockExpression body = Expression.Block(  
    new[] { result },  
    initializeResult,  
    Expression.Loop(  
        Expression.IfThenElse(  
            Expression.GreaterThan(nArgument, Expression.Constant(1)),  
            block,  
            Expression.Break(label, result)  
        ),  
        label  
    )  
);
```

계승 함수에 대한 식 트리를 작성하는 코드는 훨씬 더 길고 더 복잡하며, 레이블과 break 문 및 일상적인 코딩 작업에서 방지하려는 기타 요소로 인해 복잡해집니다.

이 섹션에서는 이 식 트리의 모든 노드를 방문하고 이 샘플에서 만들어진 노드에 대한 정보를 작성하는 코드를 작성했습니다. GitHub의 dotnet/docs 리포지토리에서 [샘플 코드를 보거나 다운로드](#) 할 수 있습니다. 샘플을 빌드하고 실행하여 직접 실험합니다.

## 코드 구문을 식에 매핑

다음 코드 예에서는 API를 사용하여 람다 식 `num => num < 5`를 나타내는 식 트리를 보여 줍니다.

C#

```
// Manually build the expression tree for
// the lambda expression num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });
```

식 트리 API는 루프, 조건부 블록 및 `try-catch` 블록과 같은 할당 및 제어 흐름 식도 지원합니다. API를 사용하면 C# 컴파일러를 통해 람다 식에서 작성할 수 있는 것보다 복잡한 식 트리를 만들 수 있습니다. 다음 예제에서는 숫자의 계승을 계산하는 식 트리를 만드는 방법을 보여 줍니다.

C#

```
// Creating a parameter expression.
ParameterExpression value = Expression.Parameter(typeof(int), "value");

// Creating an expression to hold a local variable.
ParameterExpression result = Expression.Parameter(typeof(int), "result");

// Creating a label to jump to from a loop.
LabelTarget label = Expression.Label(typeof(int));

// Creating a method body.
BlockExpression block = Expression.Block(
    // Adding a local variable.
    new[] { result },
    // Assigning a constant to a local variable: result = 1
    Expression.Assign(result, Expression.Constant(1)),
    // Adding a loop.
    Expression.Loop(
        // Adding a conditional block into the loop.
        Expression.IfThenElse(
            // Condition: value > 1
            Expression.GreaterThan(value, Expression.Constant(1)),
            // If true: result *= value --
            Expression.MultiplyAssign(result,
                Expression.PostDecrementAssign(value)),
            // If false, exit the loop and go to the label.
            Expression.Break(label, result)
        ),
        // Label to jump to.
        label
    )
);
```

```
)  
;  
  
// Compile and execute an expression tree.  
int factorial = Expression.Lambda<Func<int, int>>(block, value).Compile()  
(5);  
  
Console.WriteLine(factorial);  
// Prints 120.
```

자세한 내용은 [Generating Dynamic Methods with Expression Trees in Visual Studio 2010](#) (Visual Studio 2010에서 식 트리를 사용하여 동적 메서드 생성)을 참조하세요. 이 내용은 Visual Studio의 최신 버전에도 적용됩니다.

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

[설명서 문제 열기](#)

[제품 사용자 의견 제공](#)

# 식 트리 변환

아티클 • 2024. 02. 21.

이 문서에서는 식 트리의 각 노드를 방문하고 해당 식 트리의 수정된 복사본을 작성하는 방법을 알아봅니다. 다른 환경으로 변환할 수 있도록 식 트리를 변환하여 알고리즘을 이해합니다. 생성된 알고리즘을 변경합니다. 로깅을 추가하고, 메서드 호출을 가로채고, 추적하거나, 다른 용도로 사용할 수 있습니다.

식 트리를 변환하기 위해 작성하는 코드는 트리의 모든 노드를 방문하기 위해 이미 살펴본 코드의 확장입니다. 식 트리를 변환할 때는 모든 노드를 방문하고 노드를 방문하는 동안 새 트리를 작성합니다. 새 트리에는 원래 노드에 대한 참조 또는 트리에 배치한 새 노드가 포함될 수 있습니다.

식 트리를 방문하여 일부 대체 노드가 있는 새 트리를 만들어 보겠습니다. 이 예제에서는 특정 상수를 10배 더 큰 상수로 대체합니다. 그러지 않으면 식 트리를 그대로 유지합니다. 상수의 값을 읽고 새 상수로 대체하는 대신 곱하기를 수행하는 새 노드로 상수 노드를 대체하여 이를 대체합니다.

여기에서 상수 노드를 찾은 다음에는 자식이 원래 상수 및 상수 10인 새 곱하기 노드를 만듭니다.

C#

```
private static Expression ReplaceNodes(Expression original)
{
    if (original.NodeType == ExpressionType.Constant)
    {
        return Expression.Multiply(original, Expression.Constant(10));
    }
    else if (original.NodeType == ExpressionType.Add)
    {
        var binaryExpression = (BinaryExpression)original;
        return Expression.Add(
            ReplaceNodes(binaryExpression.Left),
            ReplaceNodes(binaryExpression.Right));
    }
    return original;
}
```

원래 노드를 대체 노드로 바꿔 새 트리를 만듭니다. 대체된 트리를 컴파일하고 실행하여 변경 내용을 확인합니다.

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
```

```

var addition = Expression.Add(one, two);
var sum = ReplaceNodes(addition);
var executableFunc = Expression.Lambda(sum);

var func = (Func<int>)executableFunc.Compile();
var answer = func();
Console.WriteLine(answer);

```

새 트리 작성은 기존 트리의 노드를 방문하고 새 노드를 만들어 트리에 삽입하는 작업의 조합입니다. 이전 예제에서는 변경할 수 없는 식 트리의 중요도를 보여 줍니다. 이전 코드에서 만든 새 트리에는 새로 만든 노드와 기존 트리의 노드가 함께 포함됩니다. 기존 트리의 노드를 수정할 수 없으므로 두 트리에서 노드를 사용할 수 있습니다. 노드를 다시 사용하면 상당한 메모리 효율성이 발생합니다. 트리 전체 또는 여러 식 트리에서 같은 노드를 사용할 수 있습니다. 노드는 수정할 수 없기 때문에 필요할 때마다 같은 노드를 다시 사용할 수 있습니다.

## 추가 트래버스 및 실행

추가 노드의 트리를 이동하고 결과를 계산하는 두 번째 방문자를 작성하여 확인해 보겠습니다. 지금까지 본 방문자를 몇 가지 수정합니다. 이 새 버전에서 방문자는 이 지점까지 더하기 작업의 부분합을 반환합니다. 상수 식의 경우 단순히 상수 식의 값입니다. 더하기 식의 경우 해당 트리가 트래버스된 후의 결과는 왼쪽 및 오른쪽 피연산자의 합계입니다.

C#

```

var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var three = Expression.Constant(3, typeof(int));
var four = Expression.Constant(4, typeof(int));
var addition = Expression.Add(one, two);
var add2 = Expression.Add(three, four);
var sum = Expression.Add(addition, add2);

// Declare the delegate, so you can call it
// from itself recursively:
Func<Expression, int> aggregate = null!;
// Aggregate, return constants, or the sum of the left and right operand.
// Major simplification: Assume every binary expression is an addition.
aggregate = (exp) =>
    exp.NodeType == ExpressionType.Constant ?
        (int)((ConstantExpression)exp).Value :
        aggregate(((BinaryExpression)exp).Left) +
        aggregate(((BinaryExpression)exp).Right);

var theSum = aggregate(sum);
Console.WriteLine(theSum);

```

코드는 상당히 많지만 개념은 쉽게 이해할 수 있습니다. 이 코드는 깊이 우선 검색으로 자식을 방문합니다. 상수 노드가 나타나면 방문자는 상수의 값을 반환합니다. 방문자가 두 자식을 방문한 후에 자식은 해당 하위 트리에 대해 계산된 합계를 계산합니다. 이제 더하기 노드에서 해당 합계를 컴퓨팅할 수 있습니다. 식 트리의 모든 노드를 방문하면 합계가 계산됩니다. 디버거에서 샘플을 실행하고 실행을 추적하여 실행을 추적할 수 있습니다.

노드가 분석되는 방법 및 트리를 트래버스하여 합계를 계산하는 방법을 더 쉽게 추적할 수 있도록 만들어 보겠습니다. 다음은 매우 많은 추적 정보를 포함하는 집계 메서드의 업데이트된 버전입니다.

C#

```
private static int Aggregate(Expression exp)
{
    if (exp.NodeType == ExpressionType.Constant)
    {
        var constantExp = (ConstantExpression)exp;
        Console.Error.WriteLine($"Found Constant: {constantExp.Value}");
        if (constantExp.Value is int value)
        {
            return value;
        }
        else
        {
            return 0;
        }
    }
    else if (exp.NodeType == ExpressionType.Add)
    {
        var addExp = (BinaryExpression)exp;
        Console.Error.WriteLine("Found Addition Expression");
        Console.Error.WriteLine("Computing Left node");
        var leftOperand = Aggregate(addExp.Left);
        Console.Error.WriteLine($"Left is: {leftOperand}");
        Console.Error.WriteLine("Computing Right node");
        var rightOperand = Aggregate(addExp.Right);
        Console.Error.WriteLine($"Right is: {rightOperand}");
        var sum = leftOperand + rightOperand;
        Console.Error.WriteLine($"Computed sum: {sum}");
        return sum;
    }
    else throw new NotSupportedException("Haven't written this yet");
}
```

`sum` 식에서 이 메서드를 실행하면 다음과 같이 출력됩니다.

출력

10

Found Addition Expression

```
Computing Left node
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Constant: 2
Right is: 2
Computed sum: 3
Left is: 3
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 10
10
```

출력을 추적하고 이전 코드에서 따릅니다. 코드가 트리를 이동하고 합계를 찾을 때 각 노드를 방문하고 합계를 계산하는 방법을 이해할 수 있어야 합니다.

이제 `sum1`에 지정된 식을 사용하여 다른 실행을 살펴보겠습니다.

C#

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
```

이 식을 검사한 출력은 다음과 같습니다.

출력

```
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 2
Left is: 2
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
```

```
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 9
Right is: 9
Computed sum: 10
10
```

최종 해답은 같지만 트리 탐색은 다릅니다. 먼저 발생하는 다른 작업으로 트리가 생성되었기 때문에 다른 순서로 노드를 이동합니다.

## 수정된 복사본 만들기

콘솔 애플리케이션 프로젝트를 새로 만듭니다. `System.Linq.Expressions` 네임스페이스에 대한 `using` 지시문을 파일에 추가합니다. 프로젝트에 `AndAlsoModifier` 클래스를 추가합니다.

C#

```
public class AndAlsoModifier : ExpressionVisitor
{
    public Expression Modify(Expression expression)
    {
        return Visit(expression);
    }

    protected override Expression VisitBinary(BinaryExpression b)
    {
        if (b.NodeType == ExpressionType.AndAlso)
        {
            Expression left = this.Visit(b.Left);
            Expression right = this.Visit(b.Right);

            // Make this binary expression an OrElse operation instead of an
            // AndAlso operation.
            return Expression.MakeBinary(ExpressionType.OrElse, left, right,
                b.IsLiftedToNull, b.Method);
        }

        return base.VisitBinary(b);
    }
}
```

이 클래스는 `ExpressionVisitor` 클래스를 상속하며, 조건부 `AND` 작업을 나타내는 식을 수정하도록 특수화되었습니다. 해당 작업을 조건부 `AND`에서 조건부 `OR`로 변경합니다. 조건부 `AND` 식은 이진 식으로 표현되기 때문에 클래스는 기본 형식의 `VisitBinary` 메서드를 재정의합니다. `VisitBinary` 메서드에서 전달된 식이 조건부 `AND` 작업을 나타내는 경우 코

드는 조건부 `AND` 연산자 대신 조건부 `OR` 연산자가 포함된 새 식을 생성합니다.

`VisitBinary`에 전달된 식이 조건부 `AND` 작업을 나타내지 않는 경우 메서드는 기본 클래스 구현을 따릅니다. 기본 클래스 메서드는 전달된 식 트리와 유사한 노드를 생성하지만 노드의 하위 트리가 방문자에 의해 재귀적으로 생성된 식 트리로 바뀌었습니다.

`System.Linq.Expressions` 네임스페이스에 대한 `using` 지시문을 파일에 추가합니다.

Program.cs 파일의 `Main` 메서드에 코드를 추가하여 식 트리를 만들고 이 식 트리를 수정할 메서드에 전달합니다.

C#

```
Expression<Func<string, bool>> expr = name => name.Length > 10 &&
name.StartsWith("G");
Console.WriteLine(expr);

AndAlsoModifier treeModifier = new AndAlsoModifier();
Expression modifiedExpr = treeModifier.Modify((Expression)expr);

Console.WriteLine(modifiedExpr);

/* This code produces the following output:

name => ((name.Length > 10) && name.StartsWith("G"))
name => ((name.Length > 10) || name.StartsWith("G"))
*/
```

코드에서 조건부 `AND` 작업이 포함된 식을 만듭니다. 그런 다음 `AndAlsoModifier` 클래스 인스턴스를 만들고 이 클래스의 `Modify` 메서드에 식을 전달합니다. 원본 및 수정된 식 트리가 둘 다 출력되어 변경 내용을 표시합니다. 애플리케이션을 컴파일하고 실행합니다.

## 자세한 정보

이 샘플에서는 식 트리로 표시되는 알고리즘을 트래버스하고 해석하기 위해 작성하는 코드의 작은 하위 집합을 보여 줍니다. 식 트리를 다른 언어로 번역하는 범용 라이브러리를 빌드하는 방법에 대한 자세한 내용은 Matt Warren의 [이 시리즈](#)를 참조하세요. 이 시리즈는 식 트리에서 찾을 수 있는 코드를 변환하는 방법에 대해 상세히 설명합니다.

이제 식 트리의 진정한 기능을 살펴보았습니다. 코드 집합을 검사하고, 해당 코드를 원하는 대로 변경하고, 변경된 버전을 실행합니다. 식 트리는 변경할 수 없기 때문에 기존 트리의 구성 요소를 사용하여 새 트리를 만듭니다. 노드를 다시 사용하면 수정된 식 트리를 만드는 데 필요한 메모리 양이 최소화됩니다.

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

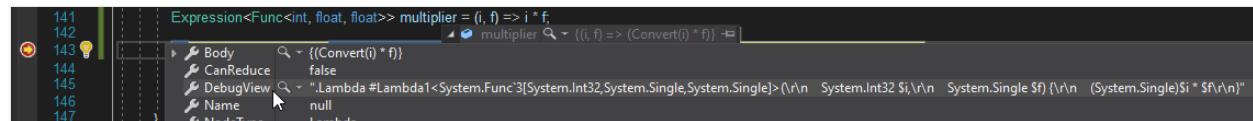
 설명서 문제 열기

 제품 사용자 의견 제공

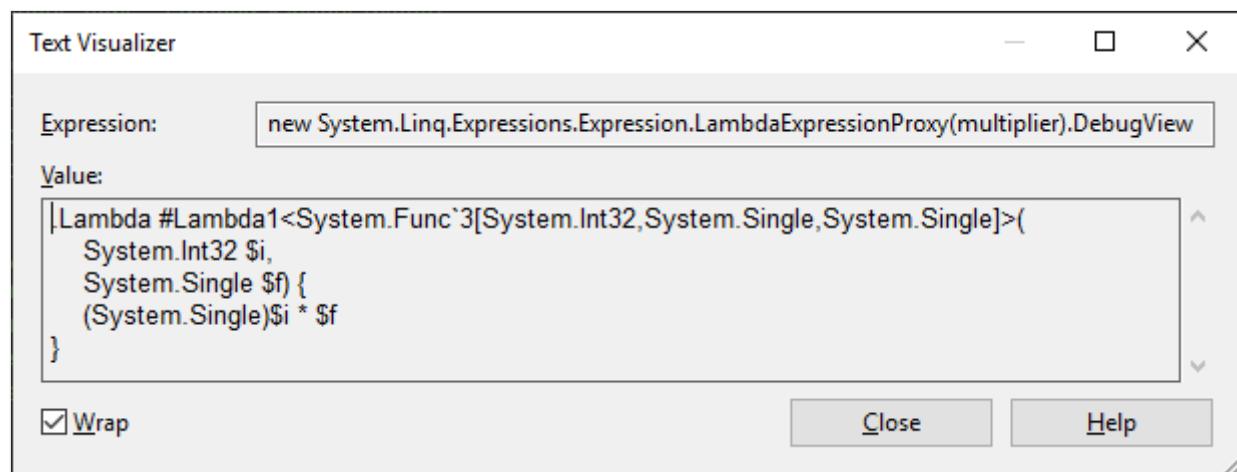
# Visual Studio에서 식 트리 디버그

아티클 • 2024. 02. 18.

애플리케이션을 디버그할 때 식 트리의 구조 및 내용을 분석할 수 있습니다. 식 트리 구조에 대한 간략한 개요를 보려면 [특수 구문](#)을 사용하여 식 트리를 나타내는 `DebugView` 속성을 사용합니다. `DebugView`는 디버그 모드에서만 사용할 수 있습니다.

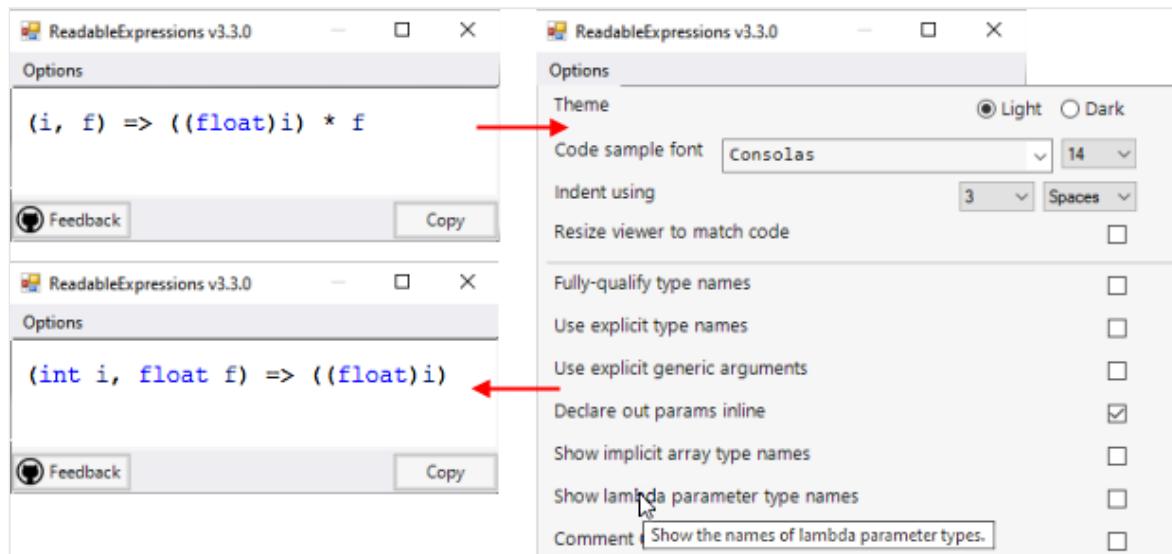


`DebugView`는 문자열이므로 `DebugView` 레이블 옆의 돋보기 아이콘에서 [텍스트 시작화 도우미](#)를 선택하여 [기본 제공 텍스트 시작화 도우미](#)를 사용하여 여러 줄에서 볼 수 있습니다.

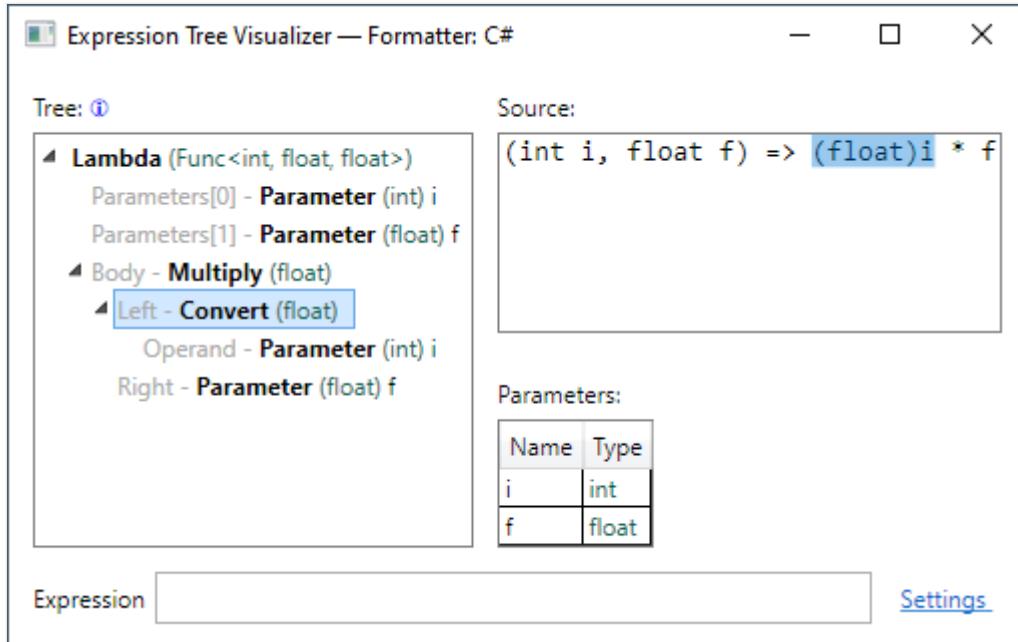


또는 다음과 같은 식 트리에 사용자 지정 시작화 도우미를 설치하여 사용할 수 있습니다.

- 읽을 수 있는 식 [\(MIT 라이선스\)](#), [Visual Studio Marketplace](#)에서 사용 가능)은 다양한 렌더링 옵션을 사용하여 식 트리를 테마 지정 가능 C# 코드로 렌더링합니다.

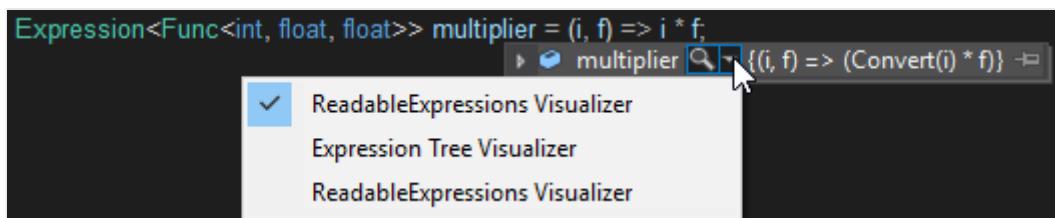


- 식 트리 시각화 도우미 ([MIT 라이선스](#))에서는 식 트리 및 개별 노드의 트리 뷔를 제공합니다.



## 식 트리에 대한 시각화 도우미 열기

DataTips, 조사식 창, 자동 창 또는 지역 창의 식 트리 옆에 나타나는 돋보기 아이콘을 클릭합니다. 사용 가능한 시각화 도우미의 목록이 나타납니다.



사용할 시각화 도우미를 클릭합니다.

## 참고 항목

- Visual Studio의 디버깅
- 사용자 지정 시각화 도우미 만들기
- DebugView 구문

GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

끌어오기 요청을 만들고 검토할  
수도 있습니다. 자세한 내용은  
[참여자 가이드](#)를 참조하세요.

☞ 설명서 문제 열기

☞ 제품 사용자 의견 제공

# DebugView 구문

아티클 • 2024. 02. 18.

DebugView 속성(디버깅할 때만 사용 가능)은 식 트리의 문자열 렌더링을 제공합니다. 대부분의 구문은 이해하기 쉽습니다. 특별한 경우는 다음 섹션에서 설명합니다.

각 예제 다음에는 DebugView를 포함한 블록 주석이 이어집니다.

## ParameterExpression

ParameterExpression 변수 이름의 시작 부분에 `$` 기호가 표시됩니다.

매개 변수에 이름이 없으면 자동으로 생성된 이름이 할당됩니다(예: `$var1` 또는 `$var2`).

```
C#  
  
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");  
/*  
 $num  
 */  
  
ParameterExpression numParam = Expression.Parameter(typeof(int));  
/*  
 $var1  
 */
```

## ConstantExpression

정수 값, 문자열 및 `null`을 나타내는 ConstantExpression 개체의 경우 상수 값이 표시됩니다.

표준 접미사인 C# 리터럴이 있는 숫자 형식의 경우 접미사가 값에 추가됩니다. 다음 표에서는 다양한 숫자 형식과 연결된 접미사를 보여 줍니다.

[+] 테이블 확장

Type	키워드	접미사
System.UInt32	uint	U
System.Int64	long	L
System.UInt64	ulong	UL

Type	키워드	접미사
System.Double	double	D
System.Single	float	F
System.Decimal	decimal	M

C#

```
int num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10
*/

double num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10D
*/
```

## BlockExpression

[BlockExpression](#) 개체의 형식이 블록에 있는 마지막 식의 형식과 다를 경우 형식은 꺼挈괄호(< 및 >) 안에 표시됩니다. 그렇지 않은 경우, [BlockExpression](#) 개체의 형식이 표시되지 않습니다.

C#

```
BlockExpression block = Expression.Block(Expression.Constant("test"));
/*
    .Block() {
        "test"
    }
*/

BlockExpression block = Expression.Block(typeof(Object),
Expression.Constant("test"));
/*
    .Block<System.Object>() {
        "test"
    }
*/
```

## LambdaExpression

[LambdaExpression](#) 개체는 대리자 형식과 함께 표시됩니다.

람다 식에 이름이 없으면 자동으로 생성된 이름이 할당됩니다(예: #Lambda1 또는 #Lambda2).

C#

```
LambdaExpression lambda = Expression.Lambda<Func<int>>()
(Expression.Constant(1));
/*
    .Lambda #Lambda1<System.Func'1[System.Int32]>() {
        1
    }
*/
LambdaExpression lambda = Expression.Lambda<Func<int>>()
(Expression.Constant(1), "SampleLambda", null);
/*
    .Lambda #SampleLambda<System.Func'1[System.Int32]>() {
        1
    }
*/
```

## LabelExpression

[LabelExpression](#) 개체의 기본값을 지정하면 이 값은 [LabelTarget](#) 개체 앞에 표시됩니다.

.Label 토큰은 레이블의 시작을 나타냅니다. .LabelTarget 토큰은 이동할 대상의 목적지를 나타냅니다.

레이블에 이름이 없으면 자동으로 생성된 이름이 할당됩니다(예: #Label1 또는 #Label2).

C#

```
LabelTarget target = Expression.Label(typeof(int), "SampleLabel");
BlockExpression block = Expression.Block(
    Expression.Goto(target, Expression.Constant(0)),
    Expression.Label(target, Expression.Constant(-1))
);
/*
    .Block() {
        .Goto SampleLabel { 0 };
        .Label
            -1
        .LabelTarget SampleLabel:
    }
*/
LabelTarget target = Expression.Label();
```

```
BlockExpression block = Expression.Block(
    Expression.Goto(target),
    Expression.Label(target)
);
/*
    .Block() {
        .Goto #Label1 { };
        .Label
        .LabelTarget #Label1:
    }
*/

```

## 확인된 연산자

확인된 연산자는 연산자 앞에 `#` 기호가 표시됩니다. 예를 들어 확인된 더하기 연산자는 `#+`로 표시됩니다.

C#

```
Expression expr = Expression.AddChecked( Expression.Constant(1),
    Expression.Constant(2));
/*
    1 #+ 2
*/

Expression expr = Expression.ConvertChecked( Expression.Constant(10.0),
    typeof(int));
/*
    #(System.Int32)10D
*/
```

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

[설명서 문제 열기](#)

[제품 사용자 의견 제공](#)

# System.Linq.Expressions.Expression.Add 메서드

아티클 • 2024. 01. 31.

이 문서에서는 이 API에 대한 참조 설명서에 대한 추가 설명서를 제공합니다.

메서드는 `Add` 구현 메서드로 `MethodInfo` 설정된 속성이 있는 값을 반환 `BinaryExpression` 합니다. 속성 `Type`은 노드의 형식으로 설정됩니다. 노드가 해제되면 `IsLifted` 속성과 `IsLiftedToNull` 속성은 둘 다 `true`입니다. 그렇지 않으면 다음과 같습니다 `false`. `Conversion` 속성은 `null`입니다.

다음 정보는 구현 방법, 노드 형식 및 노드 해제 여부를 설명합니다.

## 구현 방법

다음 규칙은 작업에 대해 선택한 구현 방법을 결정합니다.

- `Type` 더하기 연산 `MethodInfo` 자를 오버로드하는 사용자 정의 형식 중 하나 `left` 또는 `right` 해당 메서드의 속성을 나타내는 경우 해당 메서드를 나타내는 형식은 구현 메서드입니다.
- 그렇지 않으면 `.left` 형식 및 `right` 형식은 숫자 형식이고 구현 메서드는 `.입니다 null`.

## 노드 유형 및 해제된 노드 및 해제되지 않은 노드

구현 메서드가 아닌 `null` 경우:

- 이면 `left` 형식 및 `right` 형식은 구현 메서드의 해당 인수 형식에 할당할 수 있으며 노드는 해제되지 않습니다. 노드의 형식은 구현 메서드의 반환 형식입니다.
- 다음 두 조건이 충족되면 노드가 해제되고 노드 형식은 구현 메서드의 반환 형식에 해당하는 `null` 허용 형식입니다.
  - `left` 형식 및 `right` 형식은 둘 다 하나 이상의 `null`을 허용하고 해당 `nullable`이 아닌 형식 구현 메서드의 해당 인수 형식과 같은 값 형식입니다.
  - 구현 메서드의 반환 형식은 `nullable`이 아닌 값 형식입니다.

구현 메서드가 다음과 같은 경우: `null`

- 이면 `left` 형식 및 `right` 형식은 모두 `null`을 허용하지 않으며 노드가 해제되지 않습니다. 노드의 형식은 미리 정의된 더하기 연산자의 결과 형식입니다.

- 이면 `left` 형식 및 `right`. 형식은 모두 `null` 허용이며 노드가 해제됩니다. 노드의 형식은 미리 정의된 더하기 연산자의 결과 형식에 해당하는 nullable 형식입니다.

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

🌟 설명서 문제 열기

↗️ 제품 사용자 의견 제공

# System.Linq.Expressions.BinaryExpression 클래스

아티클 • 2024. 01. 10.

이 문서에서는 이 API에 대한 참조 설명서에 대한 추가 설명서를 제공합니다.

클래스는 이 [BinaryExpression](#) 진 연산자가 있는 식을 나타냅니다.

다음 표에는 속성이 나타내는 특정 노드 형식을 가진 팩터리 메서드를 [BinaryExpression](#) 만드는 데 사용할 수 있는 팩터리 메서드가 [NodeType](#) 요약되어 있습니다. 각 테이블에는 산술 또는 비트와 같은 특정 연산 클래스에 대한 정보가 포함되어 있습니다.

## 이진 산술 연산

[+] 테이블 확장

노드 형식	팩터리 메서드
Add	Add
AddChecked	AddChecked
Divide	Divide
Modulo	Modulo
Multiply	Multiply
MultiplyChecked	MultiplyChecked
Power	Power
Subtract	Subtract
SubtractChecked	SubtractChecked

## 비트 연산

[+] 테이블 확장

노드 형식	팩터리 메서드
And	And

노드 형식	팩터리 메서드
Or	Or
ExclusiveOr	ExclusiveOr

## Shift 작업

[+] 테이블 확장

노드 형식	팩터리 메서드
LeftShift	LeftShift
RightShift	RightShift

## 조건부 부울 작업

[+] 테이블 확장

노드 형식	팩터리 메서드
AndAlso	AndAlso
OrElse	OrElse

## 비교 작업

[+] 테이블 확장

노드 형식	팩터리 메서드
Equal	Equal
NotEqual	NotEqual
GreaterThanOrEqual	GreaterThanOrEqual
GreaterThan	GreaterThan
LessThan	LessThan
LessThanOrEqual	LessThanOrEqual

# 병합 작업

[+] 테이블 확장

노드 형식	팩터리 메서드
Coalesce	Coalesce

# 배열 인덱싱 작업

[+] 테이블 확장

노드 형식	팩터리 메서드
ArrayIndex	ArrayIndex

또한 합니다 [MakeBinary](#) 를 만드는 방법을 사용할 수도 있습니다는 [BinaryExpression](#)합니다. 이러한 팩터리 메서드를 사용하여 이진 작업을 나타내는 모든 노드 형식을 만들 [BinaryExpression](#) 수 있습니다. 형식의 이러한 메서드의 매개 변수 [NodeType](#) 원하는 노드 형식을 지정 합니다.

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

💬 설명서 문제 열기

💡 제품 사용자 의견 제공

# 상호 운용성 개요

아티클 • 2024. 02. 17.

상호 운용성은 비관리 코드에 대한 기존 투자를 보존하고 활용할 수 있도록 합니다. CLR(공용 언어 런타임)의 제어로 실행되는 코드는 관리 코드이고 CLR 외부에서 실행되는 코드는 관리되지 않는 코드입니다. COM, COM+, C++ 구성 요소, ActiveX 구성 요소 및 Microsoft Windows API는 비관리 코드의 예입니다.

.NET에서는 플랫폼 호출 서비스, [System.Runtime.InteropServices](#) 네임스페이스, C++ 상호 운용성 및 COM 상호 운용성(COM interop)을 통해 비관리 코드와의 상호 운용이 가능합니다.

## 플랫폼 호출

플랫폼 호출은 Windows API의 함수와 같이 DLL(동적 연결 라이브러리)에서 구현된 관리되지 않는 함수를 관리 코드가 호출할 수 있도록 하는 서비스입니다. 이 서비스는 내보낸 함수를 찾아서 호출하고 필요에 따라 상호 운용 경계를 가로질러 인수(정수, 문자열, 배열, 구조체 등)를 마샬링합니다.

자세한 내용은 [관리되지 않는 DLL 함수 사용](#) 및 [플랫폼 호출을 사용하여 WAV 파일을 재생하는 방법](#)을 참조하세요.

### ① 참고

CLR(공용 언어 런타임)은 시스템 리소스에 대한 액세스를 관리합니다. CLR 외부에 있는 비관리 코드를 호출하면 이 보안 메커니즘을 우회하므로 보안 위험이 제기됩니다. 예를 들어 비관리 코드는 CLR 보안 메커니즘을 우회하여 비관리 코드의 리소스를 직접 호출할 수 있습니다. 자세한 내용은 [.NET의 보안](#)을 참조하세요.

## C++ Interop

IJW(It Just Works)라고도 하는 C++ interop를 사용하여 네이티브 C++ 클래스를 래핑할 수 있습니다. C++ interop를 사용하면 C# 또는 다른 .NET 언어로 작성된 코드에서 액세스 할 수 있습니다. 네이티브 DLL 또는 COM 구성 요소를 래핑하는 C++ 코드를 작성합니다. 다른 .NET 언어와 달리 Visual C++에는 동일한 애플리케이션과 동일한 파일에서도 관리 및 관리되지 않는 코드를 사용할 수 있는 상호 운용성 지원이 있습니다. 그런 다음 /clr 컴파일러 스위치로 관리되는 어셈블리를 생성하여 C++ 코드를 빌드합니다. 마지막으로, C# 프로젝트의 어셈블리에 대한 참조를 추가하고 다른 관리되는 클래스를 사용하는 것처럼 래핑된 개체를 사용합니다.

# C#에 COM 구성 요소 노출

C# 프로젝트에서 COM 구성 요소를 사용할 수 있습니다. 일반적인 단계는 다음과 같습니다.

1. COM 구성 요소를 찾아서 사용하고 등록합니다. regsvr32.exe를 사용하여 COM DLL을 등록하거나 등록을 취소합니다.
2. COM 구성 요소 또는 형식 라이브러리에 대한 참조를 프로젝트에 추가합니다. 참조를 추가하면 Visual Studio에서는 형식 라이브러리를 입력으로 사용하는 [Tlbimp.exe\(형식 라이브러리 가져오기\)](#)를 통해 .NET interop 어셈블리를 출력합니다. RCW(런타임 호출 가능 래퍼)라고도 하는 어셈블리는 형식 라이브러리에 있는 인터페이스 및 COM 클래스를 래핑하는 인터페이스 및 관리되는 클래스를 포함합니다. Visual Studio에서는 생성된 어셈블리에 대한 참조를 프로젝트에 추가합니다.
3. RCW에 정의된 클래스의 인스턴스를 만듭니다. 해당 클래스의 인스턴스를 만들면 COM 개체의 인스턴스가 만들어집니다.
4. 다른 관리되는 개체와 동일한 방식으로 개체를 사용합니다. 개체가 가비지 수집에 의해 회수되면 COM 개체 인스턴스도 메모리에서 해제됩니다.

자세한 내용은 [.NET Framework에 COM 구성 요소 노출](#)을 참조하세요.

# COM에 C# 노출

COM 클라이언트는 올바르게 노출된 C# 형식을 사용할 수 있습니다. C# 형식을 노출하는 기본 단계는 다음과 같습니다.

1. C# 프로젝트에서 Interop 특성을 추가합니다. C# 프로젝트 속성을 수정하여 어셈블리 COM을 표시할 수 있습니다. 자세한 내용은 [어셈블리 정보 대화 상자](#)를 참조하세요.
2. COM 형식 라이브러리를 생성하고 COM 사용을 위해 등록합니다. C# 프로젝트 속성을 수정하여 COM interop에 대한 C# 어셈블리를 자동으로 등록할 수 있습니다. Visual Studio에서는 [Regasm.exe\(어셈블리 등록 도구\)](#)를 사용하며, 관리되는 어셈블리를 입력으로 사용하는 `/t1b` 명령줄 스위치를 통해 형식 라이브러리를 생성합니다. 이 형식 라이브러리는 어셈블리의 `public` 형식을 설명하고, COM 클라이언트가 관리되는 클래스를 만들 수 있도록 레지스트리 항목을 추가합니다.

자세한 내용은 [.NET Framework 구성 요소를 COM에 노출 및 예제 COM 클래스](#)를 참조하세요.

## 참고 항목

- [Interop 성능 향상](#)

- Introduction to Interoperability between COM and .NET(COM과 .NET 간 상호 운용 성 소개)
- Visual Basic의 COM Interop 소개
- 관리 코드와 비관리 코드 간의 마샬링
- 비관리 코드와의 상호 운용

### ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

ଓ 설명서 문제 열기

ଓ 제품 사용자 의견 제공

# COM 클래스 예제

아티클 • 2023. 04. 08.

다음 코드는 COM 개체로 노출할 클래스의 예입니다. 프로젝트에 추가된 .cs 파일에 이 코드를 배치한 후 **COM Interop에 등록** 속성을 **True**로 설정합니다. 자세한 내용은 [방법: COM Interop에 대한 구성 요소 등록](#)을 참조하세요.

C# 개체를 COM에 노출하려면 클래스 인터페이스, 필요한 경우 "이벤트 인터페이스" 및 클래스 자체를 선언해야 합니다. 클래스 멤버가 COM에 표시되려면 다음 규칙을 따라야 합니다.

- 클래스는 `public`이어야 합니다.
- 속성, 메서드 및 이벤트는 `public`이어야 합니다.
- 속성 및 메서드는 클래스 인터페이스에서 선언되어야 합니다.
- 이벤트는 이벤트 인터페이스에서 선언되어야 합니다.

이러한 인터페이스에서 선언하지 않은 클래스의 다른 공용 멤버는 COM에 표시되지 않지만 다른 .NET 개체에 표시됩니다. 속성 및 메서드를 COM에 노출하려면 클래스 인터페이스에서 선언하고 `DispId` 특성을 사용하여 표시한 후 클래스에서 구현해야 합니다. 인터페이스에서 멤버를 선언하는 순서는 COM vtable에 사용되는 순서입니다. 클래스에서 이벤트를 노출하려면 이벤트 인터페이스에서 선언하고 `DispId` 특성을 사용하여 표시해야 합니다. 클래스는 이 인터페이스를 구현하면 안 됩니다.

클래스는 클래스 인터페이스를 구현합니다. 둘 이상의 인터페이스를 구현할 수 있지만 첫 번째 구현은 기본 클래스 인터페이스입니다. 여기에서 COM에 노출된 메서드 및 속성을 구현합니다. 공용이어야 하며 클래스 인터페이스의 선언과 일치해야 합니다. 또한 여기에서 클래스에 의해 발생된 이벤트를 선언합니다. 공용이어야 하며 이벤트 인터페이스의 선언과 일치해야 합니다.

## 예제

C#

```
using System.Runtime.InteropServices;

namespace project_name
{
    [Guid("EAA4976A-45C3-4BC5-BC0B-E474F4C3C83F")]
    public interface ComClass1Interface
    {

    }

    [Guid("7BD20046-DF8C-44A6-8F6B-687FAA26FA71"),

```

```
    InterfaceType(ComInterfaceType.InterfaceIsIDispatch) ]  
    public interface ComClass1Events  
    {  
    }  
  
    [Guid("0D53A3E8-E51A-49C7-944E-E72A2064F938"),  
     ClassInterface(ClassInterfaceType.None),  
     ComSourceInterfaces(typeof(ComClass1Events))]  
    public class ComClass1 : ComClass1Interface  
    {  
    }  
}
```

## 참조

- 상호 운용성
- 프로젝트 디자이너, 빌드 페이지(C#)

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

### .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# 연습: C#의 Office 프로그래밍

아티클 • 2024. 02. 20.

C#은 Microsoft Office 프로그래밍을 개선하는 기능을 제공합니다. 유용한 C# 기능으로는 명명된 인수 및 선택적 인수, `dynamic` 형식의 반환 값 등이 있습니다. COM 프로그래밍에서 `ref` 키워드를 생략하면 인덱싱된 속성에 액세스할 수 있게 됩니다.

이 두 언어에서는 PIA(주 Interop 어셈블리)를 사용자 컴퓨터에 배포하지 않아도 COM 구성 요소와 상호 작용하는 어셈블리를 배포할 수 있도록 하는 형식 정보를 포함할 수 있습니다. 자세한 내용은 [연습: 관리되는 어셈블리의 형식 포함](#)을 참조하세요.

이 연습에서는 Office 프로그래밍과 관련해서 이러한 기능을 설명하지만 대부분 일반 프로그래밍에서도 유용합니다. 연습에서는 Excel 추가 기능 애플리케이션을 사용하여 Excel 통합 문서를 만듭니다. 그런 다음 통합 문서 링크를 포함하는 Word 문서를 만듭니다. 마지막으로 PIA 종속성을 사용 및 사용하지 않도록 설정하는 방법을 알아봅니다.

## ① 중요

VSTO(Visual Studio Tools for Office)는 .NET Framework를 사용합니다. COM 추가 기능은 .NET Framework를 사용하여 작성할 수도 있습니다. Office 추가 기능은 최신 버전인 .NET Core 및 .NET 5 이상으로 만들 수 없습니다. .NET Core 및 .NET 5 이상은 동일한 프로세스에서 .NET Framework와 함께 작동할 수 없으며 추가 기능 로드 실패로 이어질 수 있기 때문입니다. 계속해서 .NET Framework를 사용하여 Office용 VSTO 및 COM 추가 기능을 작성할 수 있습니다. Microsoft는 .NET Core 또는 .NET 5 이상을 사용하도록 VSTO 또는 COM 추가 기능 플랫폼을 업데이트하지 않습니다. ASP.NET Core를 포함한 .NET Core 및 .NET 5 이상을 활용하여 Office 웹 추가 기능의 서버 쪽을 만들 수 있습니다.

## 필수 조건

이 연습을 완료하려면 Microsoft Office Excel 및 Microsoft Office Word가 컴퓨터에 설치되어 있어야 합니다.

## ① 참고

일부 Visual Studio 사용자 인터페이스 요소의 경우 다음 지침에 설명된 것과 다른 이름 또는 위치가 시스템에 표시될 수 있습니다. 이러한 요소는 사용하는 Visual Studio 버전 및 설정에 따라 결정됩니다. 자세한 내용은 IDE 개인 설정을 참조하세요.

# Excel 추가 기능 애플리케이션 설정

1. Visual Studio를 시작합니다.
2. 파일 메뉴에서 새로 만들기를 가리킨 다음, 프로젝트를 선택합니다.
3. 설치된 템플릿 창에서 C#을 확장하고 Office를 확장한 다음 Office 제품의 버전 연도를 선택합니다.
4. 템플릿 창에서 Excel <버전> 추가 기능을 선택합니다.
5. 템플릿 창 위쪽의 대상 프레임워크 상자에 .NET Framework 4 이상 버전이 표시되어 있는지 확인합니다.
6. 원하는 경우 이름 상자에 프로젝트의 이름을 입력합니다.
7. 확인을 선택합니다.
8. 솔루션 탐색기에 새 프로젝트가 표시됩니다.

## 참조 추가

1. 솔루션 탐색기에서 프로젝트 이름을 마우스 오른쪽 단추로 클릭한 다음 참조 추가를 선택합니다. 참조 추가 대화 상자가 나타납니다.
2. 어셈블리 탭의 구성 요소 이름 목록에서 Microsoft.Office.Interop.Excel, 버전 <version>.0.0.0(Office 제품 버전 번호에 대한 자세한 내용은 [Microsoft 버전 참조](#))을 선택하고 Ctrl 키를 누른 상태로 Microsoft.Office.Interop.Word, version <version>.0.0.0을 선택합니다. 어셈블리가 표시되지 않으면 어셈블리를 설치해야 할 수 있습니다([방법: Office 기본 Interop 어셈블리 설치 참조](#)).
3. 확인을 선택합니다.

## 필요한 Import 문 또는 using 지시문 추가

솔루션 탐색기에서 ThisAddIn.cs 파일을 마우스 오른쪽 단추로 클릭한 다음 코드 보기 를 선택합니다. 코드 파일의 맨 위에 다음 using 지시문(C#)이 아직 없는 경우 추가합니다.

C#

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

## 은행 계좌 목록 만들기

솔루션 탐색기에서 프로젝트 이름을 마우스 오른쪽 단추로 클릭하고 추가를 선택한 다음 클래스를 선택합니다. 클래스 이름을 Account.cs로 지정합니다. 추가를 선택합니다.

`Account` 클래스 정의를 다음 코드로 바꿉니다. 클래스 정의는 자동으로 구현된 속성을 사용합니다.

C#

```
class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

두 개의 계정이 포함된 `bankAccounts` 목록을 만들려면 `ThisAddIn.cs`에서 `ThisAddIn_Startup` 메서드에 다음 코드를 추가합니다. 목록 선언은 컬렉션이나 셀라이저를 사용합니다.

C#

```
var bankAccounts = new List<Account>
{
    new Account
    {
        ID = 345,
        Balance = 541.27
    },
    new Account
    {
        ID = 123,
        Balance = -127.44
    }
};
```

## Excel로 데이터 내보내기

같은 파일에서 `ThisAddIn` 클래스에 다음 메서드를 추가합니다. 이 메서드는 Excel 통합 문서를 설정하고 데이터를 해당 통합 문서로 내보냅니다.

C#

```
void DisplayInExcel(IEnumerable<Account> accounts,
                     Action<Account, Excel.Range> DisplayFunc)
{
    var excelApp = this.Application;
    // Add a new Excel workbook.
    excelApp.Workbooks.Add();
    excelApp.Visible = true;
    excelApp.Range["A1"].Value = "ID";
    excelApp.Range["B1"].Value = "Balance";
```

```

excelApp.Range["A2"].Select();

foreach (var ac in accounts)
{
    DisplayFunc(ac, excelApp.ActiveCell);
    excelApp.ActiveCell.Offset[1, 0].Select();
}
// Copy the results to the Clipboard.
excelApp.Range["A1:B3"].Copy();
}

```

- `Add` 메서드에는 특정 템플릿을 지정하기 위한 선택적 매개 변수가 있습니다. 선택적 매개 변수를 사용하면 매개 변수의 기본값을 사용하려는 경우 해당 매개 변수의 인수를 생략할 수 있습니다. 이전 예제에는 인수가 없으므로 `Add(은)`는 기본 템플릿을 사용하고 새 통합 문서를 만듭니다. 이전 버전의 C#에서 이와 동일한 문을 사용하려면 자리 표시자 인수인 `excelApp.Workbooks.Add(Type.Missing)`를 사용해야 했습니다. 자세한 내용은 [명명된 인수 및 선택적 인수](#)를 참조하세요.
- 범위 개체의 `Range` 및 `Offset` 속성은 인덱싱된 속성 기능을 사용합니다. 이 기능을 사용하면 다음과 같은 일반적인 C# 구문을 통해 COM 형식에서 이러한 속성을 사용할 수 있습니다. 또한 인덱싱된 속성에서는 `Value` 개체의 `Range` 속성을 사용할 수 있으므로 `Value2` 속성을 사용할 필요가 없습니다. `Value` 속성은 인덱싱된 속성이지만 인덱스는 선택 사항입니다. 다음 예제에서는 선택적 인수와 인덱싱된 속성이 함께 사용됩니다.

C#

```

// Visual C# 2010 provides indexed properties for COM programming.
excelApp.Range["A1"].Value = "ID";
excelApp.ActiveCell.Offset[1, 0].Select();

```

인덱싱된 속성을 직접 만들 수는 없습니다. 이 기능은 기존 인덱싱된 속성의 사용만을 지원합니다.

열 너비를 콘텐츠에 맞게 조정하려면 `DisplayInExcel` 끝에 다음 코드를 추가합니다.

C#

```

excelApp.Columns[1].AutoFit();
excelApp.Columns[2].AutoFit();

```

여기서 추가하는 코드는 C#의 또 다른 기능, 즉 Office 등의 COM 호스트에서 반환되는 `Object` 값을 `dynamic` 형식인 것처럼 처리하는 기능을 보여 줍니다. COM 개체는 `Embed Interop Types`에 해당 기본값 `True`(이)가 있는 경우 혹은 동등하게 `EmbedInteropTypes` 컴파일러 옵션을 사용하여 어셈블리를 참조하는 경우 자동으로 `dynamic`(으)로 처리됩니다.

다. `interop` 형식을 포함하는 방법에 대한 자세한 내용은 이 문서의 뒷부분에 있는 "PIA 참조 찾기" 및 "PIA 종속성 복원" 절차를 참조하세요. `dynamic`에 대한 자세한 내용은 [dynamic 또는 dynamic 형식 사용](#)을 참조하세요.

## DisplayInExcel 호출

`ThisAddIn_StartUp` 메서드의 끝에 다음 코드를 추가합니다. `DisplayInExcel` 호출에는 두 개의 인수가 포함됩니다. 첫 번째 인수는 처리된 계정 목록의 이름입니다. 두 번째 인수는 데이터를 처리하는 방법을 정의하는 여러 줄 람다 식입니다. 각 계좌의 `ID` 및 `balance` 값은 인접 셀에 표시되며 잔액이 0보다 작으면 행은 빨간색으로 표시됩니다. 자세한 내용은 [람다 식](#)을 참조하세요.

C#

```
DisplayInExcel(bankAccounts, (account, cell) =>
    // This multiline lambda expression sets custom processing rules
    // for the bankAccounts.
{
    cell.Value = account.ID;
    cell.Offset[0, 1].Value = account.Balance;
    if (account.Balance < 0)
    {
        cell.Interior.Color = 255;
        cell.Offset[0, 1].Interior.Color = 255;
    }
});
```

F5 키를 눌러 프로그램을 실행합니다. 그러면 계좌의 데이터가 포함된 Excel 워크시트가 표시됩니다.

## Word 문서 추가

`ThisAddIn_StartUp` 메서드 끝에 다음 코드를 추가하여 Excel 통합 문서에 대한 링크가 포함된 Word 문서를 만듭니다.

C#

```
var wordApp = new Word.Application();
wordApp.Visible = true;
wordApp.Documents.Add();
wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
```

이 코드는 C#의 몇 가지 기능, 즉 COM 프로그래밍, 명명된 인수 및 선택적 인수에서 `ref` 키워드를 생략하는 기능을 보여 줍니다. `PasteSpecial` 메서드에는 7개의 매개 변수가 있으

며 모두 선택적 참조 매개 변수입니다. 명명된 인수와 선택적 인수를 사용하면 액세스할 매개 변수를 이름으로 지정하고 해당 매개 변수에만 인수를 보낼 수 있습니다. 이 예제에서 인수는 클립보드(매개 변수 `Link`)에 통합 문서에 대한 링크를 만들고 Word 문서에 해당 링크를 아이콘(매개 변수 `DisplayAsIcon`)으로 표시함을 나타냅니다. 또한 C#을 사용하면 이러한 인수에 대한 `ref` 키워드를 생략할 수 있습니다.

## 애플리케이션 실행

F5 키를 눌러 애플리케이션을 실행합니다. Excel이 시작되고 `bankAccounts`의 두 계좌 정보가 포함된 표가 표시됩니다. 그런 후에 Excel 표의 링크가 포함된 Word 문서가 표시됩니다.

## 완료된 프로젝트 정리

Visual Studio의 **빌드** 메뉴에서 **솔루션 정리**를 선택합니다. 그렇지 않으면 컴퓨터에서 Excel을 열 때마다 추가 기능이 실행됩니다.

## PIA 참조 찾기

1. 애플리케이션을 다시 실행하지만 **솔루션 정리**는 선택하지 마세요.
2. **시작**을 선택합니다. Microsoft Visual Studio <버전>(을)를 찾고 개발자 명령 프롬프트를 엽니다.
3. Visual Studio용 개발자 명령 프롬프트 창에 `ildasm`을 입력하고 Enter 키를 누릅니다. IL DASM 창이 나타납니다.
4. IL DASM 창의 **파일** 메뉴에서 **파일>열기**를 선택합니다. Visual Studio <버전>(을)를 두 번 클릭한 다음 **프로젝트**를 두 번 클릭합니다. 프로젝트 폴더를 열고 **프로젝트 이름.dll**에서 bin/Debug 폴더를 확인한 후 **프로젝트 이름.dll**을 두 번 클릭합니다. 새 창에 프로젝트 특성과 기타 모듈 및 어셈블리에 대한 참조가 표시됩니다. 어셈블리에는 네임스페이스 `Microsoft.Office.Interop.Excel` 및 `Microsoft.Office.Interop.Word`(이)가 포함되어 있습니다. 기본적으로 Visual Studio에서 컴파일하는 필요한 형식을 참조된 PIA에서 어셈블리로 가져옵니다. 자세한 내용은 [방법: 어셈블리 내용 보기](#)를 참조하세요.
5. **MANIFEST** 아이콘을 두 번 클릭합니다. 프로젝트가 참조하는 항목이 들어 있는 어셈블리 목록이 포함된 창이 표시됩니다. 목록에 `Microsoft.Office.Interop.Excel` 및 `Microsoft.Office.Interop.Word`(이)가 없습니다. 프로젝트에 필요한 형식을 어셈블리로 가져왔기 때문에 PIA에 대한 참조를 설치할 필요가 없습니다. 형식을 어셈블리로 가져오면 배포가 더 쉬워집니다. PIA는 사용자의 컴퓨터에 있을 필요가 없습니다. 애플리케이션은 특정 버전의 PIA를 배포할 필요가 없습니다. 필요한 API가 모든 버전에 있는 경우 애플리케이션은 여러 버전의 Office에서 작동할 수 있습니다. PIA

를 배포할 필요가 없으므로 이전 버전을 비롯한 여러 Office 버전에서 사용 가능한 애플리케이션을 고급 시나리오에서 만들 수 있습니다. 코드는 작업 중인 Office 버전에서 사용할 수 없는 API를 사용할 수 없습니다. 특정 API를 이전 버전에서 사용할 수 있는지 여부가 항상 명확하지는 않습니다. 이전 버전의 Office를 사용하는 것은 권장되지 않습니다.

6. 매니페스트 창과 어셈블리 창을 닫습니다.

## PIA 종속성 복원

1. 솔루션 탐색기에서 모든 파일 표시 단추를 선택합니다. References 폴더를 선택한 다음 Microsoft.Office.Interop.Excel을 선택합니다. F4 키를 눌러 속성 창을 표시합니다.
2. 속성 창에서 Interop 형식 포함 속성을 True에서 False로 변경합니다.
3. Microsoft.Office.Interop.Word에 대해 이 절차의 1-2단계를 반복합니다.
4. C#에서 Autofit 메서드 끝의 DisplayInExcel에 대한 두 호출을 주석 처리합니다.
5. F5 키를 눌러 프로젝트가 제대로 실행되는지 확인합니다.
6. 이전 절차의 1-3 단계를 반복하여 어셈블리 창을 엽니다.

Microsoft.Office.Interop.Word 및 Microsoft.Office.Interop.Excel이 포함된 어셈블리 목록에 더 이상 표시되지 않습니다.

7. MANIFEST 아이콘을 두 번 클릭하고 참조되는 어셈블리 목록을 스크롤합니다.  
Microsoft.Office.Interop.Word 및 Microsoft.Office.Interop.Excel이 모두 목록에 있음을 확인할 수 있습니다. 애플리케이션은 Excel 및 Word PIA를 참조하고 Embed Interop Types 속성이 False이기 때문에 두 어셈블리가 모두 최종 사용자의 컴퓨터에 있어야 합니다.
8. Visual Studio의 빌드 메뉴에서 솔루션 정리를 선택하여 완료된 프로젝트를 정리합니다.

## 참고 항목

- 자동으로 구현된 속성(C#)
- 개체 이니셜라이저 및 컬렉션 이니셜라이저
- VSTO(Visual Studio Tools for Office)
- 명명된 인수 및 선택적 인수
- dynamic
- dynamic 형식 사용
- 람다 식(C#)
- 연습: Visual Studio에서 Microsoft Office 어셈블리의 형식 정보 포함
- 연습: 관리되는 어셈블리의 형식 포함
- 연습: Excel용 첫 VSTO 추가 기능 만들기

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 플랫폼 호출을 사용하여 WAV 파일을 재생하는 방법

아티클 • 2024. 02. 18.

다음 C# 코드 예제에서는 플랫폼 호출 서비스를 사용하여 Windows 운영 체제에서 WAV 사운드 파일을 재생하는 방법을 설명합니다.

## 예시

이 예제 코드에서는 `[DllImportAttribute]`를 사용하여 `winmm.dll`의 `PlaySound` 메서드 진입점을 `Form1 PlaySound()`로 가져옵니다. 이 예제에는 단추를 포함하는 간단한 Windows Form이 있습니다. 단추를 클릭하면 재생할 파일을 열 수 있도록 표준 Windows `OpenFileDialog` 대화 상자가 열립니다. 웨이브 파일을 선택하면 `winmm.dll` 라이브러리의 `PlaySound()` 메서드를 사용하여 재생됩니다. 이 메서드에 대한 자세한 내용은 [파형 오디오 파일과 함께 PlaySound 함수 사용](#)을 참조하세요. .wav 확장명을 가진 파일을 찾아서 선택한 다음 [열기](#)를 선택하여 플랫폼 호출을 통해 웨이브 파일을 재생합니다. 텍스트 상자에 선택한 파일의 전체 경로가 표시됩니다.

C#

```
using System.Runtime.InteropServices;

namespace WinSound;

public partial class Form1 : Form
{
    private TextBox textBox1;
    private Button button1;

    public Form1() // Constructor.
    {
        InitializeComponent();
    }

    [DllImport("winmm.dll", EntryPoint = "PlaySound", SetLastError = true,
    CharSet = CharSet.Unicode, ThrowOnUnmappableChar = true)]
    private static extern bool PlaySound(string szSound, System.IntPtr hMod,
    PlaySoundFlags flags);

    [System.Flags]
    public enum PlaySoundFlags : int
    {
        SND_SYNC = 0x0000,
        SND_ASYNC = 0x0001,
        SND_NODEFAULT = 0x0002,
```

```

        SND_LOOP = 0x0008,
        SND_NOSTOP = 0x0010,
        SND_NOWAIT = 0x00002000,
        SND_FILENAME = 0x00020000,
        SND_RESOURCE = 0x00040004
    }

    private void button1_Click(object sender, System.EventArgs e)
    {
        var dialog1 = new OpenFileDialog();

        dialog1.Title = "Browse to find sound file to play";
        dialog1.InitialDirectory = @"c:\";
        //<Snippet5>
        dialog1.Filter = "Wav Files (*.wav)|*.wav";
        //</Snippet5>
        dialog1.FilterIndex = 2;
        dialog1.RestoreDirectory = true;

        if (dialog1.ShowDialog() == DialogResult.OK)
        {
            textBox1.Text = dialog1.FileName;
            PlaySound(dialog1.FileName, new System.IntPtr(),
PlaySoundFlags.SND_SYNC);
        }
    }

    private void Form1_Load(object sender, EventArgs e)
    {
        // Including this empty method in the sample because in the IDE,
        // when users click on the form, generates code that looks for a
default method
        // with this name. We add it here to prevent confusion for those
using the samples.
    }
}

```

파일 열기 대화 상자가 필터 설정을 통해 .wav 확장명을 가진 파일만 표시하도록 필터링 됩니다.

## 코드 컴파일

Visual Studio에서 새 C# Windows Forms 애플리케이션 프로젝트를 만들고 이름을 WinSound로 지정합니다. 위의 코드를 복사하여 *Form1.cs* 파일 내용에 붙여넣습니다. 다음 코드를 복사하여 *Form1.Designer.cs* 파일의 `InitializeComponent()` 메서드에서 기존 코드 뒤에 붙여넣습니다.

```
this.button1 = new System.Windows.Forms.Button();
this.textBox1 = new System.Windows.Forms.TextBox();
this.SuspendLayout();
//
// button1
//
this.button1.Location = new System.Drawing.Point(192, 40);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(88, 24);
this.button1.TabIndex = 0;
this.button1.Text = "Browse";
this.button1.Click += new System.EventHandler(this.button1_Click);
//
// textBox1
//
this.textBox1.Location = new System.Drawing.Point(8, 40);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(168, 20);
this.textBox1.TabIndex = 1;
this.textBox1.Text = "File path";
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(5, 13);
this.ClientSize = new System.Drawing.Size(292, 266);
this.Controls.Add(this.textBox1);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "Platform Invoke WinSound C#";
this.ResumeLayout(false);
this.PerformLayout();
```

코드를 컴파일하고 실행합니다.

## 참고 항목

- [플랫폼 호출 자세히 보기](#)
- [플랫폼 호출을 사용하여 데이터 마샬링](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.



### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

### 설명서 문제 열기

☞ 제품 사용자 의견 제공

# COM interop 프로그래밍에서 인덱싱된 속성을 사용하는 방법

아티클 • 2024. 02. 17.

인덱싱된 속성은 C#의 다른 기능(예: [명명된 인수 및 선택적 인수](#)), 새 형식([dynamic](#)), [포함된 형식 정보](#)와 함께 작동하여 Microsoft Office 프로그래밍을 개선합니다.

## ① 중요

[VSTO\(Visual Studio Tools for Office\)](#)는 [.NET Framework](#)를 사용합니다. COM 추가 기능은 .NET Framework를 사용하여 작성할 수도 있습니다. Office 추가 기능은 최신 버전인 [.NET Core 및 .NET 5 이상](#)으로 만들 수 없습니다. .NET Core 및 .NET 5 이상은 동일한 프로세스에서 .NET Framework와 함께 작동할 수 없으며 추가 기능 로드 실패로 이어질 수 있기 때문입니다. 계속해서 .NET Framework를 사용하여 Office용 VSTO 및 COM 추가 기능을 작성할 수 있습니다. Microsoft는 .NET Core 또는 .NET 5 이상을 사용하도록 VSTO 또는 COM 추가 기능 플랫폼을 업데이트하지 않습니다. ASP.NET Core를 포함한 .NET Core 및 .NET 5 이상을 활용하여 [Office 웹 추가 기능](#)의 서버 쪽을 만들 수 있습니다.

이전 버전의 C#에서는 `get` 메서드에 매개 변수가 없고 `set` 메서드에 하나의 값 매개 변수가 있는 경우에만 메서드를 속성으로 액세스할 수 있습니다. 그러나 모든 COM 속성이 이러한 제한을 충족하는 것은 아닙니다. 예를 들어 Excel [Range\[\]](#) 속성에는 범위 이름에 대한 매개 변수가 필요한 `get` 접근자가 있습니다. 이전에는 `Range` 속성에 직접 액세스할 수 없었기 때문에 다음 예제와 같이 `get_Range` 메서드를 대신 사용해야 했습니다.

```
{language}
```

```
// Visual C# 2008 and earlier.  
var excelApp = new Excel.Application();  
// ...  
Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
```

인덱싱된 속성을 사용하면 대신 다음과 같이 작성할 수 있습니다.

```
{language}
```

```
// Visual C# 2010.  
var excelApp = new Excel.Application();  
// ...  
Excel.Range targetRange = excelApp.Range["A1"];
```

또한 앞의 예제에서는 선택적 인수 기능을 사용하므로 `Type.Missing`을 생략할 수 있습니다.

인덱싱된 속성을 사용하면 다음 코드를 작성할 수 있습니다.

```
{language}

// Visual C# 2010.
targetRange.Value = "Name";
```

인덱스가 만들어진 속성을 직접 만들 수는 없습니다. 이 기능은 기존 인덱싱된 속성의 사용만을 지원합니다.

## 예제

다음 코드에서는 전체 예제를 보여 줍니다. Office API에 액세스하는 프로젝트를 설정하는 방법에 대한 자세한 내용은 [Visual C# 기능을 사용하여 Office interop 개체에 액세스하는 방법](#)을 참조하세요.

```
{language}

// You must add a reference to Microsoft.Office.Interop.Excel to run
// this example.
using System;
using Excel = Microsoft.Office.Interop.Excel;

namespace IndexedProperties
{
    class Program
    {
        static void Main(string[] args)
        {
            CSharp2010();
        }

        static void CSharp2010()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add();
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.Range["A1"];
            targetRange.Value = "Name";
        }

        static void CSharp2008()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add(Type.Missing);
```

```
        excelApp.Visible = true;

        Excel.Range targetRange = excelApp.get_Range("A1",
Type.Missing);
        targetRange.set_Value(Type.Missing, "Name");
        // Or
        //targetRange.Value2 = "Name";
    }
}
```

## 참조

- 명명된 인수 및 선택적 인수
- dynamic
- dynamic 형식 사용

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

[설명서 문제 열기](#)

[제품 사용자 의견 제공](#)

# Office 상호 운용성 개체에 액세스하는 방법

아티클 • 2024. 02. 17.

C#에는 Office API 개체에 간편하게 액세스할 수 있는 기능이 있습니다. 새로운 기능에는 명명된 인수와 선택적 인수, `dynamic`이라는 새 형식 그리고 인수를 값 매개 변수처럼 COM 메서드의 참조 매개 변수로 전달하는 기능이 포함됩니다.

이 문서에서는 새로운 기능을 사용하여 Microsoft Office Excel 워크시트를 만들고 표시하는 코드를 작성합니다. Excel 워크시트에 연결된 아이콘이 포함된 Office Word 문서를 추가하는 코드를 작성합니다.

이 연습을 완료하려면 Microsoft Office Excel 2007 및 Microsoft Office Word 2007 이상 버전이 컴퓨터에 설치되어 있어야 합니다.

## ① 참고

일부 Visual Studio 사용자 인터페이스 요소의 경우 다음 지침에 설명된 것과 다른 이름 또는 위치가 시스템에 표시될 수 있습니다. 이러한 요소는 사용하는 Visual Studio 버전 및 설정에 따라 결정됩니다. 자세한 내용은 [IDE 개인 설정](#)을 참조하세요.

## ① 중요

[VSTO\(Visual Studio Tools for Office\)](#)는 [.NET Framework](#)를 사용합니다. COM 추가 기능은 .NET Framework를 사용하여 작성할 수도 있습니다. Office 추가 기능은 최신 버전인 [.NET Core 및 .NET 5 이상](#)으로 만들 수 없습니다. .NET Core 및 .NET 5 이상은 동일한 프로세스에서 .NET Framework와 함께 작동할 수 없으며 추가 기능 로드 실패로 이어질 수 있기 때문입니다. 계속해서 .NET Framework를 사용하여 Office용 VSTO 및 COM 추가 기능을 작성할 수 있습니다. Microsoft는 .NET Core 또는 .NET 5 이상을 사용하도록 VSTO 또는 COM 추가 기능 플랫폼을 업데이트하지 않습니다. ASP.NET Core를 포함한 .NET Core 및 .NET 5 이상을 활용하여 [Office 웹 추가 기능](#)의 서버 쪽을 만들 수 있습니다.

## 새 콘솔 애플리케이션을 만들려면

1. Visual Studio를 시작합니다.
2. 파일 메뉴에서 **새로 만들기**를 가리킨 다음, **프로젝트**를 선택합니다. 새 프로젝트 대화 상자가 나타납니다.

- 설치된 템플릿 창에서 C#을 확장한 다음 Windows를 선택합니다.
- 새 프로젝트 대화 상자 상단을 확인하여 .NET Framework 4(이상 버전)를 대상 프레임워크로 선택했는지 확인합니다.
- 템플릿 창에서 콘솔 애플리케이션을 선택합니다.
- 이름 필드에 프로젝트의 이름을 입력합니다.
- 확인을 선택합니다.

솔루션 탐색기에 새 프로젝트가 표시됩니다.

## 참조를 추가하려면

- 솔루션 탐색기에서 프로젝트 이름을 마우스 오른쪽 단추로 클릭한 다음 참조 추가를 선택합니다. 참조 추가 대화 상자가 나타납니다.
- 어셈블리 페이지의 구성 요소 이름 목록에서 Microsoft.Office.Interop.Word를 선택하고 Ctrl 키를 누른 상태로 Microsoft.Office.Interop.Excel을 선택합니다. 어셈블리가 표시되지 않으면 설치해야 할 수도 있습니다. 방법: Office 주 Interop 어셈블리 설치를 참조하세요.
- 확인을 선택합니다.

## 필요한 using 지시문을 추가하려면

솔루션 탐색기에서 Program.cs 파일을 마우스 오른쪽 단추로 클릭한 다음 코드 보기 를 선택합니다. 다음 using 지시문을 코드 파일의 맨 위에 추가합니다.

C#

```
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

## 은행 계좌 목록을 만들려면

다음 클래스 정의를 Program.cs의 Program 클래스 아래에 붙여 넣습니다.

C#

```
public class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

다음 코드를 `Main` 메서드에 추가하여 계좌 두 개가 포함된 `bankAccounts` 목록을 만듭니다.

C#

```
// Create a list of accounts.  
var bankAccounts = new List<Account> {  
    new Account {  
        ID = 345678,  
        Balance = 541.27  
    },  
    new Account {  
        ID = 1230221,  
        Balance = -127.44  
    }  
};
```

## 계좌 정보를 Excel로 내보내는 메서드를 선언하려면

1. 다음 메서드를 `Program` 클래스에 추가하여 Excel 워크시트를 설정합니다. `Add` 메서드에는 특정 템플릿을 지정하기 위한 선택적 매개 변수가 있습니다. 선택적 매개 변수를 사용하면 매개 변수의 기본값을 사용하려는 경우 해당 매개 변수의 인수를 생략할 수 있습니다. 인수를 제공하지 않았기 때문에 `Add`는 기본 템플릿을 사용하고 새 통합 문서를 만듭니다. 이전 버전의 C#에서 이와 동일한 문을 사용하려면 자리 표시자 인수인 `ExcelApp.Workbooks.Add(Type.Missing)`를 사용해야 했습니다.

C#

```
static void DisplayInExcel(IEnumerable<Account> accounts)  
{  
    var excelApp = new Excel.Application();  
    // Make the object visible.  
    excelApp.Visible = true;  
  
    // Create a new, empty workbook and add it to the collection returned  
    // by property Workbooks. The new workbook becomes the active workbook.  
    // Add has an optional parameter for specifying a particular template.  
    // Because no argument is sent in this example, Add creates a new  
    // workbook.  
    excelApp.Workbooks.Add();  
  
    // This example uses a single workSheet. The explicit type casting is  
    // removed in a later procedure.  
    Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;  
}
```

`DisplayInExcel` 끝에 다음 코드를 추가합니다. 이 코드는 워크시트 첫 번째 행의 처음 두 열에 값을 삽입합니다.

C#

```
// Establish column headings in cells A1 and B1.  
workSheet.Cells[1, "A"] = "ID Number";  
workSheet.Cells[1, "B"] = "Current Balance";
```

`DisplayInExcel` 끝에 다음 코드를 추가합니다. `foreach` 루프는 계좌 목록의 정보를 워크 시트에서 연속 행의 처음 두 열에 삽입합니다.

C#

```
var row = 1;  
foreach (var acct in accounts)  
{  
    row++;  
    workSheet.Cells[row, "A"] = acct.ID;  
    workSheet.Cells[row, "B"] = acct.Balance;  
}
```

열 너비를 콘텐츠에 맞게 조정하려면 `DisplayInExcel` 끝에 다음 코드를 추가합니다.

C#

```
workSheet.Columns[1].AutoFit();  
workSheet.Columns[2].AutoFit();
```

이전 버전의 C#에서는 이러한 작업을 명시적으로 캐스트해야 했습니다.

`ExcelApp.Columns[1]`은 `Object`를 반환하며 `AutoFit`은 Excel `Range` 메서드이기 때문입니다. 다음 줄에는 캐스팅이 나와 있습니다.

C#

```
((Excel.Range)workSheet.Columns[1]).AutoFit();  
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

C#에서는 `EmbedInteropTypes` 컴파일러 옵션이 어셈블리를 참조한 경우 또는 이와 동등하게 Excel의 **Interop 형식 포함** 속성이 `true`로 설정되어 있는 경우, 반환된 `Object`를 `dynamic`으로 자동 변환합니다. 이 속성의 기본값은 `true`입니다.

## 프로젝트를 실행하려면

Main 끝에 다음 줄을 추가합니다.

C#

```
// Display the list in an Excel spreadsheet.  
DisplayInExcel(bankAccounts);
```

Ctrl+F5를 누릅니다. 두 계좌의 데이터가 포함된 Excel 워크시트가 표시됩니다.

## Word 문서를 추가하려면

다음 코드는 Word 애플리케이션을 열고 Excel 워크시트에 연결되는 아이콘을 만듭니다. 이 단계의 뒷부분에서 제공되는 `CreateIconInWordDoc` 메서드를 `Program` 클래스에 붙여 넣습니다. `CreateIconInWordDoc`는 명명된 인수와 선택적 인수를 사용하여 `Add` 및 `PasteSpecial`에 대한 메서드 호출의 복잡성을 줄입니다. 이러한 호출에는 참조 매개 변수가 있는 COM 메서드에 대한 호출을 간소화하는 두 가지 다른 기능이 통합되어 있습니다. 그 중 첫 번째 기능은 인수를 값 매개 변수처럼 참조 매개 변수로 전달하는 것입니다. 즉, 각 참조 매개 변수에 대한 변수를 만들지 않고 직접 값을 보낼 수 있습니다. 컴파일러는 인수 값을 저장하기 위한 임시 변수를 생성하고 호출에서 값이 반환되면 해당 변수를 삭제합니다. 두 번째 기능은 인수 목록의 `ref` 키워드를 생략하는 것입니다.

`Add` 메서드에는 참조 매개 변수 4개가 있습니다(모두 선택적 매개 변수임). 기본값을 사용하려는 경우 매개 변수 전체 또는 일부에 대한 인수를 생략할 수 있습니다.

`PasteSpecial` 메서드는 클립보드의 내용을 삽입합니다. 이 메서드에는 참조 매개 변수 7개가 있습니다(모두 선택적 매개 변수임). 다음 코드는 이러한 매개 변수 중 두 개에 대해 인수를 지정합니다. 그 중 하나는 클립보드 내용의 소스에 대한 링크를 만드는 `Link`이고 다른 하나는 링크를 아이콘으로 표시하는 `DisplayAsIcon`입니다. 두 인수에 대해 명명된 인수를 사용하고 다른 인수는 생략할 수 있습니다. 이러한 인수는 참조 매개 변수이지만 `ref` 키워드를 사용하거나 인수로 보낼 변수를 만들 필요는 없습니다. 값을 직접 보낼 수 있습니다.

C#

```
static void CreateIconInWordDoc()  
{  
    var wordApp = new Word.Application();  
    wordApp.Visible = true;  
  
    // The Add method has four reference parameters, all of which are  
    // optional. Visual C# allows you to omit arguments for them if  
    // the default values are what you want.  
    wordApp.Documents.Add();
```

```
// PasteSpecial has seven reference parameters, all of which are  
// optional. This example uses named arguments to specify values  
// for two of the parameters. Although these are reference  
// parameters, you do not need to use the ref keyword, or to create  
// variables to send in as arguments. You can send the values directly.  
wordApp.Selection.PasteSpecial( Link: true, DisplayAsIcon: true);  
}
```

Main 끝에 다음 문을 추가합니다.

C#

```
// Create a Word document that contains an icon that links to  
// the spreadsheet.  
CreateIconInWordDoc();
```

DisplayInExcel 끝에 다음 문을 추가합니다. **Copy** 메서드는 클립보드에 워크시트를 추가합니다.

C#

```
// Put the spreadsheet contents on the clipboard. The Copy method has one  
// optional parameter for specifying a destination. Because no argument  
// is sent, the destination is the Clipboard.  
workSheet.Range["A1:B3"].Copy();
```

Ctrl+F5를 누릅니다. 아이콘이 포함된 Word 문서가 나타납니다. 아이콘을 두 번 클릭하여 워크시트를 포그라운드로 가져옵니다.

## Interop 형식 포함 속성을 설정하려면

런타임에 PIA(주 interop 어셈블리)가 필요하지 않은 COM 형식을 호출하면 더 많은 기능이 향상될 수 있습니다. PIA에 대한 종속성을 제거하면 버전을 독립적으로 실행할 수 있으며 보다 쉽게 배포할 수 있습니다. PIA를 사용하지 않는 프로그래밍의 장점에 대한 자세한 내용은 [Walkthrough: Embedding Types from Managed Assemblies](#)(연습: 관리되는 어셈블리의 형식 포함)를 참조하세요.

또한 **dynamic** 형식은 COM 메서드에 선언된 필수 형식과 반환된 형식을 나타내기 때문에 프로그래밍이 더 쉽습니다. **dynamic** 형식이 포함된 변수는 런타임까지 평가되지 않으므로 명시적 캐스팅을 수행할 필요가 없습니다. 자세한 내용은 [dynamic 형식 사용](#)을 참조하세요.

PIA를 사용하는 대신 형식 정보를 포함하는 것이 기본 동작입니다. 해당 기본값으로 인해 이전 예 중 일부가 간소화되었습니다. 명시적인 캐스팅이 필요하지 않습니다. 예를 들어

`worksheet`의 `DisplayInExcel` 선언은 `Excel._Worksheet workSheet = excelApp.ActiveSheet` 가 아닌 `Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet`로 작성됩니다. 마찬가지로 기본 동작이 없으면 같은 메서드의 `AutoFit`에 대한 호출에서도 명시적 캐스팅을 수행해야 합니다. `ExcelApp.Columns[1]`는 `Object`를 반환하며 `AutoFit`은 Excel 메서드이기 때문입니다. 다음 코드에서는 캐스팅을 보여 줍니다.

C#

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

형식 정보를 포함하는 대신 기본값을 변경하여 PIA를 사용하려면 솔루션 탐색기에서 참조 노드를 확장하고 `Microsoft.Office.Interop.Excel` 또는 `Microsoft.Office.Interop.Word`를 선택합니다. 속성 창이 표시되지 않으면 F4를 누릅니다. 속성 목록에서 `Interop 형식 포함`을 찾은 다음 해당 값을 `False`로 변경합니다. 마찬가지로 명령 프롬프트에서 `EmbedInteropTypes` 대신 `References` 컴파일러 옵션을 사용하여 컴파일할 수 있습니다.

## 표에 서식을 더 추가하려면

`AutoFit`에서 `DisplayInExcel`에 대한 두 호출을 다음 문으로 바꿉니다.

C#

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

`AutoFormat` 메서드에는 모두 선택적 매개 변수인 값 매개 변수 7개가 있습니다. 명명된 인수와 선택적 인수를 사용하여 이러한 매개 변수 중 일부 또는 모두에 대해 인수를 제공 할 수도 있고 모든 매개 변수에 인수를 제공하지 않을 수도 있습니다. 이전 문에서는 매개 변수 중 하나인 `Format`에 대해서만 인수를 제공합니다. `Format`은 매개 변수 목록의 첫 번째 매개 변수이므로 매개 변수 이름은 지정하지 않아도 됩니다. 그러나 다음 코드에 표시 된대로 매개 변수 이름을 포함하면 문을 더 쉽게 이해할 수 있습니다.

C#

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(Format:
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

결과를 보려면 CTRL+F5를 누릅니다. XIRangeAutoFormat 열거형에 나열된 다른 형식을 찾을 수 있습니다.

## 예시

다음 코드에서는 전체 예제를 보여 줍니다.

C#

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeProgrammingWalkthruComplete
{
    class Walkthrough
    {
        static void Main(string[] args)
        {
            // Create a list of accounts.
            var bankAccounts = new List<Account>
            {
                new Account {
                    ID = 345678,
                    Balance = 541.27
                },
                new Account {
                    ID = 1230221,
                    Balance = -127.44
                }
            };

            // Display the list in an Excel spreadsheet.
            DisplayInExcel(bankAccounts);

            // Create a Word document that contains an icon that links to
            // the spreadsheet.
            CreateIconInWordDoc();
        }

        static void DisplayInExcel(IEnumerable<Account> accounts)
        {
            var excelApp = new Excel.Application();
            // Make the object visible.
            excelApp.Visible = true;

            // Create a new, empty workbook and add it to the collection
            returned
                // by property Workbooks. The new workbook becomes the active
            workbook.
                // Add has an optional parameter for specifying a particular
            template.
        }
    }
}
```

```

        // Because no argument is sent in this example, Add creates a
new workbook.
        excelApp.Workbooks.Add();

        // This example uses a single workSheet.
Excel._Worksheet workSheet = excelApp.ActiveSheet;

        // Earlier versions of C# require explicit casting.
//Excel._Worksheet workSheet =
(Excel.Worksheet)excelApp.ActiveSheet;

        // Establish column headings in cells A1 and B1.
workSheet.Cells[1, "A"] = "ID Number";
workSheet.Cells[1, "B"] = "Current Balance";

var row = 1;
foreach (var acct in accounts)
{
    row++;
    workSheet.Cells[row, "A"] = acct.ID;
    workSheet.Cells[row, "B"] = acct.Balance;
}

workSheet.Columns[1].AutoFit();
workSheet.Columns[2].AutoFit();

        // Call to AutoFormat in Visual C#. This statement replaces the
// two calls to AutoFit.
workSheet.Range["A1", "B3"].AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);

        // Put the spreadsheet contents on the clipboard. The Copy
method has one
        // optional parameter for specifying a destination. Because no
argument
        // is sent, the destination is the Clipboard.
workSheet.Range["A1:B3"].Copy();
}

static void CreateIconInWordDoc()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

        // The Add method has four reference parameters, all of which
are
        // optional. Visual C# allows you to omit arguments for them if
        // the default values are what you want.
    wordApp.Documents.Add();

        // PasteSpecial has seven reference parameters, all of which are
        // optional. This example uses named arguments to specify values
        // for two of the parameters. Although these are reference
        // parameters, you do not need to use the ref keyword, or to
create

```

```
// variables to send in as arguments. You can send the values  
directly.  
        wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);  
    }  
}  
  
public class Account  
{  
    public int ID { get; set; }  
    public double Balance { get; set; }  
}  
}
```

## 참고 항목

- [Type.Missing](#)
- [dynamic](#)
- [명명된 인수 및 선택적 인수](#)
- [Office 프로그래밍에 명명된 인수와 선택적 인수 사용 방법](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# Office 프로그래밍에 명명된 인수와 선택적 인수 사용 방법

아티클 • 2024. 02. 17.

명명된 인수 및 선택적 인수는 C# 프로그래밍의 편의성, 유연성 및 가독성을 향상시킵니다. 또한 이러한 기능은 Microsoft Office 자동화 API와 같은 COM 인터페이스에 대한 액세스에 큰 도움이 됩니다.

## ① 중요

VSTO(Visual Studio Tools for Office)는 .NET Framework를 사용합니다. COM 추가 기능은 .NET Framework를 사용하여 작성할 수도 있습니다. Office 추가 기능은 최신 버전인 .NET Core 및 .NET 5 이상으로 만들 수 없습니다. .NET Core 및 .NET 5 이상은 동일한 프로세스에서 .NET Framework와 함께 작동할 수 없으며 추가 기능 로드 실패로 이어질 수 있기 때문입니다. 계속해서 .NET Framework를 사용하여 Office용 VSTO 및 COM 추가 기능을 작성할 수 있습니다. Microsoft는 .NET Core 또는 .NET 5 이상을 사용하도록 VSTO 또는 COM 추가 기능 플랫폼을 업데이트하지 않습니다. ASP.NET Core를 포함한 .NET Core 및 .NET 5 이상을 활용하여 Office 웹 추가 기능의 서버 쪽을 만들 수 있습니다.

다음 예제의 `ConvertToTable` 메서드에는 열과 행 수, 서식, 테두리, 글꼴, 색 등의 테이블 특성을 나타내는 매개 변수 16개가 있습니다. 16개 매개 변수는 모두 선택 사항입니다. 대부분의 경우 모든 매개 변수에 대해 특정 값을 지정하고 싶지 않기 때문입니다. 그러나 명명된 인수와 선택적 인수가 없으면 값이나 자리 표시자 값을 제공해야 합니다. 명명된 인수와 선택적 인수를 사용하여 프로젝트에 필요한 매개 변수에 대해서만 값을 지정합니다.

이 절차를 완료하려면 컴퓨터에 Microsoft Office Word가 설치되어 있어야 합니다.

## ① 참고

일부 Visual Studio 사용자 인터페이스 요소의 경우 다음 지침에 설명된 것과 다른 이름 또는 위치가 시스템에 표시될 수 있습니다. 이러한 요소는 사용하는 Visual Studio 버전 및 설정에 따라 결정됩니다. 자세한 내용은 IDE 개인 설정을 참조하세요.

## 새 콘솔 애플리케이션 만들기

Visual Studio를 시작합니다. 파일 메뉴에서 새로 만들기를 가리킨 다음, 프로젝트를 선택합니다. 템플릿 범주 창에서 C#을 확장한 다음 Windows를 선택합니다. 템플릿 창의 맨

위에서 .NET Framework 4가 대상 프레임워크 상자에 표시되는지 확인합니다. 템플릿 창에서 콘솔 애플리케이션을 선택합니다. 이름 필드에 프로젝트의 이름을 입력합니다. 확인을 선택합니다. 솔루션 탐색기에 새 프로젝트가 표시됩니다.

## 참조 추가

솔루션 탐색기에서 프로젝트 이름을 마우스 오른쪽 단추로 클릭한 다음 참조 추가를 선택합니다. 참조 추가 대화 상자가 나타납니다. .NET 페이지의 구성 요소 목록에서 Microsoft.Office.Interop.Word를 선택합니다. 확인을 선택합니다.

## 필요한 using 지시문 추가

솔루션 탐색기에서 Program.cs 파일을 마우스 오른쪽 단추로 클릭한 다음 코드 보기 를 선택합니다. 다음 `using` 지시문을 코드 파일의 맨 위에 추가합니다.

C#

```
using Word = Microsoft.Office.Interop.Word;
```

## Word 문서에 텍스트 표시

Program.cs의 `Program` 클래스에서 다음 메서드를 추가하여 Word 애플리케이션과 Word 문서를 만듭니다. `Add` 메서드에는 선택적 매개 변수 4개가 있습니다. 이 예제에서는 해당 기본값을 사용합니다. 따라서 호출하는 문에 인수가 필요하지 않습니다.

C#

```
static void DisplayInWord()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;
    // docs is a collection of all the Document objects currently
    // open in Word.
    Word.Documents docs = wordApp.Documents;

    // Add a document to the collection and name it doc.
    Word.Document doc = docs.Add();
}
```

메서드의 끝에 다음 코드를 추가하여 문서에서 텍스트를 표시할 위치 및 표시할 텍스트를 정의합니다.

C#

```
// Define a range, a contiguous area in the document, by specifying
// a starting and ending character position. Currently, the document
// is empty.
Word.Range range = doc.Range(0, 0);

// Use the InsertAfter method to insert a string at the end of the
// current range.
range.InsertAfter("Testing, testing, testing. . .");
```

## 애플리케이션 실행

다음 문을 Main에 추가합니다.

C#

```
DisplayInWord();
```

**Ctrl**+**F5**를 눌러 프로젝트를 실행합니다. 지정된 텍스트를 포함하는 Word 문서가 나타납니다.

## 텍스트를 표로 변경

`ConvertToTable` 메서드를 사용하여 텍스트를 표로 묶습니다. 이 메서드에는 16개의 선택적 매개 변수가 있습니다. IntelliSense는 다음 그림과 같이 선택적 매개 변수를 중괄호로 묶습니다.

```
range.ConvertToTable()
    Word.Table Range.ConvertToTable([ref object Separator = Type.Missing], [ref object NumRows =
Type.Missing], [ref object NumColumns = Type.Missing], [ref object InitialColumnWidth =
Type.Missing], [ref object Format = Type.Missing], [ref object ApplyBorders = Type.Missing],
[ref object ApplyShading = Type.Missing], [ref object ApplyFont = Type.Missing], [ref object
ApplyColor = Type.Missing], [ref object ApplyHeadingsRows = Type.Missing], [ref object
ApplyLastRow = Type.Missing], [ref object ApplyFirstColumn = Type.Missing], [ref object
ApplyLastColumn = Type.Missing], [ref object AutoFit = Type.Missing], [ref object
AutoFitBehavior = Type.Missing], [ref object DefaultTableBehavior = Type.Missing])
```

명명된 인수 및 선택적 인수를 사용하면 변경하려는 매개 변수의 값만 지정할 수 있습니다. 테이블을 만들려면 `DisplayInWord` 메서드 끝에 다음 코드를 추가합니다. 인수는 `range`의 텍스트 문자열에 있는 쉼표가 표의 셀을 구분하도록 지정합니다.

C#

```
// Convert to a simple table. The table will have a single row with
// three columns.
range.ConvertToTable(Separator: ",");
```

**Ctrl**+**F5**를 눌러 프로젝트를 실행합니다.

## 다른 매개 변수로 실험

열 1개와 행 3개가 포함되도록 표를 변경하고 `DisplayInWord`의 마지막 줄을 다음 문으로 바꾼 다음 **Ctrl**+**F5**를 입력합니다.

C#

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);
```

미리 정의된 표 형식을 지정하고 `DisplayInWord`의 마지막 줄을 다음 문으로 바꾼 다음 **Ctrl**+**F5**를 입력합니다. 형식은 `WdTableFormat` 상수 중 하나일 수 있습니다.

C#

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,  
Format: Word.WdTableFormat.wdTableFormatElegant);
```

## 예제

다음 코드에는 전체 예제가 포함되어 있습니다.

C#

```
using System;  
using Word = Microsoft.Office.Interop.Word;  
  
namespace OfficeHowTo  
{  
    class WordProgram  
    {  
        static void Main(string[] args)  
        {  
            DisplayInWord();  
        }  
  
        static void DisplayInWord()  
        {  
            var wordApp = new Word.Application();  
            wordApp.Visible = true;  
            // docs is a collection of all the Document objects currently  
            // open in Word.  
            Word.Documents docs = wordApp.Documents;  
  
            // Add a document to the collection and name it doc.  
        }  
    }  
}
```

```

Word.Document doc = docs.Add();

        // Define a range, a contiguous area in the document, by
specifying
        // a starting and ending character position. Currently, the
document
        // is empty.
Word.Range range = doc.Range(0, 0);

        // Use the InsertAfter method to insert a string at the end of
the
        // current range.
range.InsertAfter("Testing, testing, testing. . .");

        // You can comment out any or all of the following statements to
        // see the effect of each one in the Word document.

        // Next, use the ConvertToTable method to put the text into a
table.
        // The method has 16 optional parameters. You only have to
specify
        // values for those you want to change.

        // Convert to a simple table. The table will have a single row
with
        // three columns.
range.ConvertToTable(Separator: ",");

        // Change to a single column with three rows..
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns:
1);

        // Format the table.
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns:
1,
                    Format: Word.WdTableFormat.wdTableFormatElegant);
    }
}
}

```

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.



## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 유형 동적 사용

아티클 • 2024. 02. 17.

`dynamic` 형식은 정적 형식이지만 `dynamic` 형식의 개체가 정적 형식 검사를 건너뜁니다. 대부분의 경우 이 형식은 `object` 형식을 가지고 있는 것처럼 작동합니다. 컴파일러는 `dynamic` 요소가 모든 작업을 지원한다고 가정합니다. 따라서 개체가 COM API, IronPython과 같은 동적 언어, HTML DOM(문서 개체 모델), 리플렉션 또는 프로그램의 다른 위치에서 값을 가져오는지 여부를 확인할 필요가 없습니다. 그러나 코드가 유효하지 않으면 런타임 시 오류가 나타납니다.

예를 들어 다음 코드의 인스턴스 메서드 `exampleMethod1`에 매개 변수가 하나뿐인 경우, 컴파일러는 메서드 `ec.exampleMethod1(10, 4)`에 대한 첫 번째 호출이 유효하지 않음을 인식합니다. 여기에 인수가 두 개 포함되었기 때문입니다. 호출 시 컴파일러 오류가 발생합니다. `dynamic_ec`의 형식이 `dynamic`이므로 컴파일러는 `dynamic_ec.exampleMethod1(10, 4)` 메서드에 대한 두 번째 호출을 확인하지 않습니다. 따라서 컴파일러 오류가 보고되지 않습니다. 그러나 오류가 무기한 통지를 벗어나는 것은 아닙니다. 런타임에 나타나며 런타임 예외가 발생합니다.

C#

```
static void Main(string[] args)
{
    ExampleClass ec = new ExampleClass();
    // The following call to exampleMethod1 causes a compiler error
    // if exampleMethod1 has only one parameter. Uncomment the line
    // to see the error.
    //ec.exampleMethod1(10, 4);

    dynamic dynamic_ec = new ExampleClass();
    // The following line is not identified as an error by the
    // compiler, but it causes a run-time exception.
    dynamic_ec.exampleMethod1(10, 4);

    // The following calls also do not cause compiler errors, whether
    // appropriate methods exist or not.
    dynamic_ec.someMethod("some argument", 7, null);
    dynamic_ec.nonexistentMethod();
}
```

C#

```
class ExampleClass
{
    public ExampleClass() { }
    public ExampleClass(int v) { }
```

```
public void exampleMethod1(int i) { }

public void exampleMethod2(string str) { }
}
```

이 예에서 컴파일러의 역할은 각 문이 `dynamic` 개체 또는 식에 수행할 작업에 대한 정보를 함께 패키지하는 것입니다. 런타임은 저장된 정보를 검사하며 유효하지 않은 문은 런타임 예외를 발생시킵니다.

대부분의 동적 작업은 결과 그 자체가 `dynamic`입니다. 다음 예제에서 `testSum`이 사용된 곳에 마우스 포인터를 올려두면 IntelliSense에서 (지역 변수) `dynamic testSum` 형식을 표시합니다.

C#

```
dynamic d = 1;
var testSum = d + 3;
// Rest the mouse pointer over testSum in the following statement.
System.Console.WriteLine(testSum);
```

결과가 `dynamic`이 아닌 작업은 다음과 같습니다.

- `dynamic`에서 다른 형식으로의 전환.
- `dynamic` 형식의 인수를 포함하는 생성자 호출.

예를 들어 다음 선언에서 `testInstance`의 형식은 `dynamic`이 아니라 `ExampleClass`입니다.

C#

```
var testInstance = new ExampleClass(d);
```

## 변환

동적 개체와 다른 형식 간에 손쉽게 변환할 수 있습니다. 변환을 통해 개발자는 동적 동작과 비동적 동작 사이를 전환할 수 있습니다.

다음 예에 표시된 것처럼 `any`를 암시적으로 `dynamic`으로 변환할 수 있습니다.

C#

```
dynamic d1 = 7;
dynamic d2 = "a string";
```

```
dynamic d3 = System.DateTime.Today;
dynamic d4 = System.Diagnostics.Process.GetProcesses();
```

반대로, `dynamic` 형식의 식에 암시적 변환을 동적으로 적용할 수 있습니다.

C#

```
int i = d1;
string str = d2;
DateTime dt = d3;
System.Diagnostics.Process[] procs = d4;
```

## 동적 형식의 인수로 오버로드 확인

메서드 호출 내 하나 이상의 인수에 `dynamic` 형식이 있거나 메서드 호출의 수신자가 `dynamic` 형식인 경우 오버로드 확인은 컴파일 시간이 아니라 런타임에 발생합니다. 다음 예에서 액세스 가능한 유일한 `exampleMethod2` 메서드가 문자열 인수를 사용하는 경우 `d1` 을 인수로 보내면 컴파일러 오류가 발생하지 않지만 런타임 예외가 발생합니다. `d1`의 런타임 형식은 `int`인데 `exampleMethod2`에는 문자열이 필요하므로 오버로드 확인이 런타임에 실패합니다.

C#

```
// Valid.
ec.exampleMethod2("a string");

// The following statement does not cause a compiler error, even though ec
is not
// dynamic. A run-time exception is raised because the run-time type of d1
is int.
ec.exampleMethod2(d1);
// The following statement does cause a compiler error.
//ec.exampleMethod2(7);
```

## 동적 언어 런타임

DLR(동적 언어 런타임)은 C#에서 `dynamic` 형식을 지원하는 인프라와 IronPython 및 IronRuby와 같은 동적 프로그래밍 언어의 구현도 제공합니다. DLR에 대한 자세한 내용은 [동적 언어 런타임 개요](#)를 참조하세요.

## COM interop

많은 COM 메서드는 형식을 `object`로 지정하여 인수 형식 및 반환 형식의 변환을 허용합니다. COM interop에서는 C#의 강력한 형식의 변수와 조화를 이루기 위해 값을 명시적으로 캐스팅해야 합니다. [EmbedInteropTypes\(C# 컴파일러 옵션\)](#) 옵션을 사용하여 컴파일 하는 경우 `dynamic` 형식을 도입하면 COM 서명에서 `object`의 발생을 마치 `dynamic` 형식인 것처럼 취급하여 캐스팅을 상당 부분 피할 수 있습니다. COM 개체와 함께 `dynamic` 형식을 사용하는 방법에 대한 자세한 내용은 [C# 기능을 사용하여 Office 상호 운용성 개체에 액세스하는 방법](#) 문서를 참조하세요.

## 관련 문서

 테이블 확장

제목	설명
<a href="#">dynamic</a>	<code>dynamic</code> 키워드의 사용법을 설명합니다.
<a href="#">동적 언어 런타임 개요</a>	동적 언어에 대한 서비스 집합을 CLR(공용 언어 런타임)에 추가하는 런타임 환경인 DLR 개요를 제공합니다.
<a href="#">연습: 동적 개체 만들기 및 사용</a>	사용자 지정 동적 개체를 만들고 <code>IronPython</code> 라이브러리에 액세스하는 프로젝트를 만드는데 필요한 단계별 지침을 제공합니다.



### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.



### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

[설명서 문제 열기](#)

[제품 사용자 의견 제공](#)

# 연습: C에서 동적 개체 만들기 및 사용 #

아티클 • 2023. 04. 08.

동적 개체는 컴파일 시간이 아닌 런타임에 속성 및 메서드와 같은 멤버를 노출합니다. 동적 개체를 사용하면 정적 형식 또는 형식과 일치하지 않는 구조체로 작업할 개체를 만들 수 있습니다. 예를 들어 동적 개체를 사용하여 DOM(문서 개체 모델)을 참조할 수 있습니다. DOM에는 유효한 HTML 마크업 요소 및 특성의 조합을 포함할 수 있습니다. 각 HTML 문서는 고유하므로, 특정 HTML 문서에 대한 멤버는 런타임에 의해 결정됩니다. HTML 요소의 특성을 참조하는 일반적인 방법은 요소의 `GetProperty` 메서드에 특성의 이름을 전달하는 것입니다. HTML 요소 `<div id="Div1">`의 `id` 특성을 참조하려면 먼저 `<div>` 요소에 대한 참조를 가져온 다음 `divElement.GetProperty("id")`를 사용합니다. 동적 개체를 사용하는 경우 `id` 특성을 `divElement.id`로서 참조할 수 있습니다.

동적 개체를 사용하면 IronPython 및 IronRuby와 같은 동적 언어에 편리하게 액세스할 수 있습니다. 동적 개체를 사용하여 런타임에 해석된 동적 스크립트를 참조할 수 있습니다.

런타임에 바인딩을 사용하여 동적 개체를 참조합니다. 런타임에 바인딩된 개체의 형식으로 `dynamic` 지정합니다. 자세한 내용은 [동적을 참조하세요](#).

`System.Dynamic` 네임스페이스의 클래스를 사용하여 사용자 지정 동적 개체를 만들 수 있습니다. 예를 들어 `ExpandoObject`를 만들고 해당 개체의 멤버를 런타임에 지정할 수 있습니다. `DynamicObject` 클래스를 상속하는 고유한 형식을 만들 수도 있습니다. 그런 다음 런타임 동적 기능을 제공하도록 `DynamicObject` 클래스의 멤버를 재정의할 수 있습니다.

이 문서에는 두 개의 독립적인 연습이 포함됩니다.

- 텍스트 파일의 내용을 개체의 속성으로서 동적으로 노출하는 사용자 지정 개체를 만듭니다.
- `IronPython` 라이브러리를 사용하는 프로젝트를 만듭니다.

## 사전 요구 사항

- .NET 데스크톱 개발 워크로드가 설치된 `Visual Studio 2022` 버전 17.3 이상 버전.  
.NET 7 SDK는 이 워크로드를 선택할 때 포함됩니다.

### ① 참고

일부 Visual Studio 사용자 인터페이스 요소의 경우 다음 지침에 설명된 것과 다른 이름 또는 위치가 시스템에 표시될 수 있습니다. 이러한 요소는 사용하는 Visual Studio 버전 및 설정에 따라 결정됩니다. 자세한 내용은 [IDE 개인 설정을 참조하세요](#).

- 두 번째 연습에서는 .NET용 [IronPython](#) 을 설치합니다. [다운로드 페이지](#) 로 이동하여 최신 버전을 다운로드하세요.

## 사용자 지정 동적 개체 만들기

첫 번째 연습에서는 텍스트 파일의 콘텐츠를 검색하는 사용자 지정 동적 개체를 정의합니다. 동적 속성은 검색할 텍스트를 지정합니다. 예를 들어, 호출 코드가

`dynamicFile.Sample`을 지정하면 동적 클래스는 "Sample"로 시작하는 파일의 모든 줄을 포함하는 문자열의 제네릭 목록을 반환합니다. 검색은 대/소문자를 구분합니다. 동적 클래스는 또한 두 개의 선택적 인수를 지원합니다. 첫 번째 인수는 동적 클래스가 행의 시작, 행의 끝 또는 행의 어디에서나 일치를 검색하도록 지정하는 검색 옵션 열거형 값입니다. 두 번째 인수는 동적 클래스가 검색 전에 각 행의 선행 및 후행 공백을 잘라내야 함을 지정합니다. 예를 들어 호출 코드가 `dynamicFile.Sample(StringSearchOption.Contains)` 을 지정하면 동적 클래스는 한 줄의 어디에서나 "Sample"을 검색합니다. 호출 코드가 를 지정 `dynamicFile.Sample(StringSearchOption.StartsWith, false)` 하는 경우 동적 클래스는 각 줄의 시작 부분에 있는 "샘플"을 검색하고 선행 및 후행 공백을 제거하지 않습니다. 동적 클래스의 기본 동작은 각 줄의 시작 부분에서 일치를 검색하고 선행 및 후행 공백을 제거하는 것입니다.

## 사용자 지정 동적 클래스 만들기

Visual Studio를 시작합니다. 새 프로젝트 만들기를 선택합니다. 새 프로젝트 만들기 대화 상자에서 C#을 선택하고 콘솔 애플리케이션을 선택한 다음, 다음을 선택합니다. 새 프로젝트 구성 대화 상자에서 프로젝트 이름으로 `DynamicSample`을 입력하고 다음을 선택합니다. 추가 정보 대화 상자에서 대상 프레임워크에 대한 .NET 7.0(현재)을 선택한 다음, 만들기를 선택합니다. 솔루션 탐색기에서 `DynamicSample` 프로젝트를 마우스 오른쪽 단추로 클릭하고 추가>클래스를 선택합니다. 이름 상자에 `ReadOnlyFile`을 입력한 다음, 추가를 선택합니다. `ReadOnlyFile.cs` 또는 `ReadOnlyFile.vb` 파일 맨 위에 다음 코드를 추가하여 `System.IO` 및 `System.Dynamic` 네임스페이스를 가져옵니다.

C#

```
using System.IO;
using System.Dynamic;
```

사용자 지정 동적 개체는 열거형을 사용하여 검색 기준을 결정합니다. Class 문 앞에 다음 열거형 정의를 추가합니다.

C#

```
public enum StringSearchOption
{
    StartsWith,
    Contains,
    EndsWith
}
```

다음 코드 예제와 같이, `DynamicObject` 클래스를 상속하도록 class 문을 업데이트합니다.

C#

```
class ReadOnlyFile : DynamicObject
```

다음 코드를 `ReadOnlyFile` 클래스에 추가하여 파일 경로의 전용 필드 및 `ReadOnlyFile` 클래스의 생성자를 정의합니다.

C#

```
// Store the path to the file and the initial line count value.
private string p_filePath;

// Public constructor. Verify that file exists and store the path in
// the private variable.
public ReadOnlyFile(string filePath)
{
    if (!File.Exists(filePath))
    {
        throw new Exception("File path does not exist.");
    }

    p_filePath = filePath;
}
```

1. 다음 `GetPropertyValue` 메서드를 `ReadOnlyFile` 클래스에 추가합니다.

`GetPropertyValue` 메서드는 검색 기준을 입력으로 가져오고 해당 검색 기준과 일치하는 텍스트 파일로부터 줄을 반환합니다. `ReadOnlyFile` 클래스가 제공하는 동적 메서드는 `GetPropertyValue` 메서드를 호출하여 각 결과를 검색합니다.

C#

```
public List<string> GetPropertyValue(string propertyName,
                                      StringSearchOption StringSearchOption =
StringSearchOption.StartsWith,
                                      bool trimSpaces = true)
{
    StreamReader sr = null;
    List<string> results = new List<string>();
```

```

string line = "";
string testLine = "";

try
{
    sr = new StreamReader(p_filePath);

    while (!sr.EndOfStream)
    {
        line = sr.ReadLine();

        // Perform a case-insensitive search by using the specified
        search options.
        testLine = line.ToUpper();
        if (trimSpaces) { testLine = testLine.Trim(); }

        switch (StringSearchOption)
        {
            case StringSearchOption.StartsWith:
                if (testLine.StartsWith(propertyName.ToUpper())) {
results.Add(line); }
                break;
            case StringSearchOption.Contains:
                if (testLine.Contains(propertyName.ToUpper())) {
results.Add(line); }
                break;
            case StringSearchOption.EndsWith:
                if (testLine.EndsWith(propertyName.ToUpper())) {
results.Add(line); }
                break;
        }
    }
}
catch
{
    // Trap any exception that occurs in reading the file and return
null.
    results = null;
}
finally
{
    if (sr != null) {sr.Close();}
}

return results;
}

```

`GetPropertyValues` 메서드 뒤에 다음 코드를 추가하여 `DynamicObject` 클래스의 `TryGetMember` 메서드를 재정의합니다. 동적 클래스의 멤버를 요청했는데 지정된 인수가 없는 경우 `TryGetMember` 메서드가 호출됩니다. `binder` 인수는 참조된 멤버에 대한 정보를 포함하며, `result` 인수는 지정된 멤버에 대해 반환된 결과를 참조합니다.

`TryGetMember` 메서드는 요청한 멤버가 있는 경우 `true`, 없는 경우 `false`를 반환하는 부울 값을 반환합니다.

C#

```
// Implement the TryGetMember method of the DynamicObject class for dynamic
// member calls.
public override bool TryGetMember(GetMemberBinder binder,
                                    out object result)
{
    result = GetPropertyValue(binder.Name);
    return result == null ? false : true;
}
```

`TryGetMember` 메서드 뒤에 다음 코드를 추가하여 `DynamicObject` 클래스의 `TryInvokeMember` 메서드를 재정의합니다. 인수를 사용하여 동적 클래스의 멤버를 요청하면 `TryInvokeMember` 메서드가 호출됩니다. `binder` 인수는 참조된 멤버에 대한 정보를 포함하며, `result` 인수는 지정된 멤버에 대해 반환된 결과를 참조합니다. `args` 인수는 멤버에 전달되는 인수의 배열을 포함합니다. `TryInvokeMember` 메서드는 요청한 멤버가 있는 경우 `true`, 없는 경우 `false`를 반환하는 부울 값을 반환합니다.

`TryInvokeMember` 메서드의 사용자 지정 버전은 첫 번째 인수로 이전 단계에서 정의한 `StringSearchOption` 열거형의 값을 예상합니다. `TryInvokeMember` 메서드는 두 번째 인수로 부울 값을 예상합니다. 하나 또는 두 인수가 유효한 값이면 결과를 검색하기 위해 `GetPropertyValues` 메서드에 전달됩니다.

C#

```
// Implement the TryInvokeMember method of the DynamicObject class for
// dynamic member calls that have arguments.
public override bool TryInvokeMember(InvokeMemberBinder binder,
                                      object[] args,
                                      out object result)
{
    StringSearchOption StringSearchOption = StringSearchOption.StartsWith;
    bool trimSpaces = true;

    try
    {
        if (args.Length > 0) { StringSearchOption =
(StringSearchOption)args[0]; }
    }
    catch
    {
        throw new ArgumentException("StringSearchOption argument must be a
StringSearchOption enum value.");
    }
}
```

```

try
{
    if (args.Length > 1) { trimSpaces = (bool)args[1]; }
}
catch
{
    throw new ArgumentException("trimSpaces argument must be a Boolean
value.");
}

result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces);

return result == null ? false : true;
}

```

파일을 저장하고 닫습니다.

## 샘플 텍스트 파일 만들기

솔루션 탐색기에서 DynamicSample 프로젝트를 마우스 오른쪽 단추로 클릭하고 **추가>새 항목**을 선택합니다. 설치된 템플릿 창에서 **일반**을 선택한 다음 **텍스트 파일** 템플릿을 선택합니다. 이름 **상자에** *TextFile1.txt* 기본 이름을 그대로 두고 **추가**를 선택합니다. *TextFile1.txt* 파일에 다음 텍스트를 복사합니다.

```

text

List of customers and suppliers

Supplier: Lucerne Publishing (https://www.lucernepublishing.com/)
Customer: Preston, Chris
Customer: Hines, Patrick
Customer: Cameron, Maria
Supplier: Graphic Design Institute (https://www.graphicdesigninstitute.com/)
Supplier: Fabrikam, Inc. (https://www.fabrikam.com/)
Customer: Seubert, Roxanne
Supplier: Proseware, Inc. (http://www.proseware.com/)
Customer: Adolphi, Stephan
Customer: Koch, Paul

```

파일을 저장한 후 닫습니다.

## 사용자 지정 동적 개체를 사용하는 샘플 애플리케이션 만들기

솔루션 탐색기 *Program.cs* 파일을 두 번 클릭합니다. **Main** 프로시저에 다음 코드를 추가하여 *TextFile1.txt* 파일에 대한 **ReadOnlyFile** 클래스의 인스턴스를 만듭니다. 코드는 런타임

에 바인딩을 사용하여 동적 멤버를 호출하고 "Customer" 문자열이 포함된 텍스트의 줄을 검색합니다.

```
C#  
  
dynamic rFile = new ReadOnlyFile(@"..\..\..\TextFile1.txt");  
foreach (string line in rFile.Customer)  
{  
    Console.WriteLine(line);  
}  
Console.WriteLine("-----");  
foreach (string line in rFile.Customer(StringSearchOption.Contains, true))  
{  
    Console.WriteLine(line);  
}
```

파일을 저장하고 **ctrl+F5**를 눌러 애플리케이션을 빌드하고 실행합니다.

## 동적 언어 라이브러리 호출

다음 연습에서는 동적 언어 IronPython으로 작성된 라이브러리에 액세스하는 프로젝트를 만듭니다.

### 사용자 지정 동적 클래스를 만들려면

Visual Studio에서 파일>새로 만들기>프로젝트를 선택합니다. 새 프로젝트 만들기 대화 상자에서 C#을 선택하고 콘솔 애플리케이션을 선택한 다음, 다음을 선택합니다. 새 프로젝트 구성 대화 상자에서 프로젝트 이름으로 `DynamicIronPythonSample`을 입력하고 다음을 선택합니다. 추가 정보 대화 상자에서 대상 프레임워크에 대한 .NET 7.0(현재)을 선택한 다음, 만들기를 선택합니다. [IronPython](#) NuGet 패키지를 설치합니다. `Program.cs` 파일을 편집합니다. 파일 맨 위에 다음 코드를 추가하여 IronPython 라이브러리의 `Microsoft.Scripting.Hosting` 및 `IronPython.Hosting` 네임스페이스와 `System.Linq` 네임스페이스를 가져옵니다.

```
C#  
  
using System.Linq;  
using Microsoft.Scripting.Hosting;  
using IronPython.Hosting;
```

Main 메서드에서 다음 코드를 추가하여 IronPython 라이브러리를 호스트하기 위한 새 `Microsoft.Scripting.Hosting.ScriptRuntime` 개체를 만듭니다. `ScriptRuntime` 개체는 IronPython 라이브러리 모듈 `random.py`를 로드합니다.

C#

```
// Set the current directory to the IronPython libraries.  
System.IO.Directory.SetCurrentDirectory(  
    Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles) +  
    @"\IronPython 2.7\Lib");  
  
// Create an instance of the random.py IronPython library.  
Console.WriteLine("Loading random.py");  
ScriptRuntime py = Python.CreateRuntime();  
dynamic random = py.UseFile("random.py");  
Console.WriteLine("random.py loaded.");
```

random.py 모듈을 로드할 코드 뒤에 다음 코드를 추가하여 정수 배열을 만듭니다. 배열은 random.py 모듈의 `shuffle` 메서드로 전달되며, 이 모듈은 배열의 값을 임의로 정렬합니다.

C#

```
// Initialize an enumerable set of integers.  
int[] items = Enumerable.Range(1, 7).ToArray();  
  
// Randomly shuffle the array of integers by using IronPython.  
for (int i = 0; i < 5; i++)  
{  
    random.shuffle(items);  
    foreach (int item in items)  
    {  
        Console.WriteLine(item);  
    }  
    Console.WriteLine("-----");  
}
```

파일을 저장하고 `ctrl+F5`를 눌러 애플리케이션을 빌드하고 실행합니다.

## 참조

- [System.Dynamic](#)
- [System.Dynamic.DynamicObject](#)
- [dynamic 형식 사용](#)
- [dynamic](#)
- [동적 인터페이스 구현\(Microsoft TechNet에서 다운로드 가능한 PDF\)](#)

 Collaborate with us on  
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

# 새로운 C# 기능을 사용하여 메모리 할당 줄이기

아티클 • 2024. 02. 17.

## ① 중요

이 섹션에 설명된 기술은 코드의 실행 부하 과다 경로에 적용할 때 성능을 개선합니다. 실행 부하 과다 경로는 일반 작업에서 자주 반복적으로 실행되는 코드베이스의 섹션입니다. 자주 실행되지 않는 코드에 이러한 기술을 적용하면 영향이 최소화됩니다. 성능 개선을 위해 변경하기 전에 기준을 측정해야 합니다. 그런 다음 해당 기준을 분석하여 메모리 병목 현상이 발생하는 위치를 확인합니다. [진단 및 계측](#) 섹션에서 애플리케이션 성능을 측정하는 다양한 플랫폼 간 도구에 대해 알아볼 수 있습니다. Visual Studio 설명서의 [메모리 사용량 측정](#) 자습서에서 프로파일링 세션을 실습할 수 있습니다.

메모리 사용량을 측정하고 할당을 줄일 수 있다고 판단한 후에는 이 섹션의 기술을 사용하여 할당을 줄입니다. 연속적인 변경이 있을 때마다 메모리 사용량을 다시 측정합니다. 각 변경 내용이 애플리케이션의 메모리 사용량에 긍정적인 영향을 미치는지 확인합니다.

.NET에서의 성능 작업은 종종 코드에서 할당을 제거하는 것을 의미합니다. 할당한 모든 메모리 블록은 결국 해제되어야 합니다. 할당량이 적으면 가비지 수집에 소요되는 시간이 줄어듭니다. 특정 코드 경로에서 가비지 수집을 제거하여 보다 예측 가능한 실행 시간을 허용합니다.

할당을 줄이는 일반적인 전략은 중요한 데이터 구조를 `class` 형식에서 `struct` 형식으로 변경하는 것입니다. 이 변경 내용은 해당 형식을 사용하는 의미 체계에 영향을 미칩니다. 이제 매개 변수와 반환값이 참조 대신 값으로 전달됩니다. 형식이 작고 세 단어 이하인 경우 값을 복사하는 비용은 무시할 수 있습니다(한 단어가 하나의 정수의 보통 크기인 것을 고려). 이는 측정 가능하며 더 큰 형식의 경우 실제 성능에 영향을 미칠 수 있습니다. 복사 효과를 방지하기 위해 개발자는 이러한 형식을 `ref`로 전달하여 의도한 의미 체계를 다시 가져올 수 있습니다.

C# `ref` 기능을 사용하면 전반적인 유용성에 부정적인 영향을 주지 않고 `struct` 형식에 대해 원하는 의미 체계를 표현할 수 있습니다. 이러한 개선 사항 이전에 개발자는 동일한 성능 영향을 얻기 위해 포인터와 원시 메모리가 포함된 `unsafe` 구문에 의존해야 했습니다. 컴파일러는 새로운 `ref` 관련 기능에 대해 검증 가능한 안전한 코드를 생성합니다. 확실히 안전한 코드는 컴파일러가 가능한 버퍼 오버런이나 할당되지 않거나 해제된 메모리

에 액세스하는 것을 검색한다는 것을 의미합니다. 컴파일러는 일부 오류를 검색하고 방지합니다.

## 참조로 전달 및 반환

C#의 변수는 값을 저장합니다. `struct` 형식에서 같은 해당 형식의 인스턴스 콘텐츠입니다. `class` 형식에서 같은 해당 형식의 인스턴스를 저장하는 메모리 블록에 대한 참조입니다. `ref` 한정자를 추가한다는 것은 변수가 값에 대한 참조를 저장한다는 의미입니다. `struct` 형식에서 참조는 값이 포함된 스토리지를 가리킵니다. `class` 형식에서 참조는 메모리 블록에 대한 참조를 포함하는 스토리지를 가리킵니다.

C#에서 메서드에 대한 매개 변수는 값으로 전달되고, 반환 값은 값으로 반환됩니다. 인수의 값이 메서드에 전달됩니다. 반환 인수의 값은 반환 값입니다.

`ref`, `in`, `ref readonly` 또는 `out` 한정자는 인수가 참조로 전달되었음을 나타냅니다. 스토리지 위치에 대한 참조가 메서드에 전달됩니다. 메서드 서명에 `ref`를 추가하면 반환 값이 참조로 반환된다는 의미입니다. 스토리지 위치에 대한 참조가 반환 값입니다.

또한 `ref` 할당을 사용하여 변수가 다른 변수를 참조하도록 할 수도 있습니다. 일반적인 할당은 오른쪽의 값을 할당의 왼쪽에 있는 변수에 복사합니다. `ref` 할당은 오른쪽 변수의 메모리 위치를 왼쪽 변수에 복사합니다. 이제 `ref`는 원래 변수를 참조하세요.

C#

```
int anInteger = 42; // assignment.  
ref int location = ref anInteger; // ref assignment.  
ref int sameLocation = ref location; // ref assignment  
  
Console.WriteLine(location); // output: 42  
  
sameLocation = 19; // assignment  
  
Console.WriteLine(anInteger); // output: 19
```

변수를 할당하면 해당 값이 변경됩니다. 변수를 `ref` 할당하면 변수가 참조하는 내용이 변경됩니다.

`ref` 변수, 참조로 전달 및 `ref` 할당을 사용하여 값 스토리지로 직접 작업할 수 있습니다. 컴파일러에 의해 적용되는 범위 규칙은 스토리지로 직접 작업할 때 안전을 보장합니다.

`ref readonly` 및 `in` 한정자는 모두 인수가 참조로 전달되어야 하며 메서드에서 다시 할당될 수 없음을 나타냅니다. 차이점은 `ref readonly`가 메서드가 매개 변수를 변수로 사용한다는 것을 나타냅니다. 메서드는 매개 변수를 캡처하거나 읽기 전용 참조로 매개 변수를 반환할 수 있습니다. 이러한 경우에는 `ref readonly` 한정자를 사용해야 합니다. 그렇

지 않으면 `in` 한정자가 더 많은 유연성을 제공합니다. `in` 매개 변수의 인수에 `in` 한정자를 추가할 필요가 없으므로 `in` 한정자를 사용하여 기존 API 서명을 안전하게 업데이트할 수 있습니다. `ref readonly` 매개 변수의 인수에 `ref` 또는 `in` 한정자를 추가하지 않으면 컴파일러는 경고를 표시합니다.

## ref safe 컨텍스트

C#에는 참조하는 스토리지가 더 이상 유효하지 않은 경우 `ref` 식에 액세스할 수 없도록 하는 `ref` 식에 대한 규칙이 포함되어 있습니다. 다음 예제를 참조하세요.

C#

```
public ref int CantEscape()
{
    int index = 42;
    return ref index; // Error: index's ref safe context is the body of
CantEscape
}
```

메서드에서 지역 변수에 대한 참조를 반환할 수 없기 때문에 컴파일러는 오류를 보고합니다. 호출자는 참조되는 스토리지에 액세스할 수 없습니다. `ref safe` 컨텍스트는 `ref` 식이 액세스하거나 수정하기에 safe 범위를 정의합니다. 다음 표에는 변수 형식에 대한 `ref safe` 컨텍스트가 나열되어 있습니다. `ref` 필드는 `class` 또는 `ref`가 아닌 `struct`에서 선언될 수 없으므로 해당 행은 테이블에 없습니다.

[+] 테이블 확장

선언	<code>ref safe</code> 컨텍스트
<code>ref</code> 가 아닌 로컬	local이 선언된 블록
<code>ref</code> 가 아닌 매개 변수	현재 메서드
<code>ref</code> , <code>ref readonly</code> , <code>in</code> 매개 변수	호출 메서드
<code>out</code> 매개 변수	현재 메서드
<code>class</code> 필드	호출 메서드
<code>ref</code> 가 아닌 <code>struct</code> 필드	현재 메서드
<code>ref struct</code> 의 <code>ref</code> 필드	호출 메서드

`ref safe` 컨텍스트가 호출 메서드인 경우 변수는 `ref` 반환될 수 있습니다. `ref safe` 컨텍스트가 현재 메서드 또는 블록인 경우 `ref` 반환이 허용되지 않습니다. 다음 코드 조각은 두

가지 예를 보여 줍니다. 메서드를 호출하는 범위에서 멤버 필드에 액세스할 수 있으므로 클래스 또는 구조체 필드의 `ref safe` 컨텍스트가 호출 메서드입니다. `ref` 또는 `in` 한정자가 있는 매개 변수에 대한 `ref safe` 컨텍스트는 전체 메서드입니다. 둘 다 멤버 메서드에서 `ref` 반환될 수 있습니다.

C#

```
private int anIndex;

public ref int RetrieveIndexRef()
{
    return ref anIndex;
}

public ref int RefMin(ref int left, ref int right)
{
    if (left < right)
        return ref left;
    else
        return ref right;
}
```

## ① 참고

`ref readonly` 또는 `in` 한정자가 매개 변수에 적용되면 해당 매개 변수는 `ref`가 아닌 `ref readonly`에 의해 반환될 수 있습니다.

컴파일러는 참조가 `ref safe` 컨텍스트를 벗어날 수 없도록 보장합니다. 스토리지가 유효하지 않을 때 `ref` 식에 액세스할 수 있는 코드를 실수로 작성했는지 컴파일러가 검색하므로 `ref` 매개 변수, `ref return` 및 `ref` 지역 변수를 안전하게 사용할 수 있습니다.

## safe 컨텍스트 및 ref 구조체

`ref struct` 형식을 안전하게 사용하려면 더 많은 규칙이 필요합니다. `ref struct` 형식에는 `ref` 필드가 포함될 수 있습니다. 이를 위해서는 `safe` 컨텍스트의 도입이 필요합니다. 대부분의 형식에서 `safe` 컨텍스트는 호출 메서드입니다. 즉, `ref struct` 가 아닌 값은 항상 메서드에서 반환될 수 있습니다.

비공식적으로 `ref struct` 의 `safe` 컨텍스트는 모든 `ref` 필드에 액세스할 수 있는 범위입니다. 즉, 모든 `ref` 필드의 `ref safe` 컨텍스트가 교차하는 지점입니다. 다음 메서드는 멤버 필드에 `ReadOnlySpan<char>` 를 반환하므로 해당 `safe` 컨텍스트가 메서드입니다.

C#

```

private string longMessage = "This is a long message";

public ReadOnlySpan<char> Safe()
{
    var span = longMessage.AsSpan();
    return span;
}

```

반면에 다음 코드는 `Span<int>`의 `ref field` 멤버가 스택에 할당된 정수 배열을 참조하기 때문에 오류를 발생시킵니다. 메서드를 벗어날 수 없습니다.

C#

```

public Span<int> M()
{
    int length = 3;
    Span<int> numbers = stackalloc int[length];
    for (var i = 0; i < length; i++)
    {
        numbers[i] = i;
    }
    return numbers; // Error! numbers can't escape this method.
}

```

## 메모리 형식 통합

`System.Span<T>` 및 `System.Memory<T>`의 도입으로 메모리 작업을 위한 통합 모델이 제공됩니다. `System.ReadOnlySpan<T>` 및 `System.ReadOnlyMemory<T>`는 메모리 액세스를 위한 읽기 전용 버전을 제공합니다. 이들은 모두 유사한 요소의 배열을 저장하는 메모리 블록에 대한 추상화를 제공합니다. 차이점은 `Span<T>`와 `ReadOnlySpan<T>`는 `ref struct` 형식이고 `Memory<T>`와 `ReadOnlyMemory<T>`는 `struct` 형식이라는 것입니다. 범위에는 `ref field` 가 포함되어 있습니다. 따라서 범위의 인스턴스는 `safe` 컨텍스트를 벗어날 수 없습니다. `ref struct`의 `safe` 컨텍스트는 `ref field`의 `ref safe` 컨텍스트입니다. `Memory<T>` 및 `ReadOnlyMemory<T>`를 구현하면 이 제한이 제거됩니다. 이러한 형식을 사용하여 메모리 버퍼에 직접 액세스합니다.

## ref safety로 성능 개선

성능을 개선시키기 위해 이러한 기능을 사용하는 작업에는 다음이 포함됩니다.

- **할당 방지:** 형식을 `class`에서 `struct`로 변경하면 저장 방법이 변경됩니다. 지역 변수는 스택에 저장됩니다. 멤버는 컨테이너 자체가 할당될 때 인라인으로 저장됩니다.

다. 이 변경은 할당 횟수가 줄어들고 가비지 수집기의 작업이 줄어드는 것을 의미합니다. 또한 메모리 압력을 줄여 가비지 수집기 실행 빈도를 줄일 수도 있습니다.

- **참조 의미 체계 유지:** 형식을 `class`에서 `struct`로 변경하면 변수를 메서드에 전달하는 의미 체계가 변경됩니다. 매개 변수의 상태를 수정한 코드는 수정이 필요합니다. 이제 매개 변수가 `struct`이므로 메서드가 원본 개체의 복사본을 수정합니다. 해당 매개 변수를 `ref` 매개 변수로 전달하여 원래 의미 체계를 복원할 수 있습니다. 변경 후 메서드는 원래 `struct`를 다시 수정합니다.
- **데이터 복사 방지:** 더 큰 `struct` 형식을 복사하면 일부 코드 경로의 성능에 영향을 미칠 수 있습니다. 또한 `ref` 한정자를 추가하여 값 대신 참조로 더 큰 데이터 구조를 메서드에 전달할 수도 있습니다.
- **수정 제한:** `struct` 형식이 참조로 전달되면 호출된 메서드가 구조체의 상태를 수정할 수 있습니다. `ref` 한정자를 `ref readonly` 또는 `in` 한정자로 바꿔 인수를 수정할 수 없음을 나타낼 수 있습니다. 메서드가 매개 변수를 캡처하거나 읽기 전용 참조로 반환하는 경우 `ref readonly`를 선호합니다. `readonly` 멤버로 `readonly struct` 형식 또는 `struct` 형식을 만들어 수정할 수 있는 `struct` 멤버를 더 효과적으로 제어할 수도 있습니다.
- **메모리 직접 조작:** 일부 알고리즘은 데이터 구조를 일련의 요소가 포함된 메모리 블록으로 처리할 때 가장 효율적입니다. `Span` 및 `Memory` 형식은 메모리 블록에 대한 안전한 액세스를 제공합니다.

이러한 기술에는 `unsafe` 코드가 필요하지 않습니다. 현명하게 사용하면 이전에는 안전하지 않은 기술을 통해서만 가능했던 안전한 코드로부터 성능 특성을 가져올 수 있습니다. [메모리 할당 줄이기에 대한 자습서](#)에서 기술을 직접 시도해 볼 수 있습니다.

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

💡 설명서 문제 열기

↗️ 제품 사용자 의견 제공

# 자습서: `ref` 안전성으로 메모리 할당 줄이기

아티클 • 2024. 02. 17.

.NET 애플리케이션의 성능 튜닝에는 두 가지 기술이 필요한 경우가 많습니다. 먼저, 힙 할당의 수와 크기를 줄입니다. 둘째, 데이터가 복사되는 빈도를 줄입니다. Visual Studio는 애플리케이션이 메모리를 사용하는 방식을 분석하는 데 도움이 되는 훌륭한 [도구](#)를 제공합니다. 앱에서 불필요한 할당을 수행하는 위치를 확인한 후에는 해당 할당을 최소화하도록 변경합니다. `class` 형식을 `struct` 형식으로 변환합니다. 의미 체계를 유지하고 추가 복사를 최소화하려면 `ref` 안전 [기능](#)을 사용합니다.

이 자습서를 최대한 활용하려면 [Visual Studio 17.5](#) 를 사용합니다. 메모리 사용량을 분석하는 데 사용되는 .NET 개체 할당 도구는 Visual Studio의 일부입니다. [Visual Studio Code](#) 와 명령줄을 사용하여 애플리케이션을 실행하고 모든 변경 작업을 수행할 수 있습니다. 그러나 변경 내용에 대한 분석 결과를 볼 수는 없습니다.

사용하게 될 애플리케이션은 침입자가 귀중품을 가지고 비밀 갤러리에 들어왔는지 확인하기 위해 여러 센서를 모니터링하는 IoT 애플리케이션의 시뮬레이션입니다. IoT 센서는 공기 중 산소(O<sub>2</sub>)와 이산화탄소(CO<sub>2</sub>)의 혼합을 측정하는 데이터를 지속적으로 전송합니다. 또한 온도와 상대 습도도 보고합니다. 이러한 각 값은 항상 약간씩 변동합니다. 그러나 사람이 방에 들어가면 변화가 조금 더 심해지고 항상 같은 방향으로 나타납니다. 즉, 산소는 감소하고 이산화탄소는 증가하며 온도는 증가하고 상대 습도도 증가합니다. 센서가 결합되어 증가를 표시하면 침입자 경보가 트리거됩니다.

이 자습서에서는 애플리케이션을 실행하고 메모리 할당을 측정한 다음 할당 수를 줄여 성능을 개선합니다. 소스 코드는 [샘플 브라우저](#)에서 사용할 수 있습니다.

## 시작 애플리케이션 살펴보기

애플리케이션을 다운로드하고 시작 샘플을 실행합니다. 시작 애플리케이션은 올바르게 작동하지만 각 측정 주기마다 많은 작은 개체를 할당하기 때문에 시간이 지남에 따라 실행되면서 성능이 서서히 저하됩니다.

콘솔

Press <return> to start simulation

Debounced measurements:

Temp: 67.332

Humidity: 41.077%

Oxygen: 21.097%

CO<sub>2</sub> (ppm): 404.906

```
Average measurements:
```

```
Temp: 67.332  
Humidity: 41.077%  
Oxygen: 21.097%  
CO2 (ppm): 404.906
```

```
Debounced measurements:
```

```
Temp: 67.349  
Humidity: 46.605%  
Oxygen: 20.998%  
CO2 (ppm): 408.707
```

```
Average measurements:
```

```
Temp: 67.349  
Humidity: 46.605%  
Oxygen: 20.998%  
CO2 (ppm): 408.707
```

많은 행이 제거되었습니다.

```
콘솔
```

```
Debounced measurements:
```

```
Temp: 67.597  
Humidity: 46.543%  
Oxygen: 19.021%  
CO2 (ppm): 429.149
```

```
Average measurements:
```

```
Temp: 67.568  
Humidity: 45.684%  
Oxygen: 19.631%  
CO2 (ppm): 423.498
```

```
Current intruders: 3
```

```
Calculated intruder risk: High
```

```
Debounced measurements:
```

```
Temp: 67.602  
Humidity: 46.835%  
Oxygen: 19.003%  
CO2 (ppm): 429.393
```

```
Average measurements:
```

```
Temp: 67.568  
Humidity: 45.684%  
Oxygen: 19.631%  
CO2 (ppm): 423.498
```

```
Current intruders: 3
```

```
Calculated intruder risk: High
```

코드를 탐색하여 애플리케이션 작동 방식을 알아볼 수 있습니다. 메인 프로그램은 시뮬레이션을 실행합니다. <Enter>를 누르면 방이 만들어지고 일부 초기 데이터가 수집됩니다.

C#

```
Console.WriteLine("Press <return> to start simulation");
Console.ReadLine();
var room = new Room("gallery");
var r = new Random();

int counter = 0;

room.TakeMeasurements(
    m =>
{
    Console.WriteLine(room.Debounce);
    Console.WriteLine(room.Average);
    Console.WriteLine();
    counter++;
    return counter < 20000;
});
```

기준 데이터가 설정되면 방에서 시뮬레이션을 실행합니다. 여기서 난수 생성기는 침입자가 방에 들어왔는지 확인합니다.

C#

```
counter = 0;
room.TakeMeasurements(
    m =>
{
    Console.WriteLine(room.Debounce);
    Console.WriteLine(room.Average);
    room.Intruders += (room.Intruders, r.Next(5)) switch
    {
        ( > 0, 0 ) => -1,
        ( < 3, 1 ) => 1,
        _ => 0
    };

    Console.WriteLine($"Current intruders: {room.Intruders}");
    Console.WriteLine($"Calculated intruder risk: {room.RiskStatus}");
    Console.WriteLine();
    counter++;
    return counter < 200000;
});
```

다른 형식에는 측정값, 최근 50개 측정값의 평균인 디바운싱된 측정값 및 수행된 모든 측정값의 평균이 포함됩니다.

그런 다음 [.NET 개체 할당 도구](#)를 사용하여 애플리케이션을 실행합니다. `Debug` 빌드가 아닌 `Release` 빌드를 사용하고 있는지 확인합니다. `Ctrl+Shift+F5` 또는 `Ctrl+Shift+Shift+F5`를 엽니다. [.NET 개체 할당](#) 추적 옵션을 확인하고 다른 옵션은 확인하지 않습니다. 애플리케이

션을 완료할 때까지 실행합니다. 프로파일러는 개체 할당을 측정하고 할당 및 가비지 수집 주기에 대해 보고합니다. 다음 이미지와 유사한 그래프가 표시됩니다.



이전 그림에서는 할당을 최소화하는 작업이 성능상의 이점을 제공한다는 것을 보여 줍니다. 실시간 개체 그래프에 톱니 모양 패턴이 표시됩니다. 이는 빠르게 쓰레기�이 되는 수 많은 개체가 만들어진다는 것을 알려 줍니다. 개체 델타 그래프에 표시된 대로 나중에 수집됩니다. 아래쪽 빨간색 막대는 가비지 수집 주기를 나타냅니다.

다음으로 그림 아래에 있는 할당 탭을 살펴봅니다. 다음 표는 가장 많이 할당된 형식을 보여 줍니다.

Type	Allocations
Small Object Heap	
System.String	842,451
IntruderAlert.SensorMeasurement	220,000
IntruderAlert.IntruderRisk	200,000
System.Char[]	80
System.Diagnostics.Tracing.EventSource.EventMetadata[]	7
System.Byte[]	339

`System.String` 형식은 가장 많은 할당을 설명합니다. 가장 중요한 작업은 문자열 할당 빈도를 최소화하는 것입니다. 이 애플리케이션은 다양한 형식의 출력을 콘솔에 지속적으로 인쇄합니다. 이 시뮬레이션에서는 메시지를 유지하려고 하므로 다음 두 행인 `SensorMeasurement` 형식과 `IntruderRisk` 형식에 집중할 예정입니다.

`SensorMeasurement` 라인을 두 번 클릭합니다. 모든 할당이 `static` 메서드 `SensorMeasurement.TakeMeasurement`에서 발생하는 것을 볼 수 있습니다. 다음 코드 조각에서 메서드를 볼 수 있습니다.

```
C#
```

```
public static SensorMeasurement TakeMeasurement(string room, int intruders)
{
    return new SensorMeasurement
    {
        CO2 = (CO2Concentration + intruders * 10) + (20 *
```

```

        generator.NextDouble() - 10.0),
        O2 = (O2Concentration - intruders * 0.01) + (0.005 *
generator.NextDouble() - 0.0025),
        Temperature = (TemperatureSetting + intruders * 0.05) + (0.5 *
generator.NextDouble() - 0.25),
        Humidity = (HumiditySetting + intruders * 0.005) + (0.20 *
generator.NextDouble() - 0.10),
        Room = room,
        TimeRecorded = DateTime.Now
    );
}

```

모든 측정은 `class` 형식인 새로운 `SensorMeasurement` 객체를 할당합니다. 모든 `SensorMeasurement` 가 만들어지면 힙이 할당됩니다.

## 클래스를 구조체로 변경

다음 코드는 `SensorMeasurement` 의 초기 선언을 보여 줍니다.

C#

```

public class SensorMeasurement
{
    private static readonly Random generator = new Random();

    public static SensorMeasurement TakeMeasurement(string room, int
intruders)
    {
        return new SensorMeasurement
        {
            CO2 = (CO2Concentration + intruders * 10) + (20 *
generator.NextDouble() - 10.0),
            O2 = (O2Concentration - intruders * 0.01) + (0.005 *
generator.NextDouble() - 0.0025),
            Temperature = (TemperatureSetting + intruders * 0.05) + (0.5 *
generator.NextDouble() - 0.25),
            Humidity = (HumiditySetting + intruders * 0.005) + (0.20 *
generator.NextDouble() - 0.10),
            Room = room,
            TimeRecorded = DateTime.Now
        };
    }

    private const double CO2Concentration = 409.8; // increases with people.
    private const double O2Concentration = 0.2100; // decreases
    private const double TemperatureSetting = 67.5; // increases
    private const double HumiditySetting = 0.4500; // increases

    public required double CO2 { get; init; }
    public required double O2 { get; init; }
    public required double Temperature { get; init; }
}

```

```

public required double Humidity { get; init; }
public required string Room { get; init; }
public required DateTime TimeRecorded { get; init; }

public override string ToString() => $"""
    Room: {Room} at {TimeRecorded}:
        Temp:      {Temperature:F3}
        Humidity:  {Humidity:P3}
        Oxygen:    {O2:P3}
        CO2 (ppm): {CO2:F3}
"""
}

```

이 형식은 수많은 `double` 측정값을 포함하므로 원래 `class`로 만들어졌습니다. 실행 부하 과다 경로에 복사하려는 것보다 큽니다. 그러나 그 결정은 많은 할당을 의미했습니다. 형식을 `class`에서 `struct`로 변경합니다.

`class`에서 `struct`로 변경하면 원본 코드가 몇 군데에서 `null` 참조 확인을 사용했기 때문에 몇 가지 컴파일러 오류가 발생합니다. 첫 번째는 `DebounceMeasurement` 클래스의 `AddMeasurement` 메서드에 있습니다.

C#

```

public void AddMeasurement(SensorMeasurement datum)
{
    int index = totalMeasurements % debounceSize;
    recentMeasurements[index] = datum;
    totalMeasurements++;
    double sumCO2 = 0;
    double sumO2 = 0;
    double sumTemp = 0;
    double sumHumidity = 0;
    for (int i = 0; i < debounceSize; i++)
    {
        if (recentMeasurements[i] is not null)
        {
            sumCO2 += recentMeasurements[i].CO2;
            sumO2+= recentMeasurements[i].O2;
            sumTemp+= recentMeasurements[i].Temperature;
            sumHumidity += recentMeasurements[i].Humidity;
        }
    }
    O2 = sumO2 / ((totalMeasurements > debounceSize) ? debounceSize :
    totalMeasurements);
    CO2 = sumCO2 / ((totalMeasurements > debounceSize) ? debounceSize :
    totalMeasurements);
    Temperature = sumTemp / ((totalMeasurements > debounceSize) ?
    debounceSize : totalMeasurements);
    Humidity = sumHumidity / ((totalMeasurements > debounceSize) ?

```

```
debounceSize : totalMeasurements);  
}
```

`DebounceMeasurement` 형식에는 50개의 측정값 배열이 포함되어 있습니다. 센서 판독값은 최근 50회 측정의 평균으로 보고됩니다. 그러면 판독값의 노이즈가 줄어듭니다. 50개 전체를 판독하기 전의 값은 `null`입니다. 코드는 시스템 시작 시 올바른 평균을 보고하기 위해 `null` 참조를 확인합니다. `SensorMeasurement` 형식을 구조체로 변경한 후에는 다른 테스트를 사용해야 합니다. `SensorMeasurement` 형식에는 방 식별자에 대한 `string`이 포함되어 있으므로 해당 테스트를 대신 사용할 수 있습니다.

C#

```
if (recentMeasurements[i].Room is not null)
```

다른 세 가지 컴파일러 오류는 모두 방에서 반복적으로 측정하는 방법에 있습니다.

C#

```
public void TakeMeasurements(Func<SensorMeasurement, bool>  
MeasurementHandler)  
{  
    SensorMeasurement? measure = default;  
    do {  
        measure = SensorMeasurement.TakeMeasurement(Name, Intruders);  
        Average.AddMeasurement(measure);  
        Debounce.AddMeasurement(measure);  
    } while (MeasurementHandler(measure));  
}
```

시작 메서드에서 `SensorMeasurement`의 지역 변수는 `null` 허용 참조입니다.

C#

```
SensorMeasurement? measure = default;
```

이제 `SensorMeasurement`는 `class` 대신 `struct`이므로 `null` 허용 항목은 `null` 허용 값 형식입니다. 선언을 값 형식으로 변경하여 나머지 컴파일러 오류를 수정할 수 있습니다.

C#

```
SensorMeasurement measure = default;
```

이제 컴파일러 오류가 해결되었으므로 코드를 검사하여 의미 체계가 변경되지 않았는지 확인해야 합니다. `struct` 형식은 값으로 전달되므로 메서드 매개 변수에 대한 수정 사항

은 메서드가 반환된 후에 표시되지 않습니다.

## ① 중요

형식을 `class`에서 `struct`로 변경하면 프로그램의 의미 체계가 변경될 수 있습니다. `class` 형식이 메서드에 전달되면 메서드에서 발생한 모든 변형이 인수에 적용됩니다. `struct` 형식이 메서드에 전달되고 메서드에서 발생한 변형이 인수의 복사본에 만들어지는 경우. 즉, 설계상 인수를 수정하는 모든 메서드는 `class`에서 `struct`로 변경한 모든 인수 형식에 대해 `ref` 한정자를 사용하도록 업데이트되어야 함을 의미 합니다.

`SensorMeasurement` 형식에는 상태를 변경하는 메서드가 포함되어 있지 않으므로 이 샘플에서는 문제가 되지 않습니다. `SensorMeasurement` 구조체에 `readonly` 한정자를 추가하여 이를 증명할 수 있습니다.

C#

```
public readonly struct SensorMeasurement
```

컴파일러는 `SensorMeasurement` 구조체의 `readonly` 특성을 적용합니다. 코드 검사에서 상태를 수정하는 일부 메서드가 누락된 경우 컴파일러가 이를 알려 줍니다. 앱은 여전히 오류 없이 빌드되므로 이 형식은 `readonly`입니다. 형식을 `class`에서 `struct`로 변경할 때 `readonly` 한정자를 추가하면 `struct`의 상태를 수정하는 멤버를 찾는 데 도움이 될 수 있습니다.

## 복사본 만들기 방지

앱에서 불필요한 할당을 많이 제거했습니다. `SensorMeasurement` 형식은 테이블 어디에도 표시되지 않습니다.

이제 매개 변수나 반환 값으로 사용될 때마다 `SensorMeasurement` 구조를 복사하는 추가 작업을 수행하고 있습니다. `SensorMeasurement` 구조체에는 4개의 `double`, 즉 `DateTime`과 `string`이 포함되어 있습니다. 해당 구조는 참조보다 측정 가능하게 더 큽니다. `SensorMeasurement` 형식이 사용되는 위치에 `ref` 또는 `in` 한정자를 추가해 보겠습니다.

다음 단계는 측정값을 반환하거나 측정값을 인수로 사용하고 가능한 경우 참조를 사용하는 메서드를 찾는 것입니다. `SensorMeasurement` 구조체에서 시작합니다. 정적 `TakeMeasurement` 메서드는 새 `SensorMeasurement`를 만들고 반환합니다.

C#

```

public static SensorMeasurement TakeMeasurement(string room, int intruders)
{
    return new SensorMeasurement
    {
        CO2 = (CO2Concentration + intruders * 10) + (20 *
generator.NextDouble() - 10.0),
        O2 = (O2Concentration - intruders * 0.01) + (0.005 *
generator.NextDouble() - 0.0025),
        Temperature = (TemperatureSetting + intruders * 0.05) + (0.5 *
generator.NextDouble() - 0.25),
        Humidity = (HumiditySetting + intruders * 0.005) + (0.20 *
generator.NextDouble() - 0.10),
        Room = room,
        TimeRecorded = DateTime.Now
    };
}

```

이를 그대로 두고 값으로 반환할 예정입니다. `ref`로 반환하려고 하면 컴파일러 오류가 발생합니다. 메서드에서 로컬로 만들어진 새 구조에 `ref`를 반환할 수 없습니다. 변경이 불가능한 구조체의 디자인은 생성 시 측정 값만 설정할 수 있음을 의미합니다. 이 메서드는 새로운 측정 구조체를 만들어야 합니다.

`DebounceMeasurement.AddMeasurement`를 다시 살펴보겠습니다. `measurement` 매개 변수에 `in` 한정자를 추가해야 합니다.

C#

```

public void AddMeasurement(in SensorMeasurement datum)
{
    int index = totalMeasurements % debounceSize;
    recentMeasurements[index] = datum;
    totalMeasurements++;
    double sumCO2 = 0;
    double sumO2 = 0;
    double sumTemp = 0;
    double sumHumidity = 0;
    for (int i = 0; i < debounceSize; i++)
    {
        if (recentMeasurements[i].Room is not null)
        {
            sumCO2 += recentMeasurements[i].CO2;
            sumO2+= recentMeasurements[i].O2;
            sumTemp+= recentMeasurements[i].Temperature;
            sumHumidity += recentMeasurements[i].Humidity;
        }
    }
    O2 = sumO2 / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    CO2 = sumCO2 / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
}

```

```

        Temperature = sumTemp / ((totalMeasurements > debounceSize) ?
debounceSize : totalMeasurements);
        Humidity = sumHumidity / ((totalMeasurements > debounceSize) ?
debounceSize : totalMeasurements);
    }

```

그러면 하나의 복사 작업이 절약됩니다. `in` 매개 변수는 호출자가 이미 만든 복사본에 대한 참조입니다. `Room` 형식의 `TakeMeasurement` 메서드를 사용하여 복사본을 저장할 수도 있습니다. 이 메서드는 `ref`로 인수를 전달할 때 컴파일러가 어떻게 안전을 제공하는지 보여 줍니다. `Room` 형식의 초기 `TakeMeasurement` 메서드는 `Func<SensorMeasurement, bool>` 인수를 사용합니다. 해당 선언에 `in` 또는 `ref` 한정자를 추가하려고 하면 컴파일러가 오류를 보고합니다. 람다 식에 `ref` 인수를 전달할 수 없습니다. 컴파일러는 호출된 식이 참조를 복사하지 않는다고 보장할 수 없습니다. 람다 식이 참조를 캡처하는 경우 참조는 참조하는 값보다 수명이 길어질 수 있습니다. `ref safe` 컨택스트 외부에서 액세스하면 메모리가 손상될 수 있습니다. `ref safe` 규칙에서는 이를 허용하지 않습니다. `ref safety` 가능 개요에서 자세히 알아볼 수 있습니다.

## 의미 체계 보존

형식이 실행 부하 과다 경로에 만들어지지 않기 때문에 최종 변경 내용 집합은 이 애플리케이션의 성능에 큰 영향을 미치지 않습니다. 이러한 변경 내용은 성능 튜닝에 사용할 몇 가지 다른 기술을 보여 줍니다. 초기 `Room` 클래스를 살펴보겠습니다.

C#

```

public class Room
{
    public AverageMeasurement Average { get; } = new ();
    public DebounceMeasurement Debounce { get; } = new ();
    public string Name { get; }

    public IntruderRisk RiskStatus
    {
        get
        {
            var CO2Variance = (Debounce.CO2 - Average.CO2) > 10.0 / 4;
            var O2Variance = (Average.O2 - Debounce.O2) > 0.005 / 4.0;
            var TempVariance = (Debounce.Temperature - Average.Temperature)
> 0.05 / 4.0;
            var HumidityVariance = (Debounce.Humidity - Average.Humidity) >
0.20 / 4;

            IntruderRisk risk = IntruderRisk.None;
            if (CO2Variance) { risk++; }
            if (O2Variance) { risk++; }
            if (TempVariance) { risk++; }
            if (HumidityVariance) { risk++; }

            return risk;
        }
    }
}

```

```

        }

    public int Intruders { get; set; }

    public Room(string name)
    {
        Name = name;
    }

    public void TakeMeasurements(Func<SensorMeasurement, bool>
MeasurementHandler)
    {
        SensorMeasurement? measure = default;
        do {
            measure = SensorMeasurement.TakeMeasurement(Name, Intruders);
            Average.AddMeasurement(measure);
            Debounce.AddMeasurement(measure);
        } while (MeasurementHandler(measure));
    }
}

```

이 형식에는 여러 속성이 포함되어 있습니다. 일부는 `class` 형식입니다. `Room` 개체 만들기에는 여러 할당이 포함됩니다. 하나는 `Room` 자체에 대한 것이고 다른 하나는 포함된 `class` 형식의 각 멤버에 대한 것입니다. 이러한 속성 중 두 개(`DebounceMeasurement` 및 `AverageMeasurement` 형식)를 `class` 형식에서 `struct` 형식으로 변환할 수 있습니다. 두 가지 형식을 모두 사용하여 해당 변환을 살펴보겠습니다.

`DebounceMeasurement` 형식을 `class`에서 `struct`로 변경합니다. 이로 인해 컴파일러 오류 CS8983: A 'struct' with field initializers must include an explicitly declared constructor가 발생합니다. 매개 변수가 없는 빈 생성자를 추가하여 이 문제를 해결할 수 있습니다.

C#

```
public DebounceMeasurement() { }
```

구조체에 대한 언어 참조 문서에서 이 요구 사항에 대해 자세히 알아볼 수 있습니다.

`Object.ToString()` 재정의는 구조체 값을 수정하지 않습니다. 해당 메서드 선언에 `readonly` 한정자를 추가할 수 있습니다. `DebounceMeasurement` 형식은 변경 가능하므로 수정 사항이 삭제되는 복사본에 영향을 미치지 않도록 주의해야 합니다. `AddMeasurement` 메서드는 개체의 상태를 수정합니다. `TakeMeasurements` 메서드의 `Room` 클래스에서 호출됩니다. 메서드를 호출한 후에도 이러한 변경 내용을 유지하려고 합니다. `Room.Debounce` 속

성을 변경하여 `DebounceMeasurement` 형식의 단일 인스턴스에 대한 참조를 반환할 수 있습니다.

C#

```
private DebounceMeasurement debounce = new();
public ref readonly DebounceMeasurement Debounce { get { return ref
debounce; } }
```

이전 예에는 몇 가지 변경 내용이 있습니다. 첫째, 속성은 이 방이 소유한 인스턴스에 대한 읽기 전용 참조를 반환하는 읽기 전용 속성입니다. 이제 `Room` 개체가 인스턴스화될 때 초기화되는 선언된 필드에 의해 지원됩니다. 이렇게 변경한 후에는 `AddMeasurement` 메서드 구현을 업데이트하게 됩니다. 읽기 전용 속성 `Debounce`가 아닌 프라이빗 지원 필드 `debounce`를 사용합니다. 이렇게 하면 초기화 중에 만들어진 단일 인스턴스에 변경 내용이 적용됩니다.

동일한 기술이 `Average` 속성에도 적용됩니다. 먼저, `AverageMeasurement` 형식을 `class`에서 `struct`로 수정하고 `ToString` 메서드에 `readonly` 한정자를 추가합니다.

C#

```
namespace IntruderAlert;

public struct AverageMeasurement
{
    private double sumCO2 = 0;
    private double sumO2 = 0;
    private double sumTemperature = 0;
    private double sumHumidity = 0;
    private int totalMeasurements = 0;

    public AverageMeasurement() { }

    public readonly double CO2 => sumCO2 / totalMeasurements;
    public readonly double O2 => sumO2 / totalMeasurements;
    public readonly double Temperature => sumTemperature /
totalMeasurements;
    public readonly double Humidity => sumHumidity / totalMeasurements;

    public void AddMeasurement(in SensorMeasurement datum)
    {
        totalMeasurements++;
        sumCO2 += datum.CO2;
        sumO2 += datum.O2;
        sumTemperature += datum.Temperature;
        sumHumidity+= datum.Humidity;
    }

    public readonly override string ToString() => $"""
```

```

        Average measurements:
        Temp:      {Temperature:F3}
        Humidity:   {Humidity:P3}
        Oxygen:     {O2:P3}
        CO2 (ppm): {CO2:F3}
    """";
}

```

그런 다음 `Debounce` 속성에 사용한 것과 동일한 기술에 따라 `Room` 클래스를 수정합니다. `Average` 속성은 평균 측정을 위해 프라이빗 필드에 `readonly ref`를 반환합니다. `AddMeasurement` 메서드는 내부 필드를 수정합니다.

C#

```

private AverageMeasurement average = new();
public ref readonly AverageMeasurement Average { get { return ref average; }
} }

```

## boxing 방지

성능을 개선시키기 위한 마지막 변경 내용이 하나 있습니다. 주요 프로그램은 위험 평가를 포함하여 방에 대한 통계를 인쇄합니다.

C#

```

Console.WriteLine($"Current intruders: {room.Intruders}");
Console.WriteLine($"Calculated intruder risk: {room.RiskStatus}");

```

생성된 `ToString`에 대한 호출은 열거형 값을 문자에 넣습니다. 예상 위험 값에 따라 문자열 형식을 지정하는 `Room` 클래스에 재정의를 작성하면 이를 방지할 수 있습니다.

C#

```

public override string ToString() =>
    $"Calculated intruder risk: {RiskStatus switch
    {
        IntruderRisk.None => "None",
        IntruderRisk.Low => "Low",
        IntruderRisk.Medium => "Medium",
        IntruderRisk.High => "High",
        IntruderRisk.Extreme => "Extreme",
        _ => "Error!"
    }}, Current intruders: {Intruders.ToString()}";

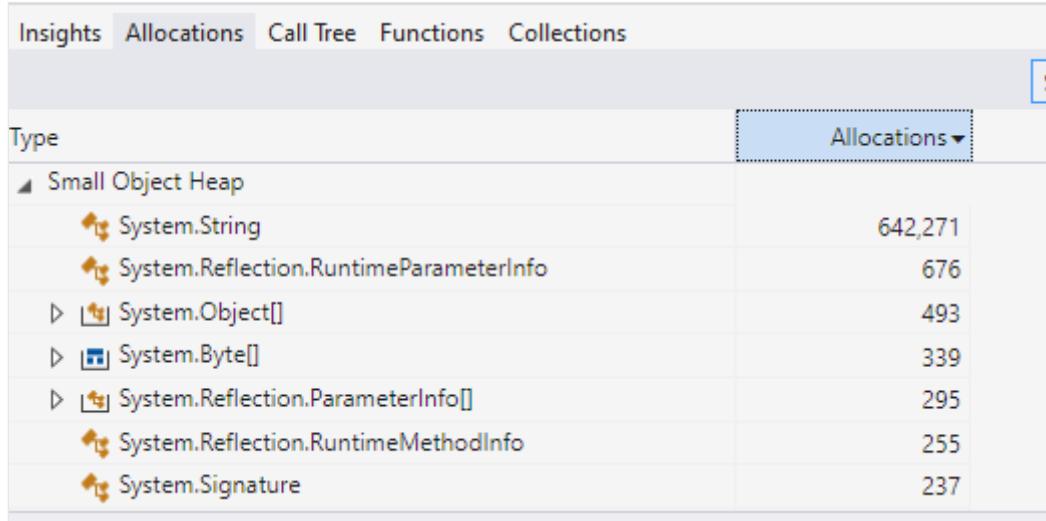
```

그런 다음 기본 프로그램의 코드를 수정하여 이 새 `ToString` 메서드를 호출합니다.

C#

```
Console.WriteLine(room.ToString());
```

프로파일러를 사용하여 앱을 실행하고 업데이트된 할당 테이블을 살펴봅니다.



수많은 할당을 제거하고 앱 성능을 향상시켰습니다.

## 애플리케이션에 ref safety 사용

이러한 기술은 낮은 수준의 성능 튜닝입니다. 실행 부하 과다 경로에 적용하고 변경 전후의 영향을 측정하면 애플리케이션의 성능이 향상될 수 있습니다. 대부분의 경우 따르는 주기는 다음과 같습니다.

- **할당 측정:** 가장 많이 할당되는 형식과 힙 할당을 줄일 수 있는 시기를 결정합니다.
- **클래스를 구조체로 변환:** 형식을 `class`에서 `struct`로 변환할 수 있는 경우가 많습니다. 앱은 힙을 할당하는 대신 스택 공간을 사용합니다.
- **의미 체계 보존:** `class`를 `struct`로 변환하면 매개 변수 및 반환 값의 의미 체계에 영향을 미칠 수 있습니다. 매개 변수를 수정하는 모든 메서드는 이제 해당 매개 변수를 `ref` 한정자로 표시해야 합니다. 이를 통해 올바른 개체에 대한 수정이 이루어집니다. 마찬가지로 호출자가 속성이나 메서드 반환 값을 수정해야 하는 경우 해당 반환은 `ref` 한정자로 표시되어야 합니다.
- **복사 방지:** 큰 구조체를 매개 변수로 전달하는 경우 `in` 한정자로 매개 변수를 표시할 수 있습니다. 더 적은 바이트로 참조를 전달할 수 있으며 메서드가 원래 값을 수정하지 않도록 할 수 있습니다. 또한 `readonly ref`로 값을 반환하여 수정할 수 없는 참조를 반환할 수도 있습니다.

이러한 기술을 사용하면 코드의 실행 부하 과다 경로 성능을 개선시킬 수 있습니다.

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# .NET Compiler Platform SDK

아티클 • 2024. 04. 11.

컴파일러는 애플리케이션 코드의 구문 및 의미 체계의 유효성을 검사할 때 애플리케이션 코드의 세부 모델을 빌드합니다. 컴파일러는 이 모델을 사용하여 소스 코드에서 실행 가능 출력을 빌드합니다. .NET Compiler Platform SDK는 이 모델에 대한 액세스를 제공합니다. 우리는 점점 더 IntelliSense, 리팩터링, 지능형 이름 바꾸기, “모든 참조 찾기” 및 “정의로 이동”과 같은 IDE(통합 개발 환경) 기능에 의존하여 생산성을 높입니다. 또한 코드 분석 도구를 사용하여 코드 품질을 개선하고 코드 생성기를 사용하여 애플리케이션 구성에서 도움을 받습니다. 이러한 도구가 더 스마트해짐에 따라 컴파일러가 애플리케이션 코드를 처리할 때 컴파일러만이 만드는 모델의 점점 더 많은 부분에 이러한 도구가 액세스해야 합니다. 이것이 바로 Roslyn API의 핵심 임무입니다. 불투명 상자를 열고 도구 및 최종 사용자가 컴파일러가 코드에 대해 가진 다양한 정보를 공유할 수 있도록 하는 것 말입니다. 컴파일러는 불투명한 소스 코드 입력 변환기 및 개체 코드 출력 변환기 대신 Roslyn을 통해 플랫폼이 됩니다. 즉, 도구 및 애플리케이션에서 코드 관련 작업에 사용할 수 있는 API입니다.

## .NET Compiler Platform SDK 개념

.NET Compiler Platform SDK는 코드 중심 도구 및 애플리케이션을 만들기 위한 진입에 대한 장벽을 크게 낮춰줍니다. 메타 프로그래밍, 코드 생성 및 변환, C# 및 Visual Basic 언어의 대화형 사용, 도메인 특정 언어에 C# 및 Visual Basic 포함과 같은 영역에서 다양한 혁신 기회를 창출합니다.

.NET Compiler Platform SDK를 사용하면 코딩 실수를 찾아 수정하는 **분석기 및 코드 수정을 빌드할 수 있습니다.** **분석기는** 구문(코드 구조) 및 의미 체계를 이해하여 수정해야 하는 사례를 검색합니다. **코드 수정은** 분석기 또는 컴파일러 진단에서 발견된 코딩 실수를 해결하기 위해 하나 이상의 제안된 수정 사항을 제공합니다. 일반적으로 분석기 및 연관된 코드 수정 사항은 단일 프로젝트에서 함께 패키지됩니다.

분석기 및 코드 수정 사항은 정적 분석을 사용하여 코드를 이해하며, 코드를 실행하거나 다른 테스트 이점을 제공하지 않습니다. 하지만 종종 버그, 유지 관리할 수 없는 코드, 표준 지침 위반으로 이어질 수 있는 사례를 짚어줄 수 있습니다.

분석기 및 코드 수정 외에도 .NET Compiler Platform SDK를 사용하면 **코드 리팩터링을 빌드할 수 있습니다.** 또한 C# 또는 Visual Basic 코드베이스를 검사하고 이해할 수 있게 해주는 단일 API 집합도 제공합니다. 이 단일 코드베이스를 사용할 수 있으므로 .NET Compiler Platform SDK에서 제공하는 구문 및 의미 체계 분석 API를 활용하여 분석기 및 코드 수정 사항을 더 쉽게 작성할 수 있습니다. 컴파일러가 수행한 분석을 복제하는 대규모 작업에

서 벗어난 사용자는 프로젝트 또는 라이브러리에 대한 일반적인 코딩 오류를 찾아 수정하는 더 중심이 되는 작업에 집중할 수 있습니다.

작은 이점 한 가지는 사용자가 직접 프로젝트의 코드를 이해하기 위해 자체 코드베이스를 작성하는 경우보다 Visual Studio에 로드되었을 때 분석기 및 코드 수정 사항이 메모리를 훨씬 더 적게 사용하고 규모가 작다는 것입니다. 컴파일러 및 Visual Studio에서 사용되는 것과 같은 클래스를 활용하여 자체 정적 분석 도구를 만들 수 있습니다. 즉, 팀은 IDE 성능에 대한 눈에 띄는 영향 없이 분석기 및 코드 수정 사항을 사용할 수 있습니다.

분석기 및 코드 수정 사항을 작성하는 세 가지 주요 시나리오가 있습니다.

1. [팀 코딩 표준 적용](#)
2. [라이브러리 패키지로 지침 제공](#)
3. [일반 지침 제공](#)

## 팀 코딩 표준 적용

많은 팀에는 다른 팀 구성원과의 코드 검토를 통해 적용되는 코딩 표준이 있습니다. 분석기 및 코드 수정 사항은 이 프로세스를 훨씬 더 효율적으로 만들 수 있습니다. 코드 검토는 개발자가 다른 팀 구성원과 자신의 작업을 공유할 때 발생합니다. 개발자는 의견을 듣기 전에 새로운 기능을 완성하는 데 필요한 모든 시간을 투자했을 것입니다. 개발자가 팀의 습관과 일치하지 않는 습관을 강화하는 동안 몇 주가 흐를 수도 있습니다.

개발자가 코드를 작성할 때 분석기가 실행됩니다. 개발자는 즉시 지침을 따르도록 권장하는 즉각적인 피드백을 받습니다. 개발자는 프로토타입 생성을 시작하는 즉시 규격 코드를 작성하는 습관이 붙게 됩니다. 기능이 사람에 의한 검토를 받을 준비가 되면 모든 표준 지침이 적용된 상태가 되어 있습니다.

팀은 팀 코딩 습관을 위반하는 가장 일반적인 습관을 찾는 분석기 및 코드 수정 사항을 빌드할 수 있습니다. 이러한 분석기 및 코드 수정 사항을 각 개발자의 컴퓨터에 설치하여 표준을 적용할 수 있습니다.

### 💡 팁

사용자 고유의 분석기를 빌드하기 전에 기본 제공되는 분석기를 확인합니다. 자세한 내용은 [코드 스타일 규칙](#)을 참조하세요.

## 라이브러리 패키지로 지침 제공

NuGet에는 .NET 개발자가 사용할 수 있는 다양한 라이브러리가 있습니다. 이러한 라이브러리 중 일부는 Microsoft에서 제공한 것이고, 또 다른 일부는 타사에서 제공한 것이며, 나

머지는 커뮤니티 회원 및 지원자가 제공한 것입니다. 개발자가 이러한 라이브러리로 성공할 수 있는 경우 해당 라이브러리는 더 많이 채택되고 더 많은 검토를 받게 됩니다.

설명서를 제공하는 것 외에 라이브러리의 일반적인 오용을 찾아 수정하는 분석기 및 코드 수정 사항을 제공할 수 있습니다. 이러한 즉각적인 수정 사항은 개발자가 더 빠르게 성공하도록 도와줍니다.

NuGet의 라이브러리를 사용하여 분석기 및 코드 수정 사항을 패키지할 수 있습니다. 해당 시나리오에서 NuGet 패키지를 설치하는 모든 개발자는 분석기 패키지도 설치합니다. 라이브러리를 사용하는 모든 개발자는 실수 및 제안된 수정 사항에 대한 즉각적인 피드백의 형태로 팀으로부터 즉시 지침을 받게 됩니다.

## 일반 지침 제공

.NET 개발자 커뮤니티는 경험을 통해 잘 작동하는 패턴과 가장 피해야 할 패턴을 간파해왔습니다. 여러 커뮤니티 회원은 이러한 권장 패턴을 적용하는 분석기를 만들었습니다. 자세히 알아보다 보면 항상 새로운 아이디어를 위한 여지가 있음을 알게 됩니다.

이러한 분석기를 [Visual Studio Marketplace](#)에 업로드하고 Visual Studio를 사용하는 개발자가 다운로드할 수 있습니다. 언어 및 플랫폼을 처음 사용하는 초보자는 일반적으로 인정된 습관을 신속하게 배우고 .NET 여정에서 조기에 생산성을 높일 수 있습니다. 이러한 습관이 더 널리 사용됨에 따라 커뮤니티에서는 이러한 습관을 채택합니다.

## 다음 단계

.NET Compiler Platform SDK에는 코드 생성, 분석 및 리팩터링에 대한 최신 언어 개체 모델이 포함되어 있습니다. 이 섹션에서는 .NET Compiler Platform SDK에 대한 개념적 개요를 제공합니다. 자세한 내용은 빠른 시작, 샘플 및 자습서 섹션에서 확인할 수 있습니다.

다음 다섯 가지 항목에서 .NET Compiler Platform SDK의 개념에 대해 자세히 알아볼 수 있습니다.

- 구문 시각화 도우미를 사용하여 코드 탐색
- 컴파일러 API 모델 이해
- 구문 작업
- 의미 체계 작업
- 작업 영역 작업

시작하려면 .NET Compiler Platform SDK를 설치해야 합니다.

## 설치 지침 - Visual Studio 설치 관리자

Visual Studio 설치 관리자에서 .NET Compiler Platform SDK를 찾는 두 가지 방법이 있습니다.

## Visual Studio 설치 관리자를 사용한 설치 - 워크로드 보기

.NET Compiler Platform SDK는 Visual Studio 확장 개발 워크로드의 일부로 자동으로 선택되지 않습니다. 선택적 구성 요소로 선택해야 합니다.

1. Visual Studio 설치 관리자를 실행합니다.
2. **수정을 선택합니다.**
3. Visual Studio 확장 개발 워크로드를 확인합니다.
4. 요약 트리에서 Visual Studio 확장 개발 노드를 엽니다.
5. .NET Compiler Platform SDK에 대한 확인란을 선택합니다. 선택적 구성 요소 아래에서 마지막에 찾을 수 있습니다.

필요에 따라, 시각화 도우미에서 DGML 편집기에 그래프도 표시할 수 있습니다.

1. 요약 트리에서 **개별 구성 요소** 노드를 엽니다.
2. DGML 편집기 확인란을 선택합니다.

## Visual Studio 설치 관리자를 사용한 설치 - 개별 구성 요소 탭

1. Visual Studio 설치 관리자를 실행합니다.
2. **수정을 선택합니다.**
3. **개별 구성 요소** 탭을 선택합니다.
4. .NET Compiler Platform SDK에 대한 확인란을 선택합니다. 컴파일러, 빌드 도구 및 런타임 섹션의 위쪽에서 찾을 수 있습니다.

필요에 따라, 시각화 도우미에서 DGML 편집기에 그래프도 표시할 수 있습니다.

1. DGML 편집기 확인란을 선택합니다. 코드 도구 섹션에서 찾을 수 있습니다.

---

## 피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공 ↗](#)

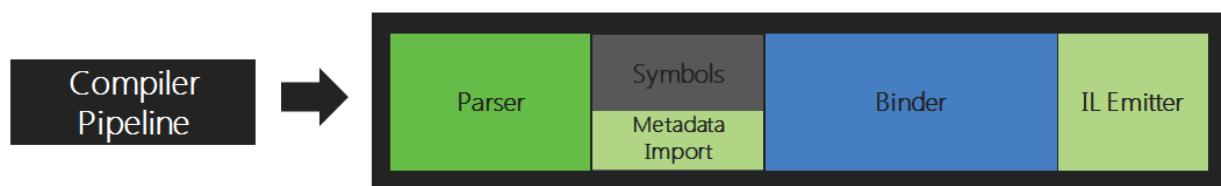
# .NET Compiler Platform SDK 모델 이해

아티클 • 2023. 05. 10.

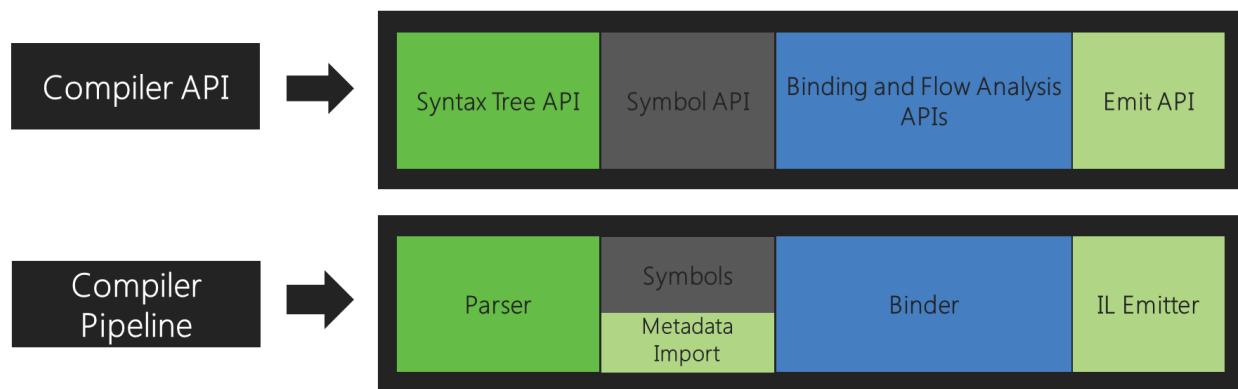
컴파일러는 종종 사람이 코드를 읽고 이해하는 방식과 다른 구조적 규칙을 따라 작성하는 코드를 처리합니다. 컴파일러에서 사용되는 모델에 대한 기본적인 이해는 Roslyn 기반 도구를 빌드할 때 사용하는 API를 이해하는 데 반드시 필요합니다.

## 컴파일러 파이프라인 기능 영역

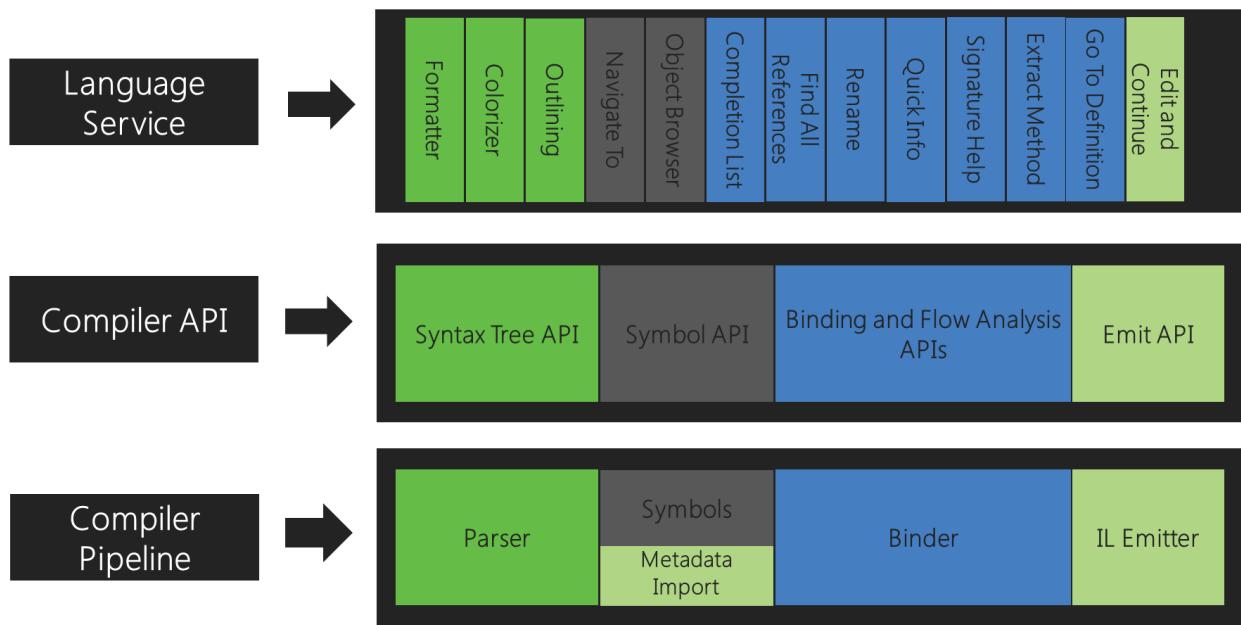
.NET Compiler Platform SDK는 기존의 컴파일러 파이프라인을 미러링하는 API 계층을 제공하여 소비자로서 사용자에게 C# 및 Visual Basic 컴파일러의 코드 분석을 노출합니다.



이 파이프라인의 각 단계는 별도 구성 요소입니다. 첫째로, 구문 분석 단계는 원본 텍스트를 언어 문법 뒤에 오는 구문으로 토큰화하고 구문 분석합니다. 둘째로, 선언 단계는 원본 및 가져온 메타데이터를 분석하여 명명된 기호를 형성합니다. 다음으로 바인딩 단계는 코드의 식별자를 기호와 일치시킵니다. 마지막으로 내보내기 단계는 컴파일러로 생성된 모든 정보와 함께 어셈블리를 내보냅니다.



이러한 각 단계에 해당하여 .NET Compiler Platform SDK는 해당 단계에서 정보에 액세스 할 수 있는 개체 모델을 노출합니다. 구문 분석 단계는 구문 트리를 노출하고, 선언 단계는 계층적 기호 테이블을 노출하고, 바인딩 단계는 컴파일러의 의미 체계 분석 결과를 노출하고, 내보내기 단계는 IL 바이트 코드를 생성하는 API입니다.



각 컴파일러는 단일 엔드투엔드 전체로 이러한 구성 요소를 하나로 결합합니다.

이러한 API는 Visual Studio에서 사용하는 API와 동일합니다. 예를 들어 코드 개요 및 서식 지정 기능은 구문 트리를 사용하고, **개체 브라우저** 및 탐색 기능은 기호 테이블을 사용하며, 리팩터링 및 **정의로 이동**은 의미 체계 모델을 사용하고, **편집하며 계속하기**는 내보내기 API를 포함하여 이러한 모든 것을 사용합니다.

## API 계층

.NET 컴파일러 SDK는 컴파일러 API, 진단 API, 스크립팅 API, 작업 영역 API 등 여러 개의 계층으로 구성되어 있습니다.

### 컴파일러 API

컴파일러 계층은 구문 및 의미 체계 모두의 컴파일러 파이프라인의 각 단계에서 노출되는 정보에 해당하는 개체 모델을 포함합니다. 컴파일러 계층은 어셈블리 참조, 컴파일러 옵션 및 소스 코드 파일을 포함하는 컴파일러 단일 호출의 변경할 수 없는 스냅샷도 포함합니다. C# 언어 및 Visual Basic 언어를 나타내는 두 개의 고유한 API가 있습니다. 두 개의 API는 모양이 유사하지만 개별 언어에 대한 충실도가 높게 조정되었습니다. 이 계층에는 Visual Studio 구성 요소에 대한 종속성이 없습니다.

### 진단 API

해당 분석의 일환으로 컴파일러는 구문, 의미 체계, 한정된 할당 오류에서 다양한 경고 및 정보 진단에 이르는 모든 항목을 포함하는 진단 정보 세트를 생성할 수 있습니다. 컴파일러 API 계층은 사용자 정의 분석기를 컴파일 프로세스에 연결할 수 있도록 하는 확장 가능한 API를 통해 진단을 노출합니다. StyleCop 같은 도구에서 생성된 진단 등의 사용자 정

의 진단을 컴파일러 정의 진단과 함께 생성할 수 있습니다. 이러한 방식으로 진단을 생성하면 정책을 기반으로 한 빌드 중지 및 편집기에서 라이브 물결선 표시 및 코드 수정 사항 제안과 같은 환경 진단을 사용하는 MSBuild 및 Visual Studio와 같은 도구와 자연스럽게 통합되는 이점이 있습니다.

## 스크립팅 API

호스팅 및 스크립팅 API는 컴파일러 계층의 일부입니다. 코드 조각을 실행하고 런타임 실행 컨텍스트를 누적하는 데 사용할 수 있습니다. C# 대화형 REPL(읽기 평가 인쇄 루프)은 이러한 API를 사용합니다. REPL을 통해 작성함에 따라 대화형으로 코드를 실행하여 스크립팅 언어로 C#을 사용할 수 있습니다.

## 작업 영역 API

작업 영역 계층은 코드 분석을 수행하고 전체 솔루션을 통해 리팩터링하는 시작 지점인 작업 영역 API를 포함합니다. 파일을 구문 분석하고, 옵션을 구성하거나 프로젝트 간 종속성을 관리하지 않고도 컴파일러 계층 개체 모델에 대한 직접 액세스를 제공하여 솔루션의 프로젝트에 대한 모든 정보를 단일 개체 모델로 구성하는 데 도움을 줍니다.

또한 작업 영역 계층은 코드 분석을 구현하고 Visual Studio IDE와 같은 호스트 환경 내에서 작동하는 도구를 리팩터링할 때 사용되는 API 집합을 나타냅니다. 모든 참조 찾기, 서식 지정 및 코드 생성 API를 예로 들 수 있습니다.

이 계층에는 Visual Studio 구성 요소에 대한 종속성이 없습니다.

# 구문 작업

아티클 • 2023. 04. 08.

'구문 트리'는 컴파일러 API에서 노출되는 변경이 불가능한 기본 데이터 구조입니다. 이러한 트리는 소스 코드의 어휘 및 구문 구조를 나타냅니다. 두 가지 중요한 용도를 제공합니다.

- 사용자의 프로젝트에서 소스 코드의 구문 구조를 보고 처리하도록 IDE, 추가 기능, 코드 분석 도구 및 리팩터링과 같은 도구 허용
- 직접 텍스트 편집을 사용하지 않고 자연스러운 방식으로 소스 코드를 만들고, 수정하고, 다시 정렬하도록 리팩터링 및 IDE와 같은 도구 활성화 트리를 만들고 조작하여 도구는 쉽게 소스 코드를 만들고 다시 정렬할 수 있습니다.

## 구문 트리

구문 트리는 컴파일, 코드 분석, 바인딩, 리팩터링, IDE 기능 및 코드 생성에 사용되는 기본 구조입니다. 소스 코드의 어떠한 부분도 먼저 식별되고 잘 알려진 많은 구조적 언어 요소 중 하나로 분류되지 않고 인식되지 않습니다.

구문 트리에는 세 가지 주요 특성이 있습니다.

- 모든 소스 정보를 최고의 충실도로 유지합니다. 최고 충실도란 구문 트리에 공백, 설명 및 전처리기 지시문을 포함하여 소스 텍스트에서 발견되는 모든 정보, 모든 문법 구문, 모든 어휘 토큰 및 사이의 모든 항목이 포함되는 것을 의미합니다. 예를 들어 원본에서 언급된 각 리터럴은 입력된 대로 정확하게 표시됩니다. 또한 구문 트리는 프로그램이 불완전하거나 형식이 잘못된 경우 건너뛰거나 누락된 토큰을 표시하여 소스 코드의 오류를 캡처합니다.
- 구문 분석된 정확한 텍스트를 생성할 수 있습니다. 구문 노드에서 해당 노드를 기반으로 하는 하위 트리의 텍스트 표현을 가져올 수 있습니다. 이 기능은 구문 트리가 원본 텍스트를 생성하고 편집하는 방법으로 사용될 수 있다는 것을 의미합니다. 암시적으로 해당 텍스트를 만든 트리를 만들고 변경 내용에서 새 트리를 기준 트리로 만들어 텍스트를 효과적으로 편집했습니다.
- 변경할 수 없으며 스레드로부터 안전합니다. 트리를 얻으면 이는 코드의 현재 상태의 스냅샷이며 변경되지 않습니다. 따라서 여러 사용자는 잠금 또는 중복 없이 서로 다른 스레드에서 동시에 동일한 구문 트리와 상호 작용할 수 있습니다. 트리를 변경할 수 없으며 트리에 직접 수정을 만들 수 없으므로 팩터리 메서드는 트리의 추가 스냅샷을 만들어 구문 트리를 만들고 수정하도록 돕습니다. 트리는 기본 노드를 다시 사용하는 방법에서 효율적이므로 빠르게 작은 추가 메모리로 새 버전을 다시 빌드 할 수 있습니다.

구문 트리는 비터미널 구조 요소가 다른 요소를 부모로 삼는 문자 그대로 트리 데이터 구조입니다. 각 구문 트리는 노드, 토큰 및 기타 정보로 구성되어 있습니다.

## 구문 노드

구문 노드는 구문 트리의 기본 요소 중 하나입니다. 이러한 노드는 선언, 명령문, 절 및 식과 같은 구문 구조를 나타냅니다. 구문 노드의 각 범주는 [Microsoft.CodeAnalysis.SyntaxNode](#)에서 파생되는 별도 클래스로 나타냅니다. 노드 클래스의 집합은 확장할 수 없습니다.

모든 구문 노드는 구문 트리에서 비터미널 노드입니다. 즉, 항상 다른 노드 및 토큰을 자식으로 갖습니다. 다른 노드의 자식으로 각 노드는 [SyntaxNode.Parent](#) 속성을 통해 액세스 할 수 있는 부모 노드를 갖습니다. 노드 및 트리를 변경할 수 없기 때문에 부모 노드는 변경되지 않습니다. 트리의 루트는 null 부모를 갖습니다.

각 노드에는 원본 텍스트에서의 위치에 따라 순서대로 자식 노드의 목록을 반환하는 [SyntaxNode.ChildNodes\(\)](#) 메서드가 있습니다. 이 목록은 토큰을 포함하지 않습니다. 각 노드에는 해당 노드의 하위 트리에 있는 모든 노드, 토큰 또는 기타 정보 목록을 나타내는 [DescendantNodes](#), [DescendantTokens](#), [DescendantTrivia](#) 등의 하위 항목을 검사하는 메서드도 있습니다.

또한 각 구문 노드 하위 클래스는 강력한 형식의 속성을 통해 동일한 모든 자식을 노출합니다. 예를 들어 [BinaryExpressionSyntax](#) 노드 클래스에는 이진 연산자, [Left](#), [OperatorToken](#) 및 [Right](#)에 해당하는 3개의 추가 속성이 있습니다. [Left](#) 및 [Right](#)의 형식은 [ExpressionSyntax](#)이며 [OperatorToken](#)의 형식은 [SyntaxToken](#)입니다.

일부 구문 노드에는 선택적 자식이 있습니다. 예를 들어 [IfStatementSyntax](#)에는 선택적 [ElseClauseSyntax](#)가 있습니다. 자식이 없는 경우 속성은 null을 반환합니다.

## 구문 토큰

구문 토큰은 코드의 가장 작은 구문 조각을 나타내는 언어 문법의 터미널입니다. 다른 노드 또는 토큰의 부모가 아닙니다. 구문 토큰은 키워드, 식별자, 리터럴 및 문장 부호로 구성됩니다.

효율성 향상을 위해 [SyntaxToken](#) 형식은 CLR 값 형식입니다. 따라서 구문 노드와 달리 표시되는 토큰의 종류에 따라 의미가 있는 속성을 조합하여 모든 종류의 토큰에 대해 하나의 구문만 있습니다.

예를 들어 정수 리터럴 토큰은 숫자 값을 나타냅니다. 토큰이 포괄하는 원시 원본 텍스트 이외에 리터럴 토큰에는 디코딩된 정확한 정수 값을 알려 주는 [Value](#) 속성이 있습니다. 많은 기본 형식 중 하나일 수 있으므로 이 속성은 [Object](#)로 입력됩니다.

`ValueText` 속성은 `Value` 속성과 동일한 정보를 알려 주지만 이 속성은 항상 `String`으로 입력됩니다. C# 원본 텍스트에서 식별자는 유니코드 이스케이프 문자를 포함할 수 있지만 이스케이프 시퀀스 자체의 구문은 식별자 이름의 일부로 간주되지 않습니다. 따라서 토큰에 포함되는 원시 텍스트는 이스케이프 시퀀스를 포함하지만 `ValueText` 속성은 포함하지 않습니다. 대신 이스케이프로 식별되는 유니코드 문자를 포함합니다. 예를 들어 원본 텍스트가 `\u03c0`으로 작성된 식별자를 포함하는 경우 이 토큰에 대한 `ValueText` 속성은 `π`를 반환합니다.

## 구문 기타 정보

구문 기타 정보는 공백, 설명 및 전처리기 지시문과 같은 코드의 일반적인 이해에 크게 중요하지 않은 원본 텍스트의 일부를 나타냅니다. 구문 토큰과 같이 기타 정보는 값 형식입니다. 단일 `Microsoft.CodeAnalysis.SyntaxTrivia` 형식은 모든 종류의 기타 정보를 설명하는 데 사용됩니다.

기타 정보는 일반 언어 구문의 일부가 아니며 두 토큰 사이의 아무 곳에나 나타날 수 있으므로 노드의 자식으로 구문 트리에 포함되지 않습니다. 리팩터링과 같은 기능을 구현하는 경우 및 원본 텍스트로 최고의 충실도를 유지하는 데 중요하므로 구문 트리의 일부로 존재합니다.

토큰의 `SyntaxToken.LeadingTrivia` 또는 `SyntaxToken.TrailingTrivia` 컬렉션을 검사하여 기타 정보에 액세스할 수 있습니다. 원본 텍스트를 구문 분석할 때 기타 정보의 시퀀스는 토큰과 연결됩니다. 일반적으로 토큰은 다음 토큰까지 동일한 줄의 다음에 모든 기타 정보를 소유합니다. 해당 줄 다음의 모든 기타 정보는 다음 토큰으로 연결됩니다. 원본 파일의 첫 번째 토큰은 모든 초기 기타 정보를 가져오고 파일에 있는 기타 정보의 마지막 시퀀스는 파일 끝 토큰으로 고정됩니다. 그렇지 않으면 0의 너비를 갖습니다.

구문 노드 및 토큰과 달리 구문 기타 정보는 부모가 없습니다. 아직 트리의 일부이며 각각은 단일 토큰에 연결되어 있으므로 `SyntaxTrivia.Token` 속성을 사용하여 연결된 토큰에 액세스할 수 있습니다.

## 범위

각 노드, 토큰 또는 기타 정보는 원본 텍스트 내의 해당 위치와 구성된 문자 수를 알고 있습니다. 텍스트 위치는 0부터 시작하는 `char` 인덱스인 32비트 정수로 표현됩니다.

`TextSpan` 개체는 시작 위치 및 문자 수이며 정수로 표현됩니다. `TextSpan`의 길이가 0인 경우 두 문자 사이의 위치를 나타냅니다.

각 노드에는 `Span` 및 `FullSpan`이라는 두 가지 `TextSpan` 속성이 있습니다.

`Span` 속성은 노드의 하위 트리에 있는 첫 번째 토큰의 시작부터 마지막 토큰의 끝에 이르는 텍스트 범위입니다. 이 범위는 모든 선행 또는 후행 기타 정보를 포함하지 않습니다.

`FullSpan` 속성은 노드의 일반 범위와 모든 선행 또는 후행 기타 정보의 범위를 포함하는 텍스트 범위입니다.

예를 들어:

C#

```
if (x > 3)
{
||      // this is bad
|throw new Exception("Not right.");| // better exception? ||
}
```

블록 내의 명령문 노드에는 단일 세로 막대(|)로 표시되는 범위가 있습니다. 여기에는 `throw new Exception("Not right.");` 문자가 포함됩니다. 전체 범위는 이중 세로 막대(||)로 표시됩니다. 여기에는 범위와 동일한 문자 및 선행 및 후행 기타 정보와 연결된 문자가 포함됩니다.

## 종류

각 노드, 토큰 또는 기타 정보에는 표시되는 정확한 구문 요소를 식별하는 `System.Int32` 형식의 `SyntaxNode.RawKind` 속성이 있습니다. 이 값은 언어별 열거형으로 캐스팅될 수 있습니다. 각 언어(C# 또는 Visual Basic)에는 문법의 가능한 모든 노드, 토큰 및 퀴즈 요소를 나열하는 단일 `SyntaxKind` 열거형(`Microsoft.CodeAnalysis.CSharp.SyntaxKind` 및 `Microsoft.CodeAnalysis.VisualBasic.SyntaxKind`)이 있습니다. `CSharpExtensions.Kind` 또는 `VisualBasicExtensions.Kind` 확장 메서드에 액세스하여 이 변환을 자동으로 수행할 수 있습니다.

`RawKind` 속성은 동일한 노드 클래스를 공유하는 구문 노드 형식의 쉬운 명확성을 허용합니다. 토큰 및 기타 정보의 경우 이 속성은 요소의 한 형식을 다른 형식에서 구분하는 유일한 방법입니다.

예를 들어 단일 `BinaryExpressionSyntax` 클래스에는 자식으로 `Left`, `OperatorToken` 및 `Right`가 있습니다. `Kind` 속성은 구문 노드의 `AddExpression`, `SubtractExpression` 또는 `MultiplyExpression` 종류인지를 구분합니다.

### 💡 팁

`IsKind(C#)` 또는 `IsKind(VB)` 확장 메서드를 사용하여 종류를 확인하는 것이 좋습니다.

## 오류

원본 텍스트에 구문 오류가 있는 경우에도 원본에 대해 왕복 가능한 전체 구문 트리가 노출됩니다. 파서가 언어의 정의된 구문에 맞지 않는 코드를 발견하면 두 가지 기술 중 하나를 사용하여 구문 트리를 만듭니다.

- 파서에 특정 종류의 토큰이 필요하지만 찾을 수 없는 경우 토큰이 필요한 위치의 구문 트리로 누락된 토큰을 삽입할 수 있습니다. 누락된 토큰은 필요한 실제 토큰을 나타내지만 빈 범위를 가지며 해당 [SyntaxNode.IsMatch](#) 속성은 `true`를 반환합니다.
- 파서는 구문 분석을 계속할 수 있는 토큰을 찾을 때까지 토큰을 건너뛸 수 있습니다. 이 경우 건너뛴 토큰은 [SkippedTokensTrivia](#) 종류와 함께 기타 정보 노드로 연결됩니다.

# 의미 체계 작업

아티클 • 2023. 04. 08.

구문 트리는 소스 코드의 어휘 및 구문 구조를 나타냅니다. 이 정보만으로 원본의 모든 선언 및 논리를 설명하기에 충분하지만 참조되는 것을 식별하는 데 충분한 정보가 아닙니다. 이름은 다음을 나타낼 수 있습니다.

- 형식
- 필드
- 메서드
- 지역 변수

이러한 각각은 고유하게 다르지만 식별자에서 실제로 참조하는 것을 결정하는 데 종종 언어 규칙에 대한 심층적 이해가 필요합니다.

소스 코드에서 표현되는 프로그램 요소가 있으며 프로그램은 어셈블리 파일에서 패키지된 이전에 컴파일된 라이브러리를 참조할 수도 있습니다. 어셈블리에서 사용할 수 있는 소스 코드, 구문 노드 또는 트리가 없음에도 불구하고 프로그램은 여전히 내부의 요소를 참조할 수 있습니다.

이러한 작업의 경우 의미 체계 모델이 필요합니다.

소스 코드의 구문 모델 외에도 의미 체계 모델은 식별자를 참조하는 올바른 프로그램 요소와 올바르게 일치시키는 쉬운 방법을 제공하여 언어 규칙을 캡슐화합니다.

## 컴파일

컴파일은 어셈블리 참조, 컴파일러 옵션 및 원본 파일을 포함하는 C# 또는 Visual Basic 프로그램을 컴파일하는 데 필요한 모든 항목의 표현입니다.

이 정보는 모두 한 곳에 있기 때문에 소스 코드에 포함된 요소를 더 자세히 설명할 수 있습니다. 컴파일은 기호로 각 선언된 형식, 멤버 또는 변수를 나타냅니다. 컴파일은 소스 코드에서 선언되었거나 어셈블리에서 메타데이터로 가져온 기호를 찾거나 관련시키는 데 도움이 되는 다양한 메서드를 포함합니다.

구문 트리와 마찬가지로, 컴파일은 변경할 수 없습니다. 컴파일을 만든 후에 사용자나 공유하는 사용자가 변경할 수 없습니다. 그러나 그렇게 수행하는 대로 변경 내용을 지정하여 기존 컴파일에서 새 컴파일을 만들 수 있습니다. 예를 들어 추가 원본 파일 또는 어셈블리 참조를 포함할 수 있는 것을 제외하고 기존 컴파일과 모든 방식에서 동일한 컴파일을 만들 수 있습니다.

# 기호

기호는 소스 코드에 의해 선언되거나 메타데이터로 어셈블리에서 가져온 고유한 요소를 나타냅니다. 모든 네임스페이스, 형식, 메서드, 속성, 필드, 이벤트, 매개 변수 또는 지역 변수는 기호로 표시됩니다.

[Compilation](#) 형식의 다양한 메서드 및 속성은 기호를 찾는 데 도움이 됩니다. 예를 들어 일반적인 메타데이터 이름별로 선언된 형식에 대한 기호를 찾을 수 있습니다. 전역 네임스페이스로 루트된 기호의 트리로 전체 기호 테이블에 액세스할 수도 있습니다.

기호는 또한 컴파일이 다른 참조된 기호와 같은 원본 또는 메타데이터에서 결정하는 추가 정보를 포함합니다. 각 종류의 기호는 컴파일러에서 수집한 정보를 자세히 설명하는 고유한 메서드 및 속성이 있는 각 [ISymbol](#)에서 파생된 별도 인터페이스로 표시됩니다. 이러한 속성의 상당수는 다른 기호를 직접 참조합니다. 예를 들어 [IMethodSymbol.ReturnType](#) 속성은 메서드가 반환하는 실제 형식 기호를 알려 줍니다.

기호는 소스 코드와 메타데이터 간의 네임스페이스, 형식 및 멤버의 일반적인 표현을 제공합니다. 예를 들어 소스 코드에 선언된 메서드 및 메타데이터에서 가져온 메서드는 모두 동일한 속성이 있는 [IMethodSymbol](#)로 표시됩니다.

기호는 [System.Reflection](#) API로 표시된 CLR 형식 시스템과 개념상 비슷하지만 형식 이상을 모델링하므로 더 다양합니다. 네임스페이스, 지역 변수 및 레이블은 모두 기호입니다. 또한 기호는 CLR 개념이 아닌 언어 개념의 표현입니다. 겹치는 경우가 많지만 의미 있는 차이도 많습니다. 예를 들어 C# 또는 Visual Basic의 반복기 메서드는 단일 기호입니다. 그러나 반복기 메서드가 CLR 메타데이터로 번역되는 경우 이는 형식 및 여러 메서드입니다.

## 의미 체계 모델

의미 체계 모델은 단일 원본 파일에 대한 모든 의미 체계 정보를 나타냅니다. 다음 검색에 사용할 수 있습니다.

- 원본의 특정 위치에서 참조되는 기호
- 모든 식의 결과 형식
- 오류 및 경고인 모든 진단
- 원본 영역 내부 및 외부의 변수 흐름 방식
- 더 가상적인 질문에 대한 대답

# 작업 영역 작업

아티클 • 2023. 04. 08.

작업 영역 계층은 코드 분석을 수행하고 전체 솔루션을 통해 리팩터링하는 시작 지점입니다. 이 계층 내에서 작업 영역 API는 파일을 구문 분석하고, 옵션을 구성하거나 프로젝트 간 종속성을 관리하지 않고도 원본 텍스트, 구문 트리, 의미 체계 모델 및 컴파일과 같은 컴파일러 계층 개체 모델에 대한 직접 액세스를 제공하여 솔루션의 프로젝트에 대한 모든 정보를 단일 개체 모델로 구성하는 데 도움을 줍니다.

IDE와 같은 호스트 환경은 개방형 솔루션에 해당하는 작업 영역을 제공합니다. 간단히 솔루션 파일을 로드하여 IDE 외부에서 이 모델을 사용할 수도 있습니다.

## 작업 영역

작업 영역은 각각 문서의 컬렉션이 있는 프로젝트의 컬렉션으로 솔루션의 활성 표현입니다. 작업 영역은 일반적으로 사용자 형식으로 지속적으로 변경하거나 속성을 조작하는 호스트 환경에 연결됩니다.

[Workspace](#)는 솔루션의 현재 모델에 대한 액세스를 제공합니다. 호스트 환경에서 변경이 발생하는 경우 작업 영역은 해당 이벤트를 발생시키고 [Workspace.CurrentSolution](#) 속성이 업데이트됩니다. 예를 들어 텍스트 편집기에서 사용자 형식이 원본 문서 중 하나에 해당하는 경우 작업 영역은 이벤트를 사용하여 솔루션의 전반적인 모델이 변경되었고 해당 문서가 수정되었다는 신호를 보냅니다. 그런 다음 새 모델의 정확성을 분석하고, 중요성의 영역을 강조 표시하거나 코드 변경에 대해 제안하여 이러한 변경 내용에 반응할 수 있습니다.

또한 호스트 환경에서 연결이 해제되거나 호스트 환경이 없는 애플리케이션에서 사용되는 독립 실행형 작업 영역을 만들 수도 있습니다.

## 솔루션, 프로젝트, 문서

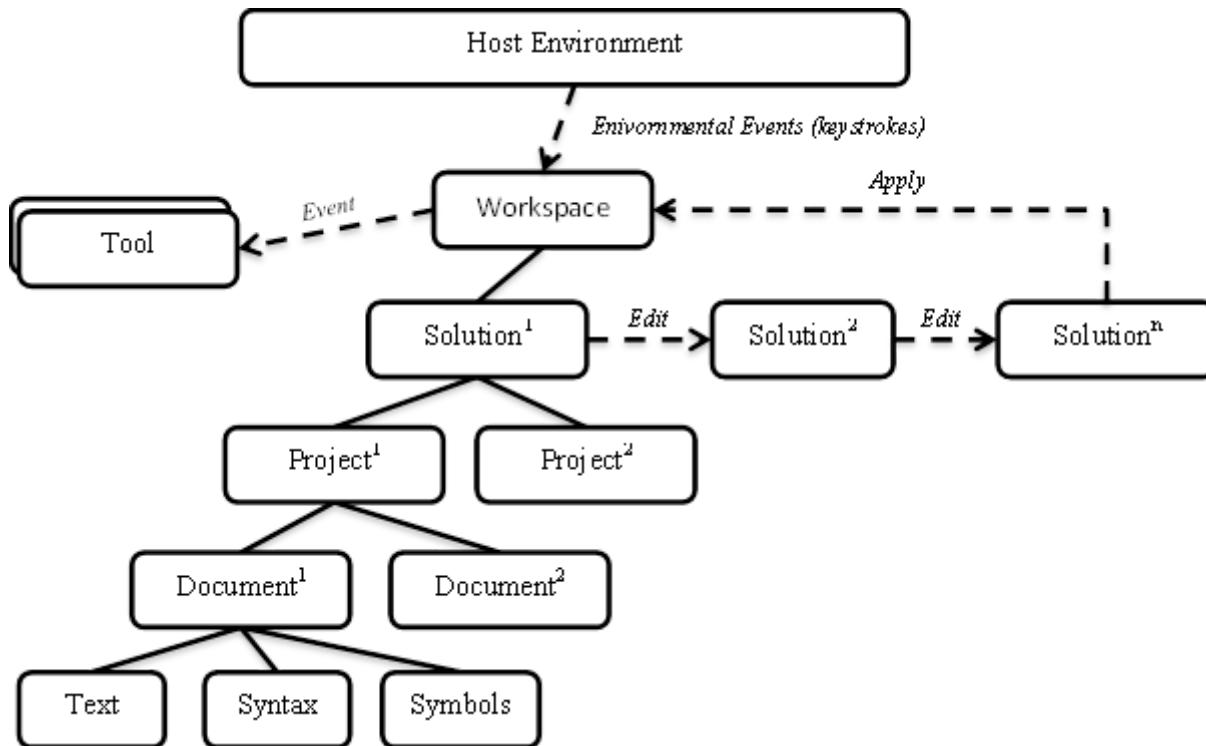
작업 영역은 키를 누를 때마다 변경될 수 있지만 격리 상태에서 솔루션의 모델을 사용하여 작업할 수 있습니다.

솔루션은 프로젝트 및 문서의 변경할 수 없는 모델입니다. 즉, 잠금 또는 중복 없이 모델을 공유할 수 있습니다. [Workspace.CurrentSolution](#) 속성에서 솔루션 인스턴스를 가져온 후 해당 인스턴스는 변경되지 않습니다. 그러나 구문 트리 및 컴파일을 사용하는 것과 같이 기존 솔루션 및 특정 변경 내용에 따라 새 인스턴스를 구성하여 솔루션을 수정할 수 있습니다. 변경 내용을 반영하도록 작업 영역을 가져오려면 변경된 솔루션을 작업 영역에 다시 명시적으로 적용해야 합니다.

프로젝트는 변경할 수 없는 전체 솔루션 모델의 일부입니다. 모든 소스 코드 문서, 구문 분석과 컴파일 옵션 및 어셈블리와 프로젝트 간 참조 모두를 나타냅니다. 프로젝트에서 프로젝트 종속성을 확인하거나 모든 원본 파일을 구문 분석할 필요 없이 해당 컴파일에 액세스할 수 있습니다.

문서는 또한 변경할 수 없는 전체 솔루션 모델의 일부입니다. 문서는 파일의 텍스트에 액세스할 수 있는 단일 원본 파일, 구문 트리 및 의미 체계 모델을 나타냅니다.

다음 다이어그램은 작업 영역이 호스트 환경, 도구에 연결되는 방법 및 편집하는 방법을 표현합니다.



## 요약

Roslyn은 소스 코드에 대한 풍부한 정보를 제공하고 C# 및 Visual Basic 언어로 완전한 충실도를 가진 컴파일러 API 및 작업 영역 API의 집합을 노출합니다. .NET Compiler Platform SDK는 코드 중심 도구 및 애플리케이션을 만들기 위한 진입에 대한 장벽을 크게 낮춰줍니다. 메타 프로그래밍, 코드 생성 및 변환, C# 및 Visual Basic 언어의 대화형 사용, 도메인 특정 언어에 C# 및 Visual Basic 포함과 같은 영역에서 다양한 혁신 기회를 창출합니다.

# Visual Studio에서 Roslyn 구문 시각화 도우미를 사용하여 코드 탐색

아티클 • 2023. 04. 08.

이 아티클에서는 .NET Compiler Platform("Roslyn") SDK의 일부로 제공되는 구문 시각화 도우미 도구에 대한 개요를 제공합니다. 구문 시각화 도우미는 구문 트리를 검사하고 탐색하는 데 도움이 되는 도구 창입니다. 분석하려는 코드의 모델을 이해하는 데 필수적인 도구입니다. 또한 .NET Compiler Platform("Roslyn") SDK를 사용하여 자체 애플리케이션을 개발할 때 도움이 되는 디버깅 도구이기도 합니다. 첫 번째 분석기를 만들 때 이 도구를 엽니다. 시각화 도우미는 API에서 사용되는 모델을 이해하는 데 도움이 됩니다. [SharpLab](#) 또는 [LINQPad](#)와 같은 도구를 사용하여 코드를 검사하고 구문 트리를 이해할 수도 있습니다.

## 설치 지침 - Visual Studio 설치 관리자

Visual Studio 설치 관리자에서 .NET Compiler Platform SDK를 찾는 두 가지 방법이 있습니다.

### Visual Studio 설치 관리자를 사용한 설치 - 워크로드 보기

.NET Compiler Platform SDK는 Visual Studio 확장 개발 워크로드의 일부로 자동으로 선택되지 않습니다. 선택적 구성 요소로 선택해야 합니다.

1. Visual Studio 설치 관리자를 실행합니다.
2. 수정을 선택합니다.
3. Visual Studio 확장 개발 워크로드를 확인합니다.
4. 요약 트리에서 Visual Studio 확장 개발 노드를 엽니다.
5. .NET Compiler Platform SDK에 대한 확인란을 선택합니다. 선택적 구성 요소 아래에서 마지막에 찾을 수 있습니다.

필요에 따라, 시각화 도우미에서 DGML 편집기에 그래프도 표시할 수 있습니다.

1. 요약 트리에서 개별 구성 요소 노드를 엽니다.
2. DGML 편집기 확인란을 선택합니다.

### Visual Studio 설치 관리자를 사용한 설치 - 개별 구성 요소 탭

1. Visual Studio 설치 관리자를 실행합니다.

2. 수정을 선택합니다.
3. 개별 구성 요소 탭을 선택합니다.
4. .NET Compiler Platform SDK에 대한 확인란을 선택합니다. 컴파일러, 빌드 도구 및 런타임 섹션의 위쪽에서 찾을 수 있습니다.

필요에 따라, 시각화 도우미에서 DGML 편집기에 그래프도 표시할 수 있습니다.

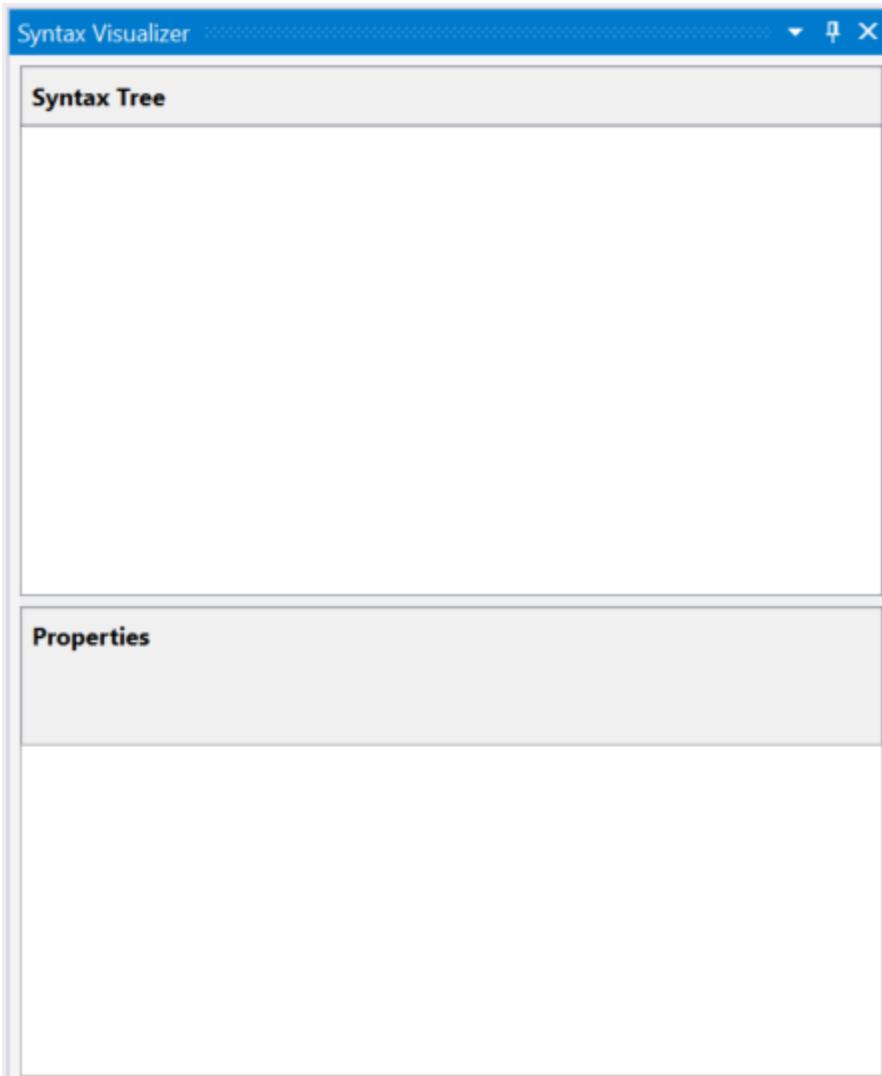
1. DGML 편집기 확인란을 선택합니다. 코드 도구 섹션에서 찾을 수 있습니다.

[개요](#) 아티클을 읽고 .NET Compiler Platform SDK에서 사용된 개념을 숙지하세요. 구문 트리, 노드, 토큰 및 퀴즈에 대한 소개를 제공합니다.

## 구문 시각화 도우미

Syntax Visualizer를 사용하면 Visual Studio IDE 내의 현재 활성 편집기 창에서 C# 또는 Visual Basic 코드 파일에 대한 구문 트리를 검사할 수 있습니다. 시각화 도우미는 보기> 다른 창>구문 시각화 도우미를 클릭하여 시작할 수 있습니다. 오른쪽 위 모서리에 있는 빠른 실행 도구 모음을 사용할 수도 있습니다. "구문"을 입력하면 구문 시각화 도우미를 여는 명령이 나타납니다.

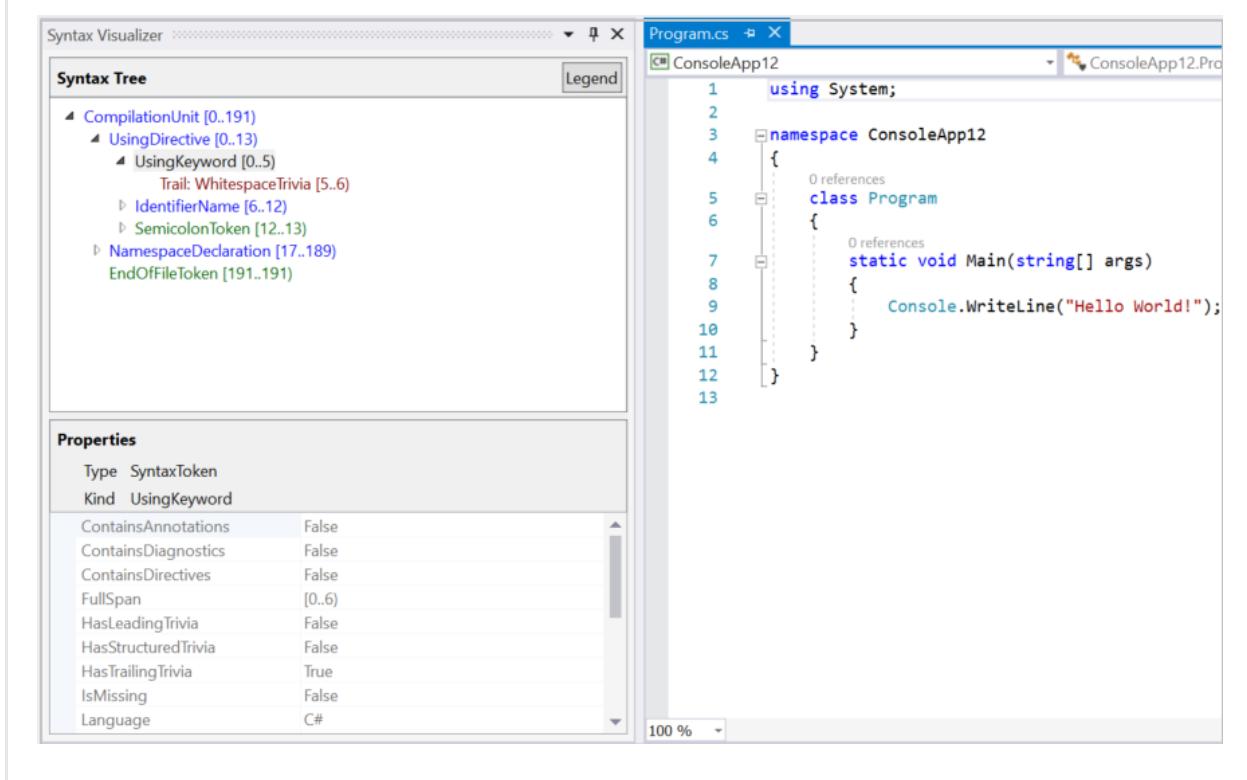
이 명령은 구문 시각화 도우미를 부동 도구 창으로 엽니다. 코드 편집기 창이 열려 있지 않은 경우 다음 그림과 같이 디스플레이가 비어 있습니다.



이 도구 창을 Visual Studio 내에서 왼쪽과 같이 편리한 위치에 고정합니다. 시각화 도우미에서는 현재 코드 파일에 대한 정보를 표시합니다.

**파일>새 프로젝트** 명령을 사용하여 새 프로젝트를 만듭니다. Visual Basic 또는 C# 프로젝트를 만들 수 있습니다. Visual Studio에서 이 프로젝트의 주 코드 파일을 열면 시각화 도우미는 구문 트리를 표시합니다. Visual Studio 인스턴스에서 기존 C#/Visual Basic 파일을 열 수 있으며 시각화 도우미는 해당 파일의 구문 트리를 표시합니다. Visual Studio 내에서 여러 코드 파일을 열면 시각화 도우미는 현재 활성 코드 파일(키보드 포커스가 있는 코드 파일)에 대한 구문 트리를 표시합니다.

C#



위 이미지에서와 같이 시각화 도우미 도구 창에는 구문 트리가 위에 표시되고 속성 그리드가 아래에 표시됩니다. 속성 그리드는 항목의 .NET 형식 및 종류(SyntaxKind)를 포함하여 트리에서 현재 선택된 항목의 속성을 표시합니다.

구문 트리는 노드, 토큰 및 큐즈라는 세 가지 유형의 항목으로 구성됩니다. [구문을 사용하여 작업](#) 아티클에서 이러한 형식에 대해 자세히 읽을 수 있습니다. 각 형식의 항목은 다른 색을 사용하여 표시됩니다. 사용된 색에 대한 개요를 보려면 '범례' 단추를 클릭하세요.

트리의 각 항목에는 자체 범위도 표시됩니다. 범위는 텍스트 파일에서 해당 노드의 인덱스(시작 및 끝 위치)입니다. 앞의 C# 예에서 선택한 "UsingKeyword [0..5)" 토큰에는 5자 너비 [0..5)인 범위가 있습니다. "[.]" 표기법은 시작 인덱스는 범위의 일부이지만 끝 인덱스는 아님을 의미합니다.

트리를 탐색하는 방법은 두 가지가 있습니다.

- 트리에서 항목을 확장하거나 클릭합니다. 시각화 도우미는 코드 편집기에서 이 항목의 범위에 해당하는 텍스트를 자동으로 선택합니다.
- 코드 편집기에서 텍스트를 클릭하거나 선택합니다. 앞의 Visual Basic 예제에서 코드 편집기에 "모듈 Module1"이 포함된 줄을 선택하면 시각화 도우미가 트리의 해당 ModuleStatement 노드로 자동 이동합니다.

시각화 도우미는 트리에서 편집기의 선택된 텍스트 범위와 가장 일치하는 항목을 강조 표시합니다.

시각화 도우미는 활성 코드 파일의 수정 내용과 일치하도록 트리를 새로 고칩니다.

Main() 내의 Console.WriteLine()에 대한 호출을 추가합니다. 입력할 때 시각화 도우미

는 트리를 새로 고칩니다.

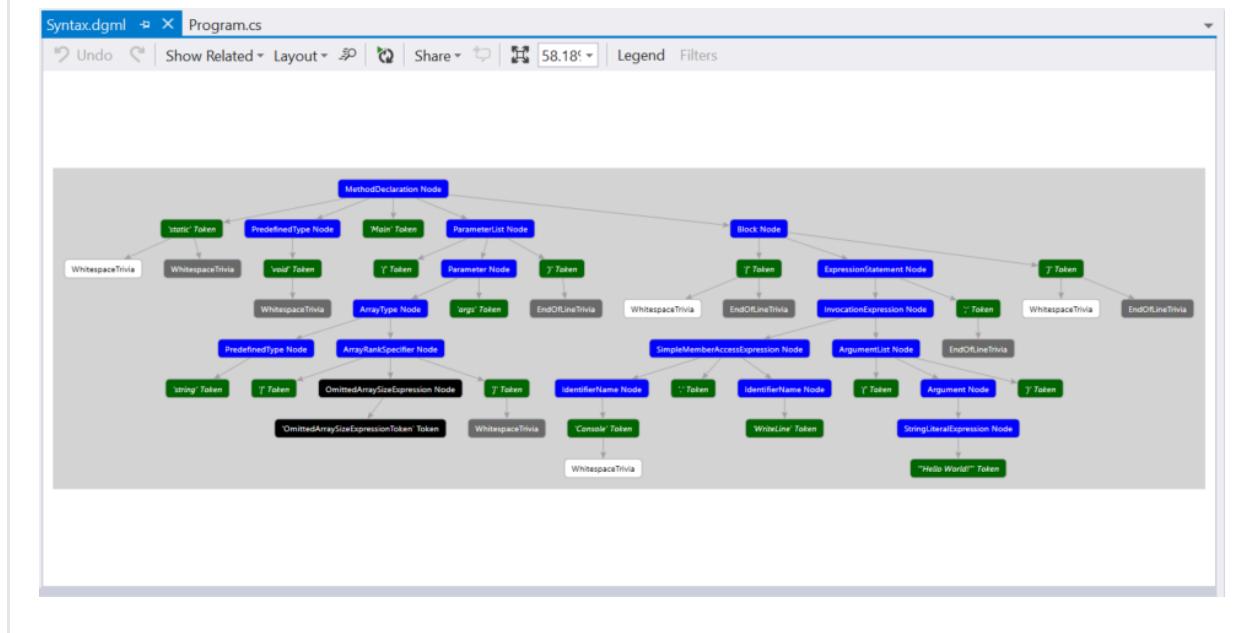
`Console`.을 입력하면 입력을 일시 중지합니다. 트리에는 분홍색으로 채색된 일부 항목이 있습니다. 이때 입력된 코드에는 오류('진단'이라고도 함)가 있습니다. 이러한 오류는 구문 트리의 노드, 토큰 및 퀴즈에 첨부됩니다. 시각화 도우미에서는 분홍색으로 배경을 강조 표시하여 어떤 항목에 오류가 첨부되어 있는지 보여줍니다. 항목을 마우스로 가리키면 분홍색으로 표시된 항목의 오류를 검사할 수 있습니다. 시각화 도우미는 구문 오류(입력된 코드의 구문과 관련된 오류)만 표시하고, 의미 체계 오류는 표시하지 않습니다.

## 구문 그래프

트리에서 원하는 항목을 마우스 오른쪽 단추로 클릭하고 **직접 구문 그래프 보기**를 클릭합니다.

C#

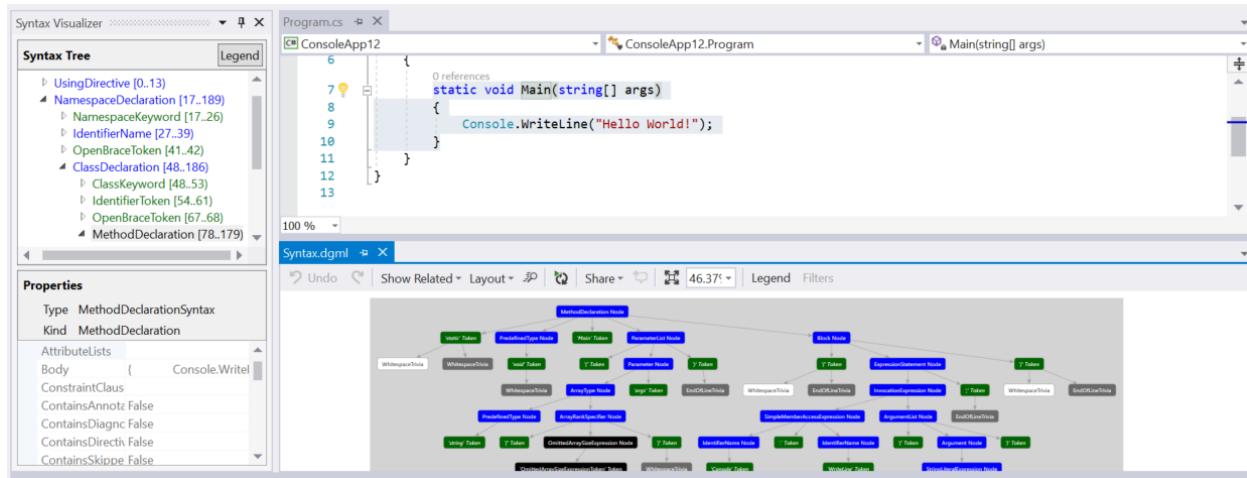
시각화 도우미는 선택한 항목을 루트로 하는 하위 트리의 그래픽 표현을 표시합니다. C# 예제에서 `Main()` 메서드에 해당하는 **MethodDeclaration** 노드에 대해 이 단계를 시도합니다. 시각화 도우미는 다음과 같이 보이는 구문 그래프를 표시합니다.



구문 그래프 뷰어에는 범례를 색칠 체계로 표시하는 옵션이 있습니다. 또한 구문 그래프의 개별 항목을 마우스로 가리키면 해당 항목에 해당하는 속성을 볼 수 있습니다.

트리의 여러 항목에 대한 구문 그래프를 반복적으로 볼 수 있으며 그레프는 항상 Visual Studio 내의 동일한 창에 표시됩니다. Visual Studio 내 편리한 위치에 이 창을 고정할 수 있으므로 새 구문 그래프를 보기 위해 탭 사이를 전환할 필요가 없습니다. 대개는 코드 편집기 창 아래쪽이 보기 편리합니다.

시각화 도우미 도구 창 및 구문 그래프 창과 함께 사용할 도킹 레이아웃은 다음과 같습니다.



또 다른 옵션은 듀얼 모니터 환경에서 두 번째 모니터에 구문 그래프 창을 배치하는 것입니다.

## 의미 체계 검사

구문 시각화 도우미는 기호 및 의미 체계 정보에 대한 기본 검사를 수행합니다. C# 예제에서는 `Main()` 안에 `double x = 1 + 1;`을 입력합니다. 그런 다음, 코드 편집기 창에서 식 `1 + 1`을 선택합니다. 시각화 도우미는 시각화 도우미에서 **AddExpression** 노드를 강조 표시합니다. 이 **AddExpression**을 마우스 오른쪽 단추로 클릭하고 **기호 보기(있는 경우)**를 클릭합니다. 대부분의 메뉴 항목에는 "해당되는 경우" 한정자가 있습니다. 구문 시각화 도우미는 모든 노드에 대해 나타나지 않을 수 있는 속성을 포함한 노드의 속성을 검사합니다.

시각화 도우미의 속성 그리드는 다음 그림과 같이 업데이트됩니다. 식의 기호는 **Kind = Method**인 **SynthesizedIntrinsicOperatorSymbol**입니다.

The screenshot shows the Syntax Visualizer window with the following details:

- Syntax Tree:** Shows the parse tree structure with nodes like `lDeclarationStatement`, `/VariableDeclaration`, `PredefinedType`, `VariableDeclarator`, `IdentifierToken`, `EqualsValueClause`, `AddExpression`, `NumericLiteralExpression`, and `PlusToken`.
- Properties:**
  - Type: `SynthesizedIntrinsicOperatorSymbol`
  - Kind: `Method`
  - Name: `op_Addition`
  - OriginalDefinition: `int.operator +(int, int)`
  - OverriddenMethod
  - Parameters: `(Collection)`
  - PartialDefinitionPart
  - PartialImplementation
  - ReceiverType: `int`

Red arrows point from the text descriptions below to the corresponding properties in the Properties grid.

```

static void Main(string[] ar
{
    double x = 1 + 1;
}

```

동일한 `AddExpression` 노드에 대해 `TypeSymbol` 보기(있는 경우)를 시도합니다. 시각화 도우미의 속성 그리드가 다음 그림과 같이 업데이트되어 선택한 식의 형식이 `Int32`임을 나타냅니다.

The screenshot shows the Syntax Visualizer window with the following details:

- Syntax Tree:** Same as the first screenshot.
- Properties:**
  - Type: `PENamedTypeSymbolNonGeneric`
  - Kind: `NamedType`
  - Name: `Int32`
  - OriginalDefinition: `int`
  - SpecialType: `System_Int32`
  - StaticConstructors: `(Collection)`
  - TupleElementName: `(Collection)`
  - TupleElements: `(Collection)`
  - TupleElementTypes: `(Collection)`

Red arrows point from the text descriptions below to the corresponding properties in the Properties grid.

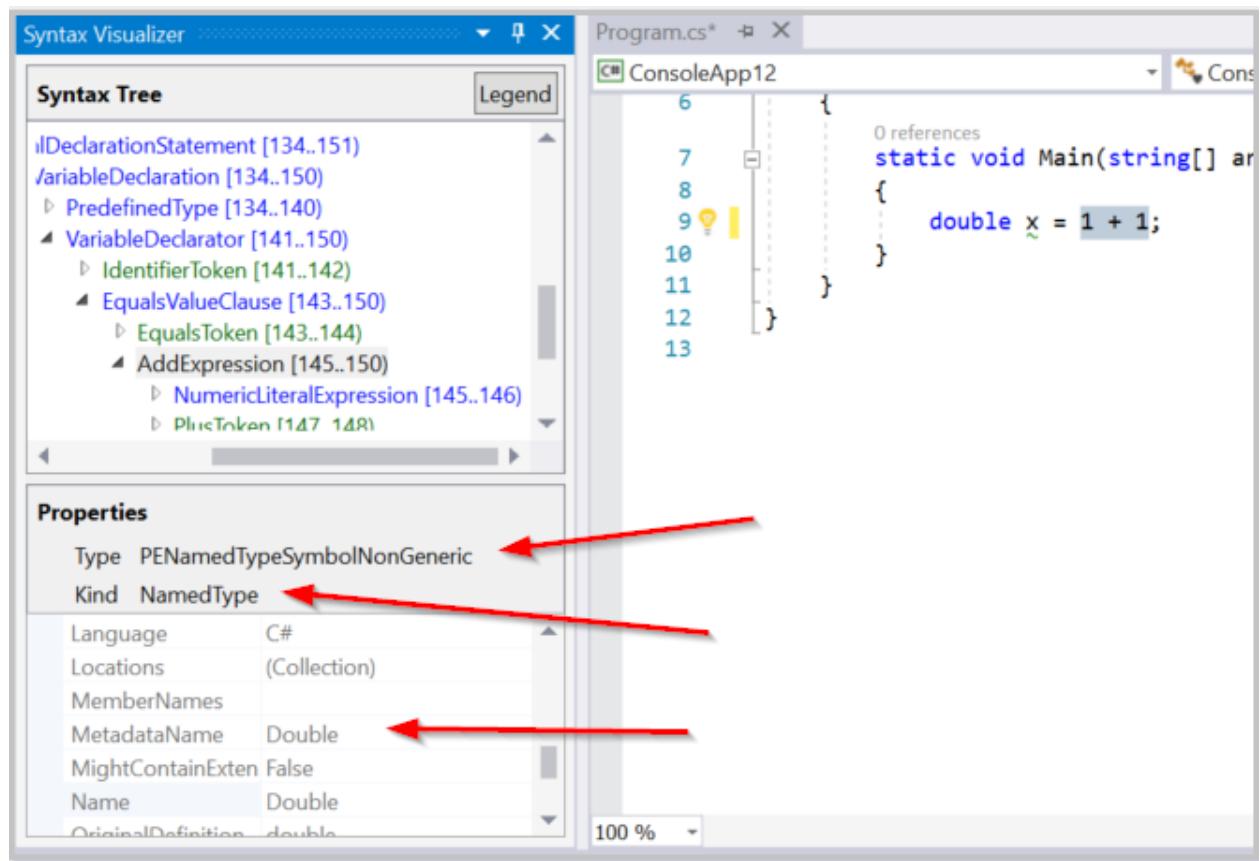
```

static void Main(string[] ar
{
    double x = 1 + 1;
}

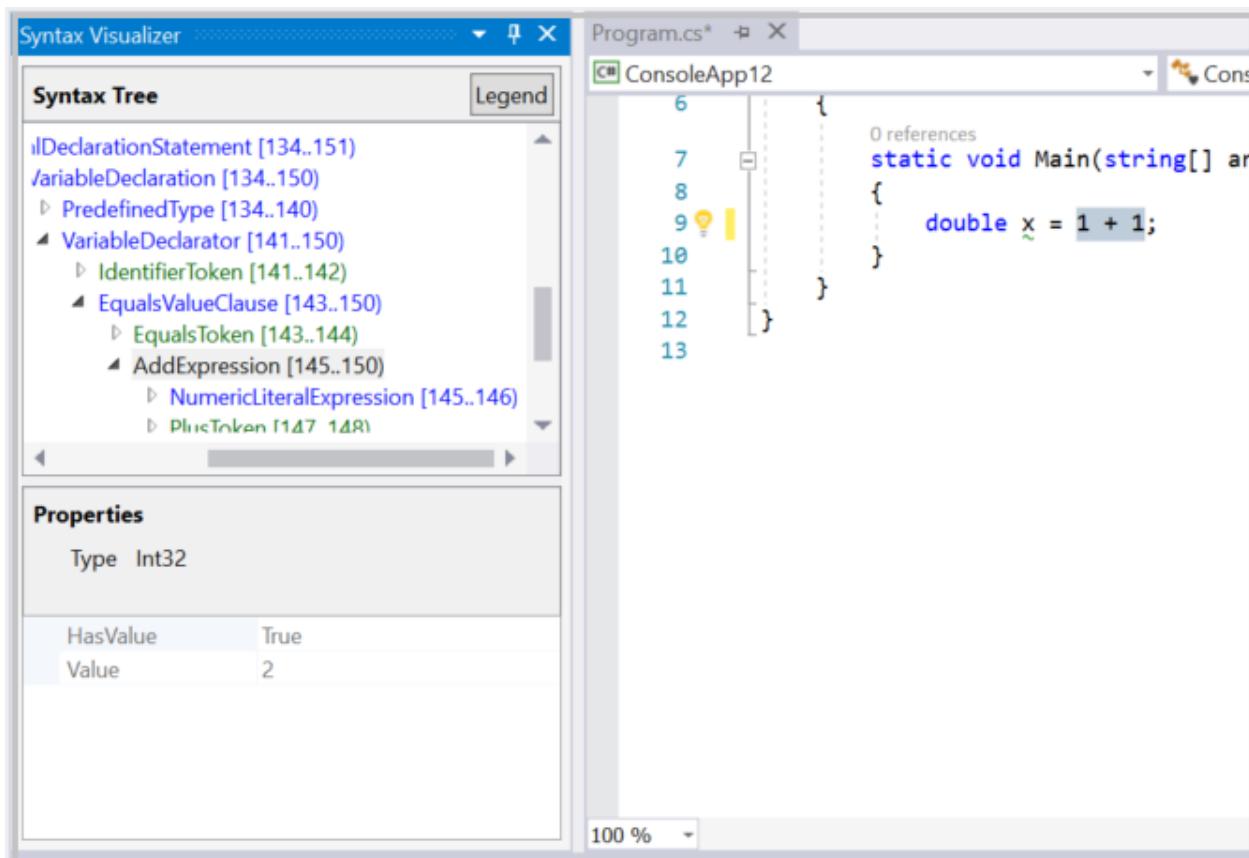
```

동일한 `AddExpression` 노드에 대해 변환된 `TypeSymbol` 보기(있는 경우)를 시도합니다. 다음 그림과 같이 식 형식이 `Int32`이지만 변환된 식 형식이 `Double`임을 나타내도록 속성

그리드가 업데이트됩니다. `Int32` 식은 `Double`로 변환되어야 하는 컨텍스트에서 발생하므로 이 노드에는 변환된 형식 기호 정보가 포함됩니다. 이 변환은 대입 연산자의 왼쪽에 있는 변수 `x`에 대해 지정된 `Double` 형식을 충족시킵니다.



마지막으로 동일한 `AddExpression` 노드에 대해 **상수 값 보기(있는 경우)**를 시도합니다. 속성 그리드에서는 식의 값이 값 2인 컴파일 시간 상수를 보여줍니다.



앞의 예제는 Visual Basic에서도 복제될 수 있습니다. Visual Basic 파일에 `Dim x As Double = 1 + 1`을 입력합니다. 코드 편집기 창에서 식 `1 + 1`을 선택합니다. 시각화 도우미는 시각화 도우미의 해당 `AddExpression` 노드를 강조 표시합니다. 이 `AddExpression`에 대해 앞의 단계를 반복하면 동일한 결과가 나타납니다.

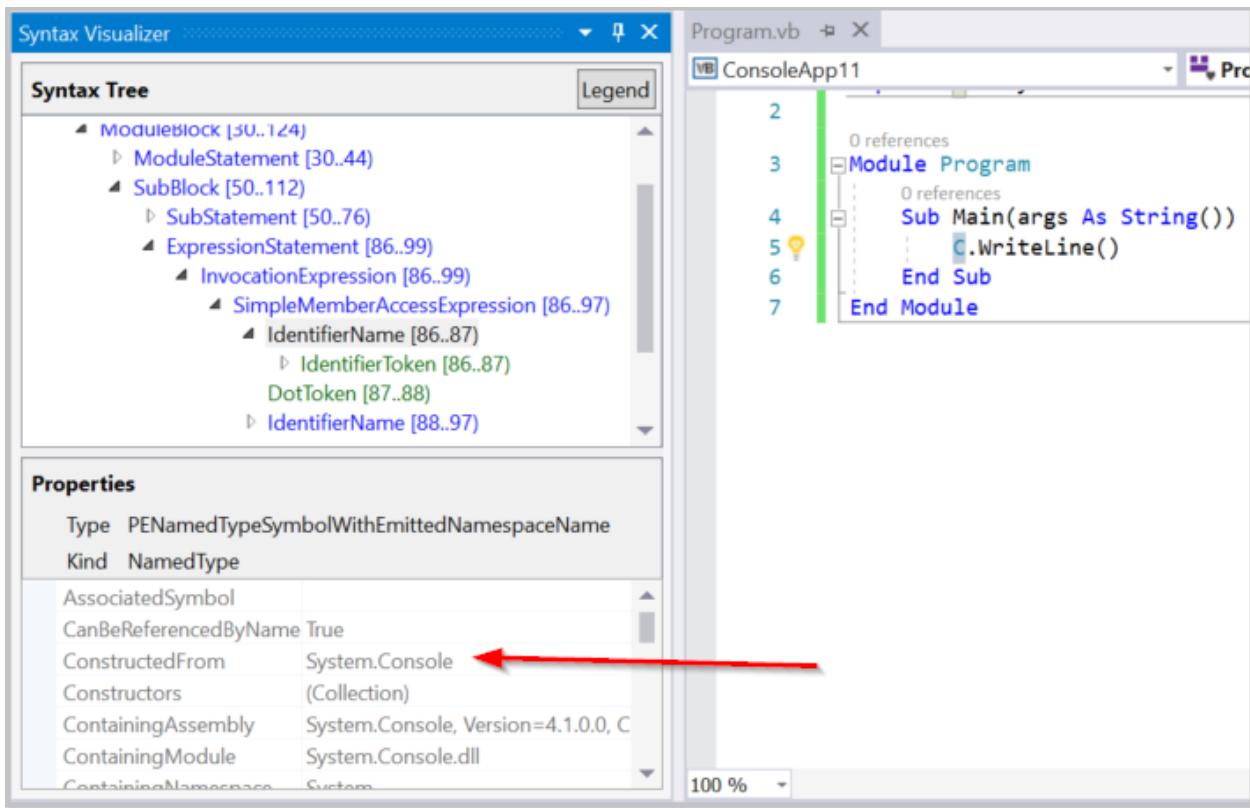
Visual Basic에서 더 많은 코드를 검사합니다. 주 Visual Basic 파일을 다음 코드로 업데이트합니다.

```
VB

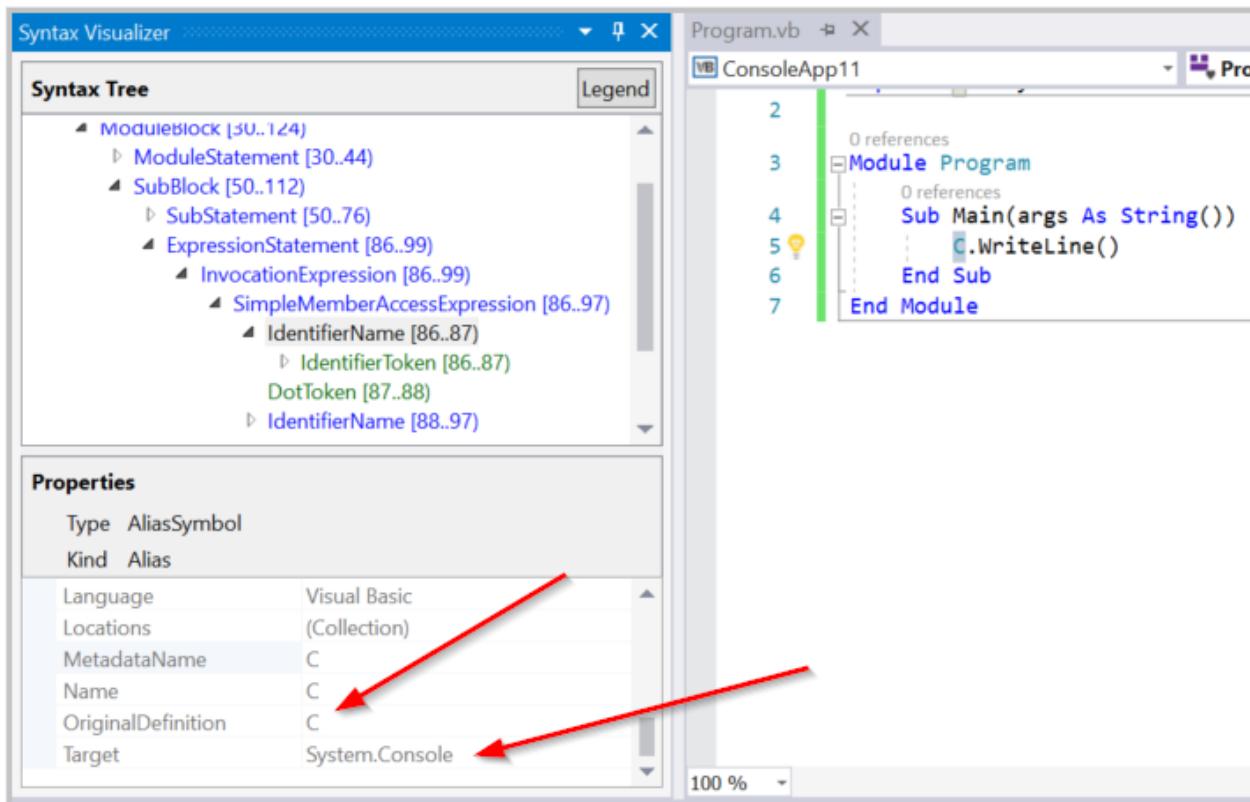
Imports C = System.Console

Module Program
    Sub Main(args As String())
        C.WriteLine()
    End Sub
End Module
```

이 코드에서는 파일 위에 있는 `System.Console` 형식에 매핑되는 `c`라는 별칭을 소개하고 `Main()` 내부에서 이 별칭을 사용합니다. `Main()` 메서드 내에서 이 별칭(`C.WriteLine()`)의 `c`의 사용을 선택합니다. 시각화 도우미는 시각화 도우미에서 해당 `IdentifierName` 노드를 선택합니다. 이 노드를 마우스 오른쪽 단추로 클릭하고 **기호 보기(있는 경우)**를 클릭합니다. 속성 그리드는 다음 그림과 같이 이 식별자가 형식 `System.Console`에 바인딩되어 있음을 나타냅니다.

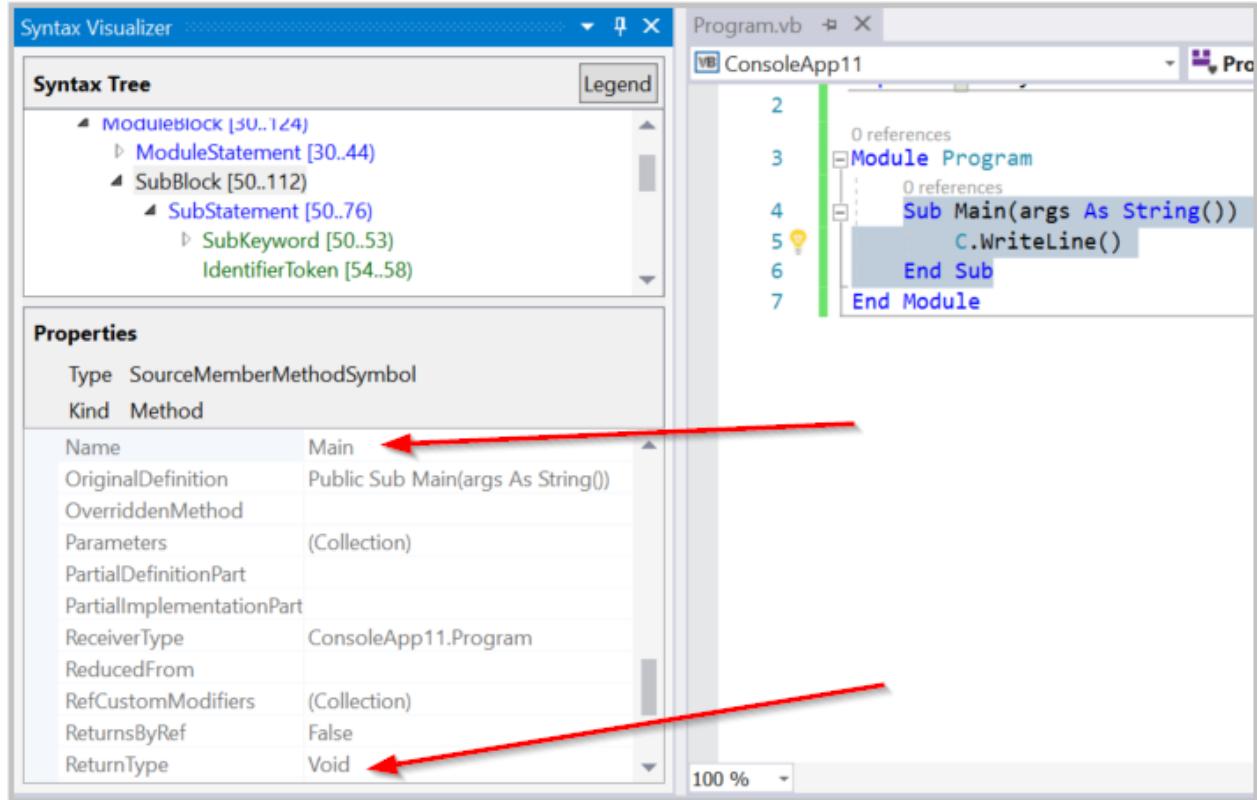


동일한 IdentifierName 노드에 대해 AliasSymbol 보기(있는 경우)를 시도합니다. 속성 그리드는 식별자가 System.Console 대상에 바인딩된 이름이 c인 별칭임을 나타냅니다. 즉, 속성 그리드는 식별자 c에 해당하는 AliasSymbol에 대한 정보를 제공합니다.



선택된 형식, 메서드, 속성에 해당하는 기호를 검사합니다. 시각화 도우미에서 해당 노드를 선택하고 기호 보기(있는 경우)를 클릭합니다. 메서드의 본문이 포함된 Sub Main() 메서드를 선택합니다. 시각화 도우미의 해당 SubBlock 노드에 대해 기호 보기(있는 경우)

를 클릭합니다. 속성 그리드는 이 SubBlock의 MethodSymbol 이름이 Main이고 반환 형식이 Void임을 보여줍니다.



위의 Visual Basic 예제는 C#에서 쉽게 복제할 수 있습니다. 별칭에 대해 `Imports C = System.Console` 대신 `using C = System.Console;` 을 입력합니다. C#의 앞 단계는 시각화 도우미 창에서 동일한 결과를 생성합니다.

의미 체계 검사 작업은 노드에서만 사용할 수 있습니다. 토큰 또는 퀴즈에는 사용할 수 없습니다. 모든 노드에 검사할 흥미 있는 의미 체계 정보가 있는 것은 아닙니다. 노드에 흥미 있는 의미 체계 정보가 없으면 \* 기호 보기(있는 경우)를 클릭하면 빈 속성 그리드가 표시됩니다.

[의미 체계와 함께 작업](#) 개요 문서에서 의미 체계 분석을 수행하기 위한 API에 대해 자세히 읽을 수 있습니다.

## 구문 시각화 도우미 닫기

소스 코드 검사에 사용하고 있지 않을 때 시각화 도우미 창을 닫을 수 있습니다. 구문 시각화 도우미는 코드를 탐색하고 소스를 편집 및 변경하면서 해당 디스플레이를 업데이트 합니다. 사용하지 않을 때 혼란을 가져올 수 있습니다.

# 원본 생성기

아티클 • 2023. 06. 25.

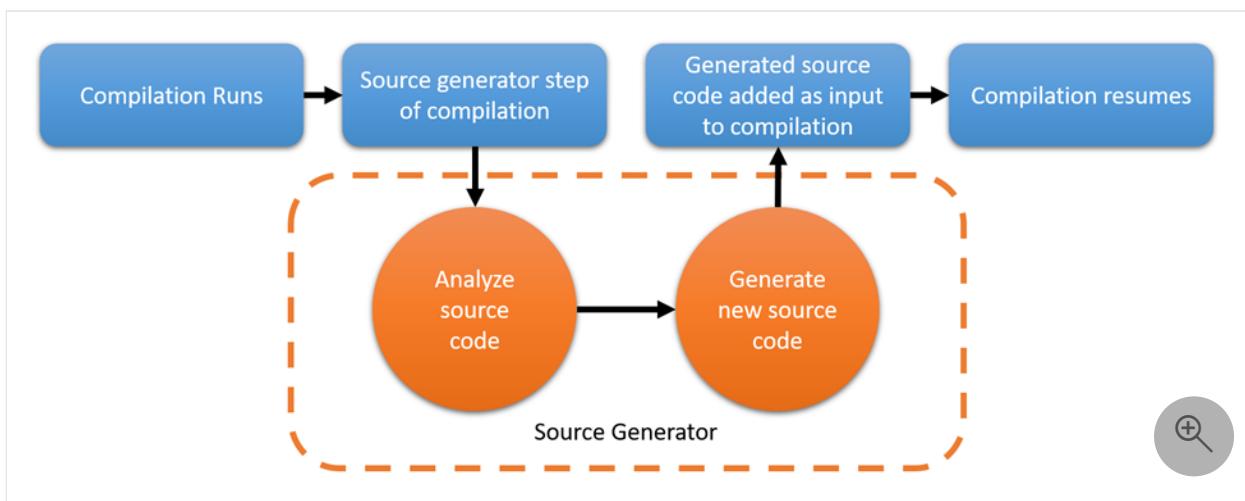
이 문서에서는 .NET Compiler Platform("Roslyn") SDK의 일부로 제공되는 원본 생성기에 대한 개요를 제공합니다. 원본 생성기를 사용하면 C# 개발자가 컴파일되는 사용자 코드를 검사할 수 있습니다. 생성기는 사용자의 컴파일에 추가되는 새 C# 원본 파일을 즉시 만들 수 있습니다. 이러한 방식으로 컴파일하는 동안 실행되는 코드가 있습니다. 프로그램을 검사하여 나머지 코드와 함께 컴파일되는 추가 소스 파일을 생성합니다.

원본 생성기는 C# 개발자가 작성할 수 있는 새로운 종류의 구성 요소로, 다음 두 가지 주요 작업을 수행할 수 있습니다.

1. 컴파일되는 모든 사용자 코드를 나타내는 **컴파일 개체**를 검색합니다. 이 개체를 검사할 수 있으며 현재의 분석기와 마찬가지로 컴파일 중인 코드에 대한 구문 및 의미 모델과 함께 작동하는 코드를 작성할 수 있습니다.
2. 컴파일 중에 컴파일 개체에 추가할 수 있는 C# 원본 파일을 생성합니다. 즉, 코드를 컴파일하는 동안 추가 소스 코드를 컴파일에 대한 입력으로 제공할 수 있습니다.

이러한 두 가지를 결합하면 원본 생성기가 매우 유용합니다. 컴파일 중에 컴파일러가 빌드하는 모든 풍부한 메타데이터를 사용하여 사용자 코드를 검사할 수 있습니다. 그런 다음 생성기는 분석한 데이터를 기반으로 하는 동일한 컴파일로 C# 코드를 다시 내보낸다. Roslyn 분석기에 익숙한 경우 원본 생성기를 C# 소스 코드를 내보낼 수 있는 분석기라고 생각할 수 있습니다.

원본 생성기는 아래에 시각화된 컴파일 단계로 실행됩니다.



원본 생성기는 모든 분석기와 함께 컴파일러에 의해 로드되는 .NET Standard 2.0 어셈블리입니다. .NET Standard 구성 요소를 로드하고 실행할 수 있는 환경에서 사용할 수 있습니다.

## ① 중요

현재는 .NET Standard 2.0 어셈블리만 원본 생성기로 사용할 수 있습니다.

# 일반적인 시나리오

최신 기술에서 사용하는 분석을 바탕으로 사용자 코드를 검사하고 정보나 코드를 생성하는 대표적인 세 가지 방법이 있습니다.

- 런타임 리플렉션,
- MSBuild 작업 저글링,
- (이 문서에서는 설명하지 않는) IL(중간 언어) 위빙입니다.

원본 생성기는 이를 각각의 방법보다 개선될 수 있습니다.

## 런타임 리플렉션

런타임 리플렉션은 오래 전에 .NET에 추가된 강력한 기술입니다. 이를 사용하는 수많은 시나리오가 있습니다. 일반적인 시나리오는 앱이 시작될 때 사용자 코드의 일부 분석을 수행하고 해당 데이터를 사용하여 작업을 생성하는 것입니다.

예를 들어 ASP.NET Core 웹 서비스가 처음 실행되면 리플렉션을 사용하여 정의한 구문을 검색하여 컨트롤러 및 razor 페이지와 같은 것을 "연결"할 수 있습니다. 이렇게 하면 강력한 추상화로 간단한 코드를 작성할 수 있지만 런타임 시 성능 저하가 발생합니다. 웹 서비스 또는 앱이 처음 시작될 때 코드에 대한 정보를 검색하는 모든 런타임 리플렉션 코드 실행이 완료될 때까지 요청을 수락할 수 없습니다. 이 성능 저하는 크지 않지만, 자체 앱에서 개선할 수 없는 고정 비용입니다.

원본 생성기를 사용하면 시작의 컨트롤러 검색 단계가 컴파일 시간에 대신 발생할 수 있습니다. 생성기는 소스 코드를 분석하고 앱을 "연결"하는 데 필요한 코드를 내보낼 수 있습니다. 원본 생성기를 사용하면 오늘 런타임에 발생하는 작업이 컴파일 시간으로 푸시될 수 있으므로 시작 시간이 더 빨라질 수 있습니다.

## MSBuild 작업 저글링

원본 생성기는 런타임 시 리플렉션에 제한되지 않도록 성능을 개선하여 형식도 검색할 수 있습니다. 일부 시나리오에서는 컴파일에서 데이터를 검사할 수 있도록 MSBuild C# 작업(CSC라고 함)을 여러 번 호출합니다. 짐작하시겠지만, 컴파일러를 두 번 이상 호출하면 앱을 구축하는 데 걸리는 총 시간이 영향을 받습니다. 원본 생성기는 일부 성능 혜택을 제공할 뿐만 아니라 도구가 올바른 추상화 수준에서 작동할 수 있도록 해주므로, 이와 같

이 MSBuild 작업을 저글링할 필요가 없도록 원본 생성기를 사용할 방법을 조사하고 있습니다.

원본 생성기가 제공할 수 있는 또 다른 기능은 컨트롤러와 razor 페이지 간의 ASP.NET Core 라우팅이 작동하는 방식과 같은 일부 "문자열 형식" API의 사용을 없애는 것입니다. 원본 생성기를 사용하면 컴파일 시간 세부 정보로 생성되는 데 필요한 문자열로 라우팅을 강력하게 입력할 수 있습니다. 이렇게 하면 잘못 입력된 문자열 리터럴이 올바른 컨트롤러에 도달하지 않는 요청으로 이어지는 횟수가 줄어듭니다.

## 원본 생성기 시작

이 가이드에서는 [IIncrementalGenerator API](#)를 사용하여 원본 생성기를 만드는 방법을 살펴봅니다.

1. .NET 콘솔 애플리케이션을 만듭니다. 이 예제에서는 .NET 7을 사용합니다.
2. `Program` 클래스를 다음 코드로 바꿉니다. 다음 코드는 최상위 문을 사용하지 않습니다. 이 첫 번째 소스 생성기는 해당 클래스에서 partial 메서드를 작성하므로 클래식 양식이 `Program` 필요합니다.

```
C#  
  
namespace ConsoleApp;  
  
partial class Program  
{  
    static void Main(string[] args)  
    {  
        HelloFrom("Generated Code");  
    }  
  
    static partial void HelloFrom(string name);  
}
```

### ⚠ 참고

이 샘플을 있는 그대로 실행할 수 있지만 아무 일도 발생하지 않습니다.

3. 지금부터는 `partial void HelloFrom` 메서드를 구현하는 원본 생성기 프로젝트를 만들겠습니다.
4. TFM(대상 프레임워크 모니터)을 `netstandard2.0` 대상으로 하는 .NET 표준 라이브러리 프로젝트를 만듭니다. `NuGet 패키지 Microsoft.CodeAnalysis.Analyzers` 및 `Microsoft.CodeAnalysis.CSharp`를 추가합니다.

## XML

```
<Project Sdk="Microsoft.NET.Sdk">

    <PropertyGroup>
        <TargetFramework>netstandard2.0</TargetFramework>
        <LangVersion>11.0</LangVersion>
        <EnforceExtendedAnalyzerRules>true</EnforceExtendedAnalyzerRules>
    </PropertyGroup>

    <ItemGroup>
        <PackageReference Include="Microsoft.CodeAnalysis.CSharp"
Version="4.6.0" PrivateAssets="all" />
        <PackageReference Include="Microsoft.CodeAnalysis.Analyzers"
Version="3.3.4" PrivateAssets="all" />
    </ItemGroup>

</Project>
```

### 💡 팀

원본 생성기 프로젝트는 netstandard2.0 TFM를 대상으로 해야 합니다. 그렇지 않으면 정상적으로 작동하지 않습니다.

5. 다음과 같이 자체 원본 생성기를 지정하는 *HelloSourceGenerator.cs*라는 새 C# 파일을 만듭니다.

## C#

```
using Microsoft.CodeAnalysis;

namespace SourceGenerator;

public sealed class HelloSourceGenerator : IIncrementalGenerator
{
    public void Initialize(IncrementalGeneratorInitializationContext
context)
    {
        var compilationProvider = context.CompilationProvider;

        context.RegisterSourceOutput(
            compilationProvider,
            static (context, compilation) =>
            {
                // Code generation goes here
            });
    }
}
```

원본 생성기는 [Microsoft.CodeAnalysis.IIncrementalGenerator](#) 인터페이스를 구현하고 [Microsoft.CodeAnalysis.GeneratorAttribute](#)가 있어야 합니다.

1. [IIncrementalGenerator.Initialize](#) 메서드의 내용을 다음 구현으로 바꿉니다.

C#

```
using Microsoft.CodeAnalysis;

namespace SourceGenerator;

[Generator]
public sealed class HelloSourceGenerator : IIncrementalGenerator
{
    public void Initialize(IncrementalGeneratorInitializationContext context)
    {
        var compilationIncrementalValue = context.CompilationProvider;

        context.RegisterSourceOutput(
            compilationIncrementalValue,
            static (context, compilation) =>
        {
            // Get the entry point method
            var mainMethod =
                compilation.GetEntryPoint(context.CancellationToken);
            var typeName = mainMethod.ContainingType.Name;

            string source = $$"""
                // Auto-generated code
                namespace
                {{mainMethod.ContainingNamespace.ToString()}};

                public static partial class {{typeName}}
                {
                    static partial void HelloFrom(string name) =>
                        Console.WriteLine($"Generator says: Hi {name}");
                }
            """;

            // Add the source code to the compilation
            context.AddSource($"{typeName}.g.cs", source);
        });
    }
}
```

`compilationIncrementalValue` 변수는 의 메서드 `context`에 전달되는 `RegisterSourceOutput` 컴파일의 증분 모델 값을 저장하는 데 사용됩니다. 매개 변수에서 `compilation` 컴파일의 진입점 또는 `Main` 메서드에 액세스할 수 있습니다. `mainMethod` 인스턴스는 [IMethodSymbol](#)이며, 메서드 또는 메서드형 기호(생성자,

소멸자, 연산자 또는 속성/이벤트 접근자 포함)를 나타냅니다. 메서드는 `Microsoft.CodeAnalysis.Compilation.GetEntryPoint` 프로그램의 진입점에 대한 를 반환 `IMethodSymbol` 합니다. 다른 메서드를 사용하면 프로젝트에서 메서드 기호를 찾을 수 있습니다. 이 개체에서 포함된 네임스페이스(있는 경우)와 형식에 대해 추론 할 수 있습니다. 이 예제의 는 `source` 보간된 구멍이 포함된 네임스페이스 및 형식 정보로 채워지는 소스 코드를 템플릿하는 보간된 문자열입니다. `source`는 힌트 이름과 함께 `context`에 추가됩니다. 이 예제에서 생성기는 콘솔 애플리케이션에서 메서드의 `partial` 구현을 포함하는 생성된 새 원본 파일을 만듭니다. 원본 생성기를 작성하여 원하는 원본을 추가할 수 있습니다.

### 💡 팁

`SourceProductionContext.AddSource` 메서드의 `hintName` 매개 변수는 고유한 이름이라면 무엇이든 가능합니다. 대부분의 경우에는 이름에 `".g.cs"` 나 `".generated.cs"` 같은 명시적 C# 파일 확장명을 제공합니다. 파일 이름을 사용 하면 원본으로 생성 중인 파일을 식별할 수 있습니다.

2. 이제 작동하는 생성기가 생겼지만, 이를 콘솔 애플리케이션에 연결해야 합니다. 원래 콘솔 애플리케이션 프로젝트를 편집하고 다음을 추가하여 프로젝트 경로를 위에 서 만든 .NET Standard 프로젝트의 경로로 바꿉니다.

#### XML

```
<!-- Add this as a new ItemGroup, replacing paths and names  
appropriately -->  
<ItemGroup>  
    <ProjectReference Include=".\\PathTo\\SourceGenerator.csproj"  
        OutputItemType="Analyzer"  
        ReferenceOutputAssembly="false" />  
</ItemGroup>
```

이 새 참조는 기존 프로젝트 참조가 아니며 및 `ReferenceOutputAssembly` 특성을 포 함 `OutputItemType` 하도록 수동으로 편집해야 합니다. 의 및 특성에 대한 `OutputItemType` 자세한 내용은 [일반적인 MSBuild 프로젝트 항목: ProjectReference](#) 를 참조하세요. `ProjectReference ReferenceOutputAssembly`

3. 이제 콘솔 애플리케이션을 실행하면 생성된 코드가 실행되어 화면에 출력되는 것을 볼 수 있습니다. 콘솔 애플리케이션 자체는 메서드를 `HelloFrom` 구현하지 않고 원본 생성기 프로젝트에서 컴파일하는 동안 생성된 원본입니다. 다음 텍스트는 애플리케이션의 출력 예제입니다.

#### 콘솔

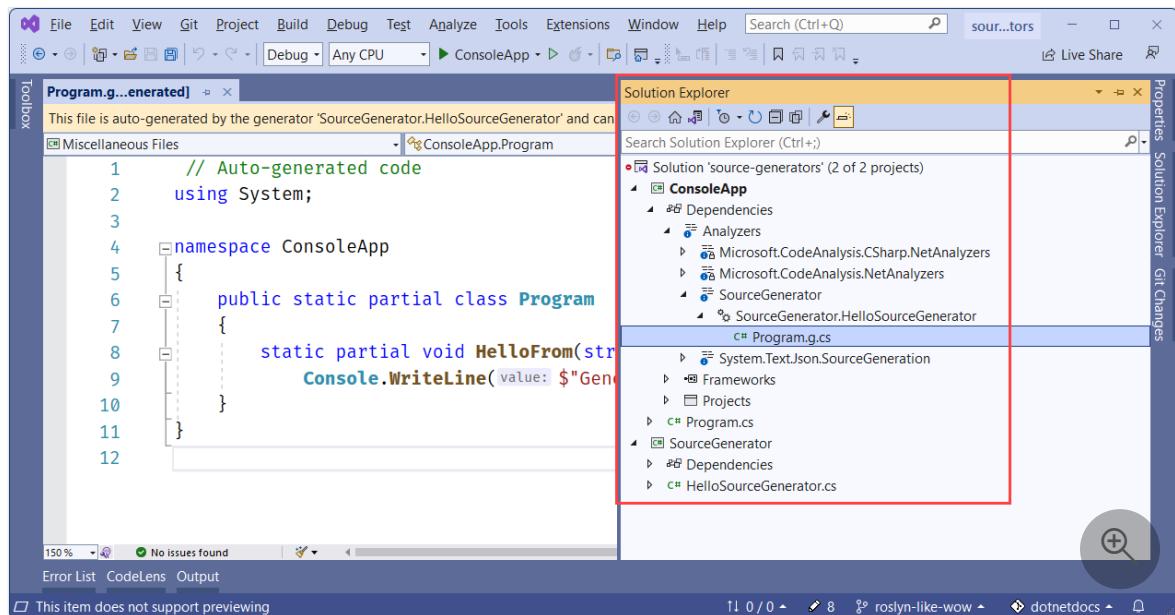
Generator says: Hi from 'Generated Code'

## ① 참고

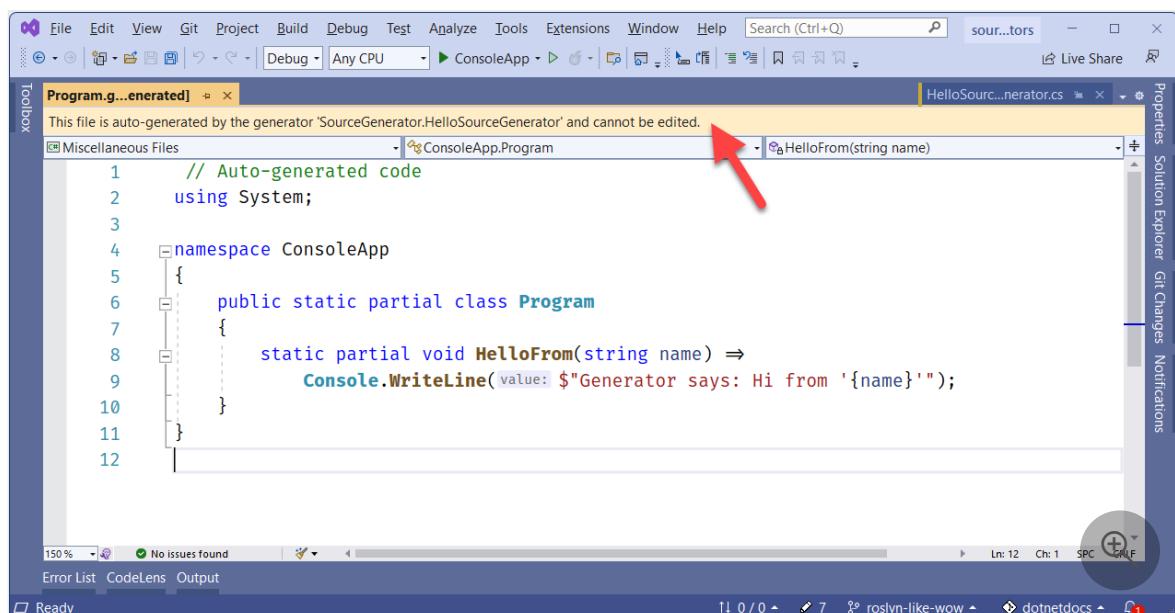
IntelliSense를 확인하고 오류를 제거하려면 Visual Studio를 다시 시작하여 도구 환경을 적극적으로 개선해야 할 수 있습니다.

## 4. Visual Studio 사용하는 경우에는 원본에서 생성한 파일을 볼 수 있습니다. 솔루션 탐색기 창에서 종속성>분석기>

SourceGeneratorSourceGenerator.HelloSourceGenerator>를 확장하고  
Program.g.cs 파일을 두 번 클릭합니다.



이 생성된 파일을 열면 Visual Studio는 파일이 자동으로 생성되고 편집할 수 없음을 나타냅니다.



5. 생성된 파일을 저장하고 생성된 파일이 저장되는 위치를 제어하도록 빌드 속성을 설정할 수도 있습니다. 콘솔 애플리케이션의 프로젝트 파일에서 요소를에 `<PropertyGroup>` 추가하고 `<EmitCompilerGeneratedFiles>` 해당 값을로 `true` 설정합니다. 프로젝트를 다시 빌드합니다. 이제 생성된 파일은 `obj/Debug/net7.0/generated/SourceGenerator/SourceGenerator.HelloSourceGenerator` 아래에 만들어집니다. 경로의 구성 요소는 생성기의 빌드 구성, 대상 프레임워크, 원본 생성기 프로젝트 이름 및 정규화된 형식 이름에 매핑됩니다. 애플리케이션의 프로젝트 파일에 요소를 추가하여 `<CompilerGeneratedFilesOutputPath>` 더 편리한 출력 폴더를 선택할 수 있습니다.

## 다음 단계

원본 생성기 [Cookbook](#)은 이러한 예제 중 일부를 해결하는 몇 가지 권장 방법을 설명합니다. 또한 직접 시도해 볼 수 있는 [GitHub에서 사용할 수 있는 샘플 세트](#)도 있습니다.

다음 문서에서 원본 생성기에 대해 자세히 알아볼 수 있습니다.

- [원본 생성기 디자인 문서](#)
- [원본 생성기 cookbook](#)

# 진단 ID 선택

아티클 • 2024. 03. 12.

진단 ID는 컴파일러 오류 또는 분석기에서 생성된 진단과 같은 특정 진단과 연결된 문자열입니다.

ID는 다음과 같은 다양한 API에서 표시됩니다.

- `DiagnosticDescriptor.Id`
- `ObsoleteAttribute.DiagnosticId`
- `ExperimentalAttribute.DiagnosticId`

진단 ID는 원본(예: `#pragma warning 사용 안 함` 또는 `.editorconfig` 파일)의 식별자로도 사용됩니다.

## 고려 사항

- 진단 ID는 고유해야 함
- 진단 ID는 C#에서 유효한 식별자여야 함
- 진단 ID는 15자 미만이어야 함
- 진단 ID는 `<PREFIX><number>` 형식이어야 함
  - 접두사는 프로젝트에 따라 다름
  - 숫자는 특정 진단을 나타냄

### ① 참고

진단 ID를 변경하는 것은 원본 호환성이 손상되는 변경입니다. ID가 변경되면 기존 제거가 무시되기 때문입니다.

접두사를 두 문자(예: `csxxxx` 및 `caxxxx`)로 제한하지 마세요. 대신 충돌을 방지하려면 더 긴 접두사를 사용합니다. 예를 들어, `System.*` 진단은 `SYSLIB`를 접두사로 사용합니다.

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.



### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

### 설명서 문제 열기

☞ 제품 사용자 의견 제공

# 구문 분석 시작

아티클 • 2023. 04. 08.

이 자습서에서는 **구문 API**를 탐색합니다. 구문 API는 C# 또는 Visual Basic 프로그램을 설명하는 데이터 구조에 대한 액세스를 제공합니다. 이러한 데이터 구조에는 모든 규모의 프로그램을 완벽하게 나타낼 수 있는 충분한 세부 정보가 있습니다. 이러한 구조는 컴파일하고 올바르게 실행하는 전체 프로그램을 설명할 수 있습니다. 또한 편집기에서 작성한 대로 불완전한 프로그램을 설명합니다.

다양한 식을 사용하도록 설정하려면 구문 API를 구성하는 데이터 구조 및 API는 복잡해질 수 밖에 없습니다. 일반적인 "Hello World" 프로그램에 있는 데이터 구조의 모양부터 시작하겠습니다.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;

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

이전 프로그램의 텍스트를 확인합니다. 친숙한 요소를 인식합니다. 전체 텍스트는 단일 원본 파일 또는 **컴파일 단위**를 나타냅니다. 해당 원본 파일의 처음 세 줄은 **using** 지시문입니다. 나머지 원본은 **네임스페이스 선언**에 포함됩니다. 네임스페이스 선언에는 자식 **클래스 선언**이 포함됩니다. 클래스 선언에는 하나의 **메서드 선언**이 포함됩니다.

구문 API는 컴파일 단위를 나타내는 루트를 사용하여 트리 구조를 만듭니다. 트리의 노드는 **using** 지시문, **네임스페이스 선언** 및 프로그램의 다른 모든 요소를 나타냅니다. 트리 구조는 가장 낮은 수준으로 계속됩니다. "헬로 월드!" 문자열은 **인수**의 하위 항목인 **문자열 리터럴 토큰**입니다. 구문 API는 프로그램의 구조에 대한 액세스를 제공합니다. 특정 코드 사례에 대해 쿼리하고, 전체 트리를 통해 코드를 이해하고, 기존 트리를 수정하여 새 트리를 만들 수 있습니다.

간략한 설명을 통해 구문 API를 사용하여 액세스할 수 있는 정보의 종류에 대한 개요를 제공합니다. 구문 API는 C#에서 알아낸 친숙한 코드 구문을 설명하는 공식 API입니다. 줄

바꿈, 공백 및 들여쓰기를 비롯하여 코드의 형식을 지정하는 방법에 대한 정보가 포함된 완전한 기능입니다. 이 정보를 사용하여 코드를 작성한 대로 완벽하게 나타내고 휴먼 프로그래머 또는 컴파일러가 읽을 수 있습니다. 이 구조를 사용하면 의미 있는 수준에서 소스 코드와 상호 작용할 수 있습니다. 더 이상 텍스트 문자열이 아니지만 C# 프로그램의 구조를 나타내는 데이터입니다.

시작하려면 .NET Compiler Platform SDK를 설치해야 합니다.

## 설치 지침 - Visual Studio 설치 관리자

Visual Studio 설치 관리자에서 .NET Compiler Platform SDK를 찾는 두 가지 방법이 있습니다.

### Visual Studio 설치 관리자를 사용한 설치 - 워크로드 보기

.NET Compiler Platform SDK는 Visual Studio 확장 개발 워크로드의 일부로 자동으로 선택되지 않습니다. 선택적 구성 요소로 선택해야 합니다.

1. Visual Studio 설치 관리자를 실행합니다.
2. 수정을 선택합니다.
3. Visual Studio 확장 개발 워크로드를 확인합니다.
4. 요약 트리에서 Visual Studio 확장 개발 노드를 엽니다.
5. .NET Compiler Platform SDK에 대한 확인란을 선택합니다. 선택적 구성 요소 아래에서 마지막에 찾을 수 있습니다.

필요에 따라, 시각화 도우미에서 DGML 편집기에 그래프도 표시할 수 있습니다.

1. 요약 트리에서 개별 구성 요소 노드를 엽니다.
2. DGML 편집기 확인란을 선택합니다.

### Visual Studio 설치 관리자를 사용한 설치 - 개별 구성 요소 탭

1. Visual Studio 설치 관리자를 실행합니다.
2. 수정을 선택합니다.
3. 개별 구성 요소 탭을 선택합니다.
4. .NET Compiler Platform SDK에 대한 확인란을 선택합니다. 컴파일러, 빌드 도구 및 런타임 섹션의 위쪽에서 찾을 수 있습니다.

필요에 따라, 시각화 도우미에서 DGML 편집기에 그래프도 표시할 수 있습니다.

1. DGML 편집기 확인란을 선택합니다. 코드 도구 섹션에서 찾을 수 있습니다.

# 구문 트리 이해

C# 코드 구조의 분석에 구문 API를 사용합니다. **구문 API**는 구문 트리를 분석하고 생성하기 위한 파서, 구문 트리 및 유틸리티를 노출합니다. 그렇게 특정 구문 요소에 대한 코드를 검색하거나 프로그램에 대한 코드를 읽을 수 있습니다.

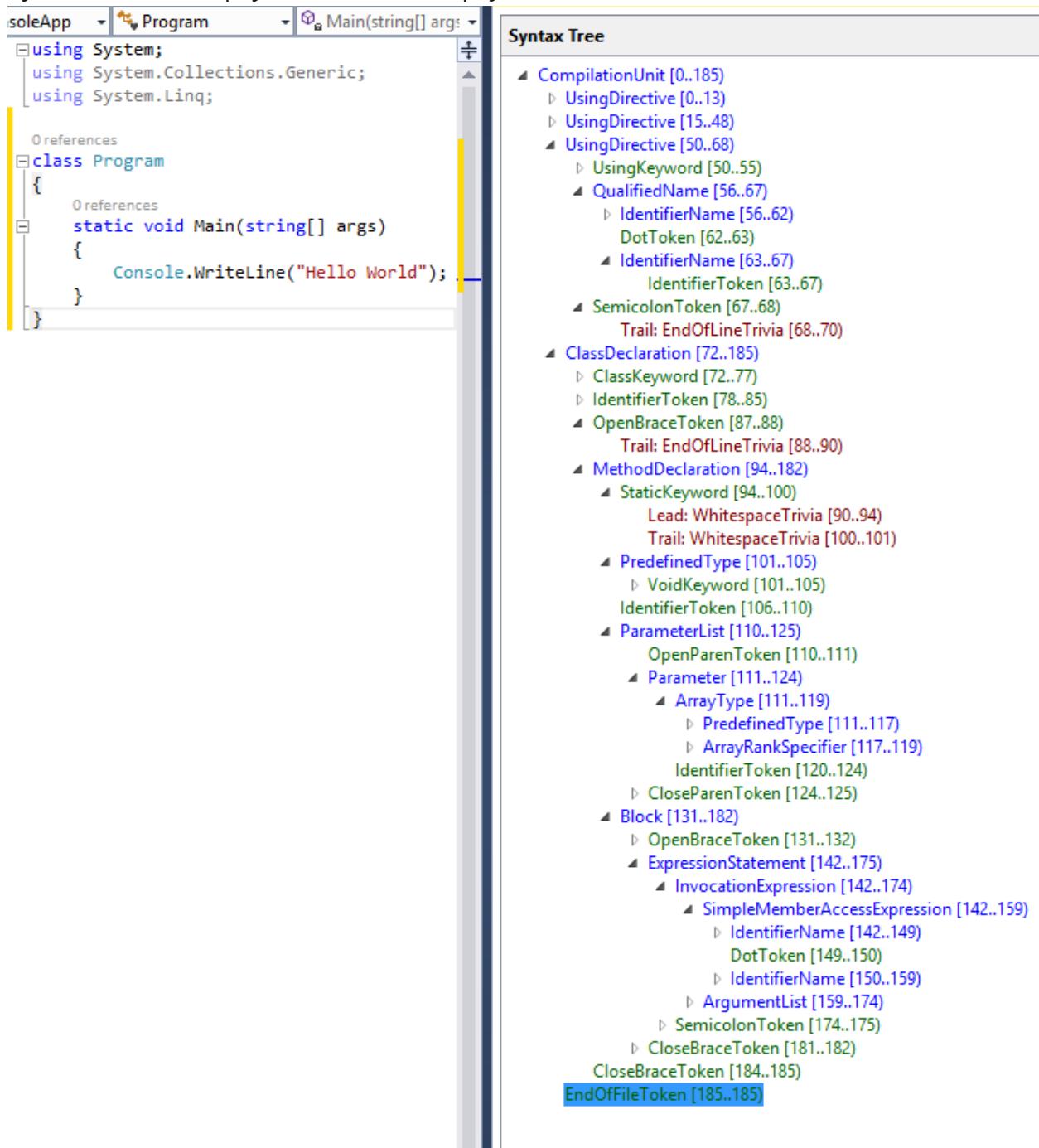
구문 트리는 C# 및 Visual Basic 프로그램을 이해하기 위해 C# 및 Visual Basic 컴파일러에서 사용하는 데이터 구조입니다. 구문 트리는 프로젝트가 빌드되거나 개발자가 F5 키를 누를 때 실행되는 동일한 파서에서 생성됩니다. 구문 트리는 언어를 완전히 제공합니다. 코드 파일의 모든 정보가 트리에 표시됩니다. 텍스트에 대한 구문 트리를 작성하면 구문 분석된 정확한 원본 텍스트를 재현합니다. 또한 구문 트리는 **변경할 수 없습니다**. 만들어진 구문 트리는 변경할 수 없습니다. 트리의 소비자는 잠금 또는 기타 동시성 조치 없이 여러 스레드에서 트리를 분석할 수 있으며 데이터가 바뀌지 않는다는 점을 인식합니다. API를 사용하여 기존 트리를 수정한 결과로 나타난 새 트리를 만들 수 있습니다.

구문 트리의 네 가지 기본 구성 요소는 다음과 같습니다.

- [Microsoft.CodeAnalysis.SyntaxTree](#) 클래스는 전체 구문 분석 트리를 나타내는 인스턴스입니다. [SyntaxTree](#)은 언어별 파생물을 포함하는 추상 클래스입니다. [Microsoft.CodeAnalysis.CSharp.CSharpSyntaxTree](#)(또는 [Microsoft.CodeAnalysis.VisualBasic.VisualBasicSyntaxTree](#)) 클래스의 구문 분석 메서드를 사용하여 C#(또는 Visual Basic)에서 텍스트를 구문 분석합니다.
- [Microsoft.CodeAnalysis\\_SyntaxNode](#) 클래스는 선언, 명령문, 절 및 식과 같은 구문 구조를 나타내는 인스턴스입니다.
- [Microsoft.CodeAnalysis\\_SyntaxToken](#) 구조는 개별 키워드, 식별자, 연산자 또는 문장 부호를 나타냅니다.
- 마지막으로 [Microsoft.CodeAnalysis\\_SyntaxTrivia](#) 구조는 토큰 간의 공백, 전처리 지시문 및 주석 등의 구문상으로 중요하지 않은 정보를 나타냅니다.

퀴즈, 토큰 및 노드는 Visual Basic 또는 C# 코드의 일부에 있는 모든 항목을 완전히 나타내는 트리를 형성하기 위해 계층적으로 구성됩니다. **구문 시각화 도우미** 창을 사용하여 이 구조를 확인할 수 있습니다. Visual Studio에서 **보기>다른 창>구문 시각화 도우미**를 선택합니다. 예를 들어 **구문 시각화 도우미**를 사용하여 검사된 위의 C# 원본 파일은 다음 그림처럼 표시됩니다.

SyntaxNode: Blue | SyntaxToken: 녹색 | SyntaxTrivia: Red



이 트리 구조를 탐색하여 코드 파일에서 문, 식, 토큰 또는 공백을 찾을 수 있습니다.

구문 API를 사용하여 코드 파일에서 무언가 찾을 수 있지만 대부분의 시나리오에는 작은 코드 조각을 검사하거나 특정 문 또는 조각을 검색하는 작업이 포함됩니다. 이후의 두 가지 예제에서는 코드의 구조를 찾거나 단일 문을 검색하는 일반적인 사용법을 보여줍니다.

## 트리 트래버스

두 가지 방법으로 구문 트리에서 노드를 검사할 수 있습니다. 트리를 트래버스하여 각 노드를 검사하거나 특정 요소 또는 노드에 대해 쿼리할 수 있습니다.

# 수동 트래버스

[GitHub 리포지토리](#)에서 이 샘플의 완성된 코드를 볼 수 있습니다.

## ① 참고

구문 트리 형식은 상속을 사용하여 프로그램의 여러 위치에서 유효한 다른 구문 요소를 설명합니다. 종종 이러한 API를 사용하면 속성이나 컬렉션 멤버를 파생된 특정 형식에 캐스팅하게 됩니다. 다음 예제에서 할당 및 캐스팅은 명시적으로 형식화된 변수를 사용하는 별도의 문입니다. API의 반환 형식 및 반환되는 개체의 런타임 형식을 확인하기 위해 코드를 읽을 수 있습니다. 이 연습에서는 암시적으로 형식화된 변수를 사용하고 API 이름을 사용하여 검사된 개체의 형식을 설명하는 것이 더 일반적입니다.

새 C# 독립 실행형 코드 분석 도구 프로젝트를 만듭니다.

- Visual Studio에서 파일>새로 만들기>프로젝트를 선택하여 새 프로젝트 대화 상자를 표시합니다.
- Visual C#>확장성 아래에서 독립 실행형 코드 분석 도구를 선택합니다.
- 프로젝트 이름을 "SyntaxTreeManualTraversal"이라고 지정하고 확인을 클릭합니다.

앞에서 보여 준 기본 "헬로 월드!" 프로그램을 분석합니다. Hello World 프로그램의 텍스트를 Program 클래스의 상수로 추가합니다.

C#

```
const string programText =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

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

다음으로 `programText` 상수에서 코드 텍스트의 구문 트리를 빌드하는 다음 코드를 추가합니다. `Main` 메서드에 다음 줄을 추가합니다.

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

해당 두 줄은 트리를 만들고 해당 트리의 루트 노드를 검색합니다. 이제 트리에서 노드를 검사할 수 있습니다. 다음과 같은 줄을 `Main` 메서드에 추가하여 트리에서 루트 노드 속성 중 일부를 표시합니다.

C#

```
WriteLine($"The tree is a {root.Kind()} node.");
WriteLine($"The tree has {root.Members.Count} elements in it.");
WriteLine($"The tree has {root.Usings.Count} using statements. They are:");
foreach (UsingDirectiveSyntax element in root.Usings)
    WriteLine($"{element.Name}");
```

애플리케이션을 실행하여 코드가 이 트리에서 루트 노드에 대해 검색한 내용을 확인합니다.

일반적으로 코드에 대해 자세히 알아보려면 트리를 탐색합니다. 이 예제에서는 API를 탐색하기 위해 알아야 하는 코드를 분석합니다. 다음 코드를 추가하여 `root` 노드의 첫 번째 멤버를 검사합니다.

C#

```
MemberDeclarationSyntax firstMember = root.Members[0];
WriteLine($"The first member is a {firstMember.Kind()}.");
var helloWorldDeclaration = (NamespaceDeclarationSyntax)firstMember;
```

해당 멤버는 [Microsoft.CodeAnalysis.CSharp.Syntax.NamespaceDeclarationSyntax](#)입니다. `namespace HelloWorld` 선언 범위에 있는 모든 항목을 나타냅니다. 다음 코드를 추가하여 노드가 `HelloWorld` 네임스페이스 내에서 선언된 내용을 검사합니다.

C#

```
WriteLine($"There are {helloWorldDeclaration.Members.Count} members declared
in this namespace.");
WriteLine($"The first member is a
{helloWorldDeclaration.Members[0].Kind()}.");
```

배운 내용을 확인하기 위해 프로그램을 실행합니다.

이제 선언이 `Microsoft.CodeAnalysis.CSharp.Syntax.ClassDeclarationSyntax`임을 알았으므로 해당 형식의 새로운 변수를 선언하여 클래스 선언을 검사합니다. 이 클래스에는 `Main` 메서드라는 하나의 멤버만이 포함됩니다. 다음 코드를 추가하여 `Main` 메서드를 찾고 `Microsoft.CodeAnalysis.CSharp.Syntax.MethodDeclarationSyntax`로 캐스팅합니다.

C#

```
var programDeclaration =
(ClassDeclarationSyntax)helloWorldDeclaration.Members[0];
WriteLine($"There are {programDeclaration.Members.Count} members declared in
the {programDeclaration.Identifier} class.");
WriteLine($"The first member is a {programDeclaration.Members[0].Kind()}.");
var mainDeclaration =
(MethodDeclarationSyntax)programDeclaration.Members[0];
```

메서드 선언 노드에는 메서드에 대한 모든 구문 정보가 포함됩니다. `Main` 메서드의 반환 형식, 인수의 수와 형식 및 메서드의 본문 텍스트를 표시하겠습니다. 다음 코드를 추가합니다.

C#

```
WriteLine($"The return type of the {mainDeclaration.Identifier} method is
{mainDeclaration.ReturnType}.");
WriteLine($"The method has {mainDeclaration.ParameterList.Parameters.Count}
parameters.");
foreach (ParameterSyntax item in mainDeclaration.ParameterList.Parameters)
    WriteLine($"The type of the {item.Identifier} parameter is
{item.Type}.");
WriteLine($"The body text of the {mainDeclaration.Identifier} method
follows:");
WriteLine(mainDeclaration.Body?.ToFullString());

var argsParameter = mainDeclaration.ParameterList.Parameters[0];
```

프로그램을 실행하여 이 프로그램에 대해 알게 된 모든 정보를 확인합니다.

text

```
The tree is a CompilationUnit node.
The tree has 1 elements in it.
The tree has 4 using statements. They are:
    System
    System.Collections
    System.Linq
    System.Text
The first member is a NamespaceDeclaration.
There are 1 members declared in this namespace.
The first member is a ClassDeclaration.
There are 1 members declared in the Program class.
```

```
The first member is a MethodDeclaration.  
The return type of the Main method is void.  
The method has 1 parameters.  
The type of the args parameter is string[].  
The body text of the Main method follows:  
{  
    Console.WriteLine("Hello, World!");  
}
```

## 쿼리 메서드

트리를 트래버스하는 것 외에도 [Microsoft.CodeAnalysis.SyntaxNode](#)에 정의된 쿼리 메서드를 사용하여 구문 트리를 탐색할 수 있습니다. 이러한 메서드는 XPath에 익숙한 사용자라면 누구나 즉시 익숙해질 것입니다. LINQ에서 이러한 메서드를 사용하여 트리에서 신속히 작업할 수 있습니다. [SyntaxNode](#)에는 [DescendantNodes](#), [AncestorsAndSelf](#) 및 [ChildNodes](#) 등의 쿼리 메서드가 있습니다.

이러한 쿼리 메서드를 사용하여 트리를 탐색하는 대신 `Main` 메서드에 대한 인수를 찾을 수 있습니다. `Main` 메서드의 맨 아래에 다음 코드를 추가합니다.

C#

```
var firstParameters = from methodDeclaration in root.DescendantNodes()  
                      .OfType<MethodDeclarationSyntax>()  
                      where methodDeclaration.Identifier.ValueText == "Main"  
                      select  
methodDeclaration.ParameterList.Parameters.First();  
  
var argsParameter2 = firstParameters.Single();  
  
WriteLine(argsParameter == argsParameter2);
```

첫 번째 문은 LINQ 식 및 [DescendantNodes](#) 메서드를 사용하여 앞의 예제와 동일한 매개 변수를 찾습니다.

프로그램을 실행하고, LINQ 식이 트리를 수동으로 탐색할 때와 동일한 매개 변수를 찾았는지 확인할 수 있습니다.

이 샘플에서는 `WriteLine` 문을 사용하여 트래버스한 대로 구문 트리에 대한 정보를 표시합니다. 디버거에서 완성된 프로그램을 실행하여 자세히 알아볼 수 있습니다. Hello World 프로그램에서 만든 구문 트리의 일부인 속성 및 메서드를 자세히 검사할 수 있습니다.

## 구문 워커

구문 트리에서 특정 종류의 모든 노드를 찾을 수도 있습니다(예: 파일의 모든 속성 선언). [Microsoft.CodeAnalysis.CSharp.CSharpSyntaxWalker](#) 클래스를 확장하고 [VisitPropertyDeclaration\(PropertyDeclarationSyntax\)](#) 메서드를 재정의하여 해당 구조를 미리 알지 못해도 구문 트리에서 모든 속성 선언을 처리할 수 있습니다. [CSharpSyntaxWalker](#)는 노드와 해당 자식 항목을 재귀적으로 방문하는 특정 종류의 [CSharpSyntaxVisitor](#)입니다.

이 예제에서는 구문 트리를 검사하는 [CSharpSyntaxWalker](#)를 구현합니다. `System` 네임스페이스를 가져오지 않는 `using` 지시문을 찾아 수집합니다.

새 C# 독립 실행형 코드 분석 도구 프로젝트를 만들고 이름을 "SyntaxWalker"로 지정합니다.

[GitHub 리포지토리](#)에서 이 샘플의 완성된 코드를 볼 수 있습니다. GitHub의 샘플에는 이 자습서에 설명된 모든 프로젝트가 포함됩니다.

앞의 예제와 같이 분석하려는 프로그램의 텍스트를 포함하도록 문자열 상수를 정의할 수 있습니다.

```
C#  
  
    const string programText =  
    @"using System;  
    using System.Collections.Generic;  
    using System.Linq;  
    using System.Text;  
    using Microsoft.CodeAnalysis;  
    using Microsoft.CodeAnalysis.CSharp;  
  
    namespace TopLevel  
{  
        using Microsoft;  
        using System.ComponentModel;  
  
        namespace Child1  
{  
            using Microsoft.Win32;  
            using System.Runtime.InteropServices;  
  
            class Foo { }  
        }  
  
        namespace Child2  
{  
            using System.CodeDom;  
            using Microsoft.CSharp;  
  
            class Bar { }  
        }  
    }";
```

이 원본 텍스트에는 파일 수준, 최상위 네임스페이스 및 두 개의 중첩된 네임스페이스와 같은 네 가지 위치에 분산된 `using` 지시문이 포함됩니다. 이 예제에서는 코드를 쿼리하는 `CSharpSyntaxWalker` 클래스를 사용하는 핵심 시나리오를 강조 표시합니다. `using` 선언을 찾기 위해 루트 구문 트리에서 모든 노드를 방문하기는 번거롭습니다. 대신, 파생된 클래스를 만들고 트리의 현재 노드가 `using` 지시문인 경우에만 호출되는 메서드를 재정의합니다. 방문자는 다른 노드 형식에서 작업을 수행하지 않습니다. 이 단일 메서드는 각 `using` 문을 검사하고 `System` 네임스페이스에 위치하지 않는 네임스페이스의 컬렉션을 빌드합니다. 모든 `using` 문이 아닌 `using` 문을 검사하는 `CSharpSyntaxWalker`를 빌드합니다.

이제 프로그램 텍스트를 정의했으므로 `SyntaxTree`를 만들고 해당 트리의 루트를 가져와야 합니다.

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

다음으로 새 클래스를 만듭니다. Visual Studio에서 **프로젝트>새 항목 추가**를 선택합니다. **새 항목 추가** 대화 상자에서 파일 이름으로 `UsingCollector.cs`를 입력합니다.

`UsingCollector` 클래스에서 `using` 방문자 기능을 구현합니다. `CSharpSyntaxWalker`에서 파생된 `UsingCollector` 클래스를 만들기 시작합니다.

C#

```
class UsingCollector : CSharpSyntaxWalker
```

수집 중인 네임스페이스 노드를 포함하는 스토리지가 있어야 합니다. `UsingCollector` 클래스에서 공용 읽기 전용 속성을 선언합니다. 이 변수를 사용하여 찾은 `UsingDirectiveSyntax` 노드를 저장합니다.

C#

```
public ICollection<UsingDirectiveSyntax> Usings { get; } = new
List<UsingDirectiveSyntax>();
```

`CSharpSyntaxWalker` 기본 클래스는 구문 트리에서 각 노드를 방문하는 논리를 구현합니다. 파생된 클래스는 관심이 있는 특정 노드에 호출되는 메서드를 재정의합니다. 이 경우에 `using` 지시문을 사용합니다. 즉, `VisitUsingDirective(UsingDirectiveSyntax)` 메서드를 재정의해야 합니다. 이 메서드에 대한 인수는

`Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax` 개체입니다. 방문자를 사용하는 중요한 장점은 특정 노드 형식에 캐스팅된 인수를 사용하여 재정의된 메서드를 호출한다는 것입니다. `Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax` 클래스에는 가져온 네임스페이스의 이름을 저장하는 `Name` 속성이 있습니다.

`Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax`입니다.

`VisitUsingDirective(UsingDirectiveSyntax)` 재정의에서 다음 코드를 추가합니다.

C#

```
public override void VisitUsingDirective(UsingDirectiveSyntax node)
{
    WriteLine($"\\tVisitUsingDirective called with {node.Name}.");
    if (node.Name.ToString() != "System" &&
        !node.Name.ToString().StartsWith("System."))
    {
        WriteLine($"\\t\\tSuccess. Adding {node.Name}.");
        this.Usings.Add(node);
    }
}
```

이전 예제에서 다양한 `WriteLine` 문을 추가하여 이 메서드를 이해할 수 있었습니다. 호출 시기 및 이 때 전달된 인수를 확인할 수 있습니다.

마지막으로 `UsingCollector`를 만들고 루트 노드를 방문하게 만드는 두 개의 코드 줄을 추가해야 합니다. 그러면 모든 `using` 문을 수집합니다. 그런 다음, `foreach` 루프를 추가하여 수집기에서 찾은 `using` 문을 모두 표시합니다.

C#

```
var collector = new UsingCollector();
collector.Visit(root);
foreach (var directive in collector.Usings)
{
    WriteLine(directive.Name);
}
```

프로그램을 컴파일하고 실행합니다. 다음과 같은 내용이 출력됩니다.

콘솔

```
VisitUsingDirective called with System.
VisitUsingDirective called with System.Collections.Generic.
VisitUsingDirective called with System.Linq.
VisitUsingDirective called with System.Text.
VisitUsingDirective called with Microsoft.CodeAnalysis.
    Success. Adding Microsoft.CodeAnalysis.
VisitUsingDirective called with Microsoft.CodeAnalysis.CSharp.
```

```
Success. Adding Microsoft.CodeAnalysis.CSharp.  
VisitUsingDirective called with Microsoft.  
    Success. Adding Microsoft.  
VisitUsingDirective called with System.ComponentModel.  
VisitUsingDirective called with Microsoft.Win32.  
    Success. Adding Microsoft.Win32.  
VisitUsingDirective called with System.Runtime.InteropServices.  
VisitUsingDirective called with System.CodeDom.  
VisitUsingDirective called with Microsoft.CSharp.  
    Success. Adding Microsoft.CSharp.  
Microsoft.CodeAnalysis  
Microsoft.CodeAnalysis.CSharp  
Microsoft  
Microsoft.Win32  
Microsoft.CSharp  
Press any key to continue . . .
```

지금까지 **구문 API**를 사용하여 C# 소스 코드에서 특정 종류의 C# 문 및 선언을 찾았습니다.

# 의미 체계 분석 시작

아티클 • 2023. 05. 10.

이 자습서는 사용자가 구문 API에 익숙하다고 가정합니다. [구문 분석 작업 시작](#) 아티클에서는 소개를 충분히 설명합니다.

이 자습서에서는 **기호 및 바인딩 API**를 탐색합니다. 이러한 API는 프로그램의 의미 체계에 대한 정보를 제공합니다. 이를 통해 프로그램에서 기호를 나타내는 형식에 대해 질문하고 대답할 수 있습니다.

.NET Compiler Platform SDK를 설치해야 합니다.

## 설치 지침 - Visual Studio 설치 관리자

Visual Studio 설치 관리자에서 .NET Compiler Platform SDK를 찾는 두 가지 방법이 있습니다.

### Visual Studio 설치 관리자를 사용한 설치 - 워크로드 보기

.NET Compiler Platform SDK는 Visual Studio 확장 개발 워크로드의 일부로 자동으로 선택되지 않습니다. 선택적 구성 요소로 선택해야 합니다.

1. Visual Studio 설치 관리자를 실행합니다.
2. **수정**을 선택합니다.
3. **Visual Studio 확장 개발** 워크로드를 확인합니다.
4. 요약 트리에서 **Visual Studio 확장 개발** 노드를 엽니다.
5. .NET Compiler Platform SDK에 대한 확인란을 선택합니다. 선택적 구성 요소 아래에서 마지막에 찾을 수 있습니다.

필요에 따라, 시각화 도우미에서 **DGML 편집기**에 그래프도 표시할 수 있습니다.

1. 요약 트리에서 **개별 구성 요소** 노드를 엽니다.
2. **DGML 편집기** 확인란을 선택합니다.

### Visual Studio 설치 관리자를 사용한 설치 - 개별 구성 요소 탭

1. Visual Studio 설치 관리자를 실행합니다.
2. **수정**을 선택합니다.
3. **개별 구성 요소** 탭을 선택합니다.

4. .NET Compiler Platform SDK에 대한 확인란을 선택합니다. 컴파일러, 빌드 도구 및 런타임 섹션의 위쪽에서 찾을 수 있습니다.

필요에 따라, 시각화 도우미에서 DGML 편집기에 그래프도 표시할 수 있습니다.

1. DGML 편집기 확인란을 선택합니다. 코드 도구 섹션에서 찾을 수 있습니다.

## 컴파일 및 기호 이해

.NET Compiler SDK에서 작업하게 되면 구문 API와 의미 체계 API 간의 차이점을 이해할 수 있게 됩니다. 구문 API를 사용하면 프로그램의 구조를 볼 수 있습니다. 그러나 프로그램의 의미 체계 또는 의미에 대해 더 다양한 정보가 필요할 수 있습니다. 느슨한 코드 파일 또는 Visual Basic 또는 C#의 코드 조각은 격리에서 구문을 분석할 수 있습니다. "이 변수의 형식이란?"과 같은 질문은 의미가 없습니다. 형식 이름의 의미는 어셈블리 참조, 네임스페이스 가져오기 또는 기타 코드 파일에 종속될 수 있습니다. 의미 체계 API, 특히 [Microsoft.CodeAnalysis.Compilation](#) 클래스를 사용하여 해당 질문에 대답합니다.

[Compilation](#)의 인스턴스는 컴파일러에서 보는 단일 프로젝트와 유사하고, Visual Basic 또는 C# 프로그램을 컴파일하는 데 필요한 모든 항목을 나타냅니다. 컴파일에는 컴파일할 원본 파일, 어셈블리 참조 및 컴파일러 옵션의 집합이 포함됩니다. 이 컨텍스트에서 다른 모든 정보를 사용하여 코드의 의미에 대해 추정할 수 있습니다. [Compilation](#)을 사용하면 이름 및 다른 식이 참조하는 형식, 네임스페이스, 멤버 및 변수 등의 엔터티인 기호를 찾을 수 있습니다. 기호를 사용하여 이름 및 식을 연결하는 프로세스를 **바인딩**이라고 합니다.

[Microsoft.CodeAnalysis.SyntaxTree](#)과 마찬가지로 [Compilation](#)은 언어별 파생물을 포함하는 추상 클래스입니다. 컴파일의 인스턴스를 만들 때 [Microsoft.CodeAnalysis.CSharp.CSharpCompilation](#)(또는 [Microsoft.CodeAnalysis.VisualBasic.VisualBasicCompilation](#)) 클래스에서 팩터리 메서드를 호출해야 합니다.

## 기호 쿼리

이 자습서에서는 "Hello World" 프로그램을 다시 확인합니다. 이번에는 프로그램의 기호를 쿼리하여 해당 기호가 나타내는 형식을 이해합니다. 네임스페이스의 형식에 대해 쿼리하고 형식에 사용할 수 있는 메서드를 찾는 방법을 알아봅니다.

[GitHub 리포지토리](#)에서 이 샘플의 완성된 코드를 볼 수 있습니다.

### ① 참고

구문 트리 형식은 상속을 사용하여 프로그램의 여러 위치에서 유효한 다른 구문 요소를 설명합니다. 종종 이러한 API를 사용하면 속성이나 컬렉션 멤버를 파생된 특정

형식에 캐스팅하게 됩니다. 다음 예제에서 할당 및 캐스팅은 명시적으로 형식화된 변수를 사용하는 별도의 문입니다. API의 반환 형식 및 반환되는 개체의 런타임 형식을 확인하기 위해 코드를 읽을 수 있습니다. 이 연습에서는 암시적으로 형식화된 변수를 사용하고 API 이름을 사용하여 검사된 개체의 형식을 설명하는 것이 더 일반적입니다.

새 C# 독립 실행형 코드 분석 도구 프로젝트를 만듭니다.

- Visual Studio에서 파일>새로 만들기>프로젝트를 선택하여 새 프로젝트 대화 상자를 표시합니다.
- Visual C#>확장성 아래에서 독립 실행형 코드 분석 도구를 선택합니다.
- 프로젝트 이름을 "SemanticQuickStart"로 지정하고 확인을 클릭합니다.

앞에서 보여 준 기본 "헬로 월드!" 프로그램을 분석합니다. Hello World 프로그램의 텍스트를 Program 클래스의 상수로 추가합니다.

C#

```
const string programText =
@"using System;
using System.Collections.Generic;
using System.Text;

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

다음으로 programText 상수에서 코드 텍스트의 구문 트리를 빌드하는 다음 코드를 추가합니다. Main 메서드에 다음 줄을 추가합니다.

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);

CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

다음으로 이미 만든 트리에서 CSharpCompilation을 빌드합니다. "Hello World" 샘플은 String 및 Console 형식으로 사용합니다. 컴파일에서 두 가지 해당 형식을 선언하는 어셈

블리를 참조해야 합니다. 다음 줄을 `Main` 메서드에 추가하여 적절한 어셈블리에 대한 참조를 비롯한 구문 트리의 컴파일을 만듭니다.

C#

```
var compilation = CSharpCompilation.Create("HelloWorld")
    .AddReferences(MetadataReference.CreateFromFile(
        typeof(string).Assembly.Location))
    .AddSyntaxTrees(tree);
```

`CSharpCompilation.AddReferences` 메서드는 컴파일에 참조를 추가합니다.  
`MetadataReference.CreateFromFile` 메서드는 어셈블리를 참조로 로드합니다.

## 의미 체계 모델 쿼리

`Compilation`이 있다면 `SemanticModel`에 대해 해당 `Compilation`에 포함된 `SyntaxTree`를 요청할 수 있습니다. 일반적으로 모든 정보의 원본을 IntelliSense에서 가져오는 경우 의미 체계 모델을 고려할 수 있습니다. 은 `SemanticModel` "이 위치에 scope 있는 이름은 무엇인가요?", "이 메서드에서 액세스할 수 있는 멤버는 무엇인가요?", "이 텍스트 블록에서 사용되는 변수는 무엇인가요?", "이 이름/식은 무엇을 참조하나요?"와 같은 질문에 대답할 수 있습니다. 의미 체계 모델을 만들려면 다음 문을 추가합니다.

C#

```
SemanticModel model = compilation.GetSemanticModel(tree);
```

## 이름 바인딩

는 `Compilation` 에서 를 `SemanticModelSyntaxTree`만듭니다. 모델을 만든 후에 쿼리하여 첫 번째 `using` 지시문을 찾고 `System` 네임스페이스에 대한 기호 정보를 검색할 수 있습니다. `Main` 메서드에 두 줄을 추가하여 의미 체계 모델을 만들고 첫 번째 `using` 문에 대한 기호를 검색합니다.

C#

```
// Use the syntax tree to find "using System;"  
UsingDirectiveSyntax usingSystem = root.Usings[0];  
NameSyntax systemName = usingSystem.Name;  
  
// Use the semantic model for symbol information:  
SymbolInfo nameInfo = model.GetSymbolInfo(systemName);
```

위의 코드는 첫 번째 `using` 지시문의 이름을 바인딩하여 `System` 네임스페이스에서 `Microsoft.CodeAnalysis.SymbolInfo`을 검색하는 방법을 보여 줍니다. 또한 위의 코드에서는 **구문 모델**을 사용하여 코드의 구조를 찾는 것을 설명합니다. **의미 체계 모델**을 사용하여 해당 의미를 이해합니다. **구문 모델**은 `using` 문에서 `System` 문자열을 찾습니다. **의미 체계 모델**에는 `System` 네임스페이스에서 정의된 형식에 대한 모든 정보가 있습니다.

`SymbolInfo` 개체에서 `SymbolInfo.Symbol` 속성을 사용하여 `Microsoft.CodeAnalysis.ISymbol`을 가져올 수 있습니다. 이 속성은 이 식에서 참조하는 기호를 반환합니다. 아무것도 참조하지 않은 식(예: 숫자 리터럴)의 경우 이 속성은 `null`입니다. `SymbolInfo.Symbol`이 `null`이 아니면 `ISymbol.Kind`은 기호의 형식을 나타냅니다. 다음 예제에서 `ISymbol.Kind` 속성은 `SymbolKind.Namespace`입니다. `Main` 메서드에 다음 코드를 추가합니다. `System` 네임스페이스에 대한 기호를 검색한 다음, `System` 네임스페이스에 선언된 모든 자식 네임스페이스를 표시합니다.

C#

```
var systemSymbol = (INamespaceSymbol?)nameInfo.Symbol;
if (systemSymbol?.GetNamespaceMembers() is not null)
{
    foreach (INamespaceSymbol ns in systemSymbol?.GetNamespaceMembers()!)
    {
        Console.WriteLine(ns);
    }
}
```

프로그램을 실행하고 다음과 같은 출력이 표시됩니다.

출력

```
System.Collections
System.Configuration
System.Deployment
System.Diagnostics
System.Globalization
System.IO
System.Numerics
System.Reflection
System.Resources
System.Runtime
System.Security
System.StubHelpers
System.Text
System.Threading
Press any key to continue . . .
```

① 참고

출력에는 `System` 네임스페이스의 자식 네임스페이스인 모든 네임스페이스가 포함되지 않습니다. 이 컴파일에서 나타나는 모든 네임스페이스가 표시됩니다. 여기서는 `System.String`이 선언하는 어셈블리만을 참조합니다. 다른 어셈블리에 선언된 모든 네임스페이스가 이 컴파일에 알려지지 않았습니다.

## 식 바인딩

위의 코드에서는 이름에 바인딩하여 기호를 찾는 방법을 보여줍니다. 바인딩될 수 있는 C# 프로그램에 이름이 아닌 다른 식이 있습니다. 이 기능을 보여주기 위해 간단한 문자열리터럴에 대한 바인딩에 액세스하겠습니다.

"헬로 월드" 프로그램에는 콘솔에

`Microsoft.CodeAnalysis.CSharp.Syntax.LiteralExpressionSyntax` 표시되는 "Hello, World!" 문자열이 포함되어 있습니다.

프로그램에서 단일 문자열 리터럴을 찾아 "Hello, World!" 문자열을 찾습니다. 그런 다음, 구문 노드를 찾으면 의미 체계 모델에서 노드의 형식 정보를 가져옵니다. `Main` 메서드에 다음 코드를 추가합니다.

C#

```
// Use the syntax model to find the literal string:  
LiteralExpressionSyntax helloWorldString = root.DescendantNodes()  
.OfType<LiteralExpressionSyntax>()  
.Single();  
  
// Use the semantic model for type information:  
TypeInfo literalInfo = model.GetTypeInfo(helloWorldString);
```

`Microsoft.CodeAnalysis.TypeInfo` 구조체에는 리터럴 형식에 대한 의미 체계 정보에 액세스할 수 있는 `TypeInfo.Type` 속성이 포함됩니다. 이 예제에서는 `string` 형식입니다. 이 속성을 지역 변수에 할당하는 선언을 추가합니다.

C#

```
var stringTypeSymbol = (INamedTypeSymbol?)literalInfo.Type;
```

이 자습서를 완료하려면 `string`을 반환하는 `string` 형식에 선언된 모든 공용 메서드의 시퀀스를 생성하는 LINQ 쿼리를 빌드하겠습니다. 이 쿼리가 복잡해집니다. 따라서 한 줄씩 빌드한 다음, 단일 쿼리로 다시 생성합니다. 이 쿼리의 원본은 `string` 형식에 선언된 모든 멤버의 시퀀스입니다.

C#

```
var allMembers = stringTypeSymbol?.GetMembers();
```

해당 소스 시퀀스에는 속성 및 필드를 비롯한 모든 멤버가 포함됩니다. 따라서 [Microsoft.CodeAnalysis.IMethodSymbol](#) 개체인 요소를 찾기 위해 [ImmutableArray<T>.OfType](#) 메서드를 사용하여 필터링합니다.

C#

```
var methods = allMembers?.OfType<IMethodSymbol>();
```

다음으로 공용이며 `string`을 반환하는 해당 메서드만을 반환하는 다른 필터를 추가합니다.

C#

```
var publicStringReturningMethods = methods?
    .Where(m => SymbolEqualityComparer.Default.Equals(m.ReturnType,
stringTypeSymbol) &&
    m.DeclaredAccessibility == Accessibility.Public);
```

이름 속성만을 선택하고, 오버로드를 제거하여 고유 이름만을 선택합니다.

C#

```
var distinctMethods = publicStringReturningMethods?.Select(m =>
m.Name).Distinct();
```

LINQ 쿼리 구문을 사용하여 전체 쿼리를 빌드한 다음 콘솔에 모든 메서드 이름을 표시할 수도 있습니다.

C#

```
foreach (string name in (from method in stringTypeSymbol?
    .GetMembers().OfType<IMethodSymbol>()
    where
SymbolEqualityComparer.Default.Equals(method.ReturnType, stringTypeSymbol)
&&
method.DeclaredAccessibility ==
Accessibility.Public
select method.Name).Distinct())
{
    Console.WriteLine(name);
}
```

프로그램을 빌드하고 실행합니다. 다음 출력이 표시됩니다.

## 출력

```
Join
Substring
Trim
TrimStart
TrimEnd
Normalize
PadLeft
PadRight
ToLower
ToLowerInvariant
ToUpper
ToUpperInvariant
ToString
Insert
Replace
Remove
Format
Copy
Concat
Intern
IsInterned
Press any key to continue . . .
```

이 프로그램의 일부인 기호에 대한 정보를 찾고 표시하기 위해 의미 체계 API를 사용했습니다.

# 구문 변환 시작

아티클 • 2023. 04. 08.

이 자습서는 구문 분석 시작 및 의미 체계 분석 시작 빠른 시작에서 살펴본 개념과 기술을 기반으로 구성됩니다. 아직 완료하지 않은 경우 이 자습서를 시작하기 전에 해당 빠른 시작을 완료해야 합니다.

이 빠른 시작에서는 구문 트리를 만들고 변환하는 기술을 살펴봅니다. 이전 빠른 시작에서 알아본 기술을 함께 사용하여 첫 번째 명령줄 리팩터링을 만듭니다.

## 설치 지침 - Visual Studio 설치 관리자

Visual Studio 설치 관리자에서 .NET Compiler Platform SDK를 찾는 두 가지 방법이 있습니다.

### Visual Studio 설치 관리자를 사용한 설치 - 워크로드 보기

.NET Compiler Platform SDK는 Visual Studio 확장 개발 워크로드의 일부로 자동으로 선택되지 않습니다. 선택적 구성 요소로 선택해야 합니다.

1. Visual Studio 설치 관리자를 실행합니다.
2. 수정을 선택합니다.
3. Visual Studio 확장 개발 워크로드를 확인합니다.
4. 요약 트리에서 Visual Studio 확장 개발 노드를 엽니다.
5. .NET Compiler Platform SDK에 대한 확인란을 선택합니다. 선택적 구성 요소 아래에서 마지막에 찾을 수 있습니다.

필요에 따라, 시각화 도우미에서 DGML 편집기에 그래프도 표시할 수 있습니다.

1. 요약 트리에서 개별 구성 요소 노드를 엽니다.
2. DGML 편집기 확인란을 선택합니다.

### Visual Studio 설치 관리자를 사용한 설치 - 개별 구성 요소 탭

1. Visual Studio 설치 관리자를 실행합니다.
2. 수정을 선택합니다.
3. 개별 구성 요소 탭을 선택합니다.
4. .NET Compiler Platform SDK에 대한 확인란을 선택합니다. 컴파일러, 빌드 도구 및 런타임 섹션의 위쪽에서 찾을 수 있습니다.

필요에 따라, 시각화 도우미에서 DGML 편집기에 그래프도 표시할 수 있습니다.

1. DGML 편집기 확인란을 선택합니다. 코드 도구 섹션에서 찾을 수 있습니다.

## 불변성 및 .NET 컴파일러 플랫폼

불변성은 .NET 컴파일러 플랫폼의 기본 원리입니다. 변경 불가능한 데이터 구조는 생성된 후 변경할 수 없습니다. 변경 불가능한 데이터 구조는 여러 소비자가 동시에 안전하게 공유하고 분석할 수 있습니다. 한 소비자가 예기치 않은 방식으로 다른 소비자에게 영향을 줄 위험이 없습니다. 분석기에는 잠금이나 기타 동시성 측정값이 필요하지 않습니다. 이 규칙은 구문 트리, 컴파일, 기호, 의미 체계 모델 및 사용자에게 나타나는 모든 기타 데이터 구조에 적용됩니다. 기존 구조를 수정하는 대신 API는 지정된 차이점을 기반으로 새 개체를 만듭니다. 이 개념을 구문 트리에 적용하여 변환을 통해 새 트리를 만듭니다.

## 트리 만들기 및 변환

구문 변환에 대해 두 가지 전략 중 하나를 선택합니다. 팩터리 메서드는 대체할 특정 노드를 검색하거나 새 코드를 삽입할 특정 위치를 검색할 때 가장 유용합니다. 재작성기는 대체할 코드 패턴을 전체 프로젝트에서 검색하려는 경우 가장 적합합니다.

### 팩터리 메서드를 사용하여 노드 만들기

첫 번째 구문 변환은 팩터리 메서드를 보여 줍니다. `using System.Collections;` 문을 `using System.Collections.Generic;` 문으로 바꾸려고 합니다. 이 예제는 `Microsoft.CodeAnalysis.CSharp.SyntaxFactory` 팩터리 메서드를 사용하여 `Microsoft.CodeAnalysis.CSharp.CSharpSyntaxNode` 개체를 만드는 방법을 보여 줍니다. 각 종류의 노드, 토큰 또는 기타 정보의 경우 해당 형식의 인스턴스를 만드는 팩터리 메서드가 있습니다. 노드를 상향식 계층 구조로 작성하여 구문 트리를 만듭니다. 그런 다음, 기존 노드를 직접 만든 새 트리로 바꿔서 기존 프로그램을 변환합니다.

Visual Studio를 시작하고 새 C# 독립 실행형 코드 분석 도구 프로젝트를 만듭니다. Visual Studio에서 파일>새로 만들기>프로젝트를 선택하여 새 프로젝트 대화 상자를 표시합니다. Visual C#>확장성 아래에서 독립 실행형 코드 분석 도구를 선택합니다. 이 빠른 시작에는 두 개의 예제 프로젝트가 있으므로 솔루션 이름을 `SyntaxTransformationQuickStart`로 지정하고 프로젝트 이름을 `ConstructionCS`로 지정합니다. 확인을 클릭합니다.

이 프로젝트는 `Microsoft.CodeAnalysis.CSharp.SyntaxFactory` 클래스 메서드를 사용하여 `System.Collections.Generic` 네임스페이스를 나타내는 `Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax`를 생성합니다.

다음 `using` 지시문을 `Program.cs`의 맨 위에 추가합니다.

C#

```
using static Microsoft.CodeAnalysis.CSharp.SyntaxFactory;
using static System.Console;
```

name syntax nodes를 만들어 using System.Collections.Generic; 문을 나타내는 트리를 빌드합니다. NameSyntax는 C#에 나타나는 네 가지 형식의 이름에 대한 기본 클래스입니다. 다음 네 가지 형식의 이름을 함께 작성하여 C# 언어로 표시할 수 있는 이름을 만듭니다.

- Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax - System 및 Microsoft 같은 단순 단일 식별자 이름을 나타냅니다.
- Microsoft.CodeAnalysis.CSharp.Syntax.GenericNameSyntax - List<int> 같은 제네릭 형식 또는 메서드 이름을 나타냅니다.
- Microsoft.CodeAnalysis.CSharp.Syntax.QualifiedNameSyntax - System.IO 같은 폼 <left-name>.<right-identifier-or-generic-name>의 정규화된 이름을 나타냅니다.
- Microsoft.CodeAnalysis.CSharp.Syntax.AliasQualifiedNameSyntax - LibraryV2::Foo 같은 어셈블리 extern 별칭을 사용하는 이름을 나타냅니다.

IdentifierName(String) 메서드를 사용하여 NameSyntax 노드를 만듭니다. Program.cs에서 Main 메서드에 다음 코드를 추가합니다.

C#

```
NameSyntax name = IdentifierName("System");
WriteLine($"\\tCreated the identifier {name}");
```

앞의 코드는 IdentifierNameSyntax 개체를 만들고 이를 name 변수에 할당합니다. 대부분 Roslyn API는 기본 클래스를 반환하므로 관련 형식을 더 쉽게 사용할 수 있습니다. QualifiedNameSyntax를 빌드할 때 name 변수인 NameSyntax를 재사용할 수 있습니다. 샘플을 빌드할 때 형식 유추를 사용하지 마세요. 이 프로젝트에서 해당 단계를 자동화합니다.

이름을 만들었습니다. 이제 QualifiedNameSyntax를 빌드하여 더 많은 노드를 트리로 빌드해 보겠습니다. 새 트리는 name을 이름의 왼쪽으로 사용하고 Collections 네임스페이스의 새 IdentifierNameSyntax를 QualifiedNameSyntax의 오른쪽으로 사용합니다. 다음 코드를 program.cs에 추가합니다.

C#

```
name = QualifiedName(name, IdentifierName("Collections"));
WriteLine(name.ToString());
```

코드를 다시 실행하고 결과를 확인합니다. 코드를 나타내는 노드 트리를 빌드하고 있습니다. 이 패턴을 계속해서 네임스페이스 `System.Collections.Generic`에 대한 [QualifiedNameSyntax](#)를 빌드합니다. 다음 코드를 `Program.cs`에 추가합니다.

C#

```
name = QualifiedName(name, IdentifierName("Generic"));
WriteLine(name.ToString());
```

프로그램을 다시 실행하여 추가할 코드에 대한 트리를 빌드했는지 확인합니다.

## 수정된 트리 만들기

하나의 문을 포함하는 작은 구문 트리를 빌드했습니다. 단일 문 또는 기타 작은 코드 블록을 만드는 데는 새 노드를 만드는 API를 사용하는 것이 적합합니다. 그러나 더 큰 코드 블록을 빌드하려면 노드를 바꾸거나 노드를 기존 트리에 삽입하는 메서드를 사용해야 합니다. 구문 트리는 변경할 수 없습니다. [구문 API](#)는 생성 후 기존 구문 트리를 수정하기 위한 메커니즘을 제공하지 않습니다. 대신 기존 트리에 대한 변경 내용을 기반으로 새 트리를 생성하는 메서드를 제공합니다. `With*` 메서드는 [SyntaxNodeExtensions](#) 클래스에서 선언된 확장 메서드 또는 [SyntaxNode](#)에서 파생되는 구체적인 클래스에서 정의됩니다. 이러한 메서드는 기존 노드의 자식 속성에 변경 내용을 적용하여 새 노드를 만듭니다. 또한 [ReplaceNode](#) 확장 메서드를 사용하여 하위 트리의 하위 노드를 바꿀 수 있습니다. 이 메서드는 새로 만들어진 자식을 가리키도록 부모를 업데이트하고, 전체 트리에서 이 프로세스를 반복합니다(트리 '재회전'으로 알려진 프로세스).

다음 단계에서는 전체(작은) 프로그램을 나타내는 트리를 만든 다음, 수정합니다. 다음 코드를 `Program` 클래스의 시작 부분에 추가합니다.

C#

```
private const string sampleCode =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

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

## ① 참고

예제 코드는 `System.Collections.Generic` 네임스페이스가 아닌 `System.Collections` 네임스페이스를 사용합니다.

다음으로 `Main` 메서드의 맨 아래에 다음 코드를 추가하여 텍스트를 구문 분석하고 트리를 만듭니다.

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(sampleCode);
var root = (CompilationUnitSyntax)tree.GetRoot();
```

이 예제는 [WithName\(NameSyntax\)](#) 메서드를 사용하여 [UsingDirectiveSyntax](#) 노드의 이름을 이전 코드에서 생성된 이름으로 바꿉니다.

[WithName\(NameSyntax\)](#) 메서드를 통해 새 [UsingDirectiveSyntax](#) 노드를 만들어 `System.Collections` 이름을 이전 코드에서 만든 이름으로 업데이트합니다. `Main` 메서드의 맨 아래에 다음 코드를 추가합니다.

C#

```
var oldUsing = root.Usings[1];
var newUsing = oldUsing.WithName(name);
WriteLine(root.ToString());
```

프로그램을 실행하고 출력을 신중하게 확인합니다. `newUsing`이 루트 트리에 없습니다. 원래 트리가 변경되지 않았습니다.

[ReplaceNode](#) 확장 메서드를 통해 다음 코드를 추가하여 새 트리를 만듭니다. 새 트리는 기존 가져오기를 업데이트된 `newUsing` 노드로 바꾼 결과입니다. 기존 `root` 변수에 이 새 트리를 할당합니다.

C#

```
root = root.ReplaceNode(oldUsing, newUsing);
WriteLine(root.ToString());
```

프로그램을 다시 실행합니다. 이제 트리가 `System.Collections.Generic` 네임스페이스를 올바르게 가져옵니다.

## SyntaxRewriters 를 사용하여 트리 변환

`With*` 및 `ReplaceNode` 메서드는 구문 트리의 개별 분기를 변환하는 편리한 수단을 제공합니다. `Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter` 클래스는 구문 트리에서 여러 변환을 수행합니다. `Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter` 클래스는 `Microsoft.CodeAnalysis.CSharp.CSharpSyntaxVisitor<TResult>`의 서브클래스입니다.

`CSharpSyntaxRewriter`은 특정 형식의 `SyntaxNode`에 변환을 적용합니다. 구문 트리에 나타날 때마다 여러 형식의 `SyntaxNode` 개체에 변환을 적용할 수 있습니다. 이 빠른 시작의 두 번째 프로젝트는 형식 유추를 사용할 수 있는 모든 위치에서 지역 변수 선언의 명시적 형식을 제거하는 명령줄 리팩터링을 만듭니다.

새 C# 독립 실행형 코드 분석 도구 프로젝트를 만듭니다. Visual Studio에서 `SyntaxTransformationQuickStart` 솔루션 노드를 마우스 오른쪽 단추로 클릭합니다. 추가 > 새 프로젝트를 선택하여 새 프로젝트 대화 상자를 표시합니다. Visual C#>확장성 아래에서 독립 실행형 코드 분석 도구를 선택합니다. 프로젝트 이름을 `TransformationCS`로 지정하고 [확인]을 클릭합니다.

첫 번째 단계는 `CSharpSyntaxRewriter`에서 파생되는 클래스를 만들어 변환을 수행하는 것입니다. 프로젝트에 새 클래스 파일을 추가합니다. Visual Studio에서 프로젝트>클래스 추가... 를 선택합니다. 새 항목 추가 대화 상자에서 파일 이름으로 `TypeInferenceRewriter.cs`를 입력합니다.

다음 using 지시문을 `TypeInferenceRewriter.cs` 파일에 추가합니다.

```
C#  
  
using Microsoft.CodeAnalysis;  
using Microsoft.CodeAnalysis.CSharp;  
using Microsoft.CodeAnalysis.CSharp.Syntax;
```

다음으로 `TypeInferenceRewriter` 클래스가 `CSharpSyntaxRewriter` 클래스를 확장하도록 합니다.

```
C#  
  
public class TypeInferenceRewriter : CSharpSyntaxRewriter
```

다음 코드를 추가하여 `SemanticModel`을 포함할 전용 읽기 전용 필드를 선언하고 생성자에서 초기화합니다. 이 필드는 나중에 형식 유추를 사용할 수 있는 위치를 판별하는 데 필요합니다.

```
C#
```

```
private readonly SemanticModel SemanticModel;

public TypeInferenceRewriter(SemanticModel semanticModel) => SemanticModel = semanticModel;
```

[VisitLocalDeclarationStatement\(LocalDeclarationStatementSyntax\)](#) 메서드를 재정의합니다.

C#

```
public override SyntaxNode
VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax node)
{
}
```

### ① 참고

많은 Roslyn API는 반환된 실제 런타임 형식의 기본 클래스인 반환 형식을 선언합니다. 대부분 시나리오에서 한 종류의 노드는 다른 종류의 노드로 완전히 바뀌거나 제거될 수도 있습니다. 이 예제에서 메서드는

[VisitLocalDeclarationStatement\(LocalDeclarationStatementSyntax\)](#) 파생 형식 대신을 반환 [SyntaxNode](#)합니다 [LocalDeclarationStatementSyntax](#). 이 재작성기는 기존 노드를 기반으로 새 [LocalDeclarationStatementSyntax](#) 노드를 반환합니다.

이 빠른 시작은 지역 변수 선언을 처리합니다. 이 선언은 `foreach` 루프, `for` 루프, LINQ식 및 람다 식과 같은 다른 선언으로 확장할 수 있습니다. 또한 이 재작성기는 가장 단순한 형식의 선언만 변환합니다.

C#

```
Type variable = expression;
```

혼자서 살펴보려면 다음 형식의 변수 선언에 대한 완료된 샘플을 확장해 보세요.

C#

```
// Multiple variables in a single declaration.
Type variable1 = expression1,
    variable2 = expression2;
// No initializer.
Type variable;
```

다음 코드를 `VisitLocalDeclarationStatement` 메서드의 본문에 추가하여 이러한 선언 형식의 재작성을 건너뜁니다.

C#

```
if (node.Declaration.Variables.Count > 1)
{
    return node;
}
if (node.Declaration.Variables[0].Initializer == null)
{
    return node;
}
```

이 메서드는 수정되지 않은 `node` 매개 변수를 반환하여 재작성이 발생하지 않음을 나타냅니다. 해당 `if` 식 모두가 `true`가 아닌 경우 노드는 초기화를 통해 가능한 선언을 나타냅니다. 다음 문을 추가하여 선언에 지정된 형식 이름을 추출하고 `SemanticModel` 필드를 통해 형식 이름을 바인딩하여 형식 기호를 얻습니다.

C#

```
var declarator = node.Declaration.Variables.First();
var variableTypeName = node.Declaration.Type;

var variableType = (ITypeSymbol)SemanticModel
    .GetSymbolInfo(variableTypeName)
    .Symbol;
```

이제 이 문을 추가하여 이니셜라이저 식을 바인딩합니다.

C#

```
var initializerInfo =
SemanticModel.GetTypeInfo(declarator.Initializer.Value);
```

마지막으로 다음 `if` 문을 추가하여 이니셜라이저 식의 형식이 지정된 형식과 일치하는 경우 기존 형식 이름을 `var` 키워드로 바꿉니다.

C#

```
if (SymbolEqualityComparer.Default.Equals(variableType,
initializerInfo.Type))
{
    TypeSyntax varTypeName = SyntaxFactory.IdentifierName("var")
        .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
        .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());
```

```

        return node.ReplaceNode(variableTypeName, varTypeName);
    }
    else
    {
        return node;
    }
}

```

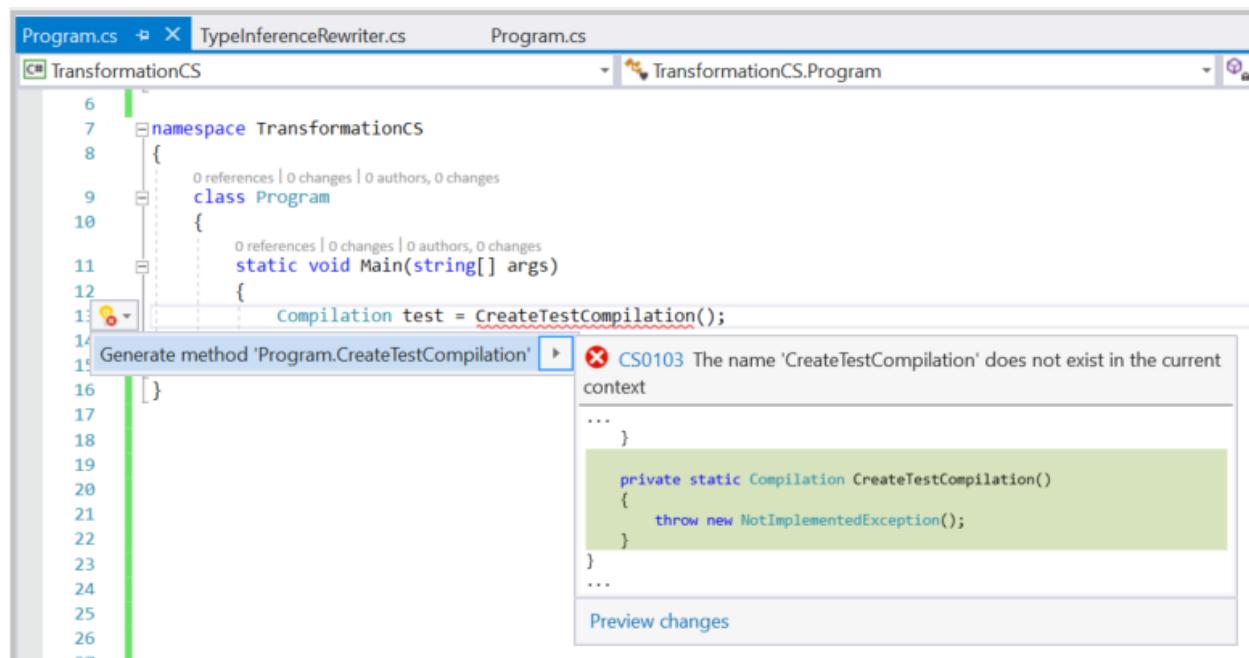
선언은 이니셜라이저 식을 기본 클래스 또는 인터페이스로 캐스트할 수 있기 때문에 조건이 필요합니다. 필요한 경우 할당의 왼쪽 및 오른쪽에 있는 형식이 일치하지 않습니다. 이러한 사례에서 명시적 형식을 제거하면 프로그램의 의미 체계가 변경됩니다. `var`은 상황별 키워드이므로 `var`은 키워드가 아니라 식별자로 지정됩니다. 선행 및 후행 기타 정보(공백)는 세로 공백과 들여쓰기를 유지하기 위해 이전 형식 이름에서 `var` 키워드로 전송됩니다. 형식 이름은 실제로 선언문의 손자이므로 `With*` 대신 `ReplaceNode`를 사용하여 `LocalDeclarationStatementSyntax`를 변환하는 것이 더 간단합니다.

`TypeInferenceRewriter`를 완료했습니다. 이제 `Program.cs` 파일로 돌아가서 예제를 완료합니다. 테스트 `Compilation`을 만들고 `SemanticModel`을 가져옵니다. `SemanticModel`을 사용하여 `TypeInferenceRewriter`를 시도합니다. 이 단계는 마지막으로 수행합니다. 그동안 테스트 컴파일을 나타내는 자리 표시자 변수를 선언합니다.

C#

```
Compilation test = CreateTestCompilation();
```

잠시 후에 `CreateTestCompilation` 메서드가 없음을 보고하는 오류 물결선이 표시됩니다. **Ctrl+마침표**를 눌러 전구를 열고 Enter 키를 눌러 **메서드 스텁 생성** 명령을 호출합니다. 이 명령은 `Program` 클래스에서 `CreateTestCompilation` 메서드에 대한 메서드 스텁을 생성합니다. 나중에 돌아와서 이 메서드를 입력합니다.



다음 코드를 작성하여 테스트 [Compilation](#)에서 각 [SyntaxTree](#)를 반복합니다. 각 트리에 대해 해당 트리의 [SemanticModel](#)을 사용하여 새 [TypeInferenceRewriter](#)를 초기화합니다.

C#

```
foreach (SyntaxTree sourceTree in test.SyntaxTrees)
{
    SemanticModel model = test.GetSemanticModel(sourceTree);

    TypeInferenceRewriter rewriter = new TypeInferenceRewriter(model);

    SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

    if (newSource != sourceTree.GetRoot())
    {
        File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
    }
}
```

직접 만든 `foreach` 문 내부에 다음 코드를 추가하여 각 소스 트리에서 변환을 수행합니다. 이 코드는 편집이 수행된 경우 변환된 새 트리를 조건부로 작성합니다. 재작성기는 형식 유추를 사용하여 단순화할 수 있는 하나 이상의 지역 변수 선언이 발생하는 경우에만 트리를 수정해야 합니다.

C#

```
SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

if (newSource != sourceTree.GetRoot())
{
    File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
}
```

`File.WriteAllText` 코드 아래에 물결선이 표시됩니다. 전구를 선택하고 필요한 `using System.IO;` 문을 추가합니다.

거의 완료되었습니다. 한 단계가 남았습니다. 테스트 [Compilation](#) 만들기입니다. 이 빠른 시작 중에 형식 유추를 사용하지 않았으므로 완벽한 테스트 사례를 만들었습니다. 그러나 C# 프로젝트 파일에서 컴파일을 만드는 작업은 이 연습의 범위를 벗어납니다. 그래도 지침을 신중하게 수행했다면 희망적입니다. `createTestCompilation` 메서드의 내용을 다음 코드로 대체합니다. 이 코드는 이 빠른 시작에 설명된 프로젝트와 조건부로 일치하는 테스트 컴파일을 만듭니다.

C#

```

String programPath = @"..\..\..\Program.cs";
String programText = File.ReadAllText(programPath);
SyntaxTree programTree =
    CSharpSyntaxTree.ParseText(programText)
        .WithFilePath(programPath);

String rewriterPath = @"..\..\..\TypeInferenceRewriter.cs";
String rewriterText = File.ReadAllText(rewriterPath);
SyntaxTree rewriterTree =
    CSharpSyntaxTree.ParseText(rewriterText)
        .WithFilePath(rewriterPath);

SyntaxTree[] sourceTrees = { programTree, rewriterTree };

MetadataReference mscorlib =
    MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
MetadataReference codeAnalysis =
    MetadataReference.CreateFromFile(typeof(SyntaxTree).Assembly.Location);
MetadataReference csharpCodeAnalysis =
    MetadataReference.CreateFromFile(typeof(CSharpSyntaxTree).Assembly.Location);

MetadataReference[] references = { mscorlib, codeAnalysis,
    csharpCodeAnalysis };

return CSharpCompilation.Create("TransformationCS",
    sourceTrees,
    references,
    new CSharpCompilationOptions(OutputKind.ConsoleApplication));

```

행운을 빌고 프로젝트를 실행합니다. Visual Studio에서 **디버그>디버깅 시작**을 선택합니다. Visual Studio에서 프로젝트의 파일이 변경되었다는 메시지가 표시됩니다. “**모두 예**”를 클릭하여 수정된 파일을 다시 로드합니다. 해당 파일을 살펴보면 놀라운 것을 관찰할 수 있습니다. 모든 명시적 및 중복 형식 지정자가 없는 코드는 놀라울 정도로 깔끔해 보입니다.

지금까지 **컴파일러 API**를 사용하여 C# 프로젝트에서 특정 구문 패턴에 대한 모든 파일을 검색하고, 해당 패턴과 일치하는 소스 코드의 의미 체계를 분석하고, 소스 코드를 변환하는 고유한 리팩터링을 작성했습니다. 이제 공식적으로 리팩터링 작성자가 되었습니다.

# 자습서: 첫 번째 분석기 및 코드 수정 작성

아티클 • 2023. 04. 08.

.NET Compiler Platform SDK는 C# 또는 Visual Basic 코드를 대상으로 하는 사용자 지정 진단(분석기), 코드 수정, 코드 리팩터링 및 진단 억제기를 만드는 데 필요한 도구를 제공합니다. **분석기**에는 규칙 위반을 인식하는 코드가 포함되어 있습니다. **코드 수정 사항**에는 위반을 수정하는 코드가 포함됩니다. 구현하는 규칙은 코드 구조에서 코딩 스타일, 명명 규칙 등에 이를 수 있습니다. .NET Compiler Platform은 개발자가 코드를 작성할 때 분석을 실행하기 위한 프레임워크와 코드를 수정하기 위한 모든 Visual Studio UI 기능(편집기에 물결선 표시, Visual Studio 오류 목록 채우기, “전구” 제안 만들기, 제안된 수정 사항의 다양한 미리 보기 표시)을 제공합니다.

이 자습서에서는 Roslyn API를 사용하여 **분석기** 및 함께 제공되는 **코드 수정 사항**을 만드는 방법을 살펴봅니다. 분석기는 소스 코드 분석을 수행하고 사용자에게 문제를 보고하는 방법입니다. 필요에 따라 코드 픽스를 분석기와 연결하여 사용자의 소스 코드에 대한 수정을 나타낼 수 있습니다. 이 자습서에서는 `const` 키워드를 사용하여 선언할 수 있는 지역 변수 선언을 찾는 분석기를 만듭니다. 함께 제공되는 코드 수정 사항은 해당 선언을 수정하여 `const` 키워드를 추가합니다.

## 사전 요구 사항

- [Visual Studio 2019](#) 버전 16.8 이상

Visual Studio 설치 관리자를 통해 .NET Compiler Platform SDK를 설치해야 합니다.

## 설치 지침 - Visual Studio 설치 관리자

Visual Studio 설치 관리자에서 .NET Compiler Platform SDK를 찾는 두 가지 방법이 있습니다.

## Visual Studio 설치 관리자를 사용한 설치 - 워크로드 보기

.NET Compiler Platform SDK는 Visual Studio 확장 개발 워크로드의 일부로 자동으로 선택되지 않습니다. 선택적 구성 요소로 선택해야 합니다.

1. Visual Studio 설치 관리자를 실행합니다.
2. **수정**을 선택합니다.
3. **Visual Studio 확장 개발** 워크로드를 확인합니다.
4. 요약 트리에서 **Visual Studio 확장 개발** 노드를 엽니다.

5. .NET Compiler Platform SDK에 대한 확인란을 선택합니다. 선택적 구성 요소 아래에서 마지막에 찾을 수 있습니다.

필요에 따라, 시각화 도우미에서 DGML 편집기에 그래프도 표시할 수 있습니다.

1. 요약 트리에서 **개별 구성 요소** 노드를 엽니다.
2. DGML 편집기 확인란을 선택합니다.

## Visual Studio 설치 관리자를 사용한 설치 - 개별 구성 요소 탭

1. Visual Studio 설치 관리자를 실행합니다.
2. **수정을 선택합니다.**
3. **개별 구성 요소** 탭을 선택합니다.
4. .NET Compiler Platform SDK에 대한 확인란을 선택합니다. 컴파일러, 빌드 도구 및 런타임 섹션의 위쪽에서 찾을 수 있습니다.

필요에 따라, 시각화 도우미에서 DGML 편집기에 그래프도 표시할 수 있습니다.

1. DGML 편집기 확인란을 선택합니다. **코드 도구** 섹션에서 찾을 수 있습니다.

분석기를 만들고 유효성 검사하는 단계는 다음과 같습니다.

1. 솔루션을 만듭니다.
2. 분석기 이름 및 설명을 등록합니다.
3. 분석기 경고 및 권장 사항을 보고합니다.
4. 권장 사항을 허용하도록 코드 수정 사항을 구현합니다.
5. 단위 테스트를 통해 분석을 개선합니다.

## 솔루션 만들기

- Visual Studio에서 파일 > 새 > 프로젝트...를 선택하여 새 프로젝트 대화 상자를 표시합니다.
- Visual C# > 확장성에서 코드 수정이 포함된 분석기(.NET 표준)를 선택합니다.
- 프로젝트 이름을 “MakeConst”로 지정하고 [확인]을 클릭합니다.

### ① 참고

컴파일 오류가 발생할 수 있습니다(MSB4062: "CompareBuildTaskVersion" 작업을 로드할 수 없음). 이 문제를 해결하려면 솔루션의 NuGet 패키지를 NuGet 패키지 관리자로 업데이트하거나 패키지 관리자 콘솔 창에서 `Update-Package` 를 사용합니다.

# 분석기 템플릿 살펴보기

코드 픽스 템플릿이 있는 분석기는 다음과 같은 다섯 가지 프로젝트를 만듭니다.

- 분석기를 포함하는 **MakeConst**.
- 코드 픽스를 포함하는 **MakeConst.CodeFixes**.
- **MakeConst.Package**: 분석기 및 코드 픽스에 대한 NuGet 패키지를 생성하는 데 사용됩니다.
- 단위 테스트 프로젝트인 **MakeConst.Test**.
- 새 분석기를 로드한 Visual Studio의 두 번째 인스턴스를 시작하는 기본 시작 프로젝트인 **MakeConst.Vsix**. **F5** 키를 눌러 VSIX 프로젝트를 시작합니다.

## ① 참고

분석기는 .NET Core 환경(명령줄 빌드) 및 .NET Framework 환경(Visual Studio)에서 실행될 수 있기 때문에 .NET Standard 2.0을 대상으로 해야 합니다.

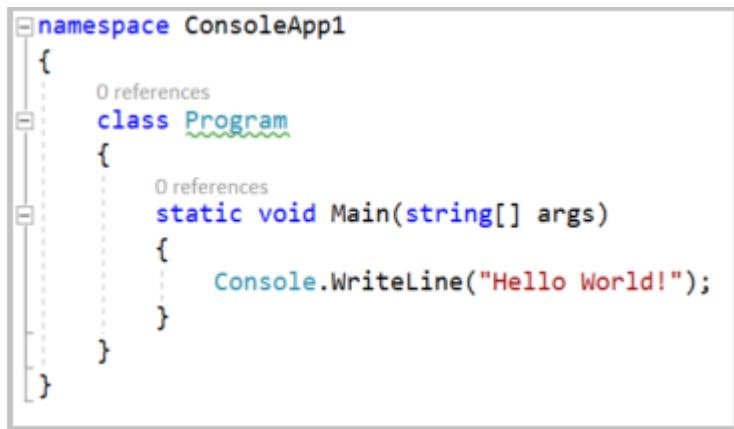
## 💡 팁

분석기를 실행할 때 Visual Studio의 두 번째 복사본을 시작합니다. 이 두 번째 복사본은 다른 레지스트리 하이브를 사용하여 설정을 저장합니다. 이렇게 하면 Visual Studio의 두 복사본에서 시각적 설정을 구별할 수 있습니다. Visual Studio의 실험 실행에 서로 다른 테마를 선택할 수 있습니다. 또한 Visual Studio의 실험 실행을 사용하여 사용자 설정 또는 로그인을 Visual Studio 계정에 로밍하지 마세요. 이렇게 하면 설정이 다르게 유지됩니다.

하이브에는 개발 중인 분석기뿐만 아니라 열려 있는 이전 분석기도 모두 포함됩니다. Roslyn 하이브를 다시 설정하려면 %LocalAppData%\Microsoft\VisualStudio에서 수동으로 삭제해야 합니다. Roslyn 하이브의 폴더 이름은 Roslyn으로 끝납니다(예: 16.0\_9ae182f9Roslyn). 하이브를 삭제한 후 솔루션을 정리하고 다시 빌드해야 할 수 있습니다.

방금 시작한 두 번째 Visual Studio 인스턴스에서 새 C# 콘솔 애플리케이션 프로젝트를 만듭니다(모든 대상 프레임워크가 작동함 - 분석기는 소스 수준에서 작동함). 물결 무늬 밑줄이 있는 토큰을 마우스로 가리키면 분석기가 제공하는 경고 텍스트가 나타납니다.

템플릿은 다음 그림에 표시된 대로 형식 이름에 소문자가 포함된 각 형식 선언에 대한 경고를 보고하는 분석기를 만듭니다.



```
namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

또한 템플릿은 소문자를 포함하는 형식 이름을 모두 대문자로 변경하는 코드 수정 사항을 제공합니다. 경고와 함께 표시된 전구를 클릭하여 제안된 변경 내용을 확인할 수 있습니다. 제안된 변경 내용을 적용하면 솔루션에서 형식 이름과 해당 형식에 대한 모든 참조가 업데이트됩니다. 이제 초기 분석기가 작동 중임을 확인했으므로 두 번째 Visual Studio 인스턴스를 닫고 분석기 프로젝트로 돌아갑니다.

분석기의 모든 변경 내용을 테스트하기 위해 Visual Studio의 두 번째 복사본을 시작하고 새 코드를 만들 필요가 없습니다. 템플릿은 사용자 대신 단위 테스트 프로젝트를 만듭니다. 해당 프로젝트에는 두 개의 테스트가 포함됩니다. `TestMethod1`은 진단을 트리거하지 않고 코드를 분석하는 테스트의 일반적인 형식을 보여 줍니다. `TestMethod2`는 진단을 트리거하는 테스트의 형식을 보여 준 후 제안된 코드 수정 사항을 적용합니다. 분석기 및 코드 수정 사항을 빌드할 때 여러 코드 구조에 대한 테스트를 작성하여 작업을 확인합니다. 분석기에 대한 단위 테스트는 Visual Studio와 대화형으로 테스트하는 것보다 훨씬 빠릅니다.

### 💡 팁

분석기 단위 테스트는 분석기를 트리거해야 하는 코드 구문과 트리거하면 안 되는 코드 구문을 알고 있을 경우 유용한 도구입니다. Visual Studio의 또 다른 복사본에서 분석기를 로드하는 기능은 아직 고려하지 않은 구문을 탐색하고 찾는데 유용한 도구입니다.

이 자습서에서는 로컬 상수로 변환할 수 있는 모든 로컬 변수 선언을 사용자에게 보고하는 분석기를 작성합니다. 예를 들어, 다음 코드를 고려하세요.

C#

```
int x = 0;
Console.WriteLine(x);
```

위의 코드에서 `x`는 상수 값이 할당되고 수정되지 않습니다. `const` 키워드를 사용하여 선언할 수 있습니다.

C#

```
const int x = 0;
Console.WriteLine(x);
```

변수를 상수로 설정할 수 있는지 여부를 판별하기 위한 분석이 포함되며, 변수가 작성되지 않는지 확인하려면 구문 분석, 상수 분석, 이니셜라이저 식의 상수 분석 및 데이터 흐름 분석이 필요합니다. .NET Compiler Platform은 이 분석을 보다 쉽게 수행할 수 있는 API를 제공합니다.

## 분석기 등록 만들기

템플릿은 `MakeConstAnalyzer.cs` 파일에서 초기 `DiagnosticAnalyzer` 클래스를 만듭니다. 이 초기 분석기는 모든 분석기의 두 가지 중요한 속성을 표시합니다.

- 모든 진단 분석기는 사용되는 언어를 설명하는 `[DiagnosticAnalyzer]` 특성을 제공해야 합니다.
- 모든 진단 분석기는 `DiagnosticAnalyzer` 클래스에서 직접 또는 간접적으로 파생되어야 합니다.

템플릿은 분석기의 일부인 기본 기능도 표시합니다.

- 작업을 등록합니다. 작업은 코드에서 위반을 검사하기 위해 분석기를 트리거해야 하는 코드 변경을 나타냅니다. Visual Studio는 등록된 작업과 일치하는 코드 편집을 검색하면 분석기의 등록된 메서드를 호출합니다.
- 진단을 만듭니다. 분석기는 위반을 검색하면 Visual Studio가 사용자에게 위반 사실을 알리는 데 사용하는 진단 개체를 만듭니다.

`DiagnosticAnalyzer.Initialize(AnalysisContext)` 메서드의 재정의에 작업을 등록합니다. 이 자습서에서는 로컬 선언을 검색하는 **구문 노드**를 방문하고 그 중 상수 값이 있는 노드를 확인합니다. 선언이 상수일 수 있으면 분석기는 진단을 만들고 보고합니다.

첫 번째 단계는 이러한 상수가 "Make Const" 분석기를 나타내도록 등록 상수 및 `Initialize` 메서드를 업데이트하는 것입니다. 대부분의 문자열 상수는 문자열 리소스 파일에 정의됩니다. 더 쉽게 지역화하려면 해당 사례를 따라야 합니다. `MakeConst` 분석기 프로젝트에 대한 `Resources.resx` 파일을 엽니다. 리소스 편집기가 표시됩니다. 다음과 같이 문자열 리소스를 업데이트합니다.

- `AnalyzerDescription`을 "Variables that are not modified should be made constants."로 변경합니다.
- `AnalyzerMessageFormat`을 "Variable '{0}' can be made constant"로 변경합니다.
- `AnalyzerTitle`을 "Variable can be made constant"로 변경합니다.

작업을 마치면 리소스 편집기가 다음 그림에서와 같이 표시됩니다.

Name	Value	Comment
AnalyzerDescription	Variables that are not modified should be made constants.	An optional longer localizable description of the diagnostic.
AnalyzerMessageFormat	Variable '{0}' can be made constant	The format-able message the diagnostic displays.
AnalyzerTitle	Variable can be made constant	The title of the diagnostic.
*		

나머지 변경 내용은 분석기 파일에 있습니다. Visual Studio에서 `MakeConstAnalyzer.cs`를 엽니다. 등록된 작업을 기호에 적용되는 작업에서 구문에 적용되는 작업으로 변경합니다. `MakeConstAnalyzerAnalyzer.Initialize` 메서드에서 기호에 대한 작업을 등록하는 줄을 찾습니다.

C#

```
context.RegisterSymbolAction(AnalyzeSymbol, SymbolKind.NamedType);
```

해당 줄을 다음 줄로 바꿉니다.

C#

```
context.RegisterSyntaxNodeAction(AnalyzeNode,  
SyntaxKind.LocalDeclarationStatement);
```

변경한 후 `AnalyzeSymbol` 메서드를 삭제할 수 있습니다. 이 분석기는 `SymbolKind.NamedType` 문이 아니라 `SyntaxKind.LocalDeclarationStatement`을 검사합니다. `AnalyzeNode` 아래에 빨간색 물결선이 있습니다. 방금 추가한 코드는 선언되지 않은 `AnalyzeNode` 메서드를 참조합니다. 다음 코드를 사용하여 메서드를 선언합니다.

C#

```
private void AnalyzeNode(SyntaxNodeAnalysisContext context)  
{  
}
```

다음 코드에 표시된 대로 `MakeConstAnalyzer.cs`에서 `Category`를 "Usage"로 변경합니다.

C#

```
private const string Category = "Usage";
```

## const일 수 있는 로컬 선언 찾기

이제 `AnalyzeNode` 메서드의 첫 번째 버전을 작성하겠습니다. 다음 코드와 같이 `const` 일 수 있지만 그렇지 않은 단일 로컬 선언을 찾아야 합니다.

C#

```
int x = 0;  
Console.WriteLine(x);
```

첫 번째 단계는 로컬 선언을 찾는 것입니다. `MakeConstAnalyzer.cs`에서 `AnalyzeNode`에 다음 코드를 추가합니다.

C#

```
var localDeclaration = (LocalDeclarationStatementSyntax)context.Node;
```

분석기는 로컬 선언의 변경 내용 및 로컬 선언만을 위해 등록되었으므로 이 캐스트는 항상 성공합니다. 다른 노드 형식은 `AnalyzeNode` 메서드 호출을 트리거하지 않습니다. 그런 다음, 선언에서 `const` 한정자를 확인합니다. 한정자를 찾으면 즉시 반환합니다. 다음 코드는 로컬 선언에서 `const` 한정자를 검색합니다.

C#

```
// make sure the declaration isn't already const:  
if (localDeclaration.Modifiers.Any(SyntaxKind.ConstKeyword))  
{  
    return;  
}
```

마지막으로 변수가 `const` 일 수 있는지 확인해야 합니다. 즉, 초기화된 후 절대 할당되지 않는지 확인합니다.

`SyntaxNodeAnalysisContext`를 사용하여 몇 가지 의미 체계 분석을 수행합니다. `context` 인수를 사용하여 지역 변수 선언을 `const`로 설정할 수 있는지 확인합니다.

`Microsoft.CodeAnalysis.SemanticModel`은 단일 소스 파일에 있는 모든 의미론적 정보를 나타냅니다. 의미 체계 모델을 다루는 문서에서 자세히 알아볼 수 있습니다.

`Microsoft.CodeAnalysis.SemanticModel`을 사용하여 로컬 선언 문에 대한 데이터 흐름 분석을 수행합니다. 그런 다음, 이 데이터 흐름 분석의 결과를 사용하여 지역 변수가 다른 곳에서 새 값으로 작성되지 않았는지 확인합니다. `GetDeclaredSymbol` 확장 메서드를 호출하여 변수의 `ILocalSymbol`을 검색하고 해당 변수가 데이터 흐름 분석의 `DataFlowAnalysis.WrittenOutside` 컬렉션에 포함되어 있지 않은지 확인합니다. `AnalyzeNode` 메서드의 끝에 다음 코드를 추가합니다.

C#

```

// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis =
context.SemanticModel.AnalyzeDataFlow(localDeclaration);

// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis
// region.
VariableDeclaratorSyntax variable =
localDeclaration.Declaration.Variables.Single();
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}

```

방금 추가된 코드는 변수가 수정되지 않았고 이에 따라 `const`로 설정될 수 있는지 확인합니다. 이제 진단을 실행하겠습니다. 다음 코드를 `AnalyzeNode`의 마지막 줄로 추가합니다.

C#

```
context.ReportDiagnostic(Diagnostic.Create(Rule, context.Node.GetLocation(),
localDeclaration.Declaration.Variables.First().Identifier.ValueText));
```

F5 키를 눌러 분석기를 실행하여 진행 상황을 확인할 수 있습니다. 이전에 만든 콘솔 애플리케이션을 로드한 후 다음 테스트 코드를 추가할 수 있습니다.

C#

```
int x = 0;
Console.WriteLine(x);
```

전구가 표시되고 분석기가 진단을 보고해야 합니다. 그러나 Visual Studio 버전에 따라 다음 중 하나가 표시됩니다.

- 템플릿에서 생성된 코드 수정을 계속 사용하는 전구는 대문자로 만들 수 있다고 알려줍니다.
- 편집기 맨 위에 'MakeConstCodeFixProvider'가 오류가 발생하여 사용하지 않도록 설정되었다는 배너 메시지가 표시됩니다. 코드 수정 공급자가 아직 변경되지 않았고 요소 대신 `LocalDeclarationStatementSyntax` 요소를 찾을 `TypeDeclarationSyntax` 것으로 예상하기 때문입니다.

다음 섹션에서는 코드 수정 사항을 작성하는 방법을 설명합니다.

## 코드 수정 사항 작성

분석기는 하나 이상의 코드 수정 사항을 제공할 수 있습니다. 코드 수정 사항은 보고된 문제를 해결하는 편집을 정의합니다. 직접 작성한 분석기의 경우 `const` 키워드를 삽입하는 코드 수정 사항을 제공할 수 있습니다.

diff

```
- int x = 0;
+ const int x = 0;
Console.WriteLine(x);
```

사용자가 편집기에서 전구 UI를 선택하면 Visual Studio가 코드를 변경합니다.

`CodeFixResources.resx` 파일을 열고 `CodeFixTitle`을 "Make constant"로 변경합니다.

템플릿에서 추가된 `MakeConstCodeFixProvider.cs` 파일을 엽니다. 이 코드 수정 사항은 진단 분석기에서 생성된 진단 ID에 연결되어 있지만 아직 적합한 코드 변환을 구현하지 않습니다.

그런 다음, `MakeUppercaseAsync` 메서드를 삭제합니다. 코드가 더 이상 적용되지 않습니다.

모든 코드 픽스 공급자는 `CodeFixProvider`에서 파생됩니다. 모두 `CodeFixProvider.RegisterCodeFixesAsync(CodeFixContext)`을 재정의하여 사용 가능한 코드 수정 사항을 보고합니다. `RegisterCodeFixesAsync`에서 진단과 일치하도록 검색 중인 상위 노드 형식을 `LocalDeclarationStatementSyntax`로 변경합니다.

C#

```
var declaration =
root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().OfType<LocalDeclarationStatementSyntax>().First();
```

그런 다음, 마지막 줄을 변경하여 코드 수정 사항을 등록합니다. 이 수정은 기존 선언에 `const` 한정자를 추가함으로써 생성되는 새 문서를 만듭니다.

C#

```
// Register a code action that will invoke the fix.
context.RegisterCodeFix(
    CodeAction.Create(
        title: CodeFixResources.CodeFixTitle,
        createChangedDocument: c => MakeConstAsync(context.Document,
declaration, c),
        equivalenceKey: nameof(CodeFixResources.CodeFixTitle)),
diagnostic);
```

방금 추가한 기호 `MakeConstAsync`에 대한 코드에 빨간색 물결선이 표시됩니다. 다음 코드와 같이 `MakeConstAsync`에 대한 선언을 추가합니다.

C#

```
private static async Task<Document> MakeConstAsync(Document document,
    LocalDeclarationStatementSyntax localDeclaration,
    CancellationToken cancellationToken)
{}
```

새 `MakeConstAsync` 메서드는 사용자의 소스 파일을 나타내는 `Document`를 현재 `const` 선언이 포함된 새 `Document`로 변환합니다.

선언 문 앞에 삽입할 새 `const` 키워드 토큰을 만듭니다. 먼저 선언 문의 첫 번째 토큰에서 수행 trivia를 제거하고 이를 `const` 토큰에 연결해야 합니다. `MakeConstAsync` 메서드에 다음 코드를 추가합니다.

C#

```
// Remove the leading trivia from the local declaration.
SyntaxToken firstToken = localDeclaration.GetFirstToken();
SyntaxTriviaList leadingTrivia = firstToken.LeadingTrivia;
LocalDeclarationStatementSyntax trimmedLocal =
localDeclaration.ReplaceToken(
    firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));

// Create a const token with the leading trivia.
SyntaxToken constToken = SyntaxFactory.Token(leadingTrivia,
SyntaxKind.ConstKeyword,
SyntaxFactory.TriviaList(SyntaxFactory.ElasticMarker));
```

그런 다음, 다음 코드를 사용하여 `const` 토큰을 선언에 추가합니다.

C#

```
// Insert the const token into the modifiers list, creating a new modifiers
list.
SyntaxTokenList newModifiers = trimmedLocal.Modifiers.Insert(0, constToken);
// Produce the new local declaration.
LocalDeclarationStatementSyntax newLocal = trimmedLocal
    .WithModifiers(newModifiers)
    .WithDeclaration(localDeclaration.Declaration);
```

그런 다음, C# 형식 지정 규칙과 일치하도록 새 선언의 형식을 지정합니다. 기존 코드와 일치하도록 변경 내용의 형식을 지정하면 향상된 환경이 생성됩니다. 기존 코드 바로 뒤에 다음 문을 추가합니다.

C#

```
// Add an annotation to format the new local declaration.  
LocalDeclarationStatementSyntax formattedLocal =  
newLocal.WithAdditionalAnnotations(Formatter.Annotation);
```

이 코드에는 새 네임스페이스가 필요합니다. 다음 `using` 지시문을 파일의 맨 위에 추가합니다.

C#

```
using Microsoft.CodeAnalysis.Formatting;
```

마지막 단계는 편집을 만드는 것입니다. 이 프로세스는 다음 3개 단계로 구성됩니다.

1. 기존 문서에 대한 핸들을 가져옵니다.
2. 기존 선언을 새 선언으로 바꿔서 새 문서를 만듭니다.
3. 새 문서를 반환합니다.

`MakeConstAsync` 메서드의 끝에 다음 코드를 추가합니다.

C#

```
// Replace the old local declaration with the new local declaration.  
SyntaxNode oldRoot = await  
document.GetSyntaxRootAsync(cancellationToken).ConfigureAwait(false);  
SyntaxNode newRoot = oldRoot.ReplaceNode(localDeclaration, formattedLocal);  
  
// Return document with transformed tree.  
return document.WithSyntaxRoot(newRoot);
```

코드 수정 사항을 시도할 준비가 되었습니다. `F5` 키를 눌러 Visual Studio의 두 번째 인스턴스에서 분석기 프로젝트를 실행합니다. Visual Studio의 두 번째 인스턴스에서 새 C# 콘솔 애플리케이션 프로젝트를 만들고 상수 값으로 초기화된 몇 개의 지역 변수 선언을 `Main` 메서드에 추가합니다. 아래와 같이 경고로 보고되었음을 알 수 있습니다.

```
static void Main(string[] args)  
{  
    int i = 1;  
    int j = 2;  
    int k = i + j;  
}
```

많은 과정을 진행했습니다. `const`로 설정될 수 있는 선언 아래에 물결선이 있습니다. 그러나 아직 해야 할 일이 있습니다. `i`, `j` 및 `k`로 시작하는 세 개의 선언에 순서대로 `const`를 추가하면 이 코드가 제대로 작동합니다. 그러나 `const` 한정자를 `k`부터 다른 순서로 추

가하면 분석기에서 오류가 발생합니다. `i` 및 `j`가 둘 다 `const`가 될 때까지 `k`는 `const`로 선언될 수 없습니다. 변수를 선언하고 초기화할 수 있는 다양한 방법을 처리하도록 하려면 추가 분석을 수행해야 합니다.

## 단위 테스트 빌드

분석기 및 코드 수정 사항은 `const`로 설정될 수 있는 단일 선언의 간단한 사례에 적용됩니다. 이 구현으로 인해 오류가 발생할 수 있는 많은 선언 문이 있습니다. 템플릿에서 작성된 단위 테스트 라이브러리를 사용하여 이러한 사례를 해결합니다. 이 방법은 Visual Studio의 두 번째 복사본을 반복적으로 여는 것보다 훨씬 더 빠릅니다.

단위 테스트 프로젝트에서 `MakeConstUnitTests.cs` 파일을 엽니다. 템플릿은 분석기 및 코드 수정 사항 단위 테스트에 대한 두 개의 공통 패턴을 따르는 두 개의 테스트를 만들었습니다. `TestMethod1`은 분석기가 보고하면 안 될 때 진단을 보고하지 않는지 확인하는 테스트 패턴을 보여 줍니다. `TestMethod2`는 진단을 보고하고 코드 수정 사항을 실행하기 위한 패턴을 보여 줍니다.

템플릿은 단위 테스트를 위해 [Microsoft.CodeAnalysis.Testing](#) 패키지를 사용합니다.

### 💡 팁

테스트 라이브러리는 다음을 비롯한 특수 태그 구문을 지원합니다.

- `[|text|]`: `text`에 대한 진단이 보고되었음을 나타냅니다. 기본적으로 이 양식은 `DiagnosticAnalyzer`.`SupportedDiagnostics`에서 제공한 `DiagnosticDescriptor`가 정확히 한 개 있는 분석기 테스트에만 사용할 수 있습니다.
- `{|ExpectedDiagnosticId:text|}`: 에 대한 진단 `Id` `ExpectedDiagnosticId` 이 에 대해 보고됨을 `text` 나타냅니다.

`MakeConstUnitTest` 클래스의 템플릿 테스트를 다음 테스트 메서드로 바꿉니다.

C#

```
[TestMethod]
public async Task LocalIntCouldBeConstant_Diagnostic()
{
    await VerifyCS.VerifyCodeFixAsync(@""
using System;

class Program
{
    static void Main()
```

```

    {
        [|int i = 0;|]
        Console.WriteLine(i);
    }
}
", @""
using System;

class Program
{
    static void Main()
    {
        const int i = 0;
        Console.WriteLine(i);
    }
}
");
    }
}

```

이 테스트를 실행하여 통과하는지 확인합니다. Visual Studio에서 **테스트>Windows>테스트 탐색기**를 선택하여 **테스트 탐색기**를 엽니다. 그런 다음, **모두 실행**을 선택합니다.

## 유효한 선언에 대한 테스트 만들기

일반적으로 분석기는 가능한 한 빨리 종료되어야 하므로 최소한의 작업을 수행합니다. Visual Studio는 등록된 분석기를 사용자 편집 코드로 호출합니다. 응답은 키 요구 사항입니다. 진단을 실행하지 않아야 하는 코드에 대한 여러 가지 테스트 사례가 있습니다. 분석기가 이미 이러한 테스트 중 하나를 처리합니다. 이 경우 변수는 초기화된 후에 할당됩니다. 해당 사례를 나타내는 다음 테스트 메서드를 추가합니다.

C#

```

[TestMethod]
public async Task VariableIsAssigned_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = 0;
        Console.WriteLine(i++);
    }
}
");
}

```

이 테스트도 성공합니다. 다음으로, 아직 처리하지 않은 조건에 대한 테스트 메서드를 추가합니다.

- 이미 상수이므로 이미 `const` 인 선언:

C#

```
[TestMethod]
public async Task VariableIsAlreadyConst_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@""
using System;

class Program
{
    static void Main()
    {
        const int i = 0;
        Console.WriteLine(i);
    }
}
"));
}
```

- 사용할 값이 없으므로 이니셜라이저가 없는 선언:

C#

```
[TestMethod]
public async Task NoInitializer_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@""
using System;

class Program
{
    static void Main()
    {
        int i;
        i = 0;
        Console.WriteLine(i);
    }
}
"));
}
```

- 컴파일 시간 상수일 수 없으므로 이니셜라이저가 상수가 아닌 선언:

C#

```
[TestMethod]
public async Task InitializerIsNotConstant_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@""
using System;

class Program
{
    static void Main()
    {
        int i = DateTime.Now.DayOfYear;
        Console.WriteLine(i);
    }
}
");
}
```

C#은 여러 선언을 하나의 문으로 허용하므로 훨씬 더 복잡할 수 있습니다. 다음 테스트 사례 문자열 상수를 고려합니다.

C#

```
[TestMethod]
public async Task MultipleInitializers_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@""
using System;

class Program
{
    static void Main()
    {
        int i = 0, j = DateTime.Now.DayOfYear;
        Console.WriteLine(i);
        Console.WriteLine(j);
    }
}
");
}
```

i 변수는 상수로 설정될 수 있지만, j 변수는 상수로 설정될 수 없습니다. 따라서 이 문은 const 선언으로 설정될 수 없습니다.

테스트를 다시 실행하면 새 테스트 사례가 실패합니다.

## 올바른 선언을 무시하도록 분석기 업데이트

이러한 조건과 일치하는 코드를 필터링하려면 분석기의 `AnalyzeNode` 메서드에 대한 몇 가지 개선 사항이 필요합니다. 개선 사항은 모두 관련된 조건이므로 유사한 변경 내용이 이러한 모든 조건을 수정합니다. `AnalyzeNode`에 다음 변경 내용을 적용합니다.

- 의미 체계 분석이 단일 변수 선언을 검사했습니다. 이 코드는 동일한 문에 선언된 모든 변수를 검사하는 `foreach` 루프에 있어야 합니다.
- 선언된 각 변수에는 이니셜라이저가 있어야 합니다.
- 선언된 각 변수의 이니셜라이저는 컴파일 시간 상수여야 합니다.

`AnalyzeNode` 메서드에서 다음 원래 의미 체계 분석을

C#

```
// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis =
    context.SemanticModel.AnalyzeDataFlow(localDeclaration);

// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis
region.
VariableDeclaratorSyntax variable =
    localDeclaration.Declaration.Variables.Single();
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable,
    context.CancellationToken);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

다음 코드 조각으로 바꿉니다.

C#

```
// Ensure that all variables in the local declaration have initializers that
// are assigned with constant values.
foreach (VariableDeclaratorSyntax variable in
localDeclaration.Declaration.Variables)
{
    EqualsValueClauseSyntax initializer = variable.Initializer;
    if (initializer == null)
    {
        return;
    }

    Optional<object> constantValue =
context.SemanticModel.GetConstantValue(initializer.Value,
    context.CancellationToken);
    if (!constantValue.HasValue)
    {
        return;
```

```

        }

    // Perform data flow analysis on the local declaration.
    DataFlowAnalysis dataFlowAnalysis =
        context.SemanticModel.AnalyzeDataFlow(localDeclaration);

    foreach (VariableDeclaratorSyntax variable in
        localDeclaration.Declaration.Variables)
    {
        // Retrieve the local symbol for each variable in the local declaration
        // and ensure that it is not written outside of the data flow analysis
        // region.
        ISymbol variableSymbol =
            context.SemanticModel.GetDeclaredSymbol(variable,
                context.CancellationToken);
        if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
        {
            return;
        }
    }
}

```

첫 번째 `foreach` 루프는 구문 분석을 사용하여 각 변수 선언을 검사합니다. 첫 번째 검사는 변수에 이니셜라이저가 포함되도록 합니다. 두 번째 검사는 이니셜라이저가 상수가 되도록 합니다. 두 번째 루프에는 원래 의미 체계 분석이 있습니다. 의미 체계 검사는 성능에 더 큰 영향을 주므로 별도의 루프에 있습니다. 테스트를 다시 실행하면 테스트가 모두 성공으로 표시되어야 합니다.

## 최종 폴란드어 추가

거의 완료되었습니다. 분석기가 처리할 몇 가지 추가 조건이 있습니다. 사용자가 코드를 작성하는 동안 Visual Studio가 분석기를 호출합니다. 분석기가 컴파일되지 않는 코드를 위해 호출되는 경우도 있습니다. 진단 분석기의 `AnalyzeNode` 메서드는 상수 값이 변수 형식으로 변환될 수 있는지 확인하지 않습니다. 따라서 현재 구현은 `int i = "abc"` 같은 잘못된 선언을 로컬 상수로 변환합니다. 이 경우 테스트 메서드를 추가합니다.

C#

```

[TestMethod]
public async Task DeclarationIsInvalid_NoDiagnostic()
{
    await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
}

```

```
        int x = {|CS0029:""abc""|};  
    }  
}  
");  
}
```

또한 참조 형식이 제대로 처리되지 않습니다. 참조 형식에 허용되는 유일한 상수 값은 `null`이지만 문자열 리터럴을 허용하는 `System.String`에서는 그렇지 않습니다. 즉, `const string s = "abc"`는 적합하지만 `const object s = "abc"`는 적합하지 않습니다. 이 코드 조각은 해당 조건을 확인합니다.

C#

```
[TestMethod]  
public async Task DeclarationIsNotString_NoDiagnostic()  
{  
    await VerifyCS.VerifyAnalyzerAsync(@"  
using System;  
  
class Program  
{  
    static void Main()  
    {  
        object s = ""abc"";  
    }  
}  
");  
}
```

철저하게 하려면 문자열에 대한 상수 선언을 만들 수 있는지 확인하는 또 다른 테스트를 추가해야 합니다. 다음 코드 조각은 진단을 실행하는 코드 및 수정 사항이 적용된 후의 코드를 둘 다 정의합니다.

C#

```
[TestMethod]  
public async Task StringCouldBeConstant_Diagnostic()  
{  
    await VerifyCS.VerifyCodeFixAsync(@"  
using System;  
  
class Program  
{  
    static void Main()  
    {  
        [|string s = ""abc"";|]  
    }  
}  
", @"  
using System;
```

```
class Program
{
    static void Main()
    {
        const string s = ""abc"";
    }
}
");
}
```

마지막으로 변수가 `var` 키워드를 사용하여 선언된 경우 코드 수정 사항은 잘못된 작업을 수행하고 `const var` 선언을 생성하며, 이는 C# 언어에서 지원되지 않습니다. 이 버그를 수정하려면 코드 수정 사항이 `var` 키워드를 유추 형식 이름으로 바꾸어야 합니다.

C#

```
[TestMethod]
public async Task VarIntDeclarationCouldBeConstant_Diagnostic()
{
    await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|var item = 4;|]
    }
},
@", @"
using System;

class Program
{
    static void Main()
    {
        const int item = 4;
    }
}
");
}

[TestMethod]
public async Task VarStringDeclarationCouldBeConstant_Diagnostic()
{
    await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
```

```

    {
        [|var item = ""abc"";|]
    }
},
@", @"
using System;

class Program
{
    static void Main()
    {
        const string item = ""abc"";
    }
}
");
}

```

다행히도 위의 버그는 모두 방금 알아본 동일한 기술을 사용하여 해결할 수 있습니다.

첫 번째 버그를 수정하려면 먼저 `MakeConstAnalyzer.cs`를 열고 상수 값과 함께 할당되었는지 확인하기 위해 각 로컬 선언의 이니셜라이저가 검사되는 `foreach` 루프를 찾습니다. 첫 번째 `foreach` 루프 바로 '앞'에서 `context.SemanticModel.GetTypeInfo()`를 호출하여 로컬 선언의 선언된 형식에 대한 자세한 정보를 검색합니다.

C#

```

TypeSyntax variableTypeName = localDeclaration.Declaration.Type;
ITypeSymbol variableType =
context.SemanticModel.GetTypeInfo(variableTypeName,
context.CancellationToken).ConvertedType;

```

그런 다음, `foreach` 루프 내부에서 각 이니셜라이저를 검사하여 변수 형식으로 변환할 수 있는지 확인합니다. 이니셜라이저가 상수인지 확인한 후 다음 검사를 추가합니다.

C#

```

// Ensure that the initializer value can be converted to the type of the
// local declaration without a user-defined conversion.
Conversion conversion =
context.SemanticModel.ClassifyConversion(initializer.Value, variableType);
if (!conversion.Exists || conversion.IsUserDefined)
{
    return;
}

```

다음 변경은 마지막 항목을 기반으로 빌드됩니다. 첫 번째 `foreach` 루프의 닫는 중괄호 앞에 다음 코드를 추가하여 상수가 문자열 또는 null일 때 로컬 선언의 형식을 검사합니다.

C#

```
// Special cases:  
// * If the constant value is a string, the type of the local declaration  
//   must be System.String.  
// * If the constant value is null, the type of the local declaration must  
//   be a reference type.  
if (constantValue.Value is string)  
{  
    if (variableType.SpecialType != SpecialType.System_String)  
    {  
        return;  
    }  
}  
else if (variableType.IsReferenceType && constantValue.Value != null)  
{  
    return;  
}
```

`var` 키워드를 올바른 형식 이름으로 바꾸려면 코드 수정 사항 공급자에서 약간의 코드를 추가로 작성해야 합니다. *MakeConstCodeFixProvider.cs*로 돌아갑니다. 추가할 코드는 다음 단계를 수행합니다.

- 선언이 `var` 선언인지, 그리고 다음과 같은지 검사합니다.
- 유추 형식에 대한 새 형식을 만듭니다.
- 형식 선언이 별칭이 아닌지 확인합니다. 별칭이 아니면 `const var`을 선언하는 것이 적합합니다.
- `var`이 이 프로그램의 형식 이름이 아닌지 확인합니다. 아닌 경우 `const var`이 적합합니다.
- 전체 형식 이름 단순화

코드가 다소 많아 보이지만 그렇지 않습니다. `newLocal`을 선언 및 초기화하는 줄을 다음 코드로 바꿉니다. 코드는 `newModifiers` 초기화 바로 뒤에 옵니다.

C#

```
// If the type of the declaration is 'var', create a new type name  
// for the inferred type.  
VariableDeclarationSyntax variableDeclaration =  
localDeclaration.Declaration;  
TypeSyntax variableTypeName = variableDeclaration.Type;  
if (variableTypeName.IsVar)  
{  
    SemanticModel semanticModel = await  
document.GetSemanticModelAsync(cancellationToken).ConfigureAwait(false);  
  
    // Special case: Ensure that 'var' isn't actually an alias to another  
    // type  
    // (e.g. using var = System.String).
```

```

IAliasSymbol aliasInfo = semanticModel.GetAliasInfo(variableTypeName,
cancellationToken);
if (aliasInfo == null)
{
    // Retrieve the type inferred for var.
    ITypeSymbol type = semanticModel.GetTypeInfo(variableTypeName,
cancellationToken).ConvertedType;

    // Special case: Ensure that 'var' isn't actually a type named
    'var'.
    if (type.Name != "var")
    {
        // Create a new TypeSyntax for the inferred type. Be careful
        // to keep any leading and trailing trivia from the var keyword.
        TypeSyntax typeName =
            SyntaxFactory.ParseTypeName(type.ToString())
                .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
                .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

        // Add an annotation to simplify the type name.
        TypeSyntax simplifiedTypeName =
            typeName.WithAdditionalAnnotations(Simplifier.Annotation);

        // Replace the type in the variable declaration.
        variableDeclaration =
            variableDeclaration.WithType(simplifiedTypeName);
    }
}
// Produce the new local declaration.
LocalDeclarationStatementSyntax newLocal =
    trimmedLocal.WithModifiers(newModifiers)
        .WithDeclaration(variableDeclaration);

```

Simplifier 형식을 사용하려면 using 지시문을 하나 추가해야 합니다.

C#

```
using Microsoft.CodeAnalysis.Simplification;
```

테스트를 실행하면 모두 성공합니다. 완료된 분석기를 직접 실행할 수 있습니다. **Ctrl + F5**를 눌러 Roslyn 미리 보기 확장이 로드된 Visual Studio의 두 번째 인스턴스에서 분석기 프로젝트를 실행합니다.

- 두 번째 Visual Studio 인스턴스에서 새 C# 콘솔 애플리케이션 프로젝트를 만들고 `int x = "abc";`을 Main 메서드에 추가합니다. 첫 번째 버그 수정 덕분에 이 지역 변수 선언에 대한 경고가 보고되지 않습니다(컴파일러 오류는 예상대로 발생함).
- 그런 다음, `object s = "abc";`을 Main 메서드에 추가합니다. 두 번째 버그 수정으로 인해 경고가 보고되지 않습니다.

- 마지막으로 `var` 키워드를 사용하는 다른 지역 변수를 추가합니다. 경고가 보고되고 제안이 왼쪽 바로 아래에 표시됩니다.
- 편집기 캐럿을 물결선 위로 이동하고 `Ctrl + .`를 누릅니다. 제안된 코드 수정 사항을 표시합니다. 코드 수정 사항을 선택하면 `var` 키워드가 올바르게 처리됩니다.

마지막으로 다음 코드를 추가합니다.

C#

```
int i = 2;  
int j = 32;  
int k = i + j;
```

이러한 변경 후에는 처음 두 개의 변수에만 빨간색 물결선이 표시됩니다. `i` 및 `j`에 `const`를 추가합니다. `const`일 수 있으므로 `k`에 새 경고가 표시됩니다.

지금까지 신속한 코드 분석을 수행하여 문제를 검색하고 수정하기 위한 빠른 수정 사항을 제공하는 첫 번째 .NET Compiler Platform 확장을 만들었습니다. 또한 .NET Compiler Platform SDK(Roslyn API)의 일부인 많은 코드 API를 알아보았습니다. 샘플 GitHub 리포지토리의 [완료된 샘플](#)을 기준으로 작업을 검사할 수 있습니다.

## 기타 리소스

- [구문 분석 시작](#)
- [의미 체계 분석 시작](#)

# 프로그래밍 개념(C#)

아티클 • 2024. 04. 11.

이 섹션에서는 C# 언어의 프로그래밍 개념을 설명합니다.

## 섹션 내용

[+] 테이블 확장

제목	설명
.NET 어셈블리	어셈블리를 만들고 사용하는 방법을 설명합니다.
async 및 await를 사용한 비동기 프로그래밍(C#)	C#에서 <code>async</code> 및 <code>await</code> 키워드를 사용하여 비동기 솔루션을 작성하는 방법을 설명합니다. 예제가 포함되어 있습니다.
특성(C#)	특성을 사용하여 형식, 필드, 메서드 및 속성 등의 요소를 프로그래밍하는 방법에 대한 추가 정보를 제공하는 방법을 설명합니다.
컬렉션(C#)	.NET에서 제공하는 컬렉션의 형식 중 일부를 설명합니다. 간단한 컬렉션 및 키/값 쌍의 컬렉션을 사용하는 방법을 보여 줍니다.
공변성(Covariance) 및 반공변성(Contravariance)(C#)	인터페이스 및 대리자에서 제네릭 형식 매개 변수의 암시적 변환을 사용하도록 설정하는 방법을 보여 줍니다.
식 트리(C#)	식 트리를 사용하여 실행 코드의 동적 수정을 허용하는 방법을 설명합니다.
반복기(C#)	컬렉션을 단계별로 실행하면서 한 번에 하나씩 요소를 반환하는 데 사용되는 반복기에 대해 설명합니다.
LINQ(Language-Integrated Query)(C#)	C#의 언어 구문의 강력한 쿼리 기능과 관계형 데이터베이스, XML 문서, 데이터 세트 및 메모리 내 컬렉션을 쿼리하기 위한 모델에 대해 설명합니다.
리플렉션(C#)	리플렉션을 사용하면 동적으로 형식 인스턴스를 만들거나, 형식을 기준 개체에 바인딩하거나, 기준 개체에서 형식을 가져와 해당 메서드를 호출하거나, 필드 및 속성에 액세스하는 방법을 설명합니다.
Serialization(C#)	이진, XML 및 SOAP serialization의 주요 개념에 대해 설명합니다.

## 관련 단원

- 성능 팁

애플리케이션의 성능을 향상시키는 데 도움이 되는 여러 가지 기본 규칙에 대해 설명합니다.

---

## 피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공 ↗](#)

# 공변성(Covariance) 및 반공변성(Contravariance)(C#)

아티클 • 2024. 04. 11.

C#에서 공변성(Covariance)과 반공변성(Contravariance)은 배열 형식, 대리자 형식 및 제네릭 형식 인수에 대한 암시적 참조 변환을 가능하게 합니다. 공변성(Covariance)은 할당 호환성을 유지하고 반공변성(Contravariance)은 할당 호환성을 유지하지 않습니다.

다음 코드에서는 할당 호환성, 공변성(Covariance) 및 반공변성(Contravariance) 간의 차이를 보여 줍니다.

```
C#  
  
// Assignment compatibility.  
string str = "test";  
// An object of a more derived type is assigned to an object of a less  
// derived type.  
object obj = str;  
  
// Covariance.  
IEnumerable<string> strings = new List<string>();  
// An object that is instantiated with a more derived type argument  
// is assigned to an object instantiated with a less derived type argument.  
// Assignment compatibility is preserved.  
IEnumerable<object> objects = strings;  
  
// Contravariance.  
// Assume that the following method is in the class:  
static void SetObject(object o) { }  
Action<object> actObject = SetObject;  
// An object that is instantiated with a less derived type argument  
// is assigned to an object instantiated with a more derived type argument.  
// Assignment compatibility is reversed.  
Action<string> actString = actObject;
```

배열에 대한 공변성(Covariance)은 더 많이 파생된 형식의 배열을 더 적게 파생된 형식의 배열로 암시적 변환을 가능하게 합니다. 하지만 다음 코드 예제와 같이 이 작업은 형식이 안전하지 않습니다.

```
C#  
  
object[] array = new String[10];  
// The following statement produces a run-time exception.  
// array[0] = 10;
```

메서드 그룹에 대한 공변성(Covariance) 및 반공변성(Contravariance) 지원으로 대리자 형식과 메서드 시그니처를 일치시킬 수 있습니다. 일치하는 시그니처가 있는 메서드뿐만 아니라 더 많이 파생된 형식(공변성(covariance))을 반환하는 메서드 또는 대리자 형식에 지정된 것보다 더 적게 파생된 형식(반공변성(contravariance))을 가지고 있는 매개 변수를 수락하는 메서드도 대리자에 할당할 수 있습니다. 자세한 내용은 [대리자의 가변성\(C#\)](#) 및 [대리자의 가변성 사용\(C#\)](#)을 참조하세요.

다음 코드 예제에서는 메서드 그룹에 대한 공변성(Covariance) 및 반공변성(Contravariance) 지원을 보여 줍니다.

C#

```
static object GetObject() { return null; }
static void SetObject(object obj) { }

static string GetString() { return ""; }
static void SetString(string str) { }

static void Test()
{
    // Covariance. A delegate specifies a return type as object,
    // but you can assign a method that returns a string.
    Func<object> del = GetString;

    // Contravariance. A delegate specifies a parameter type as string,
    // but you can assign a method that takes an object.
    Action<string> del2 = SetObject;
}
```

.NET Framework 4 이상 버전에서 C#은 제네릭 인터페이스 및 대리자의 공변성(Covariance) 및 반공변성(Contravariance)을 지원하고 제네릭 형식 매개 변수를 암시적으로 변환하도록 허용합니다. 자세한 내용은 [제네릭 인터페이스의 가변성\(C#\)](#) 및 [대리자의 가변성\(C#\)](#)을 참조하세요.

다음 코드 예제에서는 제네릭 인터페이스에 대한 암시적 참조 변환을 보여 줍니다.

C#

```
IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

제네릭 매개 변수가 선언된 공변(covariant) 또는 반공변(contravariant)인 경우 제네릭 인터페이스 또는 대리자를 *variant*라고 합니다. C#에서는 사용자 고유의 variant 인터페이스 및 대리자를 만들 수 있습니다. 자세한 내용은 [Variant 제네릭 인터페이스 만들기\(C#\)](#) 및 [대리자의 가변성\(C#\)](#)을 참조하세요.

# 관련 항목

 테이블 확장

제목	설명
제네릭 인터페이스의 가변성(C#)	제네릭 인터페이스의 공변성(Covariance) 및 반공변성(Contravariance)에 대해 설명하고 .NET의 Variant 제네릭 인터페이스 목록을 제공합니다.
Variant 제네릭 인터페이스 만들기(C#)	사용자 지정 variant 인터페이스를 만드는 방법을 보여 줍니다.
제네릭 컬렉션용 인터페이스의 가변성 사용(C#)	<code>IEnumerable&lt;T&gt;</code> 및 <code>IComparable&lt;T&gt;</code> 인터페이스의 공변성(Covariance) 및 반공변성(Contravariance) 지원을 통해 코드를 다시 사용하는 방법을 보여 줍니다.
대리자의 가변성(C#)	제네릭 및 제네릭이 아닌 대리자의 공변성(Covariance) 및 반공변성(Contravariance)을 설명하고 .NET의 Variant 제네릭 인터페이스 목록을 제공합니다.
대리자의 가변성 사용(C#)	제네릭이 아닌 대리자의 공변성(Covariance) 및 반공변성(Contravariance) 지원을 사용하여 메서드 시그니처를 대리자 형식과 일치시키는 방법을 보여 줍니다.
Func 및 Action 제네릭 대리자에 가변성 사용(C#)	<code>Func</code> 및 <code>Action</code> 대리자의 공변성(Covariance) 및 반공변성(Contravariance) 지원을 통해 코드를 다시 사용하는 방법을 보여 줍니다.

## 피드백

이 페이지가 도움이 되었나요?

 Yes

 No

제품 사용자 의견 제공 

# 제네릭 인터페이스의 가변성(C#)

아티클 • 2023. 05. 10.

.NET Framework 4에서는 기존의 몇몇 제네릭 인터페이스에 대한 가변성 지원이 추가되었습니다. 가변성 지원은 이러한 인터페이스를 구현하는 클래스의 암시적 변환을 가능하게 합니다.

.NET Framework 4부터 다음 인터페이스는 variant입니다.

- `IEnumerable<T>`(`T`는 공변(covariant)임)
- `IEnumerator<T>`(`T`는 공변(covariant)임)
- `IQueryable<T>`(`T`는 공변(covariant)임)
- `IGrouping< TKey, TElement >`(`TKey` 및 `TElement`는 공변(covariant)임)
- `IComparer<T>`(`T`는 반공변(contravariant)임)
- `IEqualityComparer<T>`(`T`는 반공변(contravariant)임)
- `IComparable<T>`(`T`는 반공변(contravariant)임)

.NET Framework 4.5부터 다음 인터페이스는 variant입니다.

- `IReadOnlyList<T>`(`T`는 공변(covariant)임)
- `IReadOnlyCollection<T>`(`T`는 공변(covariant)임)

공변성(covariance)은 메서드가 인터페이스의 제네릭 형식 매개 변수에 정의된 것보다 더 많은 수의 파생된 반환 형식을 갖도록 허용합니다. 공변성(covariance) 기능을 설명하려면 `IEnumerable<Object>` 및 `IEnumerable<String>`이라는 제네릭 인터페이스를 고려하세요. `IEnumerable<String>` 인터페이스는 `IEnumerable<Object>` 인터페이스를 상속하지 않습니다. 그러나 `String` 형식은 `Object` 형식을 상속하며, 경우에 따라 이러한 인터페이스의 개체를 서로 할당할 수 있습니다. 다음 코드 예제에서 이를 확인할 수 있습니다.

C#

```
IEnumerator<String> strings = new List<String>();  
IEnumerator<Object> objects = strings;
```

.NET Framework의 이전 버전에서는 이 코드로 인해 C# 및 Visual Basic(`Option Strict`가 `on`인 경우)에서 컴파일 오류가 발생합니다. 그러나 `IEnumerable<T>` 인터페이스는 공변(covariant) 이므로 이제 다음 예제와 같이 `objects` 대신 `strings`를 사용할 수 있습니다.

반공변성(Contravariance)은 메서드가 인터페이스의 제네릭 매개 변수에 지정된 것보다 더 적은 수의 파생된 형식의 인수 형식을 갖도록 허용합니다. 반공변성(contravariance)을 설명하기 위해 사용자가 `BaseComparer` 클래스를 만들어 `BaseClass` 클래스의 인스턴스를 비교한다고 가정합니다. `BaseComparer` 클래스가 `IEqualityComparer<BaseClass>` 인터페이스를 구현합니다. `IEqualityComparer<T>` 인터페이스는 이제 반공변(contravariant)이므로 `BaseClass` 클래스를 상속하는 클래스의 인스턴스를 비교하는 데 `BaseComparer`를 사용할 수 있습니다. 다음 코드 예제에서 이를 확인할 수 있습니다.

C#

```
// Simple hierarchy of classes.
class BaseClass { }
class DerivedClass : BaseClass { }

// Comparer class.
class BaseComparer : IEqualityComparer<BaseClass>
{
    public int GetHashCode(BaseClass baseInstance)
    {
        return baseInstance.GetHashCode();
    }
    public bool Equals(BaseClass x, BaseClass y)
    {
        return x == y;
    }
}
class Program
{
    static void Test()
    {
        IEqualityComparer<BaseClass> baseComparer = new BaseComparer();

        // Implicit conversion of IEqualityComparer<BaseClass> to
        // IEqualityComparer<DerivedClass>.
        IEqualityComparer<DerivedClass> childComparer = baseComparer;
    }
}
```

추가 예제는 [제네릭 컬렉션용 인터페이스의 가변성 사용\(C#\)](#)을 참조하세요.

제네릭 인터페이스의 가변성은 참조 형식에 대해서만 지원됩니다. 값 형식은 가변성을 지원하지 않습니다. 정수는 값 형식으로 표시되므로 예를 들어 `IEnumerable<int>`를 `IEnumerable<object>`로 암시적으로 변환할 수 없습니다.

C#

```
IEnumerable<int> integers = new List<int>();
// The following statement generates a compiler error,
```

```
// because int is a value type.  
// IEnumerable<Object> objects = integers;
```

Variant 인터페이스를 구현하는 클래스는 여전히 비 variant라는 점에 유의해야 합니다. 예를 들어 `List<T>`는 공변(covariant) 인터페이스 `IEnumerable<T>`을 구현하지만 암시적으로 `List<String>`를 `List<Object>`으로 변환할 수 없습니다. 다음 코드 예제에서 이 내용을 보여 줍니다.

C#

```
// The following line generates a compiler error  
// because classes are invariant.  
// List<Object> list = new List<String>();  
  
// You can use the interface object instead.  
IEnumerable<Object> listObjects = new List<String>();
```

## 참조

- 제네릭 컬렉션용 인터페이스의 가변성 사용(C#)
- Variant 제네릭 인터페이스 만들기(C#)
- 제네릭 인터페이스
- 대리자의 가변성(C#)

# Variant 제네릭 인터페이스 만들기(C#)

아티클 • 2023. 04. 08.

인터페이스에서 제네릭 형식 매개 변수를 공변(covariant) 또는 반공변(contravariant)으로 선언할 수 있습니다. 공변성(covariance)은 인터페이스 메서드가 제네릭 형식 매개 변수에 정의된 것보다 더 많은 수의 파생된 반환 형식을 갖도록 허용합니다. 반공변성(contravariance)은 인터페이스 메서드가 제네릭 매개 변수에 지정된 것보다 더 적은 수의 파생된 형식의 인수 형식을 갖도록 허용합니다. 공변(covariant) 또는 반공변(contravariant) 제네릭 형식 매개 변수가 포함된 제네릭 인터페이스를 *variant*라고 합니다.

## ① 참고

.NET Framework 4에서는 기존의 몇몇 제네릭 인터페이스에 대한 가변성 지원이 추가되었습니다. .NET의 variant 인터페이스 목록은 [제네릭 인터페이스의 가변성\(C#\)](#)을 참조하세요.

## Variant 제네릭 인터페이스 선언

제네릭 형식 매개 변수에 `in` 및 `out` 키워드를 사용하여 Variant 제네릭 인터페이스를 선언할 수 있습니다.

## ② 중요

C#에서 `ref`, `in` 및 `out` 매개 변수는 variant일 수 없습니다. 또한 값 형식은 가변성을 지원하지 않습니다.

`out` 키워드를 사용하여 제네릭 형식 매개 변수를 공변(covariant)으로 선언할 수 있습니다. 공변(covariant) 형식은 다음 조건을 충족해야 합니다.

- 형식은 인터페이스 메서드의 반환 형식으로만 사용되고 메서드 인수의 형식으로 사용되지 않습니다. 이 내용은 `R` 형식이 공변(covariant)으로 선언된 다음 예제에서 설명합니다.

C#

```
interface ICovariant<out R>
{
    R GetSomething();
    // The following statement generates a compiler error.
```

```
// void SetSomething(R sampleArg);  
}
```

그러나 이 규칙에는 한 가지 예외가 있습니다. 반공변(contravariant) 제네릭 대리자 가 메서드 매개 변수로 있는 경우 형식을 이 대리자에 대한 제네릭 형식 매개 변수로 사용할 수 있습니다. 다음 예제에서 `R` 형식을 통해 이를 확인할 수 있습니다. 자세한 내용은 [대리자에서 가변성 사용\(C#\)](#) 및 [Func 및 Action 제네릭 대리자에 가변성 사용\(C#\)](#)을 참조하세요.

C#

```
interface ICovariant<out R>  
{  
    void DoSomething(Action<R> callback);  
}
```

- 형식은 인터페이스 메서드에 대한 제네릭 제약 조건으로 사용되지 않습니다. 이는 다음 코드에 설명되어 있습니다.

C#

```
interface ICovariant<out R>  
{  
    // The following statement generates a compiler error  
    // because you can use only contravariant or invariant types  
    // in generic constraints.  
    // void DoSomething<T>() where T : R;  
}
```

`in` 키워드를 사용하여 제네릭 형식 매개 변수를 반공변(contravariant)으로 선언할 수 있습니다. 반공변(contravariant) 형식은 메서드 인수의 형식으로서만 사용할 수 있으며 인터페이스 메서드의 반환 형식으로는 사용할 수 없습니다. 반공변(contravariant) 형식은 제네릭 제약 조건에 사용될 수도 있습니다. 다음 코드에서는 반공변(contravariant) 인터페이스를 선언하고 해당 메서드 중 하나에 제네릭 제약 조건을 사용하는 방법을 보여 줍니다.

C#

```
interface IContravariant<in A>  
{  
    void SetSomething(A sampleArg);  
    void DoSomething<T>() where T : A;  
    // The following statement generates a compiler error.  
    // A GetSomething();  
}
```

같은 인터페이스에서 그러나 서로 다른 형식 매개 변수에 대해 공변성(covariance) 및 반공변성(contravariance)을 모두 지원하는 것도 가능합니다.

C#

```
interface IVariant<out R, in A>
{
    R GetSomething();
    void SetSomething(A sampleArg);
    R GetSetSomethings(A sampleArg);
}
```

## Variant 제네릭 인터페이스 구현

고정(invariant) 인터페이스에 사용되는 같은 구문을 사용하여 클래스에서 Variant 제네릭 인터페이스를 구현합니다. 다음 코드 예제에서는 제네릭 클래스에서 공변(covariant) 인터페이스를 구현하는 방법을 보여 줍니다.

C#

```
interface ICovariant<out R>
{
    R GetSomething();
}

class SampleImplementation<R> : ICovariant<R>
{
    public R GetSomething()
    {
        // Some code.
        return default(R);
    }
}
```

Variant 인터페이스를 구현하는 클래스는 고정입니다. 예를 들어, 다음과 같은 코드를 생각해 볼 수 있습니다.

C#

```
// The interface is covariant.
ICovariant<Button> ibutton = new SampleImplementation<Button>();
ICovariant<Object> iobj = ibutton;

// The class is invariant.
SampleImplementation<Button> button = new SampleImplementation<Button>();
// The following statement generates a compiler error
// because classes are invariant.
// SampleImplementation<Object> obj = button;
```

# Variant 제네릭 인터페이스 확장

Variant 제네릭 인터페이스를 확장할 경우 `in` 및 `out` 키워드를 사용하여 파생 인터페이스가 가변성을 지원하는지 명시적으로 지정해야 합니다. 컴파일러에서는 확장되고 있는 인터페이스에서 가변성을 유추하지 않습니다. 예를 들어 다음 인터페이스를 살펴봅니다.

C#

```
interface ICovariant<out T> { }
interface IIInvariant<T> : ICovariant<T> { }
interface IExtCovariant<out T> : ICovariant<T> { }
```

두 인터페이스가 모두 같은 인터페이스를 확장하더라도 `IIInvariant<T>` 인터페이스에서 제네릭 형식 매개 변수 `T`는 고정이지만, `IExtCovariant<out T>`에서 형식 매개 변수는 공변(covariant)입니다. 반공변(contravariant) 제네릭 형식 매개 변수에는 같은 규칙이 적용됩니다.

제네릭 형식 매개 변수 `T`가 공변(covariant)인 인터페이스 및 확장 인터페이스에서 제네릭 형식 매개 변수 `T`가 고정인 경우 반공변(contravariant)인 인터페이스를 둘 다 확장하는 인터페이스를 만들 수 있습니다. 다음 코드 예제에서 이 내용을 보여 줍니다.

C#

```
interface ICovariant<out T> { }
interface IContravariant<in T> { }
interface IIInvariant<T> : ICovariant<T>, IContravariant<T> { }
```

그러나 한 인터페이스에서 제네릭 형식 매개 변수 `T`가 공변(covariant)으로 선언되면 확장 인터페이스에서는 이 매개 변수를 반공변(contravariant)으로 선언할 수 없고, 반대의 경우도 마찬가지입니다. 다음 코드 예제에서 이 내용을 보여 줍니다.

C#

```
interface ICovariant<out T> { }
// The following statement generates a compiler error.
// interface ICoContraVariant<in T> : ICovariant<T> { }
```

## 모호성 방지

Variant 제네릭 인터페이스를 구현할 경우 가변성으로 인해 모호성이 나타나는 경우가 있습니다. 이러한 모호성을 피해야 합니다.

예를 들어 서로 다른 제네릭 형식 매개 변수가 포함된 같은 Variant 제네릭 인터페이스를 한 클래스에서 명시적으로 구현하면 모호성이 발생할 수 있습니다. 컴파일러에서는 이 경우 오류를 생성하지 않지만 런타임에 선택될 인터페이스 구현이 지정되지 않습니다. 이 모호성으로 인해 코드에서 미묘한 버그가 발생할 수 있습니다. 다음 코드 예제를 살펴봅니다.

C#

```
// Simple class hierarchy.
class Animal { }
class Cat : Animal { }
class Dog : Animal { }

// This class introduces ambiguity
// because IEnumerable<out T> is covariant.
class Pets : IEnumerable<Cat>, IEnumerable<Dog>
{
    IEnumerator<Cat> IEnumerable<Cat>.GetEnumerator()
    {
        Console.WriteLine("Cat");
        // Some code.
        return null;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        // Some code.
        return null;
    }

    IEnumerator<Dog> IEnumerable<Dog>.GetEnumerator()
    {
        Console.WriteLine("Dog");
        // Some code.
        return null;
    }
}

class Program
{
    public static void Test()
    {
        IEnumerable<Animal> pets = new Pets();
        pets.GetEnumerator();
    }
}
```

이 예제에서는 `pets.GetEnumerator` 메서드가 `Cat` 및 `Dog` 중에 선택하는 방법이 지정되어 있지 않습니다. 이로 인해 코드에서 문제가 발생할 수 있습니다.

## 참조

- 제네릭 인터페이스의 가변성(C#)
- Func 및 Action 제네릭 대리자에 가변성 사용(C#)

# 제네릭 컬렉션용 인터페이스의 가변성 사용(C#)

아티클 • 2023. 05. 10.

공변(covariant) 인터페이스는 메서드가 인터페이스에 지정된 것보다 더 많은 수의 파생된 형식을 반환하도록 허용합니다. 반공변(contravariant) 인터페이스는 메서드가 인터페이스에 지정된 것보다 더 적은 파생된 형식의 매개 변수를 수락하도록 허용합니다.

.NET Framework 4에서는 몇 가지 기존 인터페이스가 공변(covariant) 및 반공변(contravariant)이 되었습니다. 여기에는 `IEnumerable<T>` 및 `IComparable<T>`이 포함됩니다. 따라서 파생된 형식의 컬렉션에 대한 기본 형식의 제네릭 컬렉션과 함께 작동하는 메서드를 다시 사용할 수 있습니다.

.NET의 variant 인터페이스 목록은 [제네릭 인터페이스의 가변성\(C#\)](#)을 참조하세요.

## 제네릭 컬렉션 변환

다음 예제에서는 `IEnumerable<T>` 인터페이스에서 공변성(Covariance) 지원의 이점을 보여 줍니다. `PrintFullName` 메서드는 `IEnumerable<Person>` 형식의 컬렉션을 매개 변수로 수락합니다. 그러나 `Employee`는 `Person`을 상속하므로 `IEnumerable<Employee>` 형식의 컬렉션에 대해 이를 다시 사용할 수 있습니다.

C#

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

class Program
{
    // The method has a parameter of the IEnumerable<Person> type.
    public static void PrintFullName(IEnumerable<Person> persons)
    {
        foreach (Person person in persons)
        {
            Console.WriteLine("Name: {0} {1}",
                person.FirstName, person.LastName);
        }
    }
}
```

```

public static void Test()
{
    IEnumerable<Employee> employees = new List<Employee>();

    // You can pass IEnumerable<Employee>,
    // although the method expects IEnumerable<Person>.

    PrintFullName(employees);

}
}

```

## 제네릭 컬렉션 비교

다음 예제에서는 `IEqualityComparer<T>` 인터페이스에서 반공변성(Contravariance) 지원의 이점을 보여 줍니다. `PersonComparer` 클래스가 `IEqualityComparer<Person>` 인터페이스를 구현합니다. 그러나 `Employee`는 `Person`을 상속하므로 `Employee` 형식 개체의 시퀀스를 비교하기 위해 이 클래스를 다시 사용할 수 있습니다.

C#

```

// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

// The custom comparer for the Person type
// with standard implementations of Equals()
// and GetHashCode() methods.
class PersonComparer : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        if (Object.ReferenceEquals(x, y)) return true;
        if (Object.ReferenceEquals(x, null) ||
            Object.ReferenceEquals(y, null))
            return false;
        return x.FirstName == y.FirstName && x.LastName == y.LastName;
    }
    public int GetHashCode(Person person)
    {
        if (Object.ReferenceEquals(person, null)) return 0;
        int hashFirstName = person.FirstName == null
            ? 0 : person.FirstName.GetHashCode();
        int hashLastName = person.LastName.GetHashCode();
        return hashFirstName ^ hashLastName;
    }
}

```

```

        }

}

class Program
{

    public static void Test()
    {
        List<Employee> employees = new List<Employee> {
            new Employee() {FirstName = "Michael", LastName =
"Alexander"}, 
            new Employee() {FirstName = "Jeff", LastName = "Price"}
        };

        // You can pass PersonComparer,
        // which implements IEqualityComparer<Person>,
        // although the method expects IEqualityComparer<Employee>.

        IEnumerable<Employee> noduplicates =
            employees.Distinct<Employee>(new PersonComparer());
        
        foreach (var employee in noduplicates)
            Console.WriteLine(employee.FirstName + " " + employee.LastName);
    }
}

```

## 참조

- 제네릭 인터페이스의 가변성(C#)

# 대리자의 가변성(C#)

아티클 • 2023. 05. 09.

.NET Framework 3.5에는 메서드 시그니처를 C#에 있는 모든 대리자의 대리자 형식과 일치시키는 가변성 지원이 추가되었습니다. 즉, 일치하는 시그니처가 있는 메서드만이 아니라 더 많은 파생된 형식(공변성(covariance))을 반환하는 메서드 또는 대리자 형식에 지정된 것보다 더 적은 수의 파생된 형식(반공변성(contravariance))을 가지고 있는 매개 변수를 수락하는 메서드도 대리자에 할당할 수 있습니다. 여기에는 제네릭 및 비 제네릭 대리자가 모두 포함됩니다.

다음과 같이 두 개의 클래스 및 두 개의 대리자(제네릭 및 비 제네릭)를 가지고 있는 코드를 예로 들어보겠습니다.

C#

```
public class First { }
public class Second : First { }
public delegate First SampleDelegate(Second a);
public delegate R SampleGenericDelegate<A, R>(A a);
```

`SampleDelegate` 또는 `SampleGenericDelegate<A, R>` 형식의 대리자를 만들 때 다음 메서드 중 하나를 할당할 수 있습니다.

C#

```
// Matching signature.
public static First ASecondRFirst(Second second)
{ return new First(); }

// The return type is more derived.
public static Second ASecondRSecond(Second second)
{ return new Second(); }

// The argument type is less derived.
public static First AFirstRFirst(First first)
{ return new First(); }

// The return type is more derived
// and the argument type is less derived.
public static Second AFirstRSecond(First first)
{ return new Second(); }
```

다음 코드 예제에서는 메서드 시그니처 및 대리자 형식 사이의 암시적 변환을 보여 줍니다.

C#

```
// Assigning a method with a matching signature
// to a non-generic delegate. No conversion is necessary.
SampleDelegate dNonGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a non-generic delegate.
// The implicit conversion is used.
SampleDelegate dNonGenericConversion = AFirstRSecond;

// Assigning a method with a matching signature to a generic delegate.
// No conversion is necessary.
SampleGenericDelegate<Second, First> dGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a generic delegate.
// The implicit conversion is used.
SampleGenericDelegate<Second, First> dGenericConversion = AFirstRSecond;
```

더 많은 예제는 [대리자에서 가변성 사용\(C#\)](#) 및 [Func 및 Action 제네릭 대리자에 가변성 사용\(C#\)](#)을 참조하세요.

## 제네릭 형식 매개 변수에서의 가변성

.NET Framework 4 이상에서는 가변성에 필요한 대로 형식이 서로 간에 상속된 경우 제네릭 형식 매개 변수로 지정한 서로 다른 형식을 가지고 있는 제네릭 대리자를 상호 간에 할당할 수 있도록, 대리자 간 암시적 변환을 사용하도록 설정할 수 있습니다.

암시적 변환을 사용하도록 설정하려면 `in` 및 `out` 키워드를 사용하여 대리자에서 제네릭 매개 변수를 공변(covariant) 또는 반공변(contravariant)으로 선언해야 합니다.

다음 코드 예제에서는 공변(covariant) 제네릭 형식 매개 변수가 있는 대리자를 만드는 방법을 보여 줍니다.

C#

```
// Type T is declared covariant by using the out keyword.
public delegate T SampleGenericDelegate <out T>();

public static void Test()
{
    SampleGenericDelegate <String> dString = () => " ";

    // You can assign delegates to each other,
    // because the type T is declared covariant.
    SampleGenericDelegate <Object> dObject = dString;
}
```

메서드 시그니처를 대리자 형식과 일치시키는 용도로만 가변성 지원을 사용하고 `in` 및 `out` 키워드를 사용하지 않는 경우, 대리자를 동일한 람다 식 또는 메서드로 인스턴스화할 수는 있지만 한 대리자를 다른 대리자에 할당할 수는 없는 경우가 더러 있습니다.

다음 코드 예제에서, `String`은 `Object`를 상속하지만 `SampleGenericDelegate<String>`을 `SampleGenericDelegate<Object>`로 명시적으로 변환할 수 없습니다. `T` 제네릭 매개 변수를 `out` 키워드로 표시하면 이 문제를 수정할 수 있습니다.

C#

```
public delegate T SampleGenericDelegate<T>();

public static void Test()
{
    SampleGenericDelegate<String> dString = () => " ";

    // You can assign the dObject delegate
    // to the same lambda expression as dString delegate
    // because of the variance support for
    // matching method signatures with delegate types.
    SampleGenericDelegate<Object> dObject = () => " ";

    // The following statement generates a compiler error
    // because the generic type T is not marked as covariant.
    // SampleGenericDelegate <Object> dObject = dString;

}
```

## .NET에 Variant 형식 매개 변수를 가지고 있는 제네릭 대리자

.NET Framework 4에는 기존의 몇몇 제네릭 대리자에서 제네릭 형식 매개 변수에 대한 가변성 지원이 추가되었습니다.

- System 네임스페이스의 `Action` 대리자(예: `Action<T>` 및 `Action<T1,T2>`)
- System 네임스페이스의 `Func` 대리자(예: `Func<TResult>` 및 `Func<T,TResult>`)
- `Predicate<T>` 대리자
- `Comparison<T>` 대리자
- `Converter<TInput,TOOutput>` 대리자

자세한 정보 및 예제는 [Func 및 Action 제네릭 대리자에 가변성 사용\(C#\)](#)을 참조하세요.

## 제네릭 대리자에서 Variant 형식 매개 변수 선언

제네릭 대리자가 공변(covariant) 또는 반공변(contravariant) 제네릭 형식 매개 변수를 가지고 있는 경우 이를 *variant* 제네릭 대리자라고 할 수 있습니다.

`out` 키워드를 사용하여 제네릭 대리자에서 제네릭 형식 매개 변수를 공변(covariant)으로 선언할 수 있습니다. 공변(covariant) 형식은 메서드 반환 형식으로만 사용할 수 있으며 메서드 인수의 형식으로는 사용할 수 없습니다. 다음 코드 예제에서는 공변(covariant) 제네릭 대리자를 선언하는 방법을 보여 줍니다.

C#

```
public delegate R DCovariant<out R>();
```

`in` 키워드를 사용하여 제네릭 대리자에서 제네릭 형식 매개 변수를 반공변(contravariant)으로 선언할 수 있습니다. 반공변(contravariant) 형식은 메서드 인수의 형식으로서만 사용할 수 있으며 메서드 반환 형식으로는 사용할 수 없습니다. 다음 코드 예제에서는 반공변(contravariant) 제네릭 대리자를 선언하는 방법을 보여 줍니다.

C#

```
public delegate void DContravariant<in A>(A a);
```

### ① 중요

C#의 `ref`, `in` 및 `out` 매개 변수는 variant로 표시할 수 없습니다.

동일한 대리자에서, 그러나 서로 다른 형식 매개 변수에 대해 분산 및 공변성(covariance)을 모두 지원하는 것도 가능합니다. 다음 예제에서 이를 확인할 수 있습니다.

C#

```
public delegate R DVariant<in A, out R>(A a);
```

## Variant 제네릭 대리자 인스턴스화 및 호출

비 variant 대리자를 인스턴스화 및 호출하듯 variant 대리자를 인스턴스화 및 호출할 수 있습니다. 다음 예제에서는 람다 식을 사용하여 대리자가 인스턴스화됩니다.

C#

```
DVariant<String, String> dvariant = (String str) => str + " ";
dvariant("test");
```

## Variant 제네릭 대리자 결합

Variant 대리자를 결합하지 마세요. [Combine](#) 메서드는 variant 대리자 변환을 지원하지 않으며 대리자가 정확히 동일한 형식일 것으로 예상합니다. 따라서 다음 코드 예제와 같이 [Combine](#) 메서드를 사용하거나 `+` 연산자를 사용하여 대리자를 결합하면 런타임 예외가 발생할 수 있습니다.

C#

```
Action<object> actObj = x => Console.WriteLine("object: {0}", x);
Action<string> actStr = x => Console.WriteLine("string: {0}", x);
// All of the following statements throw exceptions at run time.
// Action<string> actCombine = actStr + actObj;
// actStr += actObj;
// Delegate.Combine(actStr, actObj);
```

## 값 및 참조 형식에 대한 제네릭 형식 매개 변수에서의 가변성

제네릭 형식 매개 변수에 대한 가변성은 참조 형식에 대해서만 지원됩니다. 정수는 값 형식이므로, 예를 들어 `DVariant<int>`를 명시적으로 `DVariant<Object>` 또는 `DVariant<long>`으로 변환할 수 없습니다.

다음 예제에서는 제네릭 형식 매개 변수에서의 가변성이 값 형식에 대해 지원되지 않음을 보여 줍니다.

C#

```
// The type T is covariant.
public delegate T DVariant<out T>();

// The type T is invariant.
public delegate T DInvariant<T>();

public static void Test()
{
    int i = 0;
    DInvariant<int> dInt = () => i;
    DVariant<int> dVariantInt = () => i;

    // All of the following statements generate a compiler error
```

```
// because type variance in generic parameters is not supported
// for value types, even if generic type parameters are declared
variant.
// DInvariant<Object> dObject = dInt;
// DInvariant<long> dLong = dInt;
// DVariant<Object> dVariantObject = dVariantInt;
// DVariant<long> dVariantLong = dVariantInt;
}
```

## 참조

- 제네릭
- Func 및 Action 제네릭 대리자에 가변성 사용(C#)
- 대리자를 결합하는 방법(멀티캐스트 대리자)

# 대리자의 가변성 사용(C#)

아티클 • 2024. 05. 16.

메서드를 대리자에 할당하면 공변성(covariance) 및 반공변성(Contravariance)은 대리자 형식과 메서드 시그니처의 일치를 확인하는 유연성을 제공합니다. 공변성(covariance)은 메서드가 대리자에 정의된 것보다 더 많은 수의 파생된 형식을 반환하도록 허용합니다. 반공변성(contravariance)은 메서드가 대리자 형식보다 더 적은 수의 파생된 매개 변수 형식을 갖도록 허용합니다.

## 예제 1: 공변성(Covariance)

### 설명

이 예제에서는 대리자를 대리자 시그니처의 반환 형식에서 파생된 반환 형식이 있는 메서드와 함께 사용하는 방법을 보여 줍니다. `DogsHandler`에서 반환된 데이터 형식은 `Dogs`이고, 이 형식은 대리자에 정의된 `Mammals` 형식에서 파생됩니다.

### 코드

```
C#  
  
class Mammals {}  
class Dogs : Mammals {}  
  
class Program  
{  
    // Define the delegate.  
    public delegate Mammals HandlerMethod();  
  
    public static Mammals MammalsHandler()  
    {  
        return null;  
    }  
  
    public static Dogs DogsHandler()  
    {  
        return null;  
    }  
  
    static void Test()  
    {  
        HandlerMethod handlerMammals = MammalsHandler;  
  
        // Covariance enables this assignment.  
        HandlerMethod handlerDogs = DogsHandler;
```

```
    }  
}
```

## 예제 2: 반공변성(Contravariance)

### 설명

이 예제에서는 대리자를 대리자 시그니처 매개 변수 형식의 기본 형식을 사용하는 매개 변수를 가지고 있는 메서드와 함께 사용하는 방법을 보여 줍니다. 반공변성 (contravariance)에서는 별도의 여러 처리기 대신 하나의 이벤트 처리기를 사용할 수 있습니다. 다음 예제는 두 개의 대리자를 사용합니다.

- Button.KeyDown 이벤트의 시그니처를 정의하는 KeyEventHandler 대리자. 해당 시그니처는 다음과 같습니다.

```
C#
```

```
public delegate void KeyEventHandler(object sender, KeyEventArgs e)
```

- Button.MouseClick 이벤트의 시그니처를 정의하는 MouseEventHandler 대리자. 해당 시그니처는 다음과 같습니다.

```
C#
```

```
public delegate void MouseEventHandler(object sender, MouseEventArgs e)
```

예제에서는 EventArgs 매개 변수를 사용하여 이벤트 처리기를 정의하고 이를 사용하여 Button.KeyDown 및 Button.MouseClick 이벤트를 모두 처리합니다. EventArgs은(는) KeyEventArgs 및 MouseEventArgs의 기본 형식이므로 이 작업을 수행할 수 있습니다.

### 코드

```
C#
```

```
// Event handler that accepts a parameter of the EventArgs type.  
private void MultiHandler(object sender, System.EventArgs e)  
{  
    label1.Text = System.DateTime.Now.ToString();  
}  
  
public Form1()  
{  
    InitializeComponent();  
}
```

```
// You can use a method that has an EventArgs parameter,  
// although the event expects the KeyEventArgs parameter.  
this.button1.KeyDown += this.MultiHandler;  
  
// You can use the same method  
// for an event that expects the MouseEventArgs parameter.  
this.button1.MouseClick += this.MultiHandler;  
  
}
```

## 참고 항목

- 대리자의 가변성(C#)
- Func 및 Action 제네릭 대리자에 가변성 사용(C#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# Func 및 Action 제네릭 대리자에 가변성 사용(C#)

아티클 • 2023. 04. 07.

이러한 예제는 메서드를 다시 사용하고 코드의 유연성을 높이기 위해 `Func` 및 `Action` 제네릭 대리자에서 공변성(covariance) 및 반공변성(contravariance)을 사용하는 방법을 보여 줍니다.

공변성(covariance) 및 반공변성(contravariance)에 대한 자세한 내용은 [대리자에서의 분산\(C#\)](#)을 참조하세요.

## 공변 형식 매개 변수가 있는 대리자 사용

다음 예제는 `Func` 대리자에서 공변성(covariance) 지원의 이점을 보여 줍니다.

`FindByTitle` 메서드는 `String` 형식의 매개 변수를 가져오고 `Employee` 형식의 개체를 반환합니다. 그러나 `Employee`는 `Person`을 상속하므로 이 메서드를 `Func<String, Person>` 대리자에 할당할 수 있습니다.

C#

```
// Simple hierarchy of classes.
public class Person { }
public class Employee : Person { }
class Program
{
    static Employee FindByTitle(String title)
    {
        // This is a stub for a method that returns
        // an employee that has the specified title.
        return new Employee();
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Func<String, Employee> findEmployee = FindByTitle;

        // The delegate expects a method to return Person,
        // but you can assign it a method that returns Employee.
        Func<String, Person> findPerson = FindByTitle;

        // You can also assign a delegate
        // that returns a more derived type
        // to a delegate that returns a less derived type.
        findPerson = findEmployee;
```

```
    }  
}
```

## 반공변 형식 매개 변수가 있는 대리자 사용

다음 예제에서는 제네릭 `Action` 대리자에서 반공변성(contravariance) 지원의 이점을 보여 줍니다. `AddToContacts` 메서드는 `Person` 형식의 매개 변수를 사용합니다. 그러나 `Employee`는 `Person`을 상속하므로 이 메서드를 `Action<Employee>` 대리자에 할당할 수 있습니다.

C#

```
public class Person { }  
public class Employee : Person { }  
class Program  
{  
    static void AddToContacts(Person person)  
    {  
        // This method adds a Person object  
        // to a contact list.  
    }  
  
    static void Test()  
    {  
        // Create an instance of the delegate without using variance.  
        Action<Person> addPersonToContacts = AddToContacts;  
  
        // The Action delegate expects  
        // a method that has an Employee parameter,  
        // but you can assign it a method that has a Person parameter  
        // because Employee derives from Person.  
        Action<Employee> addEmployeeToContacts = AddToContacts;  
  
        // You can also assign a delegate  
        // that accepts a less derived parameter to a delegate  
        // that accepts a more derived parameter.  
        addEmployeeToContacts = addPersonToContacts;  
    }  
}
```

## 참조

- 공변성(Covariance) 및 반공변성(Contravariance)(C#)
- 제네릭

# 반복기(C#)

아티클 • 2023. 04. 08.

반복기는 목록 및 배열과 같은 컬렉션을 단계별로 실행하는 데 사용할 수 있습니다.

반복기 메서드 또는 `get` 접근자는 컬렉션에 대해 사용자 지정 반복을 수행합니다. 반복기 메서드는 `yield return` 문을 사용하여 각 요소를 한 번에 하나씩 반환합니다. `yield return` 문에 도달하면 코드의 현재 위치가 기억됩니다. 다음에 반복기 함수가 호출되면 해당 위치에서 실행이 다시 시작됩니다.

클라이언트 코드에서 `foreach` 문 또는 LINQ 쿼리를 사용하여 반복기를 이용합니다.

다음 예제에서 `foreach` 루프의 첫 번째 반복은 첫 번째 `yield return` 문에 도달할 때까지 `SomeNumbers` 반복기 메서드에서 실행이 계속되도록 합니다. 이 반복은 3 값을 반환하며 반복기 메서드에서 현재 위치는 유지됩니다. 루프의 다음 반복에서는 반복기 메서드의 실행이 중지되었던 위치에서 계속되고 `yield return` 문에 도달하면 다시 중지됩니다. 이 반복은 값 5를 반환하며 반복기 메서드에서 현재 위치는 다시 유지됩니다. 루프는 반복기 메서드의 끝에 도달하면 완료됩니다.

C#

```
static void Main()
{
    foreach (int number in SomeNumbers())
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 3 5 8
    Console.ReadKey();
}

public static System.Collections.IEnumerable SomeNumbers()
{
    yield return 3;
    yield return 5;
    yield return 8;
}
```

반복기 메서드 또는 `get` 접근자의 반환 형식은 `IEnumerable`, `IEnumerable<T>`, `IEnumerator` 또는 `IEnumerator<T>` 일 수 있습니다.

`yield break` 문을 사용하여 반복기를 종료할 수 있습니다.

① 참고

단순 반복기 예제를 제외한 이 항목의 모든 예제에 대해 System.Collections 및 System.Collections.Generic 네임스페이스에 대한 using 지시문을 포함하세요.

## 단순 반복기

다음 예제에는 `for` 루프 내에 단일 `yield return` 문이 있습니다. `Main`에서 `foreach` 문 본문을 반복할 때마다 다음 `yield return` 문으로 진행하는 반복기 함수에 대한 호출이 생성됩니다.

C#

```
static void Main()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    // Output: 6 8 10 12 14 16 18
    Console.ReadKey();
}

public static System.Collections.Generic.IEnumerable<int>
    EvenSequence(int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (int number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```

## 컬렉션 클래스 만들기

다음 예제에서 `DaysOfWeek` 클래스는 `IEnumerable` 인터페이스를 구현하며, `GetEnumerator` 메서드가 필요합니다. 컴파일러는 `GetEnumerator` 메서드를 암시적으로 호출하며, `IEnumerator`가 반환됩니다.

`GetEnumerator` 메서드는 `yield return` 문을 사용하여 각 문자열을 한 번에 하나씩 반환합니다.

C#

```

static void Main()
{
    DaysOfTheWeek days = new DaysOfTheWeek();

    foreach (string day in days)
    {
        Console.Write(day + " ");
    }
    // Output: Sun Mon Tue Wed Thu Fri Sat
    Console.ReadKey();
}

public class DaysOfTheWeek : IEnumerable
{
    private string[] days = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri",
    "Sat" };

    public IEnumerator GetEnumerator()
    {
        for (int index = 0; index < days.Length; index++)
        {
            // Yield each day of the week.
            yield return days[index];
        }
    }
}

```

다음 예제에서는 동물 컬렉션을 포함하는 `zoo` 클래스를 만듭니다.

클래스 인스턴스(`theZoo`)를 참조하는 `foreach` 문은 `GetEnumerator` 메서드를 암시적으로 호출합니다. `Birds` 및 `Mammals` 속성을 참조하는 `foreach` 문은 `AnimalsForType` 명명된 반복기 메서드를 사용합니다.

C#

```

static void Main()
{
    Zoo theZoo = new Zoo();

    theZoo.AddMammal("Whale");
    theZoo.AddMammal("Rhinoceros");
    theZoo.AddBird("Penguin");
    theZoo.AddBird("Warbler");

    foreach (string name in theZoo)
    {
        Console.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Whale Rhinoceros Penguin Warbler
}

```

```
foreach (string name in theZoo.Birds)
{
    Console.WriteLine(name + " ");
}
Console.WriteLine();
// Output: Penguin Warbler

foreach (string name in theZoo.Mammals)
{
    Console.WriteLine(name + " ");
}
Console.WriteLine();
// Output: Whale Rhinoceros

Console.ReadKey();
}

public class Zoo : IEnumerable
{
    // Private members.
    private List<Animal> animals = new List<Animal>();

    // Public methods.
    public void AddMammal(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Mammal });
    }

    public void AddBird(string name)
    {
        animals.Add(new Animal { Name = name, Type = Animal.TypeEnum.Bird });
    }

    public IEnumerator GetEnumerator()
    {
        foreach (Animal theAnimal in animals)
        {
            yield return theAnimal.Name;
        }
    }

    // Public members.
    public IEnumerable Mammals
    {
        get { return AnimalsForType(Animal.TypeEnum.Mammal); }
    }

    public IEnumerable Birds
    {
        get { return AnimalsForType(Animal.TypeEnum.Bird); }
    }

    // Private methods.
}
```

```

private IEnumerable AnimalsForType(Animal.TypeEnum type)
{
    foreach (Animal theAnimal in animals)
    {
        if (theAnimal.Type == type)
        {
            yield return theAnimal.Name;
        }
    }
}

// Private class.
private class Animal
{
    public enum TypeEnum { Bird, Mammal }

    public string Name { get; set; }
    public TypeEnum Type { get; set; }
}
}

```

## 제네릭 목록과 함께 반복기 사용

다음 예제에서 `Stack<T>` 제네릭 클래스는 `IEnumerable<T>` 제네릭 인터페이스를 구현합니다. `Push` 메서드는 `T` 형식의 배열에 값을 할당합니다. `GetEnumerator` 메서드는 `yield return` 문을 사용하여 배열 값을 반환합니다.

제네릭 `GetEnumerator` 메서드뿐 아니라 제네릭이 아닌 `GetEnumerator` 메서드도 구현해야 합니다. `IEnumerable<T>`이 `IEnumerable`에서 상속하기 때문입니다. 제네릭이 아닌 구현은 제네릭 구현을 따릅니다.

예제에서는 명명된 반복기를 사용하여 동일한 데이터 컬렉션을 반복하는 다양한 방법을 지원합니다. 이러한 명명된 반복기는 `TopToBottom` 및 `BottomToTop` 속성과 `TopN` 메서드입니다.

`BottomToTop` 속성은 `get` 접근자에서 반복기를 사용합니다.

C#

```

static void Main()
{
    Stack<int> theStack = new Stack<int>();

    // Add items to the stack.
    for (int number = 0; number <= 9; number++)
    {
        theStack.Push(number);
    }
}

```

```

// Retrieve items from the stack.
// foreach is allowed because theStack implements IEnumerable<int>.
foreach (int number in theStack)
{
    Console.Write("{0} ", number);
}
Console.WriteLine();
// Output: 9 8 7 6 5 4 3 2 1 0

// foreach is allowed, because theStack.TopToBottom returns
IEnumerable(Of Integer).
foreach (int number in theStack.TopToBottom)
{
    Console.Write("{0} ", number);
}
Console.WriteLine();
// Output: 9 8 7 6 5 4 3 2 1 0

foreach (int number in theStack.BottomToTop)
{
    Console.Write("{0} ", number);
}
Console.WriteLine();
// Output: 0 1 2 3 4 5 6 7 8 9

foreach (int number in theStack.TopN(7))
{
    Console.Write("{0} ", number);
}
Console.WriteLine();
// Output: 9 8 7 6 5 4 3

Console.ReadKey();
}

public class Stack<T> : IEnumerable<T>
{
    private T[] values = new T[100];
    private int top = 0;

    public void Push(T t)
    {
        values[top] = t;
        top++;
    }
    public T Pop()
    {
        top--;
        return values[top];
    }

    // This method implements the GetEnumerator method. It allows
    // an instance of the class to be used in a foreach statement.
    public IEnumerator<T> GetEnumerator()

```

```

    {
        for (int index = top - 1; index >= 0; index--)
        {
            yield return values[index];
        }
    }

    IEnumarator IEnumarable.GetEnumarator()
    {
        return GetEnumarator();
    }

    public IEnumarable<T> TopToBottom
    {
        get { return this; }
    }

    public IEnumarable<T> BottomToTop
    {
        get
        {
            for (int index = 0; index <= top - 1; index++)
            {
                yield return values[index];
            }
        }
    }

    public IEnumarable<T> TopN(int itemsFromTop)
    {
        // Return less than itemsFromTop if necessary.
        int startIndex = itemsFromTop >= top ? 0 : top - itemsFromTop;

        for (int index = top - 1; index >= startIndex; index--)
        {
            yield return values[index];
        }
    }
}

```

## 구문 정보

반복기는 메서드 또는 `get` 접근자로 발생할 수 있습니다. 이벤트, 인스턴스 생성자, 정적 생성자 또는 정적 종료자에서는 반복기가 발생할 수 없습니다.

반복기에서 반환된 `IEnumarable<T>`의 형식 인수에 대한 `yield return` 문의 식 형식에는 암시적 변환이 있어야 합니다.

C#에서 반복기 메서드는 `in`, `ref` 또는 `out` 매개 변수를 사용할 수 없습니다.

C#에서 `yield`는 예약어가 아니며 `return` 또는 `break` 키워드 앞에서 사용되는 경우에만 특별한 의미가 있습니다.

## 기술 구현

반복기를 메서드로 작성하는 경우에도 컴파일러는 실제로 상태 시스템인 중첩 클래스로 변환합니다. 이 클래스는 클라이언트 코드의 `foreach` 루프가 계속되는 한 반복기의 위치를 추적합니다.

컴파일러의 용도를 확인하려면 Ilasm.exe 도구를 사용하여 반복기 메서드에 대해 생성되는 Microsoft Intermediate Language 코드를 확인할 수 있습니다.

`class` 또는 `struct`에 대해 반복기를 만드는 경우 전체 `IEnumerator` 인터페이스를 구현할 필요가 없습니다. 컴파일러는 반복기를 검색할 경우 `IEnumerator` 또는 `IEnumerator<T>` 인터페이스의 `Current`, `MoveNext` 및 `Dispose` 메서드를 자동으로 생성합니다.

`foreach` 루프를 연속 반복하거나 `IEnumerator.MoveNext`를 직접 호출하면 다음 반복기 코드 본문이 이전 `yield return` 문 다음에 다시 시작됩니다. 그런 후 반복기 본문의 끝에 도달하거나 `yield break` 문이 나타날 때까지 다음 `yield return` 문을 계속 실행합니다.

반복기는 `IEnumerator.Reset` 메서드를 지원하지 않습니다. 처음부터 다시 반복하려면 새 반복기를 가져와야 합니다. 반복기 메서드에 의해 반환된 반복기에서 `Reset`를 호출하면 `NotSupportedException`가 throw됩니다.

자세한 내용은 [C# 언어 사양](#)을 참조하세요.

## 반복기 사용

반복기를 사용하면 복잡한 코드를 사용하여 목록 시퀀스를 채워야 하는 경우 `foreach` 루프의 단순성을 유지할 수 있습니다. 이 기능은 다음을 수행하려는 경우에 유용할 수 있습니다.

- 첫 번째 `foreach` 루프 반복 후 목록 시퀀스를 수정합니다.
- 첫 번째 `foreach` 루프 반복 전에 큰 목록이 완전히 로드되지 않도록 합니다. 한 가지 예로 테이블 행을 일괄 로드하는 페이지 페치가 있으며, 또 다른 예로 .NET에서 반복기를 구현하는 `EnumerateFiles` 메서드가 있습니다.
- 반복기에서 목록 작성을 캡슐화합니다. 반복기 메서드에서 목록을 빌드한 후 루프에서 각 결과를 생성할 수 있습니다.

## 참조

- System.Collections.Generic
- IEnumerable<T>
- foreach, in
- 배열에 foreach 사용
- 제네릭

# 문(C# 프로그래밍 가이드)

아티클 • 2024. 03. 12.

프로그램이 수행하는 작업은 문으로 표현됩니다. 일반적인 작업으로 지정된 조건에 따라 변수 선언, 값 할당, 메서드 호출, 컬렉션 반복, 하나 또는 다른 코드 블록으로 분기 등이 있습니다. 프로그램에서 문이 실행되는 순서를 제어 흐름 또는 실행 흐름이라고 합니다. 제어 흐름은 프로그램이 런타임 시 수신하는 입력에 대응하는 방식에 따라 프로그램을 실행할 때마다 달라질 수 있습니다.

문은 세미콜론으로 끝나는 코드 한 줄이나 일련의 한 줄 문으로 이루어진 블록일 수 있습니다. 문 블록은 {} 괄호로 묶여 있으며 중첩 블록을 포함할 수 있습니다. 다음 코드는 한 줄 문과 여러 줄 문 블록의 두 가지 예제를 보여 줍니다.

C#

```
public static void Main()
{
    // Declaration statement.
    int counter;

    // Assignment statement.
    counter = 1;

    // Error! This is an expression, not an expression statement.
    // counter + 1;

    // Declaration statements with initializers are functionally
    // equivalent to declaration statement followed by assignment
    // statement:
    int[] radii = [15, 32, 108, 74, 9]; // Declare and initialize an
    array.
    const double pi = 3.14159; // Declare and initialize constant.

    // foreach statement block that contains multiple statements.
    foreach (int radius in radii)
    {
        // Declaration statement with initializer.
        double circumference = pi * (2 * radius);

        // Expression statement (method invocation). A single-line
        // statement can span multiple text lines because line breaks
        // are treated as white space, which is ignored by the compiler.
        System.Console.WriteLine("Radius of circle #{0} is {1}.
Circumference = {2:N2}",
                               counter, radius, circumference);

        // Expression statement (postfix increment).
        counter++;
    } // End of foreach statement block
```

```

    } // End of Main method body.
} // End of SimpleStatements class.
/*
Output:
Radius of circle #1 = 15. Circumference = 94.25
Radius of circle #2 = 32. Circumference = 201.06
Radius of circle #3 = 108. Circumference = 678.58
Radius of circle #4 = 74. Circumference = 464.96
Radius of circle #5 = 9. Circumference = 56.55
*/

```

## 문 유형

다음 표에는 다양한 유형의 C# 문과 관련 키워드가 나와 있으며 자세한 정보를 포함하는 항목에 대한 링크가 있습니다.

### 테이블 확장

범주	C# 키워드/참고 사항
선언문	선언문은 새 변수 또는 상수를 소개합니다. 필요에 따라 변수 선언에서 변수에 값을 할당할 수 있습니다. 상수 선언에서 대입은 필수입니다.
식 문	값을 계산하는 식 문은 값을 변수에 저장해야 합니다.
선택 문	선택 문을 사용하면 하나 이상의 지정된 조건에 따라 코드의 다른 섹션으로 분기할 수 있습니다. 자세한 내용은 아래 항목을 참조하세요. <ul style="list-style-type: none"> <li>• <a href="#">if</a></li> <li>• <a href="#">switch</a></li> </ul>
반복 문	반복 문을 사용하면 배열과 같은 컬렉션을 반복하거나, 지정한 조건이 충족될 때까지 동일한 문 집합을 반복해서 수행할 수 있습니다. 자세한 내용은 아래 항목을 참조하세요. <ul style="list-style-type: none"> <li>• <a href="#">do</a></li> <li>• <a href="#">for</a></li> <li>• <a href="#">foreach</a></li> <li>• <a href="#">while</a></li> </ul>
점프 문	점프 문은 컨트롤을 다른 코드 섹션으로 전송합니다. 자세한 내용은 아래 항목을 참조하세요. <ul style="list-style-type: none"> <li>• <a href="#">break</a></li> <li>• <a href="#">continue</a></li> <li>• <a href="#">goto</a></li> <li>• <a href="#">return</a></li> <li>• <a href="#">yield</a></li> </ul>

법주	C# 키워드/참고 사항
예외 처리문	예외 처리 문을 사용하면 런타임에 발생하는 예외적 조건에서 정상적으로 복구할 수 있습니다. 자세한 내용은 아래 항목을 참조하세요. <ul style="list-style-type: none"> <li>• <a href="#">throw</a></li> <li>• <a href="#">try-catch</a></li> <li>• <a href="#">try-finally</a></li> <li>• <a href="#">try-catch-finally</a></li> </ul>
<code>checked</code> 및 <code>unchecked</code>	<code>checked</code> 및 <code>unchecked</code> 문을 사용하면 결과가 저장되는 변수가 너무 작아서 결과 값을 수용할 수 없는 경우 정수 형식 산술 연산에서 오버플로가 발생할 수 있는지 여부를 지정할 수 있습니다.
<code>await</code> 문	메서드에 <code>async</code> 한정자를 표시하면 메서드에서 <code>await</code> 연산자를 사용할 수 있습니다. 컨트롤이 비동기 메서드의 <code>await</code> 식에 도달하면 컨트롤이 호출자로 돌아가고 대기 중인 작업이 완료될 때까지 메서드의 진행이 일시 중단됩니다. 작업이 완료되면 메서드가 실행이 다시 시작될 수 있습니다.  간단한 예제는 <a href="#">메서드</a> 의 "Async 메서드" 섹션을 참조하세요. 자세한 내용은 <a href="#">async 및 await를 사용한 비동기 프로그래밍</a> 을 참조하세요.
<code>yield return</code> 문	반복기는 배열 목록과 같은 컬렉션에 대해 사용자 지정 반복을 수행합니다. 반복기는 <code>yield return</code> 문을 사용하여 각 요소를 따로따로 반환할 수 있습니다. <code>yield return</code> 문에 도달하면 코드의 현재 위치가 기억됩니다. 다음에 반복기가 호출되면 해당 위치에서 실행이 다시 시작됩니다.  자세한 내용은 <a href="#">반복기</a> 를 참조하세요.
<code>fixed</code> 문	<code>fixed</code> 문은 가비지 수집기에서 이동 가능한 변수를 재배치할 수 없도록 합니다. 자세한 내용은 <a href="#">fixed</a> 를 참조하세요.
<code>lock</code> 문	<code>lock</code> 문을 사용하면 한 번에 하나의 스레드만 코드 블록에 액세스할 수 있도록 제한할 수 있습니다. 자세한 내용은 <a href="#">lock</a> 을 참조하세요.
레이블 문	문에 레이블을 지정한 다음 <code>goto</code> 키워드를 사용하여 레이블 문으로 점프합니다. 다음 행의 예제를 참조하세요.
빈 명령문	빈 문은 하나의 세미콜론으로 구성됩니다. 아무 작업도 수행하지 않으며, 문이 필요하지만 아무 작업도 수행할 필요가 없는 위치에서 사용할 수 있습니다.

## 선언문

다음 코드는 초기 할당이 있거나 할당되지 않은 변수 선언과 필요한 초기화가 있는 상수 선언의 예를 보여 줍니다.

```
// Variable declaration statements.  
double area;  
double radius = 2;  
  
// Constant declaration statement.  
const double pi = 3.14159;
```

## 식 문

다음 코드는 할당, 할당을 통한 개체 만들기 및 메서드 호출을 포함하는 식 명문의 예를 보여 줍니다.

C#

```
// Expression statement (assignment).  
area = 3.14 * (radius * radius);  
  
// Error. Not statement because no assignment:  
//circ * 2;  
  
// Expression statement (method invocation).  
System.Console.WriteLine();  
  
// Expression statement (new object creation).  
System.Collections.Generic.List<string> strings =  
    new System.Collections.Generic.List<string>();
```

## 빈 문

다음 예제에서는 빈 문의 두 가지 사용을 보여 줍니다.

C#

```
void ProcessMessages()  
{  
    while (ProcessMessage())  
        ; // Statement needed here.  
}  
  
void F()  
{  
    //...  
    if (done) goto exit;  
//...  
exit:
```

```
    ; // Statement needed here.  
}
```

## 포함 문

일부 명령문(예: [iteration 문](#))은 항상 embedded 문이 뒤에 나옵니다. 이 포함 명령문은 단일 명령문이나 명령문 블록에서 {} 괄호로 묶인 여러 명령문일 수 있습니다. 한 줄로 된 각 포함 명령문을 다음 예제와 같이 {} 괄호로 묶을 수 있습니다.

C#

```
// Recommended style. Embedded statement in block.  
foreach (string s in System.IO.Directory.GetDirectories(  
    System.Environment.CurrentDirectory))  
{  
    System.Console.WriteLine(s);  
}  
  
// Not recommended.  
foreach (string s in System.IO.Directory.GetDirectories(  
    System.Environment.CurrentDirectory))  
    System.Console.WriteLine(s);
```

{} 괄호로 묶이지 않은 포함 문은 선언문 또는 레이블 문이 될 수 없습니다. 이는 다음 예제에서 확인할 수 있습니다.

C#

```
if(pointB == true)  
    //Error CS1023:  
    int radius = 5;
```

오류를 해결하려면 포함 문을 블록에 배치합니다.

C#

```
if (b == true)  
{  
    // OK:  
    System.DateTime d = System.DateTime.Now;  
    System.Console.WriteLine(d.ToString());  
}
```

## 중첩된 문 블록

다음 코드와 같이 문 블록을 중첩할 수 있습니다.

C#

```
foreach (string s in System.IO.Directory.GetDirectories(
    System.Environment.CurrentDirectory))
{
    if (s.StartsWith("CSharp"))
    {
        if (s.EndsWith("TempFolder"))
        {
            return s;
        }
    }
}
return "Not found.";
```

## 연결할 수 없는 문

상황에 따라 제어 흐름이 특정 문에 연결할 수 없다고 확인될 경우 컴파일러는 다음 예제와 같이 경고 CS0162를 생성합니다.

C#

```
// An over-simplified example of unreachable code.
const int val = 5;
if (val < 4)
{
    System.Console.WriteLine("I'll never write anything."); //CS0162
}
```

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 문](#) 섹션을 참조하세요.

## 참고 항목

- [문 키워드](#)
- [C# 연산자 및 식](#)



GitHub에서 Microsoft와 공

.NET

.NET 피드백

## 동작

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# 식 본문 멤버(C# 프로그래밍 가이드)

아티클 • 2024. 03. 13.

식 본문 정의를 사용하면 간결하고 읽기 쉬운 형식으로 멤버의 구현을 제공할 수 있습니다. 메서드 또는 속성과 같은 지원되는 멤버에 대한 논리가 단일 식으로 구성된 경우 식 본문 정의를 사용할 수 있습니다. 식 본문 정의의 일반 구문은 다음과 같습니다.

C#

```
member => expression;
```

여기서 *expression*은 유효한 식입니다.

식 본문 정의는 다음 형식 멤버와 함께 사용할 수 있습니다.

- [방법](#)
- [읽기 전용 속성](#)
- [속성](#)
- [Constructor](#)
- [종료자](#)
- [인덱서](#)

## 메서드

식 본문 메서드는 형식이 메서드의 반환 형식과 일치하는 값을 반환하거나 `void`를 반환하는 메서드의 경우 일부 작업을 수행하는 단일 식으로 구성됩니다. 예를 들어 `ToString` 메서드를 재정의하는 형식에는 일반적으로 현재 개체의 문자열 표현을 반환하는 단일 식이 포함되어 있습니다.

다음 예제에 `ToString` 메서드를 식 본문 정의로 재정의하는 `Person` 클래스를 정의합니다. 또한 이름을 콘솔에 표시하는 `DisplayName` 메서드를 정의합니다. `return` 키워드는 `ToString` 식 본문 정의에 사용되지 않습니다.

C#

```
using System;

namespace ExpressionBodiedMembers;

public class Person
{
    public Person(string firstName, string lastName)
    {
```

```

        fname = firstName;
        lname = lastName;
    }

    private string fname;
    private string lname;

    public override string ToString() => $"{fname} {lname}".Trim();
    public void DisplayName() => Console.WriteLine(ToString());
}

class Example
{
    static void Main()
    {
        Person p = new Person("Mandy", "Dejesus");
        Console.WriteLine(p);
        p.DisplayName();
    }
}

```

자세한 내용은 [메서드\(C# 프로그래밍 가이드\)](#)를 참조하세요.

## 읽기 전용 속성

식 본문 정의를 사용하여 읽기 전용 속성을 구현할 수 있습니다. 이를 위해 사용하는 구문은 다음과 같습니다.

C#

```
PropertyType PropertyName => expression;
```

다음 예제에서는 `Location` 클래스를 정의합니다. 이 클래스의 읽기 전용 `Name` 속성은 `locationName` 비공개 필드의 값을 반환하는 식 본문 정의로 구현됩니다.

C#

```

public class Location
{
    private string locationName;

    public Location(string name)
    {
        locationName = name;
    }

    public string Name => locationName;
}

```

속성에 대한 자세한 내용은 [속성\(C# 프로그래밍 가이드\)](#)을 참조하세요.

## 속성

식 본문 정의를 사용하여 속성 `get` 및 `set` 접근자를 구현할 수 있습니다. 다음 예제에서 이를 수행하는 방법을 보여줍니다.

C#

```
public class Location
{
    private string locationName;

    public Location(string name) => Name = name;

    public string Name
    {
        get => locationName;
        set => locationName = value;
    }
}
```

속성에 대한 자세한 내용은 [속성\(C# 프로그래밍 가이드\)](#)을 참조하세요.

## 생성자

생성자에 대한 식 본문 정의는 일반적으로 생성자의 인수를 처리하거나 인스턴스 상태를 초기화하는 단일 할당 식 또는 메서드 호출로 구성됩니다.

다음 예제에서는 생성자에 `name`이라는 단일 문자열 매개 변수가 있는 `Location` 클래스를 정의합니다. 식 본문 정의에서 `Name` 속성에 인수를 할당합니다.

C#

```
public class Location
{
    private string locationName;

    public Location(string name) => Name = name;

    public string Name
    {
        get => locationName;
        set => locationName = value;
    }
}
```

자세한 내용은 [생성자\(C# 프로그래밍 가이드\)](#)를 참조하세요.

## 종료자

종료자에 대한 식 본문 정의에는 일반적으로 관리되지 않는 리소스를 해제하는 문 등의 정리 문이 포함되어 있습니다.

다음 예제에서는 식 본문 정의를 사용하여 종료자가 호출되었음을 나타내는 종료자를 정의합니다.

```
C#  
  
public class Destroyer  
{  
    public override string ToString() => GetType().Name;  
  
    ~Destroyer() => Console.WriteLine($"The {ToString()} finalizer is  
executing.");  
}
```

자세한 내용은 [종료자\(C# 프로그래밍 가이드\)](#)를 참조하세요.

## 인덱서

속성과 마찬가지로, `get` 접근자가 값을 반환하는 단일 식으로 구성되거나 `set` 접근자가 단순 할당을 수행하는 경우 인덱서의 `get` 및 `set` 접근자는 식 본문 정의로 구성됩니다.

다음 예제에서는 일부 스포츠의 이름이 포함된 내부 `String` 배열을 포함하는 `Sports`라는 클래스를 정의합니다. 인덱서의 `get` 및 `set` 접근자는 둘 다 식 본문 정의로 구현됩니다.

```
C#  
  
using System;  
using System.Collections.Generic;  
  
namespace SportsExample;  
  
public class Sports  
{  
    private string[] types = [ "Baseball", "Basketball", "Football",  
                             "Hockey", "Soccer", "Tennis",  
                             "Volleyball" ];  
  
    public string this[int i]  
    {  
        get => types[i];  
        set => types[i] = value;  
    }  
}
```

```
    }  
}
```

자세한 내용은 [인덱서\(C# 프로그래밍 가이드\)](#)를 참조하세요.

## 참고 항목

- 식 본문 멤버의 .NET 코드 스타일 규칙

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

💬 설명서 문제 열기

↗️ 제품 사용자 의견 제공

# 같음 비교(C# 프로그래밍 가이드)

아티클 • 2024. 03. 12.

두 값이 같은지를 비교해야 하는 경우가 있습니다. 때로는 두 변수에 포함된 값이 같음을 의미하는 값 같음(동등이라고도 함)을 테스트합니다. 다른 경우에는 두 변수가 메모리에서 동일한 기본 개체를 참조하는지 여부를 확인해야 합니다. 이 유형의 같음을 참조 같음 또는 *ID*라고 합니다. 이 항목에서는 이러한 두 종류의 같음을 설명하고 자세한 정보가 있는 다른 항목에 대한 링크를 제공합니다.

## 참조 같음

참조 같음은 두 개체 참조가 동일한 기본 개체를 참조함을 의미합니다. 이러한 상황은 다음 예제와 같이 단순 할당을 통해 발생할 수 있습니다.

C#

```
using System;
class Test
{
    public int Num { get; set; }
    public string Str { get; set; }

    public static void Main()
    {
        Test a = new Test() { Num = 1, Str = "Hi" };
        Test b = new Test() { Num = 1, Str = "Hi" };

        bool areEqual = System.Object.ReferenceEquals(a, b);
        // False:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);

        // Assign b to a.
        b = a;

        // Repeat calls with different results.
        areEqual = System.Object.ReferenceEquals(a, b);
        // True:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);
    }
}
```

이 코드에서는 두 개체가 생성되지만 대입문 이후 두 참조가 동일한 개체를 참조합니다. 따라서 참조 같음이 있습니다. 두 참조가 동일한 개체를 참조하는지 확인하려면 `ReferenceEquals` 메서드를 사용합니다.

참조 같음의 개념은 참조 형식에만 적용됩니다. 값 형식의 인스턴스가 변수에 할당될 때 값의 복사본이 생성되기 때문에 값 형식 개체에는 참조 같음이 있을 수 없습니다. 따라서 메모리의 동일한 위치를 참조하는 두 개의 unboxed 구조체가 있을 수 없습니다. 또한 [ReferenceEquals](#)를 사용하여 두 개의 값 형식을 비교하는 경우 개체에 포함된 값이 모두 동일한 경우에도 결과는 항상 `false`입니다. 각 변수가 별도의 개체 인스턴스에 boxed되기 때문입니다. 자세한 내용은 [참조 같음\(ID\)을 테스트하는 방법](#)을 참조하세요.

## 값 같음

값 같음은 두 개체에 동일한 값이 포함되어 있음을 의미합니다. `int` 또는 `bool`과 같은 기본 값 형식의 경우 값 같음 테스트가 간단합니다. 다음 예제와 같이 `==` 연산자를 사용할 수 있습니다.

C#

```
int a = GetOriginalValue();
int b = GetCurrentValue();

// Test for value equality.
if (b == a)
{
    // The two integers are equal.
}
```

대부분의 다른 형식에서는 값 같음 테스트가 더 복잡한데, 형식에서 정의된 방식을 알아야 하기 때문입니다. 여러 필드나 속성이 있는 클래스 및 구조체의 경우 값 같음은 대체로 모든 필드 또는 속성에 동일한 값이 있다는 의미로 정의됩니다. 예를 들어 `pointA.X`가 `pointB.X`와 같고 `pointA.Y`가 `pointB.Y`와 같으면 두 `Point` 개체가 동일한 것으로 정의할 수 있습니다. 레코드의 경우 값 같음은 형식이 일치하고 모든 속성 및 필드 값이 일치하는 경우 레코드 형식의 두 변수가 같다는 것을 의미합니다.

그러나 동등이 형식의 모든 필드를 기반으로 해야 한다는 요구 사항은 없습니다. 하위 집합을 기반으로 할 수도 있습니다. 소유하지 않은 형식을 비교하는 경우 해당 형식에 대해 동등이 정의된 방식을 구체적으로 알아야 합니다. 사용자 고유의 클래스 및 구조체에서 값 같음을 정의하는 방법에 대한 자세한 내용은 [형식의 값 같음을 정의하는 방법](#)을 참조하세요.

## 부동 소수점 값에 대한 값 같음

이진 컴퓨터의 부정확한 부동 소수점 연산 때문에 부동 소수점 값(`double` 및 `float`)의 같음 비교에서 문제가 발생합니다. 자세한 내용은 [System.Double](#) 항목의 설명을 참조하세요.

# 관련 항목

 테이블 확장

제목	설명
참조 같음(ID)을 테스트하는 방법	두 변수에 참조 같음이 있는지를 확인하는 방법을 설명 합니다.
형식의 값 같음을 정의하는 방법	형식에 대한 값 같음의 사용자 지정 정의를 제공하는 방법을 설명 합니다.
유형	C# 형식 시스템에 대한 정보 및 추가 정보 링크를 제공합니다.
레코드	기본적으로 값 같음을 테스트하는 레코드 형식에 대한 정보를 제공합니다.

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 클래스 또는 구조체에 대한 값 같음을 정의하는 방법(C# 프로그래밍 가이드)

아티클 • 2024. 03. 08.

레코드는 값 같음을 자동으로 구현합니다. 형식이 데이터를 모델링하고 값 같음을 구현해야 하는 경우 `class` 대신 `record`를 정의하는 것이 좋습니다.

클래스 또는 구조체를 정의할 때 형식에 대한 값 같음(또는 동등)의 사용자 지정 정의를 만드는 것이 적합한지 결정합니다. 일반적으로 형식의 개체를 컬렉션에 추가해야 하는 경우 또는 주요 용도가 필드 또는 속성 세트 저장인 경우 값 같음을 구현합니다. 형식의 모든 필드 및 속성 비교를 기준으로 값 같음의 정의를 만들거나, 하위 집합을 기준으로 정의를 만들 수 있습니다.

두 경우 모두 및 클래스와 구조체 둘 다에서 구현은 다음과 같은 동등의 5가지 보장 사항을 따라야 합니다(다음 규칙의 경우 `x`, `y`, `z`가 `null`이 아닌 것으로 가정).

1. 재귀 속성: `x.Equals(x)` 는 `true`를 반환합니다.
2. 대칭 속성: `x.Equals(y)` 는 `y.Equals(x)` 와 같은 값을 반환합니다.
3. 전이적 속성: `(x.Equals(y) && y.Equals(z))` 가 `true`를 반환하면 `x.Equals(z)` 는 `true`를 반환합니다.
4. `x.Equals(y)` 의 연속 호출은 `x` 및 `y`에서 참조하는 개체가 수정되지 않는 한, 같은 값을 반환합니다.
5. `null`이 아닌 값은 `null`과 같지 않습니다. 그러나 `x`가 `null`이면 `x.Equals(y)` 는 예외를 `throw`합니다. `Equals`에 대한 인수에 따라 규칙 1 또는 2가 위반됩니다.

정의하는 모든 구조체에는 `Object.Equals(Object)` 메서드의 `System.ValueType` 재정의에서 상속하는 값 같음의 기본 구현이 이미 있습니다. 이 구현은 리플렉션을 사용하여 형식의 모든 필드와 속성을 검사합니다. 이 구현은 올바른 결과를 생성하지만 해당 형식에 맞게 작성한 사용자 지정 구현에 비해 비교적 속도가 느립니다.

값 같음에 대한 구현 세부 정보는 클래스 및 구조체에서 서로 다릅니다. 그러나 클래스와 구조체는 둘 다 같은 구현을 위해 동일한 기본 단계가 필요합니다.

1. `virtual Object.Equals(Object)` 메서드를 재정의합니다. 대부분의 경우 `bool Equals(object obj)` 구현에서 `System.IEquatable<T>` 인터페이스 구현인 형식별 `Equals` 메서드만 호출하면 됩니다. 2단계를 참조하세요.

2. 형식별 `Equals` 메서드를 제공하여 `System.IEquatable<T>` 인터페이스를 구현합니다. 여기서 실제 동등 비교가 수행됩니다. 예를 들어 형식에서 한 개나 두 개의 필드만 비교하여 같음 정의를 결정할 수도 있습니다. `Equals`에서 예외를 throw하지 않습니다. 상속과 관련된 클래스의 경우:

- 이 메서드는 클래스에 선언된 필드만 검사해야 합니다. `base.Equals`를 호출하여 기본 클래스에 있는 필드를 검사해야 합니다. (`Object.Equals(Object)`의 `Object` 구현에서 참조 같음 검사를 수행하므로 형식이 `Object`에서 직접 상속하는 경우에는 `base.Equals`를 호출하지 마세요.)
- 비교하는 변수의 런타임 형식이 같으면 두 변수는 같은 것으로 간주되어야 합니다. 또한 변수의 런타임 형식과 컴파일 시간 형식이 다른 경우 런타임 형식에 대한 `Equals` 메서드의 `IEquatable` 구현이 사용되는지 확인합니다. 런타임 형식이 항상 올바르게 비교되도록 하기 위한 한 가지 전략은 `sealed` 클래스에서만 `IEquatable`을 구현하는 것입니다. 자세한 내용은 이 문서 뒷부분의 [코드 예제](#)를 참조하세요.

3. 선택 사항이지만 권장됨: `==` 및 `!=` 연산자를 오버로드합니다.

4. 값이 같은 두 개체가 동일한 해시 코드를 생성하도록 `Object.GetHashCode`를 재정의합니다.

5. 선택 사항: “보다 큼” 또는 “보다 작음”에 대한 정의를 지원하기 위해 형식에 대한 `IComparable<T>` 인터페이스를 구현하고 `<=` 및 `>=` 연산자도 오버로드합니다.

### ① 참고

불필요한 상용구 코드 없이 레코드를 사용하여 값 평등 의미 체계를 가져올 수 있습니다.

## 클래스 예제

다음 예제에서는 클래스(참조 형식)에서 값 같음을 구현하는 방법을 보여 줍니다.

C#

```
namespace ValueEqualityClass;

class TwoDPoint : IEquatable<TwoDPoint>
{
    public int X { get; private set; }
    public int Y { get; private set; }
```

```
public TwoDPoint(int x, int y)
{
    if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
    {
        throw new ArgumentException("Point must be in range 1 - 2000");
    }
    this.X = x;
    this.Y = y;
}

public override bool Equals(object obj) => this.Equals(obj as
TwoDPoint);

public bool Equals(TwoDPoint p)
{
    if (p is null)
    {
        return false;
    }

    // Optimization for a common success case.
    if (Object.ReferenceEquals(this, p))
    {
        return true;
    }

    // If run-time types are not exactly the same, return false.
    if (this.GetType() != p.GetType())
    {
        return false;
    }

    // Return true if the fields match.
    // Note that the base class is not invoked because it is
    // System.Object, which defines Equals as reference equality.
    return (X == p.X) && (Y == p.Y);
}

public override int GetHashCode() => (X, Y).GetHashCode();

public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)
{
    if (lhs is null)
    {
        if (rhs is null)
        {
            return true;
        }

        // Only the left side is null.
        return false;
    }
    // Equals handles case of null on right side.
    return lhs.Equals(rhs);
}
```

```
    public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !(lhs ==  
rhs);  
  
// For the sake of simplicity, assume a ThreeDPoint IS a TwoDPoint.  
class ThreeDPoint : TwoDPoint, IEquatable<ThreeDPoint>  
{  
    public int Z { get; private set; }  
  
    public ThreeDPoint(int x, int y, int z)  
        : base(x, y)  
    {  
        if ((z < 1) || (z > 2000))  
        {  
            throw new ArgumentException("Point must be in range 1 - 2000");  
        }  
        this.Z = z;  
    }  
  
    public override bool Equals(object obj) => this.Equals(obj as  
ThreeDPoint);  
  
    public bool Equals(ThreeDPoint p)  
    {  
        if (p is null)  
        {  
            return false;  
        }  
  
        // Optimization for a common success case.  
        if (Object.ReferenceEquals(this, p))  
        {  
            return true;  
        }  
  
        // Check properties that this class declares.  
        if (Z == p.Z)  
        {  
            // Let base class check its own fields  
            // and do the run-time type comparison.  
            return base.Equals((TwoDPoint)p);  
        }  
        else  
        {  
            return false;  
        }  
    }  
  
    public override int GetHashCode() => (X, Y, Z).GetHashCode();  
  
    public static bool operator ==(ThreeDPoint lhs, ThreeDPoint rhs)  
    {  
        if (lhs is null)  
        {
```

```

        if (rhs is null)
    {
        // null == null = true.
        return true;
    }

        // Only the left side is null.
        return false;
    }
    // Equals handles the case of null on right side.
    return lhs.Equals(rhs);
}

public static bool operator !=(ThreeDPoint lhs, ThreeDPoint rhs) => !
(lhs == rhs);
}

class Program
{
    static void Main(string[] args)
    {
        ThreeDPoint pointA = new ThreeDPoint(3, 4, 5);
        ThreeDPoint pointB = new ThreeDPoint(3, 4, 5);
        ThreeDPoint pointC = null;
        int i = 5;

        Console.WriteLine("pointA.Equals(pointB) = {0}",
pointA.Equals(pointB));
        Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
        Console.WriteLine("null comparison = {0}", pointA.Equals(pointC));
        Console.WriteLine("Compare to some other type = {0}",
pointA.Equals(i));

        TwoDPoint pointD = null;
        TwoDPoint pointE = null;

        Console.WriteLine("Two null TwoDPoints are equal: {0}", pointD ==
pointE);

        pointE = new TwoDPoint(3, 4);
        Console.WriteLine("(pointE == pointA) = {0}", pointE == pointA);
        Console.WriteLine("(pointA == pointE) = {0}", pointA == pointE);
        Console.WriteLine("(pointA != pointE) = {0}", pointA != pointE);

        System.Collections.ArrayList list = new
System.Collections.ArrayList();
        list.Add(new ThreeDPoint(3, 4, 5));
        Console.WriteLine("pointE.Equals(list[0]): {0}",
pointE.Equals(list[0]));

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

```

```

/* Output:
 pointA.Equals(pointB) = True
 pointA == pointB = True
 null comparison = False
 Compare to some other type = False
 Two null TwoDPoints are equal: True
 (pointE == pointA) = False
 (pointA == pointE) = False
 (pointA != pointE) = True
 pointE.Equals(list[0]): False
*/

```

클래스(참조 형식)에서 두 `Object.Equals(Object)` 메서드의 기본 구현은 값이 같은지 검사하지 않고 참조가 같은지 비교합니다. 구현자가 가상 메서드를 재정의하는 경우 값 같음 의미 체계를 제공하기 위한 것입니다.

클래스에서 재정의하지 않는 경우에도 `==` 및 `!=` 연산자를 클래스와 함께 사용할 수 있습니다. 그러나 기본 동작은 참조가 같은지 검사하는 것입니다. 클래스에서 `Equals` 메서드를 오버로드하는 경우 `==` 및 `!=` 연산자를 오버로드해야 하지만 필수는 아닙니다.

### ⓘ 중요

앞의 예제 코드는 모든 상속 시나리오를 원하는 방식으로 처리하지 않을 수 있습니다. 다음 코드를 생각해 봅시다.

C#

```

TwoDPoint p1 = new ThreeDPoint(1, 2, 3);
TwoDPoint p2 = new ThreeDPoint(1, 2, 4);
Console.WriteLine(p1.Equals(p2)); // output: True

```

이 코드는 `z` 값의 차이에도 불구하고 `p1`이 `p2`와 같다고 보고합니다. 컴파일러가 컴파일 시간 형식에 따라 `IEquatable`의 `TwoDPoint` 구현을 선택하므로 차이는 무시됩니다.

`record` 형식의 기본 제공 값 같음이 이와 같은 시나리오를 올바르게 처리합니다. `TwoDPoint` 및 `ThreeDPoint`가 `record` 형식인 경우 `p1.Equals(p2)`의 결과는 `False`가 됩니다. 자세한 내용은 [record 형식 상속 계층 구조의 같음](#)을 참조하세요.

## 구조체 예제

다음 예제에서는 구조체(값 형식)에서 값 같음을 구현하는 방법을 보여 줍니다.

C#

```
namespace ValueEqualityStruct
{
    struct TwoDPoint : IEquatable<TwoDPoint>
    {
        public int X { get; private set; }
        public int Y { get; private set; }

        public TwoDPoint(int x, int y)
            : this()
        {
            if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
            {
                throw new ArgumentException("Point must be in range 1 - 2000");
            }
            X = x;
            Y = y;
        }

        public override bool Equals(object? obj) => obj is TwoDPoint other
&& this.Equals(other);

        public bool Equals(TwoDPoint p) => X == p.X && Y == p.Y;

        public override int GetHashCode() => (X, Y).GetHashCode();

        public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs) =>
lhs.Equals(rhs);

        public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !
(lhs == rhs);
    }

    class Program
    {
        static void Main(string[] args)
        {
            TwoDPoint pointA = new TwoDPoint(3, 4);
            TwoDPoint pointB = new TwoDPoint(3, 4);
            int i = 5;

            // True:
            Console.WriteLine("pointA.Equals(pointB) = {0}",
pointA.Equals(pointB));
            // True:
            Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
            // True:
            Console.WriteLine("object.Equals(pointA, pointB) = {0}",
object.Equals(pointA, pointB));
            // False:
            Console.WriteLine("pointA.Equals(null) = {0}",
pointA.Equals(null));
        }
    }
}
```

```

// False:
Console.WriteLine("(pointA == null) = {0}", pointA == null);
// True:
Console.WriteLine("(pointA != null) = {0}", pointA != null);
// False:
Console.WriteLine("pointA.Equals(i) = {0}", pointA.Equals(i));
// CS0019:
// Console.WriteLine("pointA == i = {0}", pointA == i);

// Compare unboxed to boxed.
System.Collections.ArrayList list = new
System.Collections.ArrayList();
list.Add(new TwoDPoint(3, 4));
// True:
Console.WriteLine("pointA.Equals(list[0]): {0}",
pointA.Equals(list[0]));

// Compare nullable to nullable and to non-nullable.
TwoDPoint? pointC = null;
TwoDPoint? pointD = null;
// False:
Console.WriteLine("pointA == (pointC = null) = {0}", pointA ==
pointC);
// True:
Console.WriteLine("pointC == pointD = {0}", pointC == pointD);

TwoDPoint temp = new TwoDPoint(3, 4);
pointC = temp;
// True:
Console.WriteLine("pointA == (pointC = 3,4) = {0}", pointA ==
pointC);

pointD = temp;
// True:
Console.WriteLine("pointD == (pointC = 3,4) = {0}", pointD ==
pointC);

Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}
}

/* Output:
pointA.Equals(pointB) = True
pointA == pointB = True
Object.Equals(pointA, pointB) = True
pointA.Equals(null) = False
(pointA == null) = False
(pointA != null) = True
pointA.Equals(i) = False
pointE.Equals(list[0]): True
pointA == (pointC = null) = False
pointC == pointD = True
pointA == (pointC = 3,4) = True
pointD == (pointC = 3,4) = True

```

```
 */  
}
```

구조체의 경우 `Object.Equals(Object)`의 기본 구현(`System.ValueType`의 재정의된 버전)에서 리플렉션을 통해 형식에 있는 모든 필드의 값을 비교하여 값이 같은지 검사합니다. 구현자가 구조체의 가상 `Equals` 메서드를 재정의하는 경우 값이 같은지 검사하는 보다 효율적인 수단을 제공하고 필요에 따라 구조체 필드 또는 속성의 하위 집합을 기준으로 비교하기 위한 것입니다.

`==` 및 `!=` 연산자는 구조체에서 명시적으로 오버로드하지 않는 한 구조체에 대해 연산을 수행할 수 없습니다.

## 참고 항목

- [같음 비교](#)
- [C# 프로그래밍 가이드](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 참조 같음(ID)을 테스트하는 방법(C# 프로그래밍 가이드)

아티클 • 2024. 03. 08.

형식에서 참조 같음 비교를 지원하기 위해 사용자 지정 논리를 구현할 필요는 없습니다. 이 기능은 정적 `Object.ReferenceEquals` 메서드가 모든 형식에 대해 제공합니다.

다음 예제에서는 두 변수에 참조 같음이 있는지 여부, 즉 메모리의 동일한 개체를 참조하는지 여부를 확인하는 방법을 보여 줍니다.

또한 이 예제에서는 `Object.ReferenceEquals`가 값 형식에 대해 항상 `false`를 반환하는 이유 및 문자열 일치를 확인하는 데 `ReferenceEquals`를 사용하면 안 되는 이유를 보여 줍니다.

## 예시

C#

```
using System.Text;

namespace TestReferenceEquality
{
    struct TestStruct
    {
        public int Num { get; private set; }
        public string Name { get; private set; }

        public TestStruct(int i, string s) : this()
        {
            Num = i;
            Name = s;
        }
    }

    class TestClass
    {
        public int Num { get; set; }
        public string? Name { get; set; }
    }

    class Program
    {
        static void Main()
        {
            // Demonstrate reference equality with reference types.
            #region ReferenceTypes
```

```

        // Create two reference type instances that have identical
values.
        TestClass tcA = new TestClass() { Num = 1, Name = "New
TestClass" };
        TestClass tcB = new TestClass() { Num = 1, Name = "New
TestClass" };

        Console.WriteLine("ReferenceEquals(tcA, tcB) = {0}",
                           Object.ReferenceEquals(tcA, tcB)); // false

        // After assignment, tcB and tcA refer to the same object.
        // They now have reference equality.
        tcB = tcA;
        Console.WriteLine("After assignment: ReferenceEquals(tcA, tcB) =
{0}",
                           Object.ReferenceEquals(tcA, tcB)); // true

        // Changes made to tcA are reflected in tcB. Therefore, objects
        // that have reference equality also have value equality.
        tcA.Num = 42;
        tcA.Name = "TestClass 42";
        Console.WriteLine("tcB.Name = {0} tcB.Num: {1}", tcB.Name,
tcB.Num);
#endregion

        // Demonstrate that two value type instances never have
reference equality.
#region ValueTypes

        TestStruct tsC = new TestStruct( 1, "TestStruct 1");

        // Value types are copied on assignment. tsD and tsC have
        // the same values but are not the same object.
        TestStruct tsD = tsC;
        Console.WriteLine("After assignment: ReferenceEquals(tsC, tsD) =
{0}",
                           Object.ReferenceEquals(tsC, tsD)); // false
#endregion

        #region stringRefEquality
        // Constant strings within the same assembly are always interned
by the runtime.
        // This means they are stored in the same location in memory.
Therefore,
        // the two strings have reference equality although no
assignment takes place.
        string strA = "Hello world!";
        string strB = "Hello world!";
        Console.WriteLine("ReferenceEquals(strA, strB) = {0}",
                           Object.ReferenceEquals(strA, strB)); // true

        // After a new string is assigned to strA, strA and strB
        // are no longer interned and no longer have reference equality.
        strA = "Goodbye world!";
        Console.WriteLine("strA = \"{}\" strB = \"{}\"", strA, strB);

```

```

        Console.WriteLine("After strA changes, ReferenceEquals(strA,
strB) = {0}",

                           Object.ReferenceEquals(strA, strB)); // false

        // A string that is created at runtime cannot be interned.
        StringBuilder sb = new StringBuilder("Hello world!");
        string stringC = sb.ToString();
        // False:
        Console.WriteLine("ReferenceEquals(stringC, strB) = {0}",
                           Object.ReferenceEquals(stringC, strB));

        // The string class overloads the == operator to perform an
        // equality comparison.
        Console.WriteLine("stringC == strB = {0}", stringC == strB); // true

    #endregion

    // Keep the console open in debug mode.
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}

}

/*
/* Output:
ReferenceEquals(tcA, tcB) = False
After assignment: ReferenceEquals(tcA, tcB) = True
tcB.Name = TestClass 42 tcB.Num: 42
After assignment: ReferenceEquals(tsC, tsD) = False
ReferenceEquals(strA, strB) = True
strA = "Goodbye world!" strB = "Hello world!"
After strA changes, ReferenceEquals(strA, strB) = False
ReferenceEquals(stringC, strB) = False
stringC == strB = True
*/

```

`System.Object` 유니버설 기본 클래스의 `Equals` 구현에서도 참조 같음 검사를 수행하지만 클래스가 메서드를 재정의할 경우 결과가 예상과 다를 수 있기 때문에 이 기능은 사용하지 않는 것이 좋습니다. `==` 및 `!=` 연산자의 경우도 마찬가지입니다. 참조 형식에서 작동하는 경우 `==` 및 `!=`의 기본 동작은 참조 같음 검사를 수행하는 것입니다. 그러나 파생 클래스에서 연산자를 오버로드하여 값 같음 검사를 수행할 수 있습니다. 잠재적인 오류를 최소화하려면 두 개체에 참조 같음이 있는지 여부를 확인해야 할 때 항상 `ReferenceEquals`를 사용하는 것이 좋습니다.

동일한 어셈블리 내의 상수 문자열은 항상 런타임에서 인턴 지정됩니다. 즉, 고유한 각 리터럴 문자열의 인스턴스 하나만 유지됩니다. 그러나 런타임은 런타임에 생성된 문자열이 인턴 지정되도록 보장하지 않으며, 서로 다른 어셈블리에 있는 동일한 두 상수 문자열이 인턴 지정되도록 보장하지도 않습니다.

# 참고 항목

- [같음 비교](#)

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# 캐스팅 및 형식 변환(C# 프로그래밍 가이드)

아티클 • 2024. 03. 19.

C#은 컴파일 시간에 정적으로 형식화되므로 변수가 선언된 후에는 해당 형식을 변수의 형식으로 암시적으로 변환할 수 없는 한 다시 선언하거나 다른 형식의 값을 할당할 수 없습니다. 예를 들어 암시 `string` 적으로 변환 `int` 할 수 없습니다. 따라서 `i` 다음 코드와 같이 `int` 문자열 "Hello"를 할당할 수 없습니다.

C#

```
int i;  
  
// error CS0029: can't implicitly convert type 'string' to 'int'  
i = "Hello";
```

그러나 다른 형식의 변수 또는 메서드 매개 변수에 값을 복사해야 하는 경우도 있습니다. 예를 들어 매개 변수가 `double`로 형식화된 메서드에 전달해야 하는 정수 변수가 있을 수 있습니다. 또는 인터페이스 형식의 변수에 클래스 변수를 할당해야 할 수 있습니다. 이러한 작업을 형식 변환이라고 합니다. C#에서는 다음과 같은 변환을 수행할 수 있습니다.

- **암시적 변환:** 변환이 항상 성공하고 데이터가 손실되지 않으므로 특별한 구문이 필요하지 않습니다. 예제에는 작은 정수 형식에서 큰 정수 형식으로의 변환 및 파생 클래스에서 기본 클래스로의 변환이 포함됩니다.
- **명시적 변환(캐스트):** 명시적 변환에는 [캐스트 식](#)이 필요합니다. 변환 시 정보가 손실되거나 다른 이유로 변환에 실패할 경우 캐스팅이 필요합니다. 일반적인 예제에는 숫자를 정밀도가 낮거나 범위가 더 작은 형식으로 변환하는 작업과 기본 클래스 인스턴스를 파생 클래스로 변환하는 작업이 포함됩니다.
- **사용자 정의 변환:** 사용자 정의 변환은 기본 클래스 파생 클래스 관계가 없는 사용자 지정 형식 간에 명시적 및 암시적 변환을 사용하도록 정의할 수 있는 특수 메서드를 사용합니다. 자세한 내용은 [사용자 정의 변환 연산자](#)를 참조하세요.
- **도우미 클래스를 사용한 변환:** 정수와 `System.DateTime` 개체, 16진수 문자열과 바이트 배열 등 호환되지 않는 형식 간에 변환하려면 `System.BitConverter` 클래스, `System.Convert` 클래스, 기본 제공 숫자 형식(예: `Int32.Parse`)의 `Parse` 메서드를 사용할 수 있습니다. 자세한 내용은 [바이트 배열을 int로 변환하는 방법](#), [문자열을 숫자로 변환하는 방법](#) 및 [16진수 문자열과 숫자 형식 간에 변환하는 방법](#)을 참조하세요.

# 암시적 변환

기본 제공 숫자 형식의 경우 저장되는 값이 잘리거나 반올림되지 않고 변수에 맞출 수 있을 때 암시적 변환을 수행할 수 있습니다. 이것은 정수 형식의 경우 소스 형식의 범위가 대상 유형에 대한 범위의 적절한 하위 집합임을 의미합니다. 예를 들어 `long` 형식의 변수 (64비트 정수)는 `int`(32비트 정수)가 저장할 수 있는 모든 값을 저장할 수 있습니다. 다음 예제에서 컴파일러는 오른쪽의 `num` 값을 `long` 형식으로 암시적으로 변환한 후 `bigNum`에 할당합니다.

C#

```
// Implicit conversion. A long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```

모든 암시적 숫자 변환의 전체 목록은 [기본 제공 숫자 변환](#) 문서의 [암시적 숫자 변환](#) 섹션을 참조하세요.

참조 형식의 경우 클래스에서 직접 또는 간접 기본 클래스나 인터페이스로의 암시적 변환이 항상 존재합니다. 파생 클래스에 항상 기본 클래스의 모든 멤버가 포함되므로 특수 구문이 필요하지 않습니다.

C#

```
Derived d = new Derived();

// Always OK.
Base b = d;
```

# 명시적 변환

그러나 정보가 손실될 위험 없이 변환을 수행할 수 없는 경우 컴파일러는 캐스트라고 하는 명시적 변환을 수행해야 합니다. 캐스트는 변환하려는 컴파일러에 데이터 손실이 발생하거나 런타임에 캐스트가 실패할 수 있음을 알고 있음을 명시적으로 알리는 방법입니다. 캐스트를 수행하려면 변환할 값 또는 변수 앞에 있는 괄호 안에 캐스팅할 형식을 지정합니다. 다음 프로그램은 `int`에 `double` 을 [캐스팅합니다](#). 프로그램은 캐스트 없이 컴파일되지 않습니다.

C#

```
class Test
{
    static void Main()
```

```

{
    double x = 1234.7;
    int a;
    // Cast double to int.
    a = (int)x;
    System.Console.WriteLine(a);
}
// Output: 1234

```

지원되는 명시적 숫자 변환의 전체 목록은 [기본 제공 숫자 변환](#) 문서의 [명시적 숫자 변환](#) 섹션을 참조하세요.

참조 형식의 경우 기본 형식에서 파생 형식으로 변환해야 할 경우에는 명시적 캐스트가 필요합니다.

C#

```

// Create a new derived type.
Giraffe g = new Giraffe();

// Implicit conversion to base type is safe.
Animal a = g;

// Explicit conversion is required to cast back
// to derived type. Note: This will compile but will
// throw an exception at run time if the right-side
// object is not in fact a Giraffe.
Giraffe g2 = (Giraffe)a;

```

참조 형식 간의 캐스트 작업은 기본 개체의 런타임 형식을 변경하지 않습니다. 해당 개체에 대한 참조로 사용되는 값의 형식만 변경합니다. 자세한 내용은 [다형성](#)을 참조하세요.

## 런타임의 형식 변환 예외

일부 참조 형식 변환에서 컴파일러는 캐스트가 유효한지 여부를 확인할 수 없습니다. 올바르게 컴파일되는 캐스트 작업이 런타임에 실패할 수 있습니다. 다음 예제와 같이 런타임 [InvalidCastException](#)에 실패하는 형식 캐스트는 throw됩니다.

C#

```

class Animal
{
    public void Eat() => System.Console.WriteLine("Eating.");

    public override string ToString() => "I am an animal.";
}

```

```

class Reptile : Animal { }
class Mammal : Animal { }

class UnSafeCast
{
    static void Main()
    {
        Test(new Mammal());

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }

    static void Test(Animal a)
    {
        // System.InvalidCastException at run time
        // Unable to cast object of type 'Mammal' to type 'Reptile'
        Reptile r = (Reptile)a;
    }
}

```

`Test` 메서드에 `Animal` 매개 변수가 있으므로 인수 `a`를 `Reptile`로 명시적으로 캐스팅하면 위험한 가정이 생성됩니다. 가정을 하지 않고 형식을 검사 것이 안전합니다. C#에서는 실제로 캐스트를 수행하기 전에 호환성을 테스트할 수 있도록 `is` 연산자를 제공합니다. 자세한 내용은 [패턴 일치, as 및 is 연산자를 사용하여 안전하게 캐스트하는 방법](#)을 참조하세요.

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 전환 섹션](#)을 참조하세요.

## 참고 항목

- 유형
- 캐스트 식
- 사용자 정의 전환 연산자
- 일반화된 형식 변환
- 문자열을 숫자로 변환하는 방법

 GitHub에서 Microsoft와 공동 작업



### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

제공해 주세요.

💡 설명서 문제 열기

↗️ 제품 사용자 의견 제공

# Boxing 및 Unboxing(C# 프로그래밍 가이드)

아티클 • 2023. 04. 07.

Boxing은 값 형식을 `object` 형식 또는 이 값 형식에서 구현된 임의의 인터페이스 형식으로 변환하는 프로세스입니다. CLR(공용 언어 런타임)은 값 형식을 boxing할 때 값을 `System.Object` 인스턴스 내부에 래핑하고 관리되는 힙에 저장합니다. unboxing하면 개체에서 값 형식이 추출됩니다. Boxing은 암시적이며 unboxing은 명시적입니다. Boxing 및 unboxing의 개념은 개체로 처리할 수 있는 모든 값 형식에서 형식 시스템의 C#에 통합된 뷰의 기반이 됩니다.

다음 예제에서는 정수 변수 `i`를 *boxing*하고 개체 `o`에 할당합니다.

C#

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

그런 다음 `o` 개체를 unboxing하고 정수 변수 `i`에 할당할 수 있습니다.

C#

```
o = 123;
i = (int)o; // unboxing
```

다음 예제에서는 C#에서 boxing이 사용되는 방법을 보여 줍니다.

C#

```
// String.Concat example.
// String.Concat has many versions. Rest the mouse pointer on
// Concat in the following statement to verify that the version
// that is used here takes three object arguments. Both 42 and
// true must be boxed.
Console.WriteLine(String.Concat("Answer", 42, true));

// List example.
// Create a list of objects to hold a heterogeneous collection
// of elements.
List<object> mixedList = new List<object>();

// Add a string element to the list.
mixedList.Add("First Group:");
```

```
// Add some integers to the list.
for (int j = 1; j < 5; j++)
{
    // Rest the mouse pointer over j to verify that you are adding
    // an int to a list of objects. Each element j is boxed when
    // you add j to mixedList.
    mixedList.Add(j);
}

// Add another string and more integers.
mixedList.Add("Second Group:");
for (int j = 5; j < 10; j++)
{
    mixedList.Add(j);
}

// Display the elements in the list. Declare the loop variable by
// using var, so that the compiler assigns its type.
foreach (var item in mixedList)
{
    // Rest the mouse pointer over item to verify that the elements
    // of mixedList are objects.
    Console.WriteLine(item);
}

// The following loop sums the squares of the first group of boxed
// integers in mixedList. The list elements are objects, and cannot
// be multiplied or added to the sum until they are unboxed. The
// unboxing must be done explicitly.
var sum = 0;
for (var j = 1; j < 5; j++)
{
    // The following statement causes a compiler error: Operator
    // '*' cannot be applied to operands of type 'object' and
    // 'object'.
    //sum += mixedList[j] * mixedList[j]);

    // After the list elements are unboxed, the computation does
    // not cause a compiler error.
    sum += (int)mixedList[j] * (int)mixedList[j];
}

// The sum displayed is 30, the sum of 1 + 4 + 9 + 16.
Console.WriteLine("Sum: " + sum);

// Output:
// Answer42True
// First Group:
// 1
// 2
// 3
// 4
// Second Group:
// 5
// 6
```

```
// 7  
// 8  
// 9  
// Sum: 30
```

## 성능

단순 할당에서는 boxing과 unboxing을 수행하는 데 많은 계산 과정이 필요합니다. 값 형식을 boxing할 때는 새로운 개체를 할당하고 생성해야 합니다. 정도는 약간 덜하지만 unboxing에 필요한 캐스트에도 상당한 계산 과정이 필요합니다. 자세한 내용은 [성능](#)을 참조하세요.

## boxing

boxing은 가비지 수집되는 힙에 값 형식을 저장하는 데 사용됩니다. Boxing은 값 형식을 `object` 형식 또는 이 값 형식에서 구현된 임의의 인터페이스 형식으로 암시적으로 변환하는 프로세스입니다. 값 형식을 boxing하면 힙에 개체 인스턴스가 할당되고 값이 새 개체에 복사됩니다.

다음과 같이 값 형식 변수를 선언합니다.

C#

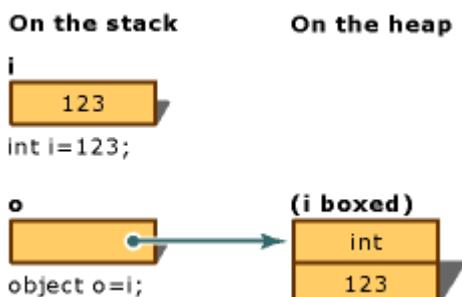
```
int i = 123;
```

다음 문에서는 변수 `i`에 암시적으로 boxing 연산을 적용합니다.

C#

```
// Boxing copies the value of i into object o.  
object o = i;
```

이 문의 결과로 힙에 있는 `o` 형식의 값을 참조하는 `int` 개체 참조가 스택에 생성됩니다. 이 같은 변수 `i`에 할당된 값 형식 값의 복사본입니다. 두 변수 `i` 및 `o`의 차이점은 boxing 변환을 보여주는 다음 이미지에 나와 있습니다.



다음 예제에서와 같이 명시적으로 boxing을 수행할 수도 있지만 명시적 boxing이 반드시 필요한 것은 아닙니다.

C#

```
int i = 123;
object o = (object)i; // explicit boxing
```

## 예제

이 예제에서는 boxing을 통해 정수 변수 `i`를 개체 `o`로 변환합니다. 그런 다음 변수 `i`에 저장된 값을 `123`에서 `456`으로 변경합니다. 이 예제에서는 원래 값 형식과 boxing된 개체에 개별 메모리 위치를 사용하여 서로 다른 값을 저장하는 방법을 보여 줍니다.

C#

```
class TestBoxing
{
    static void Main()
    {
        int i = 123;

        // Boxing copies the value of i into object o.
        object o = i;

        // Change the value of i.
        i = 456;

        // The change in i doesn't affect the value stored in o.
        System.Console.WriteLine("The value-type value = {0}", i);
        System.Console.WriteLine("The object-type value = {0}", o);
    }
}
/* Output:
   The value-type value = 456
   The object-type value = 123
*/
```

## unboxing

Unboxing은 `object` 형식에서 `값 형식`으로, 또는 인터페이스 형식에서 해당 인터페이스를 구현하는 값 형식으로 명시적으로 변환하는 프로세스입니다. unboxing 연산 과정은 다음과 같습니다.

- 개체 인스턴스가 지정한 값 형식을 boxing한 값인지 확인합니다.

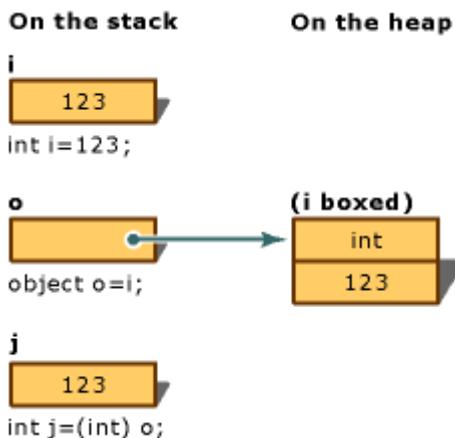
- 인스턴스의 값을 값 형식 변수에 복사합니다.

다음 문은 boxing 및 unboxing 연산을 모두 보여 줍니다.

C#

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

다음 그림에서는 이전 문의 결과를 보여 줍니다.



런타임에 값 형식의 unboxing이 성공하려면 unboxing되는 항목은 이전에 해당 값 형식의 인스턴스를 boxing하여 생성된 개체에 대한 참조여야 합니다. `null`을 unboxing하려고 하면 `NullReferenceException`이 발생합니다. 호환되지 않는 값 형식에 대한 참조를 unboxing하려고 하면 `InvalidCastException`이 발생합니다.

## 예제

다음 예제에서는 잘못된 unboxing의 경우와 그 결과로 발생하는 `InvalidCastException`을 보여 줍니다. 이 예제에서는 `try` 및 `catch`를 사용하여 오류가 발생할 때 오류 메시지를 표시합니다.

C#

```
class TestUnboxing
{
    static void Main()
    {
        int i = 123;
        object o = i; // implicit boxing

        try
        {
            int j = (short)o; // attempt to unbox
```

```
        System.Console.WriteLine("Unboxing OK.");
    }
    catch (System.InvalidCastException e)
    {
        System.Console.WriteLine("{0} Error: Incorrect unboxing.", e.Message);
    }
}
```

이 프로그램의 출력은 다음과 같습니다.

```
Specified cast is not valid. Error: Incorrect unboxing.
```

다음 문을

```
C#
```

```
int j = (short)o;
```

다음과 같이 변경합니다.

```
C#
```

```
int j = (int)o;
```

변환이 수행되고 결과가 출력됩니다.

```
Unboxing OK.
```

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)을 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스입니다.

## 참조

- [C# 프로그래밍 가이드](#)
- [참조 형식](#)
- [값 형식](#)

# 바이트 배열을 int로 변환하는 방법(C# 프로그래밍 가이드)

아티클 • 2024. 03. 08.

이 예제에서는 [BitConverter](#) 클래스를 사용하여 바이트 배열을 [int](#)로 변환하고 다시 바이트 배열로 변환하는 방법을 보여 줍니다. 예를 들어 네트워크에 바이트를 읽은 후 바이트에서 기본 제공 데이터 형식으로 변환해야 할 수 있습니다. 예제의 [ToInt32\(Byte\[\], Int32\)](#) 메서드 외에도 다음 표에서는 바이트(바이트 배열)를 다른 기본 제공 형식으로 변환하는 [BitConverter](#) 클래스의 메서드를 보여 줍니다.

[+] 테이블 확장

반환되는 형식	메서드
<code>bool</code>	<a href="#">.ToBoolean(Byte[], Int32)</a>
<code>char</code>	<a href="#">ToChar(Byte[], Int32)</a>
<code>double</code>	<a href="#">.ToDouble(Byte[], Int32)</a>
<code>short</code>	<a href="#">ToInt16(Byte[], Int32)</a>
<code>int</code>	<a href="#">ToInt32(Byte[], Int32)</a>
<code>long</code>	<a href="#">ToInt64(Byte[], Int32)</a>
<code>float</code>	<a href="#">ToSingle(Byte[], Int32)</a>
<code>ushort</code>	<a href="#">ToInt16(Byte[], Int32)</a>
<code>uint</code>	<a href="#">ToInt32(Byte[], Int32)</a>
<code>ulong</code>	<a href="#">ToInt64(Byte[], Int32)</a>

## 예제

이 예제에서는 바이트 배열을 초기화하고, 컴퓨터 아키텍처가 little-endian이면 배열을 반전한 다음(즉, 최하위 바이트가 먼저 저장됨) [ToInt32\(Byte\[\], Int32\)](#) 메서드를 호출하여 배열의 4바이트를 [int](#)로 변환합니다. [ToInt32\(Byte\[\], Int32\)](#)에 대한 두 번째 인수는 바이트 배열의 시작 인덱스를 지정합니다.

① 참고

출력은 컴퓨터 아키텍처의 endianness에 따라 달라질 수 있습니다.

C#

```
byte[] bytes = [0, 0, 0, 25];

// If the system architecture is little-endian (that is, little end first),
// reverse the byte array.
if (BitConverter.IsLittleEndian)
    Array.Reverse(bytes);

int i = BitConverter.ToInt32(bytes, 0);
Console.WriteLine("int: {0}", i);
// Output: int: 25
```

이 예제에서는 [BitConverter](#) 클래스의 [GetBytes\(Int32\)](#) 메서드를 호출하여 `int`를 바이트 배열로 변환합니다.

### ① 참고

출력은 컴퓨터 아키텍처의 endianness에 따라 달라질 수 있습니다.

C#

```
byte[] bytes = BitConverter.GetBytes(201805978);
Console.WriteLine("byte array: " + BitConverter.ToString(bytes));
// Output: byte array: 9A-50-07-0C
```

## 참고 항목

- [BitConverter](#)
- [IsLittleEndian](#)
- [유형](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

### 설명서 문제 열기

☞ 제품 사용자 의견 제공

# 문자열을 숫자로 변환하는 방법(C# 프로그래밍 가이드)

아티클 • 2024. 03. 20.

숫자 형식(`int`, `long`, `double` 등)에 있는 `Parse` 또는 `TryParse` 메서드를 호출하거나 `System.Convert` 클래스의 메서드를 사용하여 `string`을 숫자로 변환합니다.

`TryParse` 메서드(예: `int.TryParse("11", out number)`) 또는 `Parse` 메서드(예: `var number = int.Parse("11")`)를 호출하면 약간 더 효율적이고 간단합니다. `Convert` 메서드 사용은 `IConvertible`을 구현하는 일반 개체에 더 유용합니다.

`Parse` 또는 `TryParse` 메서드는 `System.Int32` 형식과 같이 문자열에 포함되는 숫자 형식에서 사용합니다. `Convert.ToInt32` 메서드는 `Parse`를 내부적으로 사용합니다. `Parse` 메서드는 변환된 숫자를 반환합니다. `TryParse` 메서드는 변환에 성공했는지 여부를 나타내는 부울 값을 반환하며 변환된 숫자를 `out` 매개 변수로 반환합니다. 문자열이 유효한 형식이 아닌 경우 `Parse`는 예외를 throw하지만 `TryParse`는 `false`를 반환합니다. `Parse` 메서드를 호출할 때는 항상 예외 처리를 사용하여 구문 분석 작업이 실패하는 경우 `FormatException`을 catch해야 합니다.

## Parse 또는 TryParse 메서드 호출

`Parse` 및 `TryParse` 메서드는 문자열의 시작과 끝에 있는 공백을 무시하지만 다른 모든 문자는 적절한 숫자 형식(`int`, `long`, `ulong`, `float`, `decimal` 등)을 구성하는 문자여야 합니다. 숫자를 구성하는 문자열 내에 공백이 있으면 오류가 발생합니다. 예를 들어 `decimal.TryParse`를 사용하여 "10", "10.3" 또는 " 10 "은 구문 분석할 수 있지만 이 메서드를 사용하여 "10X", "1 0"(공백 포함), "10 .3"(공백 포함), "10e1"(`float.TryParse` 사용) 등에서 10을 구문 분석할 수는 없습니다. 값이 `null` 또는 `String.Empty`인 문자열은 구문 분석되지 않습니다. `String.IsNullOrEmpty` 메서드를 호출하여 구문 분석하기 전에 Null 또는 빈 문자열을 확인할 수 있습니다.

다음 예제에서는 `Parse` 및 `TryParse` 호출이 성공하는 경우와 실패하는 경우를 모두 보여줍니다.

```
C#  
  
using System;  
  
public static class StringConversion  
{  
    public static void Main()  
}
```

```
{  
    string input = String.Empty;  
    try  
    {  
        int result = Int32.Parse(input);  
        Console.WriteLine(result);  
    }  
    catch (FormatException)  
    {  
        Console.WriteLine($"Unable to parse '{input}'");  
    }  
    // Output: Unable to parse ''  
  
    try  
    {  
        int numVal = Int32.Parse("-105");  
        Console.WriteLine(numVal);  
    }  
    catch (FormatException e)  
    {  
        Console.WriteLine(e.Message);  
    }  
    // Output: -105  
  
    if (Int32.TryParse("-105", out int j))  
    {  
        Console.WriteLine(j);  
    }  
    else  
    {  
        Console.WriteLine("String could not be parsed.");  
    }  
    // Output: -105  
  
    try  
    {  
        int m = Int32.Parse("abc");  
    }  
    catch (FormatException e)  
    {  
        Console.WriteLine(e.Message);  
    }  
    // Output: Input string was not in a correct format.  
  
    const string inputString = "abc";  
    if (Int32.TryParse(inputString, out int numValue))  
    {  
        Console.WriteLine(numValue);  
    }  
    else  
    {  
        Console.WriteLine($"Int32.TryParse could not parse  
'{inputString}' to an int.");  
    }  
    // Output: Int32.TryParse could not parse 'abc' to an int.
```

```
    }  
}
```

다음 예제에서는 선행 숫자(16진수 문자 포함) 및 숫자가 아닌 후행 문자를 포함해야 하는 문자열을 구문 분석하는 한 가지 방법을 보여 줍니다. [TryParse](#) 메서드를 호출하기 전에 문자열 시작 부분의 유효한 문자를 새 문자열에 할당합니다. 구문 분석할 문자열에 몇 개의 문자가 포함되어 있으므로 예제에서는 [String.Concat](#) 메서드를 호출하여 새 문자열에 유효한 문자를 할당합니다. 더 큰 문자열의 경우 [StringBuilder](#) 클래스를 대신 사용할 수 있습니다.

C#

```
using System;  
  
public static class StringConversion  
{  
    public static void Main()  
    {  
        var str = " 10FFxxx";  
        string numericString = string.Empty;  
        foreach (var c in str)  
        {  
            // Check for numeric characters (hex in this case) or leading or  
            // trailing spaces.  
            if ((c >= '0' && c <= '9') || (char.ToUpperInvariant(c) >= 'A'  
                && char.ToUpperInvariant(c) <= 'F') || c == ' ')  
            {  
                numericString = string.Concat(numericString, c.ToString());  
            }  
            else  
            {  
                break;  
            }  
        }  
  
        if (int.TryParse(numericString,  
System.Globalization.NumberStyles.HexNumber, null, out int i))  
        {  
            Console.WriteLine($"'{str}' --> '{numericString}' --> {i}");  
        }  
        // Output: ' 10FFxxx' --> ' 10FF' --> 4351  
  
        str = " -10FFXXX";  
        numericString = "";  
        foreach (char c in str)  
        {  
            // Check for numeric characters (0-9), a negative sign, or  
            // leading or trailing spaces.  
            if ((c >= '0' && c <= '9') || c == '-' || c == ' ')  
            {  
                numericString = string.Concat(numericString, c);  
            }  
        }  
    }  
}
```

```

        }
        else
        {
            break;
        }
    }

    if (int.TryParse(numericString, out int j))
    {
        Console.WriteLine($"'{str}' --> '{numericString}' --> {j}");
    }
    // Output: '-10FFXXX' --> '-10' --> -10
}
}

```

## Convert 메서드 호출

다음 표에는 문자열을 숫자로 변환하는 데 사용할 수 있는 [Convert](#) 클래스의 일부 메서드가 나와 있습니다.

[\[\] 테이블 확장](#)

숫자 형식	메서드
decimal	ToDecimal(String)
float	ToSingle(String)
double	ToDouble(String)
short	ToInt16(String)
int	ToInt32(String)
long	ToInt64(String)
ushort	ToUInt16(String)
uint	ToUInt32(String)
ulong	ToUInt64(String)

다음 예제에서는 입력 문자열을 [Convert.ToInt32\(String\)](#) int로 변환하는 메서드를 [호출합니다](#). 이 예제에서는 이 메서드 [FormatException](#)에서 throw하는 가장 일반적인 두 가지 예외를 catch합니다 [OverflowException](#). [Int32.MaxValue](#)을 초과하지 않고 결과 숫자를 증분할 수 있는 경우 예제에서는 결과에 1을 더하고 출력을 표시합니다.

C#

```
using System;

public class ConvertStringExample1
{
    static void Main(string[] args)
    {
        int numVal = -1;
        bool repeat = true;

        while (repeat)
        {
            Console.Write("Enter a number between -2,147,483,648 and
+2,147,483,647 (inclusive): ");

            string? input = Console.ReadLine();

            //ToInt32 can throw FormatException or OverflowException.
            try
            {
                numVal = Convert.ToInt32(input);
                if (numVal < Int32.MaxValue)
                {
                    Console.WriteLine("The new value is {0}", ++numVal);
                }
                else
                {
                    Console.WriteLine("numVal cannot be incremented beyond
its current value");
                }
            }
            catch (FormatException)
            {
                Console.WriteLine("Input string is not a sequence of
digits.");
            }
            catch (OverflowException)
            {
                Console.WriteLine("The number cannot fit in an Int32.");
            }

            Console.Write("Go again? Y/N: ");
            string? go = Console.ReadLine();
            if (go?.ToUpper() != "Y")
            {
                repeat = false;
            }
        }
    }
}

// Sample Output:
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive):
473
//   The new value is 474
//   Go again? Y/N: y
```

```
// Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive):  
2147483647  
// numVal cannot be incremented beyond its current value  
// Go again? Y/N: y  
// Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive):  
-1000  
// The new value is -999  
// Go again? Y/N: n
```

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

ଓ 설명서 문제 열기

ଓ 제품 사용자 의견 제공

# 16진수 문자열과 숫자 형식 간 변환 방법 (C# 프로그래밍 가이드)

아티클 • 2023. 05. 10.

이 예제에서는 다음 작업을 수행하는 방법을 보여 줍니다.

- `string`에 있는 각 문자의 16진수 값을 가져옵니다.
- 16진수 문자열의 각 값에 해당하는 `char`을 가져옵니다.
- 16진수 `string`을 `int`로 변환합니다.
- 16진수 `string`을 `float`로 변환합니다.
- `byte` 배열을 16진수 `string`으로 변환합니다.

## 예제

이 예제에서는 `string`에 있는 각 문자의 16진수 값을 출력합니다. 먼저 `string`을 문자 배열로 구문 분석합니다. 그런 다음 각 문자에서 `ToInt32(Char)`를 호출하여 해당 숫자 값을 가져옵니다. 마지막으로, `string`에서 숫자의 형식을 16진수 표현으로 지정합니다.

C#

```
string input = "Hello World!";
char[] values = input.ToCharArray();
foreach (char letter in values)
{
    // Get the integral value of the character.
    int value = Convert.ToInt32(letter);
    // Convert the integer value to a hexadecimal value in string form.
    Console.WriteLine($"Hexadecimal value of {letter} is {value:X}");
}
/* Output:
   Hexadecimal value of H is 48
   Hexadecimal value of e is 65
   Hexadecimal value of l is 6C
   Hexadecimal value of l is 6C
   Hexadecimal value of o is 6F
   Hexadecimal value of   is 20
   Hexadecimal value of W is 57
   Hexadecimal value of o is 6F
   Hexadecimal value of r is 72
   Hexadecimal value of l is 6C
   Hexadecimal value of d is 64
```

```
    Hexadecimal value of ! is 21
*/
```

이 예제에서는 16진수 값의 `string`을 구문 분석하고 각 16진수 값에 해당하는 문자를 출력합니다. 먼저 `Split(Char[])` 메서드를 호출하여 각 16진수 값을 배열의 개별 `string` 값으로 가져옵니다. 그런 다음 `ToInt32(String, Int32)`를 호출하여 16진수 값을 `int`로 표현된 10진수 값으로 변환합니다. 다음은 문자 코드에 해당하는 문자를 가져오는 두 가지 방법을 보여 줍니다. 첫 번째 방법은 정수 인수에 해당하는 문자를 `string`으로 반환하는 `ConvertFromUtf32(Int32)`를 사용합니다. 두 번째 방법은 `int`를 `char`로 명시적으로 캐스팅합니다.

C#

```
string hexValues = "48 65 6C 6C 6F 20 57 6F 72 6C 64 21";
string[] hexValuesSplit = hexValues.Split(' ');
foreach (string hex in hexValuesSplit)
{
    // Convert the number expressed in base-16 to an integer.
    int value = Convert.ToInt32(hex, 16);
    // Get the character corresponding to the integral value.
    string stringValue = Char.ConvertFromUtf32(value);
    char charValue = (char)value;
    Console.WriteLine("hexadecimal value = {0}, int value = {1}, char value
= {2} or {3}",
                      hex, value, stringValue, charValue);
}
/* Output:
   hexadecimal value = 48, int value = 72, char value = H or H
   hexadecimal value = 65, int value = 101, char value = e or e
   hexadecimal value = 6C, int value = 108, char value = l or l
   hexadecimal value = 6C, int value = 108, char value = l or l
   hexadecimal value = 6F, int value = 111, char value = o or o
   hexadecimal value = 20, int value = 32, char value =  or
   hexadecimal value = 57, int value = 87, char value = W or W
   hexadecimal value = 6F, int value = 111, char value = o or o
   hexadecimal value = 72, int value = 114, char value = r or r
   hexadecimal value = 6C, int value = 108, char value = l or l
   hexadecimal value = 64, int value = 100, char value = d or d
   hexadecimal value = 21, int value = 33, char value = ! or !
*/
```

이 예제에서는 `Parse(String, NumberStyles)` 메서드를 호출하여 16진수 `string`을 정수로 변환하는 방법을 보여 줍니다.

C#

```
string hexString = "8E2";
int num = Int32.Parse(hexString,
System.Globalization.NumberStyles.HexNumber);
```

```
Console.WriteLine(num);
//Output: 2274
```

다음 예제에서는 [System.BitConverter](#) 클래스 및 [UInt32.Parse](#) 메서드를 사용하여 16진수 `string` 을 `float`로 변환하는 방법을 보여 줍니다.

C#

```
string hexString = "43480170";
uint num = uint.Parse(hexString,
    System.Globalization.NumberStyles.AllowHexSpecifier);

byte[] floatVals = BitConverter.GetBytes(num);
float f = BitConverter.ToSingle(floatVals, 0);
Console.WriteLine("float convert = {0}", f);

// Output: 200.0056
```

다음 예제에서는 [System.BitConverter](#) 클래스를 사용하여 `byte` 배열을 16진수 문자열로 변환하는 방법을 보여 줍니다.

C#

```
byte[] vals = { 0x01, 0xAA, 0xB1, 0xDC, 0x10, 0xDD };

string str = BitConverter.ToString(vals);
Console.WriteLine(str);

str = BitConverter.ToString(vals).Replace("-", "");
Console.WriteLine(str);

/*Output:
 01-AA-B1-DC-10-DD
 01AAB1DC10DD
 */
```

다음 예제에서는 .NET 5.0에 도입된 [Convert.ToString](#) 메서드를 호출하여 `byte` 배열을 16진수 문자열로 변환하는 방법을 보여 줍니다.

C#

```
byte[] array = { 0x64, 0x6f, 0x74, 0x63, 0x65, 0x74 };

string hexValue = Convert.ToString(array);
Console.WriteLine(hexValue);

/*Output:
```

646F74636574

\*/

## 참조

- 표준 숫자 형식 문자열
- 형식
- 문자열이 숫자 값을 나타내는지 확인 방법

# Override 및 New 키워드를 사용하여 버전 관리(C# 프로그래밍 가이드)

아티클 • 2023. 04. 07.

C# 언어는 서로 다른 라이브러리의 [기본](#) 및 파생 클래스 간 버전 관리를 개발하고 이전 버전과의 호환성을 유지할 수 있도록 설계되었습니다. 예를 들어 파생 클래스의 멤버와 동일한 이름을 가진 기본 [클래스](#)에 새 멤버가 추가되면 C#이 완전히 지원되고 예기치 않은 동작이 발생하지 않습니다. 따라서 클래스는 메서드가 상속된 메서드를 재정의할지 아니면 메서드가 유사한 이름의 상속된 메서드를 숨기는 새 메서드인지를 명시적으로 지정해야 합니다.

C#에서 파생 클래스는 기본 클래스 메서드와 동일한 이름 가진 메서드를 포함할 수 있습니다.

- 파생 클래스의 메서드 앞에 `new` 또는 `override` 키워드가 있으면 컴파일러는 경고를 표시하고 메서드는 `new` 키워드가 있는 것처럼 작동합니다.
- 파생 클래스의 메서드 앞에 `new` 키워드가 있는 경우 이 메서드는 기본 클래스의 메서드와 독립적으로 정의됩니다.
- 파생 클래스의 메서드 앞에 `override` 키워드가 있는 경우 파생 클래스의 개체는 기본 클래스 메서드 대신 해당 메서드를 호출합니다.
- `override` 키워드를 파생 클래스의 메서드에 적용하려면 기본 클래스 메서드가 [가상](#)으로 정의되어야 합니다.
- 기본 클래스 메서드는 `base` 키워드를 사용하여 파생 클래스 내에서 호출할 수 있습니다.
- `override`, `virtual`, 및 `new` 키워드는 속성, 인덱서 및 이벤트에도 적용될 수 있습니다.

기본적으로 C# 메서드는 가상입니다. 메서드가 가상으로 선언되면 이 메서드를 상속하는 클래스는 자체 버전을 구현할 수 있습니다. 메서드를 가상으로 만들려면 기본 클래스의 메서드 선언에서 `virtual` 한정자를 사용합니다. 파생 클래스는 `override` 키워드를 사용하여 기본 가상 메서드를 재정의하거나 `new` 키워드를 사용하여 기본 클래스에서 가상 메서드를 숨길 수 있습니다. `override` 키워드와 `new` 키워드가 모두 지정되지 않은 경우 컴파일러는 경고를 표시하고 파생 클래스의 메서드는 기본 클래스의 메서드를 숨깁니다.

이를 실증적으로 보여 주기 위해, A 회사가 프로그램에서 사용하는 `GraphicsClass`라는 클래스를 만들었다고 잠시 가정해 보겠습니다. 다음은 `GraphicsClass`입니다.

C#

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
}
```

회사에서 이 클래스를 사용하며, 사용자는 이를 사용하여 고유한 클래스를 파생시키고 새 메서드를 추가합니다.

C#

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public void DrawRectangle() { }
}
```

회사 A에서 `GraphicsClass`의 새 버전을 출시할 때까지 애플리케이션은 문제없이 사용됩니다. 새 버전은 다음 코드와 유사합니다.

C#

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
    public virtual void DrawRectangle() { }
}
```

`GraphicsClass`의 새 버전에는 이제 `DrawRectangle`이라는 메서드가 포함되어 있습니다. 처음에는 아무것도 발생하지 않습니다. 새 버전은 여전히 이전 버전과 호환되는 이진입니다. 새 클래스가 해당 컴퓨터 시스템에 설치되어 있어도 사용자가 배포한 소프트웨어는 계속 작동합니다. `DrawRectangle` 메서드에 대한 호출은 파생 클래스에서 계속 해당 버전을 참조합니다.

그러나 `GraphicsClass`의 새 버전을 사용하여 애플리케이션을 다시 컴파일하자마자 컴파일러에서 경고(CS0108)를 표시합니다. 이 경고는 `DrawRectangle` 메서드가 애플리케이션에서 어떻게 작동할지를 고려해야 한다고 알립니다.

메서드가 새로운 기본 클래스 메서드를 재정의하도록 하려면 `override` 키워드를 사용합니다.

C#

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public override void DrawRectangle() { }
}
```

`override` 키워드는 `YourDerivedGraphicsClass`에서 파생된 개체가 `DrawRectangle`의 파생 클래스 버전을 사용하도록 합니다. `YourDerivedGraphicsClass`에서 파생된 개체는 기본 키워드를 사용하여 여전히 `DrawRectangle`의 기본 클래스 버전에 액세스할 수 있습니다.

C#

```
base.DrawRectangle();
```

메서드가 새 기본 클래스 메서드를 재정의하도록 하지 않으려면 다음 고려 사항을 적용하세요. 두 메서드 간 혼동을 피하려면 메서드의 이름을 바꿀 수 있습니다. 이 방법은 시간이 오래 걸리고 오류가 발생하기 쉬우며 어떤 경우에는 실용적이지 않습니다. 그러나 프로젝트 규모가 비교적 작으면 Visual Studio 리팩터링 옵션을 사용하여 메서드의 이름을 바꿀 수 있습니다. 자세한 내용은 [클래스 및 형식 리팩터링\(클래스 디자이너\)](#)을 참조하세요.

또는 파생 클래스 정의에서 `new` 키워드를 사용하여 경고를 방지할 수 있습니다.

C#

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public new void DrawRectangle() { }
}
```

`new` 키워드는 사용자 정의가 기본 클래스에 포함된 정의를 숨긴다는 것을 컴파일러에 알립니다. 이것은 기본적인 동작입니다.

## 재정의 및 메서드 선택

클래스에서 메서드 이름을 지정할 때, 동일한 이름을 가진 두 개의 메서드가 있고 전달된 매개 변수와 호환되는 매개 변수가 있는 경우와 같이 둘 이상의 메서드가 호출과 호환되는 경우, C# 컴파일러는 호출할 수 있는 최상의 메서드를 선택합니다. 다음 메서드는 호환 가능합니다.

C#

```
public class Derived : Base
{
```

```
public override void DoWork(int param) { }
public void DoWork(double param) { }
}
```

`Derived`의 인스턴스에서 `DoWork`가 호출되면 C# 컴파일러는 호출이 원래 `Derived`에 선언된 `DoWork` 버전과 호환되도록 만들려고 시도합니다. 재정의 메서드는 클래스에서 선언된 것으로 간주되지 않습니다. 재정의 메서드는 기본 클래스에서 선언된 메서드의 새로운 구현입니다. C# 컴파일러는 메서드 호출을 `Derived`의 원래 메서드와 일치시킬 수 없는 경우에만, 재정의된 메서드에 대한 호출을 동일한 이름 및 호환되는 매개 변수와 일치시키려고 시도합니다. 예를 들어:

C#

```
int val = 5;
Derived d = new Derived();
d.DoWork(val); // Calls DoWork(double).
```

`val` 변수를 `double`로 암시적으로 변환할 수 있으므로 C# 컴파일러는 `DoWork(int)` 대신 `DoWork(double)`을 호출합니다. 이것을 방지할 수 있는 두 가지 방법이 있습니다. 첫째, 가상 메서드와 동일한 이름을 가진 새 메서드를 선언하지 않습니다. 둘째, `Derived`의 인스턴스를 `Base`에 캐스팅하여 기본 클래스 메서드 목록을 검색하게 함으로써 가상 메서드를 호출하도록 C# 컴파일러에 지시할 수 있습니다. 메서드는 가상이므로 `Derived`에서 `DoWork(int)`의 구현이 호출됩니다. 예를 들어:

C#

```
((Base)d).DoWork(val); // Calls DoWork(int) on Derived.
```

`new` 및 `override`의 추가 예제는 [Override 및 New 키워드를 사용해야 하는 경우](#)를 참조하세요.

## 참조

- [C# 프로그래밍 가이드](#)
- [C# 형식 시스템](#)
- [메서드](#)
- [상속](#)

# Override 및 New 키워드를 사용해야 하는 경우(C# 프로그래밍 가이드)

아티클 • 2023. 04. 07.

C#에서는 파생 클래스의 메서드가 기본 클래스의 메서드와 동일한 이름을 사용할 수 있습니다. `new` 및 `override` 키워드를 사용하여 메서드가 상호 작용하는 방식을 지정할 수 있습니다. `override` 한정자는 기본 클래스 `virtual` 메서드를 확장하고, `new` 한정자는 기본 클래스 메서드를 숨깁니다. 이 항목의 예제에서는 차이점을 보여 줍니다.

콘솔 애플리케이션에서 `BaseClass` 및 `DerivedClass`라는 두 클래스를 선언합니다.

`DerivedClass`는 `BaseClass`에서 상속됩니다.

C#

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
}

class DerivedClass : BaseClass
{
    public void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}
```

`Main` 메서드에서 `bc`, `dc` 및 `bcdc` 변수를 선언합니다.

- `bc`는 `BaseClass` 형식이고, 해당 값은 `BaseClass` 형식입니다.
- `dc`는 `DerivedClass` 형식이고, 해당 값은 `DerivedClass` 형식입니다.
- `bcdc`는 `BaseClass` 형식이고, 해당 값은 `DerivedClass` 형식입니다. 이 변수에 주의해야 합니다.

`bc` 및 `bcdc`는 `BaseClass` 형식이기 때문에 캐스팅을 사용하지 않는 경우 `Method1`에 직접 액세스할 수만 있습니다. `dc` 변수는 `Method1` 및 `Method2` 둘 다에 액세스할 수 있습니다. 이러한 관계는 다음 코드에 나와 있습니다.

C#

```

class Program
{
    static void Main(string[] args)
    {
        BaseClass bc = new BaseClass();
        DerivedClass dc = new DerivedClass();
        BaseClass bcdc = new DerivedClass();

        bc.Method1();
        dc.Method1();
        dc.Method2();
        bcdc.Method1();
    }
    // Output:
    // Base - Method1
    // Base - Method1
    // Derived - Method2
    // Base - Method1
}

```

다음 `Method2` 메서드를 `BaseClass`에 추가합니다. 이 메서드의 시그니처는 `DerivedClass`에 있는 `Method2` 메서드의 시그니처와 일치합니다.

C#

```

public void Method2()
{
    Console.WriteLine("Base - Method2");
}

```

이제 `BaseClass`에 `Method2` 메서드가 있으므로 다음 코드와 같이 `BaseClass` 변수 `bc` 및 `bcdc`에 대해 두 번째 호출 문을 추가할 수 있습니다.

C#

```

bc.Method1();
bc.Method2();
dc.Method1();
dc.Method2();
bcdc.Method1();
bcdc.Method2();

```

프로젝트를 빌드할 때 `BaseClass`에 `Method2` 메서드를 추가하면 경고가 발생합니다.

`DerivedClass`의 `Method2` 메서드가 `BaseClass`의 `Method2` 메서드를 숨긴다는 경고가 표시됩니다. 이러한 결과를 원한다면 `Method2` 정의에 `new` 키워드를 사용하는 것이 좋습니다. 또는 `Method2` 메서드 중 하나의 이름을 바꿔 경고를 해결할 수 있지만 이 방법이 항상 실현 가능한 것은 아닙니다.

`new`를 추가하기 전에 프로그램을 실행하여 추가 호출 문에서 생성되는 출력을 확인합니다. 다음과 같은 결과가 표시됩니다.

C#

```
// Output:  
// Base - Method1  
// Base - Method2  
// Base - Method1  
// Derived - Method2  
// Base - Method1  
// Base - Method2
```

`new` 키워드는 해당 출력을 생성하는 관계를 유지하지만 경고는 표시하지 않습니다.

`BaseClass` 형식인 변수는 계속해서 `BaseClass`의 멤버에 액세스하고, `DerivedClass` 형식인 변수는 계속해서 `DerivedClass`의 멤버에 먼저 액세스한 다음 `BaseClass`에서 상속된 멤버를 고려합니다.

경고를 표시하지 않으려면 다음 코드와 같이 `DerivedClass`에 있는 `Method2`의 정의에 `new` 한정자를 추가합니다. `public` 앞이나 뒤에 한정자를 추가할 수 있습니다.

C#

```
public new void Method2()  
{  
    Console.WriteLine("Derived - Method2");  
}
```

프로그램을 다시 실행하여 출력이 변경되지 않았는지 확인합니다. 또한 경고가 더 이상 나타나지 않는지 확인합니다. `new`를 사용하면 수정하는 멤버가 기본 클래스에서 상속된 멤버를 숨김을 인식하고 있다고 어설션하는 것입니다. 상속을 통한 이름 숨기기에 대한 자세한 내용은 [new 한정자](#)를 참조하세요.

이 동작을 `override`를 사용할 경우의 효과와 비교하려면 다음 메서드를 `DerivedClass`에 추가합니다. `public` 앞이나 뒤에 `override` 한정자를 추가할 수 있습니다.

C#

```
public override void Method1()  
{  
    Console.WriteLine("Derived - Method1");  
}
```

`BaseClass`에 있는 `Method1`의 정의에 `virtual` 한정자를 추가합니다. `public` 앞이나 뒤에 `virtual` 한정자를 추가할 수 있습니다.

C#

```
public virtual void Method1()
{
    Console.WriteLine("Base - Method1");
}
```

프로젝트를 다시 실행합니다. 특히 다음 출력의 마지막 두 줄을 확인합니다.

C#

```
// Output:
// Base - Method1
// Base - Method2
// Derived - Method1
// Derived - Method2
// Derived - Method1
// Base - Method2
```

`override` 한정자를 사용하면 `bcdc`가 `DerivedClass`에 정의된 `Method1` 메서드에 액세스 할 수 있습니다. 일반적으로 이는 상속 계층 구조에서 원하는 동작입니다. 파생 클래스에서 생성된 값을 가진 개체에 파생 클래스에서 정의된 메서드를 사용하려고 합니다. 이 동작을 위해 `override`를 사용하여 기본 클래스 메서드를 확장합니다.

다음 코드에는 전체 예제가 포함되어 있습니다.

C#

```
using System;
using System.Text;

namespace OverrideAndNew
{
    class Program
    {
        static void Main(string[] args)
        {
            BaseClass bc = new BaseClass();
            DerivedClass dc = new DerivedClass();
            BaseClass bcdc = new DerivedClass();

            // The following two calls do what you would expect. They call
            // the methods that are defined in BaseClass.
            bc.Method1();
            bc.Method2();
            // Output:
            // Base - Method1
            // Base - Method2

            // The following two calls do what you would expect. They call
            // the methods that are defined in DerivedClass.
            dc.Method1();
            dc.Method2();
            // Output:
            // Derived - Method1
            // Derived - Method2
        }
    }
}
```

```

        // the methods that are defined in DerivedClass.
        dc.Method1();
        dc.Method2();
        // Output:
        // Derived - Method1
        // Derived - Method2

        // The following two calls produce different results, depending
        // on whether override (Method1) or new (Method2) is used.
        bcdc.Method1();
        bcdc.Method2();
        // Output:
        // Derived - Method1
        // Base - Method2
    }

}

class BaseClass
{
    public virtual void Method1()
    {
        Console.WriteLine("Base - Method1");
    }

    public virtual void Method2()
    {
        Console.WriteLine("Base - Method2");
    }
}

class DerivedClass : BaseClass
{
    public override void Method1()
    {
        Console.WriteLine("Derived - Method1");
    }

    public new void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}

```

다음 예제에서는 다른 컨텍스트의 비슷한 동작을 보여 줍니다. 이 예제에서는 `Car`라는 기본 클래스 하나와 이 클래스에서 파생된 두 클래스 `ConvertibleCar` 및 `Minivan`을 정의합니다. 기본 클래스에는 `DescribeCar` 메서드가 포함되어 있습니다. 이 메서드는 자동차에 대한 기본 설명을 표시한 다음 `ShowDetails`를 호출하여 추가 정보를 제공합니다. 세 클래스는 각각 `ShowDetails` 메서드를 정의합니다. `new` 한정자는 `ConvertibleCar` 클래스의 `ShowDetails`를 정의하는 데 사용됩니다. `override` 한정자는 `Minivan` 클래스의 `ShowDetails`를 정의하는 데 사용됩니다.

C#

```
// Define the base class, Car. The class defines two methods,
// DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each
// derived
// class also defines a ShowDetails method. The example tests which version
// of
// ShowDetails is selected, the base class method or the derived class
// method.
class Car
{
    public void DescribeCar()
    {
        System.Console.WriteLine("Four wheels and an engine.");
        ShowDetails();
    }

    public virtual void ShowDetails()
    {
        System.Console.WriteLine("Standard transportation.");
    }
}

// Define the derived classes.

// Class ConvertibleCar uses the new modifier to acknowledge that
// ShowDetails
// hides the base class method.
class ConvertibleCar : Car
{
    public new void ShowDetails()
    {
        System.Console.WriteLine("A roof that opens up.");
    }
}

// Class Minivan uses the override modifier to specify that ShowDetails
// extends the base class method.
class Minivan : Car
{
    public override void ShowDetails()
    {
        System.Console.WriteLine("Carries seven people.");
    }
}
```

예제에서는 호출되는 `ShowDetails` 버전을 테스트합니다. 다음 메서드 `TestCars1`은 각 클래스의 인스턴스를 선언한 다음 각 인스턴스에서 `DescribeCar`를 호출합니다.

C#

```

public static void TestCars1()
{
    System.Console.WriteLine("\nTestCars1");
    System.Console.WriteLine("-----");

    Car car1 = new Car();
    car1.DescribeCar();
    System.Console.WriteLine("-----");

    // Notice the output from this test case. The new modifier is
    // used in the definition of ShowDetails in the ConvertibleCar
    // class.

    ConvertibleCar car2 = new ConvertibleCar();
    car2.DescribeCar();
    System.Console.WriteLine("-----");

    Minivan car3 = new Minivan();
    car3.DescribeCar();
    System.Console.WriteLine("-----");
}

```

`TestCars1`은 다음 출력을 생성합니다. 특히 예상과 다를 수 있는 `car2`의 결과를 확인합니다. 개체 형식은 `ConvertibleCar`이지만 `DescribeCar`는 `ConvertibleCar` 클래스에 정의된 `ShowDetails` 버전에 액세스하지 않습니다. 해당 메서드는 `override` 한정자가 아니라 `new` 한정자로 선언되기 때문입니다. 결과적으로 `ConvertibleCar` 개체는 `Car` 개체와 동일한 설명을 표시합니다. `Minivan` 개체인 `car3`의 결과와 비교합니다. 이 경우 `Minivan` 클래스에서 선언된 `ShowDetails` 메서드가 `Car` 클래스에서 선언된 `ShowDetails` 메서드를 재정의하고, 표시되는 설명은 미니밴에 대해 설명합니다.

C#

```

// TestCars1
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Carries seven people.
// -----

```

`TestCars2`는 `Car` 형식인 개체 목록을 만듭니다. 개체의 값은 `Car`, `ConvertibleCar` 및 `Minivan` 클래스에서 인스턴스화됩니다. 목록의 각 요소에 대해 `DescribeCar`가 호출됩니다. 다음 코드에서는 `TestCars2`의 정의를 보여 줍니다.

C#

```
public static void TestCars2()
{
    System.Console.WriteLine("\nTestCars2");
    System.Console.WriteLine("-----");

    var cars = new List<Car> { new Car(), new ConvertibleCar(),
        new Minivan() };

    foreach (var car in cars)
    {
        car.DescribeCar();
        System.Console.WriteLine("-----");
    }
}
```

다음 출력이 표시됩니다. 이 출력은 `TestCars1`이 표시하는 출력과 같습니다. 개체의 형식이 `ConvertibleCar(TestCars1)` 또는 `Car(TestCars2)`이든 관계없이 `ConvertibleCar` 클래스의 `ShowDetails` 메서드는 호출되지 않습니다. 한편, `car3`은 `Minivan` 형식 또는 `Car` 형식이든 관계없이 두 경우 모두 `Minivan` 클래스에서 `ShowDetails` 메서드를 호출합니다.

C#

```
// TestCars2
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Carries seven people.
// -----
```

`TestCars3` 및 `TestCars4` 메서드가 예제를 완성합니다. 이러한 메서드는 `ConvertibleCar` 및 `Minivan(TestCars3)` 형식으로 선언된 개체와 `Car(TestCars4)` 형식으로 선언된 개체에서 차례로 `ShowDetails`를 직접 호출합니다. 다음 코드에서는 이러한 두 메서드를 정의합니다.

C#

```
public static void TestCars3()
{
    System.Console.WriteLine("\nTestCars3");
    System.Console.WriteLine("-----");
    ConvertibleCar car2 = new ConvertibleCar();
```

```

        Minivan car3 = new Minivan();
        car2.ShowDetails();
        car3.ShowDetails();
    }

    public static void TestCars4()
{
    System.Console.WriteLine("\nTestCars4");
    System.Console.WriteLine("-----");
    Car car2 = new ConvertibleCar();
    Car car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}

```

메서드는 이 항목에 있는 첫 번째 예제의 결과에 해당하는 다음 출력을 생성합니다.

C#

```

// TestCars3
// -----
// A roof that opens up.
// Carries seven people.

// TestCars4
// -----
// Standard transportation.
// Carries seven people.

```

다음 코드에서는 전체 프로젝트와 해당 출력을 보여 줍니다.

C#

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace OverrideAndNew2
{
    class Program
    {
        static void Main(string[] args)
        {
            // Declare objects of the derived classes and test which version
            // of ShowDetails is run, base or derived.
            TestCars1();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars2();

```

```

        // Declare objects of the derived classes and call ShowDetails
        // directly.
        TestCars3();

        // Declare objects of the base class, instantiated with the
        // derived classes, and repeat the tests.
        TestCars4();
    }

    public static void TestCars1()
{
    System.Console.WriteLine("\nTestCars1");
    System.Console.WriteLine("-----");

    Car car1 = new Car();
    car1.DescribeCar();
    System.Console.WriteLine("-----");

    // Notice the output from this test case. The new modifier is
    // used in the definition of ShowDetails in the ConvertibleCar
    // class.
    ConvertibleCar car2 = new ConvertibleCar();
    car2.DescribeCar();
    System.Console.WriteLine("-----");

    Minivan car3 = new Minivan();
    car3.DescribeCar();
    System.Console.WriteLine("-----");
}

// Output:
// TestCars1
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Standard transportation.
// -----
// Four wheels and an engine.
// Carries seven people.
// -----


public static void TestCars2()
{
    System.Console.WriteLine("\nTestCars2");
    System.Console.WriteLine("-----");

    var cars = new List<Car> { new Car(), new ConvertibleCar(),
        new Minivan() };

    foreach (var car in cars)
    {
        car.DescribeCar();
        System.Console.WriteLine("-----");
    }
}

```

```

    }

    // Output:
    // TestCars2
    // -----
    // Four wheels and an engine.
    // Standard transportation.
    // -----
    // Four wheels and an engine.
    // Standard transportation.
    // -----
    // Four wheels and an engine.
    // Carries seven people.
    // -----


    public static void TestCars3()
    {
        System.Console.WriteLine("\nTestCars3");
        System.Console.WriteLine("-----");
        ConvertibleCar car2 = new ConvertibleCar();
        Minivan car3 = new Minivan();
        car2.ShowDetails();
        car3.ShowDetails();
    }
    // Output:
    // TestCars3
    // -----
    // A roof that opens up.
    // Carries seven people.


    public static void TestCars4()
    {
        System.Console.WriteLine("\nTestCars4");
        System.Console.WriteLine("-----");
        Car car2 = new ConvertibleCar();
        Car car3 = new Minivan();
        car2.ShowDetails();
        car3.ShowDetails();
    }
    // Output:
    // TestCars4
    // -----
    // Standard transportation.
    // Carries seven people.
}

// Define the base class, Car. The class defines two virtual methods,
// DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each
derived
    // class also defines a ShowDetails method. The example tests which
version of
    // ShowDetails is used, the base class method or the derived class
method.

    class Car
    {
        public virtual void DescribeCar()

```

```

    {
        System.Console.WriteLine("Four wheels and an engine.");
        ShowDetails();
    }

    public virtual void ShowDetails()
    {
        System.Console.WriteLine("Standard transportation.");
    }
}

// Define the derived classes.

// Class ConvertibleCar uses the new modifier to acknowledge that
ShowDetails
// hides the base class method.
class ConvertibleCar : Car
{
    public new void ShowDetails()
    {
        System.Console.WriteLine("A roof that opens up.");
    }
}

// Class Minivan uses the override modifier to specify that ShowDetails
// extends the base class method.
class Minivan : Car
{
    public override void ShowDetails()
    {
        System.Console.WriteLine("Carries seven people.");
    }
}
}

```

## 참고 항목

- C# 프로그래밍 가이드
- C# 형식 시스템
- Override 및 New 키워드를 사용하여 버전 관리
- base
- abstract

# ToString 메서드 재정의 방법(C# 프로그래밍 가이드)

아티클 • 2024. 03. 12.

C#의 모든 클래스 또는 구조체는 `Object` 클래스를 암시적으로 상속합니다. 따라서 C#의 모든 개체는 해당 개체의 문자열 표현을 반환하는 `ToString` 메서드를 가져옵니다. 예를 들어 `int` 형식의 모든 변수에는 해당 내용을 문자열로 반환할 수 있도록 하는 `ToString` 메서드가 있습니다.

C#

```
int x = 42;
string strx = x.ToString();
Console.WriteLine(strx);
// Output:
// 42
```

사용자 지정 클래스 또는 구조체를 만들 때 해당 형식에 대한 정보를 클라이언트 코드에 제공하려면 `ToString` 메서드를 재정의해야 합니다.

`ToString` 메서드와 함께 형식 문자열 및 다른 형식의 사용자 지정 서식을 사용하는 방법에 대한 자세한 내용은 [형식 서식 지정](#)을 참조하세요.

## ① 중요

이 메서드를 통해 제공할 정보를 결정하는 경우 신뢰할 수 없는 코드에서 클래스 또는 구조체가 사용될지 여부를 고려합니다. 악성 코드에서 악용될 수 있는 정보를 제공하지 않으면 주의해야 합니다.

클래스 또는 구조체에서 `ToString` 메서드를 재정의하려면 다음을 수행합니다.

1. 다음 한정자 및 반환 형식으로 `ToString` 메서드를 선언합니다.

C#

```
public override string ToString(){}  
  
```

2. 문자열을 반환하도록 메서드를 구현합니다.

다음 예제에서는 클래스의 특정 인스턴스와 관련된 데이터뿐 아니라 클래스의 이름을 반환합니다.

C#

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public override string ToString()
    {
        return "Person: " + Name + " " + Age;
    }
}
```

다음 코드 예제와 같이 `ToString` 메서드를 테스트할 수 있습니다.

C#

```
Person person = new Person { Name = "John", Age = 12 };
Console.WriteLine(person);
// Output:
// Person: John 12
```

## 참고 항목

- `IFormattable`
- C# 형식 시스템
- 문자열
- `string`
- `override`
- `virtual`
- 형식 서식 지정

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

[설명서 문제 열기](#)

[제품 사용자 의견 제공](#)

# 멤버(C# 프로그래밍 가이드)

아티클 • 2023. 04. 07.

클래스 및 구조체에는 해당 데이터와 동작을 나타내는 멤버가 있습니다. 클래스의 멤버에는 클래스에서 선언된 모든 멤버가 상속 계층 구조의 모든 클래스에서 선언된 모든 멤버(생성자 및 종료자 제외)와 함께 포함됩니다. 기본 클래스의 private 멤버는 상속되지만派生 클래스에서 액세스할 수 없습니다.

다음 표에는 클래스 또는 구조체에 포함될 수 있는 멤버 종류가 나와 있습니다.

멤버 설명
<b>필드</b> 필드는 클래스 범위에서 선언된 변수입니다. 필드는 기본 제공 숫자 형식 또는 다른 클래스의 인스턴스일 수 있습니다. 예를 들어 달력 클래스에는 현재 날짜를 포함하는 필드가 있을 수 있습니다.
<b>상수</b> 상수는 해당 값이 컴파일 시간에 설정되며 변경할 수 없는 필드입니다.
<b>속성</b> 속성은 해당 클래스의 필드처럼 액세스되는 클래스의 메서드입니다. 속성은 클래스 필드에 대한 보호를 제공하여 개체 모르게 필드가 변경되지 않도록 할 수 있습니다.
<b>메서드</b> 메서드는 클래스가 수행할 수 있는 작업을 정의합니다. 메서드는 입력 데이터를 제공하는 매개 변수를 사용할 수 있으며, 매개 변수를 통해 출력 데이터를 반환할 수 있습니다. 메서드가 매개 변수를 사용하지 않고 직접 값을 반환할 수도 있습니다.
<b>이벤트</b> 이벤트는 단추 클릭, 성공적인 메서드 완료 등의 발생에 대한 알림을 다른 개체에 제공합니다. 이벤트는 대리자를 사용하여 정의 및 트리거됩니다.
<b>연산자</b> 오버로드된 연산자는 형식 멤버로 간주됩니다. 연산자를 오버로드하는 경우 유형에서 공용 정적 메서드로 정의합니다. 자세한 내용은 <a href="#">연산자 오버로드</a> 를 참조하세요.
<b>인덱서</b> 인덱서를 사용하면 배열과 유사한 방식으로 개체를 인덱싱할 수 있습니다.
<b>생성자</b> 생성자는 개체를 처음 만들 때 호출되는 메서드입니다. 대체로 개체의 데이터를 초기화하는데 사용됩니다.
<b>종료자</b> 종료자는 C#에서 매우 드물게 사용됩니다. 메모리에서 개체를 제거할 때 런타임 실행 엔진이 호출하는 메서드입니다. 일반적으로 해제해야 하는 리소스가 적절하게 처리되도록 하는 데 사용됩니다.

## 멤버 설명

중첩 형식은 다른 형식 내에서 선언된 형식입니다. 중첩 형식은 대체로 개체를 포함하는 형식에서만 사용되는 개체를 설명하는 데 사용됩니다.

### 중첩 형식

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [클래스](#)

# 추상 및 봉인 클래스와 클래스 멤버(C# 프로그래밍 가이드)

아티클 • 2024. 03. 03.

`abstract` 키워드를 사용하면, 불완전하여 파생 클래스에서 구현해야 하는 클래스 및 `클래스` 멤버를 만들 수 있습니다.

`sealed` 키워드를 사용하면, 이전에 `virtual`로 표시되었던 클래스나 특정 클래스 멤버의 상속을 방지할 수 있습니다.

## 추상 클래스 및 클래스 멤버

클래스 정의 앞에 `abstract` 키워드를 배치하여 클래스를 추상으로 선언할 수 있습니다.

예시:

```
C#  
  
public abstract class A  
{  
    // Class members here.  
}
```

추상 클래스는 인스턴스화할 수 없습니다. 추상 클래스의 목적은 여러 파생 클래스에서 공유할 수 있는 기본 클래스의 공통적인 정의를 제공하는 것입니다. 예를 들어 클래스 라이브러리에서 여러 자체 함수에 매개 변수로 사용되는 추상 클래스를 정의한 다음 해당 라이브러리를 사용하는 프로그래머가 파생 클래스를 만들어 클래스의 고유 구현을 제공하도록 할 수 있습니다.

추상 클래스에서는 추상 메서드도 정의할 수 있습니다. 메서드의 반환 형식 앞에 `abstract` 키워드를 추가하면 추상 메서드가 정의됩니다. 예시:

```
C#  
  
public abstract class A  
{  
    public abstract void DoWork(int i);  
}
```

추상 메서드에는 구현이 없으므로 메서드 정의 다음에는 일반적인 메서드 블록 대신 세미콜론이 옵니다. 추상 클래스의 파생 클래스에서는 모든 추상 메서드를 구현해야 합니다.

다. 추상 클래스에서 기본 클래스의 가상 메서드를 상속하는 경우 추상 클래스에서는 추상 메서드를 사용하여 가상 메서드를 재정의할 수 있습니다. 예시:

```
C#  
  
// compile with: -target:library  
public class D  
{  
    public virtual void DoWork(int i)  
    {  
        // Original implementation.  
    }  
}  
  
public abstract class E : D  
{  
    public abstract override void DoWork(int i);  
}  
  
public class F : E  
{  
    public override void DoWork(int i)  
    {  
        // New implementation.  
    }  
}
```

`virtual` 메서드는 `abstract`로 선언되어도 추상 클래스에서 상속된 모든 클래스에 대해 여전히 가상입니다. 추상 메서드를 상속하는 클래스에서는 메서드의 원본 구현에 액세스 할 수 없습니다. 앞의 예제에서 F 클래스의 `DoWork`에서는 D 클래스의 `DoWork`를 호출할 수 없습니다. 따라서 추상 클래스는 파생 클래스에서 가상 메서드에 대한 새 메서드 구현을 반드시 제공하도록 제한할 수 있습니다.

## 봉인 클래스 및 클래스 멤버

클래스 정의 앞에 `sealed` 키워드를 배치하여 클래스를 `sealed`로 선언할 수 있습니다. 예시:

```
C#  
  
public sealed class D  
{  
    // Class members here.  
}
```

봉인 클래스는 기본 클래스로 사용할 수 없습니다. 그러므로 추상 클래스가 될 수도 없습니다. 봉인 클래스는 상속할 수 없습니다. 봉인 클래스는 기본 클래스로 사용될 수 없으므

로 일부 런타임 최적화에서는 봉인 클래스 멤버 호출이 약간 더 빨라집니다.

기본 클래스의 가상 멤버를 재정의하는 파생 클래스의 메서드, 인덱서, 속성 또는 이벤트는 해당 멤버를 봉인으로 선언할 수 있습니다. 이렇게 하면 이후에 파생되는 클래스에서는 해당 멤버가 가상이 아니게 됩니다. 클래스 멤버 선언에서 `override` 키워드 앞에 `sealed` 키워드를 배치하면 됩니다. 다음은 그 예입니다.

C#

```
public class D : C
{
    public sealed override void DoWork() { }
```

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [C# 형식 시스템](#)
- [상속](#)
- [메서드](#)
- [필드](#)
- [추상 속성 정의 방법](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# 정적 클래스 및 정적 클래스 멤버(C# 프로그래밍 가이드)

아티클 • 2024. 03. 20.

정적 클래스는 기본적으로 비정적 클래스와 동일하지만 한 가지 차이점이 있습니다. 정적 클래스는 인스턴스화할 수 없습니다. 즉, 새 [연산자를](#) 사용하여 클래스 형식의 변수를 만들 수 없습니다. 인스턴스 변수가 없으므로 클래스 이름 자체를 사용하여 정적 클래스의 멤버에 액세스합니다. 예를 들어 `public static` 메서드 `MethodA`를 포함하는 `UtilityClass`라는 정적 클래스가 있는 경우 다음 예제와 같이 메서드를 호출합니다.

C#

```
UtilityClass.MethodA();
```

정적 클래스는 입력 매개 변수에서만 작동하고 내부 인스턴스 필드를 얻거나 설정할 필요가 없는 메서드 집합에 편리한 컨테이너로 사용할 수 있습니다. 예를 들어 .NET 클래스 라이브러리의 정적 `System.Math` 클래스에는 `Math` 클래스의 특정 인스턴스에 고유한 데이터를 저장하거나 검색할 필요 없이 수학 연산을 수행하는 메서드가 포함되어 있습니다. 즉, 다음 예제와 같이 클래스 이름과 메서드 이름을 지정하여 클래스의 멤버를 적용합니다.

C#

```
double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
Console.WriteLine(Math.Floor(dub));
Console.WriteLine(Math.Round(Math.Abs(dub)));

// Output:
// 3.14
// -4
// 3
```

모든 클래스 형식의 경우와 마찬가지로 .NET 런타임은 클래스를 참조하는 프로그램이 로드될 때 정적 클래스에 대한 형식 정보를 로드합니다. 프로그램에서 클래스가 로드되는 시기를 정확하게 지정할 수 없습니다. 그러나 클래스가 프로그램에서 처음으로 참조되기 전에 해당 필드를 로드하고 해당 필드를 초기화하고 정적 생성자를 호출하도록 보장됩니다. 정적 생성자는 한 번만 호출되며, 프로그램이 있는 애플리케이션 도메인의 수명 동안 정적 클래스가 메모리에 유지됩니다.

① 참고

자체 인스턴스 하나만 생성될 수 있도록 하는 비정적 클래스를 만들려면 [C#에서 Singleton 구현](#)을 참조하세요.

다음 목록은 정적 클래스의 주요 기능을 제공합니다.

- 정적 멤버만 포함합니다.
- 인스턴스화할 수 없습니다.
- 봉인되어 있습니다.
- 인스턴스 생성자를 [포함](#)할 수 없습니다.

따라서 정적 클래스를 만드는 것은 기본적으로 정적 멤버와 private 생성자만 포함된 클래스를 만드는 것과 동일합니다. private 생성자는 클래스가 인스턴스화되지 않도록 합니다. 정적 클래스를 사용하면 컴파일러에서 인스턴스 멤버가 실수로 추가되지 않도록 확인할 수 있다는 장점이 있습니다. 컴파일러는 이 클래스의 인스턴스를 만들 수 없게 합니다.

정적 클래스는 봉인되므로 상속할 수 없습니다. 를 제외한 [Object](#)모든 클래스 또는 인터페이스에서 상속할 수 없습니다. 정적 클래스는 인스턴스 생성자를 포함할 수 없습니다. 그러나 정적 생성자는 포함할 수 있습니다. 또한 클래스에 특수한 초기화가 필요한 정적 멤버가 포함된 경우 비정적 클래스에서 정적 생성자도 정의해야 합니다. 자세한 내용은 [정적 생성자를 참조하세요](#).

## 예시

다음은 온도를 섭씨에서 화씨로, 화씨에서 섭씨로 변환하는 두 가지 메서드를 포함하는 정적 클래스의 예입니다.

C#

```
public static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(string temperatureCelsius)
    {
        // Convert argument to double for calculations.
        double celsius = Double.Parse(temperatureCelsius);

        // Convert Celsius to Fahrenheit.
        double fahrenheit = (celsius * 9 / 5) + 32;

        return fahrenheit;
    }

    public static double FahrenheitToCelsius(string temperatureFahrenheit)
    {
        // Convert argument to double for calculations.
    }
}
```

```

        double fahrenheit = Double.Parse(temperatureFahrenheit);

        // Convert Fahrenheit to Celsius.
        double celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }
}

class TestTemperatureConverter
{
    static void Main()
    {
        Console.WriteLine("Please select the convertor direction");
        Console.WriteLine("1. From Celsius to Fahrenheit.");
        Console.WriteLine("2. From Fahrenheit to Celsius.");
        Console.Write(":");

        string? selection = Console.ReadLine();
        double F, C = 0;

        switch (selection)
        {
            case "1":
                Console.Write("Please enter the Celsius temperature: ");
                F =
TemperatureConverter.CelsiusToFahrenheit(Console.ReadLine() ?? "0");
                Console.WriteLine("Temperature in Fahrenheit: {0:F2}", F);
                break;

            case "2":
                Console.Write("Please enter the Fahrenheit temperature: ");
                C =
TemperatureConverter.FahrenheitToCelsius(Console.ReadLine() ?? "0");
                Console.WriteLine("Temperature in Celsius: {0:F2}", C);
                break;

            default:
                Console.WriteLine("Please select a convertor.");
                break;
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Example Output:
   Please select the convertor direction
   1. From Celsius to Fahrenheit.
   2. From Fahrenheit to Celsius.
   :2
   Please enter the Fahrenheit temperature: 20
   Temperature in Celsius: -6.67

```

Press any key to exit.

\*/

## 정적 멤버

비정적 클래스는 정적 메서드, 필드, 속성 또는 이벤트를 포함할 수 있습니다. 정적 멤버는 클래스의 인스턴스가 없는 경우에도 클래스에서 호출할 수 있습니다. 정적 멤버는 항상 인스턴스 이름이 아니라 클래스 이름으로 액세스됩니다. 생성된 클래스 인스턴스 개수에 관계없이 정적 멤버의 복사본 한 개만 있습니다. 정적 메서드와 속성은 포함하는 형식의 비정적 필드 및 이벤트에 액세스할 수 없으며 메서드 매개 변수에 명시적으로 전달되지 않는 한 개체의 인스턴스 변수에 액세스할 수 없습니다.

전체 클래스를 정적으로 선언하는 것보다 일부 정적 멤버를 사용하여 비정적 클래스를 선언하는 것이 더 일반적입니다. 정적 필드의 두 가지 일반적인 용도는 인스턴스화되는 개체 수의 개수를 유지하거나 모든 인스턴스 간에 공유해야 하는 값을 저장하는 것입니다.

정적 메서드는 클래스 인스턴스가 아니라 클래스에 속해 있으므로 오버로드할 수 있지만 재정의할 수 없습니다.

필드를 선언 `static const` 할 수는 없지만 `const` 필드는 기본적으로 동작에서 정적입니다. 형식의 인스턴스가 아니라 형식에 속합니다. 따라서 `const` 정적 필드에 사용되는 것과 동일한 `ClassName.MemberName` 표기법을 사용하여 필드에 액세스할 수 있습니다. 개체 인스턴스는 필요하지 않습니다.

C#은 정적 지역 변수(즉, 메서드 범위에서 선언된 변수)를 지원하지 않습니다.

다음 예제와 같이 멤버의 반환 형식 앞에 `static` 키워드를 사용하여 정적 클래스 멤버를 선언합니다.

C#

```
public class Automobile
{
    public static int NumberOfWheels = 4;

    public static int SizeOfGasTank
    {
        get
        {
            return 15;
        }
    }

    public static void Drive() { }
```

```
public static event EventType? RunOutOfGas;

// Other non-static fields and properties...
}
```

정적 멤버는 정적 멤버에 처음으로 액세스하기 전에 초기화되고 정적 생성자가 있는 경우 정적 생성자가 호출되기 전에 초기화됩니다. 정적 클래스 멤버에 액세스하려면 다음 예제와 같이 변수 이름 대신 클래스 이름을 사용하여 멤버의 위치를 지정합니다.

C#

```
Automobile.Drive();
int i = Automobile.NumberOfWheels;
```

클래스에 정적 필드가 포함된 경우 클래스가 로드될 때 정적 필드를 초기화하는 정적 생성자를 제공합니다.

정적 메서드를 호출하면 MSIL(Microsoft Intermediate Language)로 호출 명령이 생성되는 반면, 인스턴스 메서드를 호출하면 null 개체 참조도 확인하는 `callvirt` 명령이 생성됩니다. 그러나 대부분의 경우 둘 사이의 성능 차이는 중요하지 않습니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)에서 정적 클래스, 정적 및 인스턴스 멤버, 정적 생성자를 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

## 참고 항목

- [static](#)
- [클래스](#)
- [class](#)
- [정적 생성자](#)
- [인스턴스 생성자](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

### 설명서 문제 열기

수도 있습니다. 자세한 내용은  
참여자 가이드를 참조하세요.

☞ 제품 사용자 의견 제공

# 액세스 한정자(C# 프로그래밍 가이드)

아티클 • 2024. 04. 14.

모든 형식과 형식 멤버에는 액세스 수준이 있습니다. 액세스 수준은 사용 중인 어셈블리나 다른 어셈블리에서 형식 또는 형식 멤버를 사용할 수 있는지 여부를 제어합니다. 어셈블리는 단일 컴파일에서 하나 이상의 .cs 파일을 컴파일하여 만들어진 .dll 또는 .exe입니다. 다음 액세스 한정자를 사용하여 형식 또는 멤버를 선언할 때 해당 항목의 액세스 수준을 지정할 수 있습니다.

- public**: 모든 어셈블리의 코드가 이 형식이나 멤버에 액세스할 수 있습니다. 포함하는 형식의 접근성 수준은 해당 형식의 공용 멤버에 대한 접근성 수준을 제어합니다.
- private**: 동일한 `class` 또는 `struct`에 선언된 코드만 이 멤버에 액세스할 수 있습니다.
- protected**: 동일한 `class` 또는 파생된 `class`의 코드만 이 형식이나 멤버에 액세스할 수 있습니다.
- internal**: 동일한 어셈블리의 코드만 이 형식이나 멤버에 액세스할 수 있습니다.
- protected internal**: 동일한 어셈블리 또는 다른 어셈블리의 파생 클래스에 있는 코드만 이 형식이나 멤버에 액세스할 수 있습니다.
- private protected**: 동일한 어셈블리 및 동일한 클래스 또는 파생 클래스의 코드만 형식이나 멤버에 액세스할 수 있습니다.
- file**: 동일한 파일에 있는 코드만 형식이나 멤버에 액세스할 수 있습니다.

형식의 `record` 한정자는 컴파일러가 추가 멤버를 합성하도록 합니다. `record` 한정자는 `record class` 또는 `record struct`의 기본 접근성에 영향을 주지 않습니다.

## 요약표

[+] 테이블 확장

호출자의 위치	public internal	protected internal	protected internal	internal	private protected	private protected	file
파일 내	✓	✓	✓	✓	✓	✓	✓
클래스 내	✓	✓	✓	✓	✓	✓	✗
파생 클래스 (동일한 어셈블리)	✓	✓	✓	✓	✓	✗	✗
비파생 클래스(동일한 어	✓	✓	✗	✓	✗	✗	✗

호출자의 위치	public	protected internal	protected	internal	private protected	private	file
셀블리)							
파생 클래스 (다른 어셈블리)	✓	✓	✓	✗	✗	✗	✗
비파생 클래스(다른 어셈블리)	✓	✗	✗	✗	✗	✗	✗

다음 예제에서는 형식 및 멤버에 대해 액세스 한정자를 지정하는 방법을 보여 줍니다.

C#

```
public class Bicycle
{
    public void Pedal() { }
```

모든 액세스 한정자를 모든 컨텍스트에서 모든 형식 또는 멤버에서 사용할 수 있는 것은 아닙니다. 경우에 따라 포함 형식의 접근성이 해당 멤버의 접근성을 제한합니다.

[partial 클래스 또는 부분 메서드](#)의 선언 중 하나가 접근성을 선언하지 않은 경우 다른 선언의 접근성을 갖습니다. partial 클래스 또는 메서드에 대한 여러 선언이 서로 다른 접근성을 선언하는 경우 컴파일러는 오류를 생성합니다.

## 클래스 및 구조체 액세스 수준

네임스페이스 내에서 직접 선언된 클래스 및 구조체(다른 클래스나 구조체 내에 중첩되지 않음)는 `public` 또는 `internal` 일 수 있습니다. 액세스 한정자가 지정되지 않은 경우 기본값은 `internal`입니다.

구조체 멤버(중첩 클래스 및 구조체 포함)는 `public`, `internal` 또는 `private`으로 선언할 수 있습니다. 클래스 멤버(중첩 클래스 포함)는 `public`, `protected internal`, `protected`, `internal`, `private protected` 또는 `private`으로 선언할 수 있습니다. 클래스 멤버와 구조체 멤버(중첩 클래스 및 구조체 포함)의 액세스 수준은 기본적으로 `private`입니다.

파생 클래스는 기본 형식보다 높은 액세스 수준을 가질 수 없습니다. `internal` 클래스 `A`에서 파생된 `public` 클래스 `B`를 선언할 수 없습니다. 이것이 허용된다면 파생 클래스에서 `A`의 모든 `protected` 또는 `internal` 멤버에 액세스할 수 있게 되므로 결과적으로 `A`가 `public`이 되는 것과 같아집니다.

다른 특정 어셈블리에서 `internal` 형식에 액세스할 수 있도록 하려면 `InternalsVisibleToAttribute`를 사용하면 됩니다. 자세한 내용은 [Friend 어셈블리](#)를 참조하세요.

## 기타 형식

네임스페이스 내에서 직접 선언된 인터페이스는 `public` 또는 `internal` 일 수 있고, 클래스 및 구조체와 마찬가지로 인터페이스도 기본적으로 `internal` 액세스로 설정됩니다. 인터페이스는 다른 형식이 클래스나 구조체에 액세스하는 데 사용되므로 인터페이스 멤버는 기본적으로 `public`입니다. 인터페이스 멤버 선언에는 액세스 한정자가 포함될 수 있습니다. 인터페이스의 모든 구현자에게 필요한 공통 구현을 제공하려면 `interface` 멤버에 대한 액세스 한정자를 사용합니다.

네임스페이스에 직접 선언된 `delegate` 형식에는 기본적으로 `internal` 액세스 권한이 있습니다.

액세스 한정자에 대한 자세한 내용은 [접근성 수준](#) 페이지를 참조하세요.

## 멤버 접근성

`class` 또는 `struct`(중첩 클래스 및 구조체 포함)의 멤버는 6가지 액세스 형식 중 하나로 선언될 수 있습니다. 구조체는 상속을 지원하지 않으므로 구조체 멤버는 `protected`, `protected internal` 또는 `private protected`로 선언할 수 없습니다.

일반적으로 멤버의 액세스 수준은 해당 멤버를 포함하는 형식의 액세스 수준보다 크지 않습니다. 그러나 멤버가 인터페이스 메서드를 구현하거나 `public` 기본 클래스에 정의된 가상 메서드를 재정의하는 경우에는 어셈블리 외부에서 `internal` 클래스의 `public` 멤버에 액세스할 수 있습니다.

멤버의 필드, 속성 또는 이벤트 형식은 멤버 자체의 액세스 수준 이상을 가져야 합니다. 마찬가지로, 메서드, 인덱서 또는 대리자의 반환 형식과 매개 변수 형식은 멤버 자체의 액세스 수준 이상을 가져야 합니다. 예를 들어 `c`도 `public`이 아닌 이상 클래스 `c`를 반환하는 `public` 메서드 `M`이 있을 수 없습니다. 마찬가지로, `A`가 `private`으로 선언되었다면 `A` 형식의 `protected` 속성이 있을 수 없습니다.

사용자 정의 연산자는 항상 `public` 및 `static`으로 선언해야 합니다. 자세한 내용은 [연산자 오버로드](#)를 참조하세요.

`class` 또는 `struct` 멤버의 액세스 수준을 설정하려면 다음 예제와 같이 멤버 선언에 해당 키워드를 추가하세요.

C#

```
// public class:  
public class Tricycle  
{  
    // protected method:  
    protected void Pedal() { }  
  
    // private field:  
    private int _wheels = 3;  
  
    // protected internal property:  
    protected internal int Wheels  
    {  
        get { return _wheels; }  
    }  
}
```

종료자에는 접근성 한정자를 사용할 수 없습니다. `enum` 형식의 멤버는 항상 `public`이며 액세스 한정자를 적용할 수 없습니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)을 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

## 참고 항목

- 한정자 순서 지정(스타일 규칙 IDE0036)
- C# 형식 시스템
- 인터페이스
- 액세스 수준
- `private`
- `public`
- `internal`
- `protected`
- `protected internal`
- `private protected`
- `sealed`
- `class`
- `struct`
- `interface`
- 무명 형식

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 필드(C# 프로그래밍 가이드)

아티클 • 2024. 05. 17.

필드는 [클래스](#) 또는 [구조체](#)에서 직접 선언되는 모든 형식의 변수입니다. 필드는 포함하는 형식의 멤버입니다.

클래스 또는 구조체에는 인스턴스 필드, 정적 필드 또는 둘 다 있을 수 있습니다. 인스턴스 필드는 형식의 인스턴스와 관련이 있습니다. 인스턴스 필드 `F` 이(가) 있는 클래스 `T` 이(가) 있는 경우 형식이 `T`인 두 개체를 만들고 다른 개체의 값에 영향을 주지 않고 각 개체에서 `F` 값을 수정할 수 있습니다. 반면 정적 필드는 형식 자체에 속하며 해당 형식의 모든 인스턴스에서 공유됩니다. 형식 이름만 사용하여 정적 필드에 액세스할 수 있습니다. 인스턴스 이름으로 정적 필드에 액세스 하는 경우 [CS0176](#) 컴파일 시간 오류가 발생합니다.

일반적으로 필드에 대한 `private` 또는 `protected` 접근성을 선언해야 합니다. 형식에서 클라이언트 코드에 노출하는 데이터는 [메서드](#), [속성](#), [인덱서](#)를 통해 제공해야 합니다. 내부 필드에 직접 액세스하는 데 이러한 구문을 사용하면 잘못된 입력 값으로부터 보호할 수 있습니다. 공용 속성에 의해 노출된 데이터를 저장하는 `private` 필드는 백업 저장소 또는 [지원 필드](#)라고 합니다. `public` 필드를 선언할 수 있지만 해당 형식을 사용하는 코드가 해당 필드를 잘못된 값으로 설정하거나 개체의 데이터를 변경하는 것을 방지할 수는 없습니다.

필드는 일반적으로 둘 이상의 형식 메서드에서 액세스할 수 있고 단일 메서드의 수명보다 오랫동안 저장되어야 하는 데이터를 저장합니다. 예를 들어 달력 날짜를 나타내는 형식에는 각각 월, 일, 연도에 대한 세 개의 정수 필드가 있을 수 있습니다. 단일 메서드의 범위 외부에서 사용되지 않는 변수는 메서드 본문 자체 내에서 지역 변수로 선언되어야 합니다.

필드는 액세스 수준, 유형, 필드 이름을 지정하여 클래스 또는 구조체 블록에서 선언됩니다. 예시:

C#

```
public class CalendarEntry
{
    // private field (Located near wrapping "Date" property).
    private DateTime _date;

    // Public property exposes _date field safely.
    public DateTime Date
    {
        get
        {
            return _date;
        }
    }
}
```

```

    }
    set
    {
        // Set some reasonable boundaries for likely birth dates.
        if (value.Year > 1900 && value.Year <= DateTime.Today.Year)
        {
            _date = value;
        }
        else
        {
            throw new ArgumentOutOfRangeException("Date");
        }
    }
}

// public field (Generally not recommended).
public string? Day;

// Public method also exposes _date field safely.
// Example call: birthday.SetDate("1975, 6, 30");
public void SetDate(string dateString)
{
    DateTime dt = Convert.ToDateTime(dateString);

    // Set some reasonable boundaries for likely birth dates.
    if (dt.Year > 1900 && dt.Year <= DateTime.Today.Year)
    {
        _date = dt;
    }
    else
    {
        throw new ArgumentOutOfRangeException("dateString");
    }
}

public TimeSpan GetTimeSpan(string dateString)
{
    DateTime dt = Convert.ToDateTime(dateString);

    if (dt.Ticks < _date.Ticks)
    {
        return _date - dt;
    }
    else
    {
        throw new ArgumentOutOfRangeException("dateString");
    }
}
}

```

인스턴스의 필드에 액세스하려면 인스턴스 이름 뒤에 마침표를 추가하고 그 뒤에 필드 이름을 추가합니다(예: `instancename._fieldName`). 예시:

C#

```
CalendarEntry birthday = new CalendarEntry();
birthday.Day = "Saturday";
```

필드를 선언할 때 대입 연산자를 사용하여 필드에 초기 값을 지정할 수 있습니다. 예를 들어 `Day` 필드를 `"Monday"`에 자동으로 할당하려면 다음 예제와 같이 `Day`를 선언합니다.

C#

```
public class CalendarDateWithInitialization
{
    public string Day = "Monday";
    //...
}
```

필드는 개체 인스턴스에 대한 생성자를 호출 하기 직전에 초기화됩니다. 생성자가 필드 값을 할당하는 경우 필드 선언 중에 지정된 값을 덮어씁니다. 자세한 내용은 [생성자 사용](#)을 참조하세요.

### ① 참고

필드 이니셜라이저는 다른 인스턴스 필드를 참조할 수 없습니다.

필드는 `public`, `private`, `protected`, `internal`, `protected internal` 또는 `private protected`(으)로 표시할 수 있습니다. 이러한 액세스 한정자는 형식의 사용자가 필드에 액세스하는 방법을 정의합니다. 자세한 내용은 [액세스 한정자](#)를 참조하세요.

필요에 따라 필드는 `static`(으)로 선언할 수 있습니다. 정적 필드는 형식의 인스턴스가 없더라도 언제든지 호출자가 사용할 수 있습니다. 자세한 내용은 [static 클래스 및 static 클래스 멤버](#)를 참조하세요.

필드는 `readonly`(으)로 선언할 수 있습니다. 읽기 전용 필드는 초기화 중이나 생성자에서만 값을 할당할 수 있습니다. `static readonly` 필드는 C# 컴파일러가 런타임에만 컴파일 시간에 정적 읽기 전용 필드의 값에 액세스할 수 없다는 점을 제외하고 상수와 유사합니다. 자세한 내용은 [상수](#)를 참조하세요.

필드는 `required`(으)로 선언할 수 있습니다. 필수 필드는 생성자에 의해 초기화되거나 개체가 생성될 때 [개체 이니셜라이저](#)에 의해 초기화되어야 합니다. 모든 필수 멤버를 초기화하는 생성자 선언에 `System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute` 특성을 추가합니다.

`required` 한정자는 동일한 필드의 `readonly` 한정자와 결합될 수 없습니다. 그러나 속성은 단지 `required` 및 `init` 일 수 있습니다.

C# 12부터 [기본 생성자](#) 매개 변수는 필드를 선언하는 대안입니다. 형식에 초기화 시 제공해야 하는 종속성이 있는 경우 해당 종속성을 제공하는 기본 생성자를 만들 수 있습니다. 이러한 매개 변수는 캡처되어 형식의 선언된 필드 대신 사용될 수 있습니다. `record` 형식의 경우 기본 생성자 매개 변수는 `public` 속성으로 표시됩니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)을 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

## 참고 항목

- [C# 형식 시스템](#)
- [생성자 사용](#)
- [상속](#)
- [액세스 한정자](#)
- [추상/봉인된 클래스 및 클래스 멤버](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.



### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 상수(C# 프로그래밍 가이드)

아티클 • 2023. 11. 15.

상수는 컴파일 시간에 알려진 변경할 수 없는 값입니다. 프로그램 수명 동안 변경하지 마세요. 상수는 `const` 한정자로 선언됩니다. C# 기본 제공 형식 만 .로 `const` 선언할 수 있습니다. 이외의 참조 형식 상수는 `String` null 값으로만 초기화할 수 있습니다. 클래스, 구조체 및 배열을 비롯한 사용자 정의 형식은 `const` 가 될 수 없습니다. `readonly` 한정자를 사용하여 런타임에 한 번 초기화되고(예: 생성자에서) 그때부터는 변경할 수 없는 클래스, 구조체 또는 배열을 만듭니다.

C#에서는 `const` 메서드, 속성 또는 이벤트를 지원하지 않습니다.

열거형 형식을 사용하여 정수 계열 기본 제공 형식(예: `int`, `uint`, `long` 등)에 대한 명명된 상수를 정의할 수 있습니다. 자세한 내용은 `enum`을 참조하세요.

상수는 선언될 때 초기화되어야 합니다. 예시:

C#

```
class Calendar1
{
    public const int Months = 12;
}
```

이 예제에서 `Months` 상수는 항상 12이고 클래스 자체에 의해서도 변경될 수 없습니다. 실제로 컴파일러는 C# 소스 코드에서 상수 식별자를 발견할 경우(예: `Months`) 리터럴 값을 직접 컴파일러에서 생성하는 IL(중간 언어) 코드로 대체합니다. 런타임에 상수와 연결된 변수 주소가 없으므로 `const` 필드는 참조를 통해 전달될 수 없고 식에 l-value로 표시될 수 없습니다.

## ① 참고

DLL과 같이 다른 코드에 정의된 상수 값을 참조할 경우 주의하세요. DLL의 새 버전에서 상수의 새 값을 정의할 경우 프로그램은 새 버전에 대해 다시 컴파일될 때까지 이전 리터럴 값을 포함합니다.

다음과 같이 같은 형식의 여러 상수를 동시에 선언할 수 있습니다.

C#

```
class Calendar2
{
```

```
    public const int Months = 12, Weeks = 52, Days = 365;  
}
```

상수를 초기화하는 데 사용되는 식은 순환 참조를 만들지 않을 경우 다른 상수를 참조할 수 있습니다. 예시:

C#

```
class Calendar3  
{  
    public const int Months = 12;  
    public const int Weeks = 52;  
    public const int Days = 365;  
  
    public const double DaysPerWeek = (double) Days / (double) Weeks;  
    public const double DaysPerMonth = (double) Days / (double) Months;  
}
```

상수는 `public`, `private`, `protected`, `internal`, `protected internal` 또는 `private protected`로 표시될 수 있습니다. 이러한 액세스 한정자는 클래스의 사용자가 상수에 액세스하는 방법을 정의합니다. 자세한 내용은 [액세스 한정자](#)를 참조하세요.

형식의 모든 인스턴스에 대한 상수 값이 같으므로 상수가 `static` 필드인 것처럼 상수에 액세스합니다. 상수를 선언하는 데 `static` 키워드를 사용하지 않습니다. 상수를 정의하는 클래스에 포함되지 않은 식은 상수에 액세스할 때 클래스 이름, 마침표 및 상수 이름을 사용해야 합니다. 예시:

C#

```
int birthstones = Calendar.Months;
```

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)을 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [속성](#)
- [유형](#)
- [readonly](#)
- [Immutability in C# Part One: Kinds of Immutability](#)(C#의 불변성 1부: 불변성 종류)

 GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 설명서 문제 열기

 제품 사용자 의견 제공

# 추상 속성 정의 방법(C# 프로그래밍 가이드)

아티클 • 2024. 03. 12.

다음 예제에서는 `abstract` 속성을 정의하는 방법을 보여 줍니다. 추상 속성 선언은 속성 접근자의 구현을 제공하지 않습니다. 클래스가 속성을 지원하도록 선언하지만 접근자 구현은 파생 클래스에서 처리되도록 합니다. 다음 예제에서는 기본 클래스에서 상속된 추상 속성을 구현하는 방법을 보여 줍니다.

이 샘플은 개별적으로 컴파일된 파일 3개로 구성되었으며, 결과로 생성된 어셈블리는 다음 컴파일 시 참조됩니다.

- `abstractshape.cs`: 추상 `Area` 속성이 포함된 `Shape` 클래스입니다.
- `shapes.cs`: `Shape` 클래스의 서브클래스입니다.
- `shapetest.cs`: 일부 `Shape` 파생 개체의 영역을 표시할 테스트 프로그램입니다.

예제를 컴파일하려면 다음 명령을 사용합니다.

```
csc abstractshape.cs shapes.cs shapetest.cs
```

그러면 `shapetest.exe` 실행 파일이 생성됩니다.

## 예제

이 파일에서는 `double` 형식의 `Area` 속성을 포함하는 `Shape` 클래스를 선언합니다.

```
C#  
  
// compile with: csc -target:library abstractshape.cs  
public abstract class Shape  
{  
    private string name;  
  
    public Shape(string s)  
    {  
        // calling the set accessor of the Id property.  
        Id = s;  
    }  
  
    public string Id  
    {  
        get  
        {
```

```

        return name;
    }

    set
    {
        name = value;
    }
}

// Area is a read-only property - only a get accessor is needed:
public abstract double Area
{
    get;
}

public override string ToString()
{
    return $"{Id} Area = {Area:F2}";
}
}

```

- 속성의 한정자는 속성 선언 자체에 배치됩니다. 예시:

C#

```
public abstract double Area
```

- 추상 속성(이 예제의 `Area`)을 선언할 때는 단순히 사용할 수 있는 속성 접근자를 나타내고 구현하지 않습니다. 이 예제에서는 `get` 접근자만 사용할 수 있으므로 속성은 읽기 전용입니다.

다음 코드에서는 `Shape`의 세 가지 서브클래스와 이러한 서브클래스에서 `Area` 속성을 재정의하여 고유한 구현을 제공하는 방법을 보여 줍니다.

C#

```
// compile with: csc -target:library -reference:abstractshape.dll shapes.cs
public class Square : Shape
{
    private int side;

    public Square(int side, string id)
        : base(id)
    {
        this.side = side;
    }

    public override double Area
    {
        get
    }
}
```

```

        {
            // Given the side, return the area of a square:
            return side * side;
        }
    }

public class Circle : Shape
{
    private int radius;

    public Circle(int radius, string id)
        : base(id)
    {
        this.radius = radius;
    }

    public override double Area
    {
        get
        {
            // Given the radius, return the area of a circle:
            return radius * radius * System.Math.PI;
        }
    }
}

public class Rectangle : Shape
{
    private int width;
    private int height;

    public Rectangle(int width, int height, string id)
        : base(id)
    {
        this.width = width;
        this.height = height;
    }

    public override double Area
    {
        get
        {
            // Given the width and height, return the area of a rectangle:
            return width * height;
        }
    }
}

```

다음 코드에서는 많은 `Shape` 파생 개체를 만들고 해당 영역을 출력하는 테스트 프로그램을 보여 줍니다.

```

// compile with: csc -reference:abstractshape.dll;shapes.dll shapetest.cs
class TestClass
{
    static void Main()
    {
        Shape[] shapes =
        {
            new Square(5, "Square #1"),
            new Circle(3, "Circle #1"),
            new Rectangle( 4, 5, "Rectangle #1")
        };

        System.Console.WriteLine("Shapes Collection");
        foreach (Shape s in shapes)
        {
            System.Console.WriteLine(s);
        }
    }
}
/* Output:
   Shapes Collection
   Square #1 Area = 25.00
   Circle #1 Area = 28.27
   Rectangle #1 Area = 20.00
*/

```

## 참고 항목

- C# 형식 시스템
- 추상/봉인된 클래스 및 클래스 멤버
- 속성

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.



### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# C#에서 상수 정의 방법

아티클 • 2023. 04. 07.

상수는 해당 값이 컴파일 시간에 설정되며 변경할 수 없는 필드입니다. 상수를 사용하여 특수 값에 대해 숫자 리터럴("매직 넘버") 대신 의미 있는 이름을 제공할 수 있습니다.

## ① 참고

C#에서는 `#define` 전처리기 지시문을 사용하여 일반적으로 C와 C++에서 사용되는 방식으로 상수를 정의할 수 없습니다.

정수 형식(`int`, `byte` 등)의 상수 값을 정의하려면 열거 형식을 사용합니다. 자세한 내용은 [enum](#)을 참조하세요.

정수가 아닌 상수를 정의하는 한 가지 방법은 `Constants`라는 단일 정적 클래스로 그룹화하는 것입니다. 이 경우 다음 예제와 같이 상수에 대한 모든 참조 앞에 클래스 이름이 와야 합니다.

## 예제

C#

```
static class Constants
{
    public const double Pi = 3.14159;
    public const int SpeedOfLight = 300000; // km per sec.
}

class Program
{
    static void Main()
    {
        double radius = 5.3;
        double area = Constants.Pi * (radius * radius);
        int secsFromSun = 149476000 / Constants.SpeedOfLight; // in km
        Console.WriteLine(secsFromSun);
    }
}
```

클래스 이름 한정자를 통해 사용자와 상수를 사용하는 다른 사용자가 상수이며 수정할 수 없음을 쉽게 파악할 수 있습니다.

## 참조

- C# 형식 시스템

# 속성(C# 프로그래밍 가이드)

아티클 • 2024. 03. 15.

속성은 전용 필드의 값을 읽거나 쓰거나 계산하는 유연한 메커니즘을 제공하는 멤버입니다. 속성은 공용 데이터 멤버인 것처럼 사용할 수 있지만 접근자라는 특수 메서드입니다. 이 기능을 사용하면 데이터에 쉽게 액세스할 수 있으며 분석법의 안전성과 유연성을 높이는 데 도움이 됩니다.

## 속성 개요

- 속성을 사용하면 클래스가 구현 또는 검증 코드를 숨기는 동시에 값을 가져오고 설정하는 방법을 공개적으로 노출할 수 있습니다.
- `get` 속성 접근자는 속성 값을 반환하는데 사용되고 `set` 속성 접근자는 새 값을 할당하는데 사용됩니다. `init` 속성 접근자는 개체 생성 중에만 새 값을 할당하는데 사용됩니다. 이러한 접근자는 각기 다른 액세스 수준을 가질 수 있습니다. 자세한 내용은 [접근자 액세스 가능성 제한](#)을 참조하세요.
- 키워드(keyword) `값`은 `or` `init` 접근자가 할당하는 값을 `set` 정의하는 데 사용됩니다.
- 속성은 `읽기/쓰기`(`get` 및 `set` 접근자 모두 포함), `읽기 전용`(`get` 접근자는 포함하지만 `set` 접근자는 포함 안 함) 또는 `쓰기 전용`(`set` 접근자는 포함하지만 `get` 접근자는 포함 안 함)일 수 있습니다. 쓰기 전용 속성은 거의 없으며 주로 중요한 데이터에 대한 액세스를 제한하는 데 사용됩니다.
- 사용자 지정 접근자 코드가 필요 없는 단순한 속성은 식 본문 정의나 [자동 구현 속성](#)으로 구현할 수 있습니다.

## 지원 필드가 있는 속성

속성을 구현하는 한 가지 기본 패턴에는 `private` 지원 필드를 사용하여 속성 값을 설정 및 검색하는 작업이 포함됩니다. `get` 접근자는 `private` 필드의 값을 반환하고 `set` 접근자는 `private` 필드에 값을 할당하기 전에 데이터 유효성 검사를 수행할 수 있습니다. 또한 두 접근자 모두 데이터를 저장 또는 반환하기 전에 데이터에 대한 변환이나 계산을 수행할 수도 있습니다.

다음 예제에서 이 방법을 보여 줍니다. 이 예제에서 `TimePeriod` 클래스는 시간 간격을 나타냅니다. 내부적으로 이 클래스는 `_seconds`라는 `private` 필드에 시간 간격을 초 단위로 저장합니다. `Hours`라는 읽기/쓰기 속성을 사용하면 고객이 시간 간격을 시간 단위로 정할 수 있습니다. `get` 및 `set` 접근자 모두 필요에 따라 시간 및 초 간의 변환을 수행합니다. 또한 `set` 접근자는 데이터의 유효성을 검사하고 시간(시)이 잘못된 경우 `ArgumentOutOfRangeException`을 throw합니다.

C#

```
public class TimePeriod
{
    private double _seconds;

    public double Hours
    {
        get { return _seconds / 3600; }
        set
        {
            if (value < 0 || value > 24)
                throw new ArgumentOutOfRangeException(nameof(value),
                    "The valid range is between 0 and 24.");

            _seconds = value * 3600;
        }
    }
}
```

다음 예에 표시된 대로 속성에 액세스하여 값을 가져오고 설정할 수 있습니다.

C#

```
TimePeriod t = new TimePeriod();
// The property assignment causes the 'set' accessor to be called.
t.Hours = 24;

// Retrieving the property causes the 'get' accessor to be called.
Console.WriteLine($"Time in hours: {t.Hours}");
// The example displays the following output:
//      Time in hours: 24
```

## 식 본문 정의

속성 접근자는 식의 결과를 할당하거나 반환하기만 하는 한 줄로 된 문으로 구성되는 경우가 많습니다. 이러한 속성은 식 본문 멤버로 구현할 수 있습니다. 식 본문 정의는 => 기호와 속성에 할당하거나 속성에서 검색할 식으로 구성됩니다.

읽기 전용 속성은 `get` 접근자를 식 본문 멤버로 구현할 수 있습니다. 이 경우 `get` 접근자 키워드나 `return` 키워드를 모두 사용하지 않습니다. 다음 예제에서는 읽기 전용 `Name` 속성을 식 본문 멤버로 구현합니다.

C#

```
public class Person
{
```

```

private string _firstName;
private string _lastName;

public Person(string first, string last)
{
    _firstName = first;
    _lastName = last;
}

public string Name => $"{_firstName} {_lastName}";
}

```

`get` 및 `set` 접근자는 모두 식 본문 멤버로 구현될 수 있습니다. 이 경우 `get` 및 `set` 키워드가 있어야 합니다. 다음 예제에서는 두 접근자에 대해 식 본문 정의를 사용하는 방법을 보여 줍니다. `return` 키워드는 `get` 접근자와 함께 사용되지 않습니다.

C#

```

public class SaleItem
{
    string _name;
    decimal _cost;

    public SaleItem(string name, decimal cost)
    {
        _name = name;
        _cost = cost;
    }

    public string Name
    {
        get => _name;
        set => _name = value;
    }

    public decimal Price
    {
        get => _cost;
        set => _cost = value;
    }
}

```

## 자동 구현 속성

경우에 따라 속성 `get` 및 `set` 접근자는 추가 논리를 포함하지 않고 지원 필드에 값을 할당하거나 해당 필드에서 값을 검색합니다. 자동 구현 속성을 사용하면 코드를 간소화할 수 있을 뿐 아니라 C# 컴파일러에서 지원 필드를 투명하게 제공하도록 할 수 있습니다.

속성에 `get` 및 `set`(또는 `get` 및 `init`) 접근자가 모두 포함된 경우 두 접근자를 모두 자동 구현해야 합니다. 구현을 제공하지 않고 `get` 및 `set` 키워드를 사용하여 자동 구현 속성을 정의합니다. 다음 예제에서는 `Name` 및 `Price`가 자동 구현 속성인 점만 제외하고 이전 예제와 동일합니다. 이 예제에서는 매개 변수화된 생성자도 제거하므로 이제 `SaleItem` 개체가 매개 변수 없는 생성자 및 [개체 이니셜라이저](#)에 대한 호출을 통해 초기화됩니다.

C#

```
public class SaleItem
{
    public string Name
    { get; set; }

    public decimal Price
    { get; set; }
}
```

자동 구현 속성은 `get` 및 `set` 접근자에 대해 서로 다른 접근성을 선언할 수 있습니다. 일반적으로 공용 `get` 접근자와 프라이빗 `set` 접근자를 선언합니다. 자세한 내용은 [접근자 액세스 가능성 제한](#)에 대한 문서를 참조하세요.

## 필수 속성

C# 11부터 `required` 멤버를 추가하여 클라이언트 코드가 속성이나 필드를 초기화하도록 할 수 있습니다.

C#

```
public class SaleItem
{
    public required string Name
    { get; set; }

    public required decimal Price
    { get; set; }
}
```

`SaleItem`을 만들려면 다음 코드에 표시된 대로 [개체 이니셜라이저](#)를 사용하여 `Name` 및 `Price` 속성을 모두 설정해야 합니다.

C#

```
var item = new SaleItem { Name = "Shoes", Price = 19.95m };
Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
```

# 관련 단원

- 속성 사용
- 인터페이스 속성
- 속성 및 인덱서 비교
- 접근자 접근성 제한
- 자동으로 구현된 속성

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 속성을](#) 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

## 참고 항목

- [인덱서](#)
- [init 키워드](#)
- [get 키워드](#)
- [set 키워드](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# 속성 사용(C# 프로그래밍 가이드)

아티클 • 2024. 05. 27.

속성은 필드 및 메서드 모두의 측면을 결합합니다. 개체의 사용자에게 속성은 필드로 표시되며, 속성에 액세스하려면 동일한 구문이 필요합니다. 클래스의 구현자에 속성은 `get` 접근자 및/또는 `set` 또는 `init` 접근자를 나타내는 하나 또는 두 개의 코드 블록입니다. `get` 접근자에 대한 코드 블록은 속성을 읽을 때 실행됩니다. 속성에 값이 할당되면 `set` 또는 `init` 접근자에 대한 코드 블록이 실행됩니다. `set` 접근자가 없는 속성은 읽기 전용으로 간주됩니다. `get` 접근자가 없는 속성은 쓰기 전용으로 간주됩니다. 두 접근자가 모두 있는 속성은 읽기/쓰기입니다. `set` 접근자 대신 `init` 접근자를 사용하여 속성을 개체 초기화의 일부로 설정할 수 있지만 그렇지 않으면 읽기 전용으로 설정할 수 있습니다.

필드와 달리 속성은 변수로 분류되지 않습니다. 따라서 속성을 `ref` 또는 `out` 매개 변수로 전달할 수 없습니다.

속성에는 다음과 같은 여러 가지 용도가 있습니다.

- 변경을 허용하기 전에 데이터의 유효성을 검사할 수 있습니다.
- 데이터베이스와 같은 다른 원본에서 해당 데이터가 검색되는 클래스에 데이터를 투명하게 노출할 수 있습니다.
- 이벤트를 발생하거나 다른 필드의 값을 변경하는 등 데이터가 변경될 때 작업을 수행할 수 있습니다.

속성은 필드의 액세스 수준, 속성 형식, 속성 이름, `get` 접근자 및/또는 `set` 접근자를 선언하는 코드 블록을 차례로 지정하여 클래스 블록에서 선언됩니다. 예시:

```
C#  
  
public class Date  
{  
    private int _month = 7; // Backing store  
  
    public int Month  
    {  
        get => _month;  
        set  
        {  
            if ((value > 0) && (value < 13))  
            {  
                _month = value;  
            }  
        }  
    }  
}
```

이 예제에서 Month는 속성으로 선언되었으므로, set 접근자를 통해 Month 값이 1에서 12 사이로 설정되도록 할 수 있습니다. Month 속성은 전용 필드를 사용하여 실제 값을 추적합니다. 속성 데이터의 실제 위치를 속성의 “백업 저장소”라고도 합니다. 일반적으로 속성은 전용 필드를 백업 저장소로 사용합니다. 속성 호출을 통해서만 필드를 변경할 수 있도록 하기 위해 필드는 private로 표시되었습니다. 공용 및 개인 액세스 제한에 대한 자세한 내용은 [액세스 한정자](#)를 참조하세요. 자동 구현 속성은 간단한 속성 선언을 위해 간소화된 구문을 제공합니다. 자세한 내용은 [자동으로 구현된 속성을 참조하세요](#).

## get 접근자

get 접근자 본문은 메서드 본문과 유사합니다. 속성 형식의 값을 반환해야 합니다. C# 컴파일러 및 JIT(Just-In-Time) 컴파일러는 get 접근자를 구현하기 위한 일반적인 패턴을 검색하고 이러한 패턴을 최적화합니다. 예를 들어 계산을 수행하지 않고 필드를 반환하는 get 접근자가 해당 필드의 메모리 읽기에 최적화될 수 있습니다. 자동 구현 속성은 이 패턴을 따르며 이러한 최적화의 이점을 누릴 수 있습니다. 그러나 컴파일러가 실제로 런타임에 호출될 수 있는 메서드를 컴파일 시간에 알지 못하기 때문에 가상 get 접근자 메서드를 인라인화할 수 없습니다. 다음 예에서는 프라이빗 필드 \_name의 값을 반환하는 get 접근자를 보여 줍니다.

C#

```
class Employee
{
    private string _name; // the name field
    public string Name => _name; // the Name property
}
```

할당 대상을 제외하고 속성을 참조하는 경우 속성 값을 읽기 위해 get 접근자가 호출됩니다. 예시:

C#

```
var employee= new Employee();
//...

System.Console.WriteLine(employee.Name); // the get accessor is invoked here
```

get 접근자는 return 또는 throw 문으로 끝나야 하며, 제어가 접근자 본문을 벗어날 수 없습니다.

⚠ 경고

`get` 접근자를 사용하여 개체의 상태를 변경하는 것은 잘못된 프로그래밍 스타일입니다.

`get` 접근자를 사용하여 필드 값을 반환하거나 계산한 후 반환할 수 있습니다. 예시:

C#

```
class Manager
{
    private string _name;
    public string Name => _name != null ? _name : "NA";
}
```

이전 예제에서는 `Name` 속성에 값을 할당하지 않으면 `NA` 값을 반환합니다.

## set 접근자

`set` 접근자는 반환 형식이 `void`인 메서드와 비슷합니다. 형식이 속성의 형식인 `value`라는 암시적 매개 변수를 사용합니다. 컴파일러 및 JIT 컴파일러는 `set` 또는 `init` 접근자에 대한 일반적인 패턴도 인식합니다. 이러한 일반적인 패턴은 지원 필드에 대한 메모리를 직접 작성하여 최적화됩니다. 다음 예제에서는 `set` 접근자가 `Name` 속성에 추가됩니다.

C#

```
class Student
{
    private string _name; // the name field
    public string Name // the Name property
    {
        get => _name;
        set => _name = value;
    }
}
```

속성에 값을 할당하는 경우 새 값을 제공하는 인수를 사용하여 `set` 접근자가 호출됩니다. 예시:

C#

```
var student = new Student();
student.Name = "Joe"; // the set accessor is invoked here

System.Console.Write(student.Name); // the get accessor is invoked here
```

`set` 접근자의 지역 변수 선언에 대해 암시적 매개 변수 이름 `value`를 사용하면 오류가 발생합니다.

## Init 접근자

`init` 접근자를 만드는 코드는 `set` 대신 `init` 키워드를 사용한다는 점을 제외하면 `set` 접근자를 만드는 코드와 같습니다. 차이점은 `init` 접근자는 생성자 또는 `object-initializer`를 통해서만 사용할 수 있다는 것입니다.

## 설명

속성은 `public`, `private`, `protected`, `internal`, `protected internal` 또는 `private protected`로 표시될 수 있습니다. 이러한 액세스 한정자는 클래스 사용자가 속성에 액세스하는 방법을 정의합니다. 동일한 속성에 대한 `get` 및 `set` 접근자는 다른 액세스 한정자를 가질 수 있습니다. 예를 들어 `get(이)`가 `public(을)`를 형식 외부에서 읽기 전용 액세스를 허용하도록 할 수 있으며 `set(은)`는 `private` 또는 `protected`일 수 있습니다. 자세한 내용은 [액세스 한정자](#)를 참조하세요.

`static` 키워드를 사용하여 속성을 정적 속성으로 선언할 수 있습니다. 클래스의 인스턴스가 없더라도 호출자는 언제든지 정적 속성을 사용할 수 있습니다. 자세한 내용은 [static 클래스 및 static 클래스 멤버](#)를 참조하세요.

`가상` 키워드를 사용하여 속성을 가상 속성으로 표시할 수 있습니다. 가상 속성을 사용하면 파생 클래스가 `override` 키워드를 사용하여 속성 동작을 재정의할 수 있습니다. 이러한 옵션에 대한 자세한 내용은 [상속](#)을 참조하세요.

가상 속성을 재정의하는 속성이 `sealed`일 수도 있으며, 파생 클래스에 대해 더 이상 가상이 아니도록 지정합니다. 마지막으로, 속성을 `abstract`로 선언할 수 있습니다. 추상 속성은 클래스의 구현을 정의하지 않으며 파생 클래스는 자체 구현을 작성해야 합니다. 이러한 옵션에 대한 자세한 내용은 [추상 및 봉인 클래스와 클래스 멤버](#)를 참조하세요.

### ① 참고

`static` 속성의 접근자에 `virtual`, `abstract` 또는 `override` 한정자를 사용하면 오류가 발생합니다.

## 예제

이 예제에서는 인스턴스, 정적 및 읽기 전용 속성을 보여 줍니다. 키보드에서 직원 이름을 받고 `NumberOfEmployees`를 1만큼 증가한 다음 직원 이름과 번호를 표시합니다.

C#

```
public class Employee
{
    public static int NumberOfEmployees;
    private static int _counter;
    private string _name;

    // A read-write instance property:
    public string Name
    {
        get => _name;
        set => _name = value;
    }

    // A read-only static property:
    public static int Counter => _counter;

    // A Constructor:
    public Employee() => _counter = ++NumberOfEmployees; // Calculate the
    employee's number:
}
```

## 숨김 속성 예제

이 예제에서는 파생 클래스에서 이름이 같은 다른 속성에 의해 숨겨진 기본 클래스의 속성에 액세스하는 방법을 보여 줍니다.

C#

```
public class Employee
{
    private string _name;
    public string Name
    {
        get => _name;
        set => _name = value;
    }
}

public class Manager : Employee
{
    private string _name;

    // Notice the use of the new modifier:
    public new string Name
    {
```

```

        get => _name;
        set => _name = value + ", Manager";
    }
}

class TestHiding
{
    public static void Test()
    {
        Manager m1 = new Manager();

        // Derived class property.
        m1.Name = "John";

        // Base class property.
        ((Employee)m1).Name = "Mary";

        System.Console.WriteLine("Name in the derived class is: {0}",
m1.Name);
        System.Console.WriteLine("Name in the base class is: {0}",
((Employee)m1).Name);
    }
}
/* Output:
   Name in the derived class is: John, Manager
   Name in the base class is: Mary
*/

```

다음은 앞의 예제에서 중요한 사항입니다.

- 파생 클래스의 `Name` 속성은 기본 클래스의 `Name` 속성을 숨깁니다. 이러한 경우 `new` 한정자는 파생 클래스의 속성 선언에 사용됩니다.

C#

```
public new string Name
```

- `(Employee)` 캐스트는 기본 클래스의 숨겨진 속성에 액세스하는 데 사용됩니다.

C#

```
((Employee)m1).Name = "Mary";
```

멤버를 숨기는 방법에 대한 자세한 내용은 [new 한정자](#)를 참조하세요.

## 재정의 속성 예제

이 예제에서 두 클래스 `Cube` 및 `Square`는 추상 클래스 `Shape`를 구현하고 해당 `abstract Area` 속성을 재정의합니다. 속성의 `override` 한정자를 사용합니다. 프로그램은 변을 입력으로 사용하고 사각형과 정육면체의 면적을 계산합니다. 또한 면적을 입력으로 사용하고 사각형 및 정육면체의 해당 변을 계산합니다.

C#

```
abstract class Shape
{
    public abstract double Area
    {
        get;
        set;
    }
}

class Square : Shape
{
    public double side;

    //constructor
    public Square(double s) => side = s;

    public override double Area
    {
        get => side * side;
        set => side = System.Math.Sqrt(value);
    }
}

class Cube : Shape
{
    public double side;

    //constructor
    public Cube(double s) => side = s;

    public override double Area
    {
        get => 6 * side * side;
        set => side = System.Math.Sqrt(value / 6);
    }
}

class TestShapes
{
    static void Main()
    {
        // Input the side:
        System.Console.Write("Enter the side: ");
        double side = double.Parse(System.Console.ReadLine());

        // Compute the areas:
```

```

        Square s = new Square(side);
        Cube c = new Cube(side);

        // Display the results:
        System.Console.WriteLine("Area of the square = {0:F2}", s.Area);
        System.Console.WriteLine("Area of the cube = {0:F2}", c.Area);
        System.Console.WriteLine();

        // Input the area:
        System.Console.Write("Enter the area: ");
        double area = double.Parse(System.Console.ReadLine());

        // Compute the sides:
        s.Area = area;
        c.Area = area;

        // Display the results:
        System.Console.WriteLine("Side of the square = {0:F2}", s.side);
        System.Console.WriteLine("Side of the cube = {0:F2}", c.side);
    }
}

/* Example Output:
   Enter the side: 4
   Area of the square = 16.00
   Area of the cube = 96.00

   Enter the area: 24
   Side of the square = 4.90
   Side of the cube = 2.00
*/

```

## 참고 항목

- 속성
- 인터페이스 속성
- 자동으로 구현된 속성

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.



### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 인터페이스 속성(C# 프로그래밍 가이드)

아티클 • 2023. 04. 07.

[interface](#)에 속성을 선언할 수 있습니다. 다음 예제에서는 인터페이스 속성 접근자를 선언합니다.

C#

```
public interface ISampleInterface
{
    // Property declaration:
    string Name
    {
        get;
        set;
    }
}
```

일반적으로 인터페이스 속성에는 본문이 없습니다. 접근자는 속성이 읽기/쓰기인지, 읽기 전용인지, 쓰기 전용인지를 나타냅니다. 클래스 및 구조체와 달리, 접근자를 본문 없이 선언해도 [자동 구현 속성](#)이 선언되지 않습니다. 인터페이스는 멤버에 대한 기본 구현(속성 포함)을 정의할 수 있습니다. 인터페이스는 인스턴스 데이터 필드를 정의할 수 없으므로 인터페이스에서 속성에 대한 기본 구현을 정의하는 것은 드문 일입니다.

## 예제

이 예제에서 `IEmployee` 인터페이스에는 읽기/쓰기 속성 `Name`과 읽기 전용 속성 `Counter`가 있습니다. `Employee` 클래스는 `IEmployee` 인터페이스를 구현하고 이러한 두 속성을 사용합니다. 프로그램은 새 직원의 이름과 현재 직원 수를 읽고 직원 이름과 계산된 직원 수를 표시합니다.

멤버가 선언된 인터페이스를 참조하는 속성의 정규화된 이름을 사용할 수 있습니다. 예를 들어:

C#

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

앞의 예제에서는 명시적 인터페이스 구현을 보여 주었습니다. 예를 들어 Employee 클래스가 두 인터페이스 ICitizen 및 IEmployee를 구현하고 두 인터페이스에 모두 Name 속성이 있으면 명시적 인터페이스 멤버 구현이 필요합니다. 즉, 다음과 같은 속성 선언이 있다고 가정합니다.

C#

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

이 선언은 IEmployee 인터페이스의 Name 속성을 구현합니다. 또한 다음과 같은 선언이 있다고 가정합니다.

C#

```
string ICitizen.Name
{
    get { return "Citizen Name"; }
    set { }
}
```

이 선언은 ICitizen 인터페이스의 Name 속성을 구현합니다.

C#

```
interface IEmployee
{
    string Name
    {
        get;
        set;
    }

    int Counter
    {
        get;
    }
}

public class Employee : IEmployee
{
    public static int numberOfEmployees;

    private string _name;
    public string Name // read-write instance property
    {
```

```

        get => _name;
        set => _name = value;
    }

    private int _counter;
    public int Counter // read-only instance property
    {
        get => _counter;
    }

    // constructor
    public Employee() => _counter = ++numberOfEmployees;
}

```

C#

```

System.Console.Write("Enter number of employees: ");
Employee.numberOfEmployees = int.Parse(System.Console.ReadLine());

Employee e1 = new Employee();
System.Console.Write("Enter the name of the new employee: ");
e1.Name = System.Console.ReadLine();

System.Console.WriteLine("The employee information:");
System.Console.WriteLine("Employee number: {0}", e1.Counter);
System.Console.WriteLine("Employee name: {0}", e1.Name);

```

## 샘플 출력

콘솔

```

Enter number of employees: 210
Enter the name of the new employee: Hazem Abolrous
The employee information:
Employee number: 211
Employee name: Hazem Abolrous

```

## 참고 항목

- C# 프로그래밍 가이드
- 속성
- 속성 사용
- 속성 및 인덱서 비교
- 인덱서
- 인터페이스

# 접근자 액세스 가능성 제한(C# 프로그래밍 가이드)

아티클 • 2024. 03. 09.

속성 또는 인덱서의 `get` 및 `set` 부분을 접근자라고 합니다. 기본적으로 이러한 접근자는 속하는 속성 또는 인덱서와 동일한 표시 유형 또는 액세스 수준을 갖습니다. 자세한 내용은 [접근성 수준](#)을 참조하세요. 그러나 이러한 접근자 중 하나에 대한 액세스를 제한하는 것이 유용한 경우도 있습니다. 일반적으로 `get` 접근자의 공용 액세스를 유지하면서 `set` 접근자의 접근성을 제한합니다. 예시:

```
C#  
  
private string _name = "Hello";  
  
public string Name  
{  
    get  
    {  
        return _name;  
    }  
    protected set  
    {  
        _name = value;  
    }  
}
```

이 예제에서는 `Name` 속성이 `get` 및 `set` 접근자를 정의합니다. `get` 접근자는 속성 자체의 접근성 수준(이 경우 `public`)을 받는 반면, `set` 접근자는 접근자 자체에 `protected` 액세스 한정자를 적용하여 명시적으로 제한됩니다.

## ① 참고

이 문서의 예에서는 자동 구현 속성을 사용하지 않습니다. 자동 구현 속성은 사용자 지정 지원 필드가 필요하지 않은 경우 속성 선언을 위한 간결한 구문을 제공합니다.

## 접근자의 액세스 한정자에 대한 제한 사항

속성 또는 인덱서의 접근자 한정자 사용에는 다음과 같은 조건이 적용됩니다.

- 인터페이스 또는 명시적 `interface` 멤버 구현에는 접근자 한정자를 사용할 수 없습니다.

- 속성 또는 인덱서에 `set` 및 `get` 접근자가 모두 포함된 경우에만 접근자 한정자를 사용할 수 있습니다. 이 경우 두 접근자 중 하나에서만 한정자가 허용됩니다.
- 속성 또는 인덱서에 `override` 한정자가 있는 경우 접근자 한정자가 재정의된 접근자의 한정자와 일치해야 합니다(있는 경우).
- 접근자의 접근성 수준은 속성 또는 인덱서 자체의 접근성 수준보다 더 제한적이어야 합니다.

## 재정의 접근자의 액세스 한정자

속성 또는 인덱서를 재정의하는 경우 재정의 코드에서 재정의된 접근자에 액세스할 수 있어야 합니다. 또한 속성/인덱서 및 접근자의 접근성이 재정의된 속성/인덱서 및 해당 접근자와 일치해야 합니다. 예시:

C#

```
public class Parent
{
    public virtual int TestProperty
    {
        // Notice the accessor accessibility level.
        protected set { }

        // No access modifier is used here.
        get { return 0; }
    }
}

public class Kid : Parent
{
    public override int TestProperty
    {
        // Use the same accessibility level as in the overridden accessor.
        protected set { }

        // Cannot use access modifier here.
        get { return 0; }
    }
}
```

## 인터페이스 구현

접근자를 사용하여 인터페이스를 구현하는 경우 접근자에 액세스 한정자가 있을 수 없습니다. 그러나 `get` 등의 한 접근자를 사용하여 인터페이스를 구현하는 경우 다음 예제와 같이 다른 접근자에 액세스 한정자를 사용할 수 있습니다.

C#

```

public interface ISomeInterface
{
    int TestProperty
    {
        // No access modifier allowed here
        // because this is an interface.
        get;
    }
}

public class TestClass : ISomeInterface
{
    public int TestProperty
    {
        // Cannot use access modifier here because
        // this is an interface implementation.
        get { return 10; }

        // Interface property does not have set accessor,
        // so access modifier is allowed.
        protected set { }
    }
}

```

## 접근자 접근성 도메인

접근자의 액세스 한정자를 사용하는 경우 접근자의 접근성 도메인은 이 한정자에 의해 결정됩니다.

접근자의 액세스 한정자를 사용하지 않은 경우 접근자의 접근성 도메인은 속성 또는 인덱서의 접근성 수준에 의해 결정됩니다.

## 예시

다음 예제에는 세 가지 클래스 `BaseClass`, `DerivedClass`, `MainClass`가 포함되어 있습니다. `BaseClass`에는 두 클래스의 `Name` 및 `Id`인 두 가지 속성이 있습니다. 예제에서는 `protected`, `private` 등의 제한적인 액세스 한정자를 사용할 때 `DerivedClass`의 `Id` 속성을 `BaseClass`의 `Id` 속성으로 숨길 수 있는 방법을 보여 줍니다. 따라서 이 속성에 값을 할당하면 `BaseClass` 클래스의 속성이 대신 호출됩니다. 액세스 한정자를 `public`으로 바꾸면 속성에 액세스할 수 있습니다.

또한 이 예에서는 `DerivedClass`에 있는 `Name` 속성의 `set` 접근자에 있는 `private` 또는 `protected`와 같은 제한적인 액세스 한정자가 파생 클래스의 접근자에 대한 액세스를 방지한다는 것을 보여 줍니다. 이를 할당하거나 액세스 가능한 경우 동일한 이름의 기본 클래스 속성에 액세스하면 오류가 발생합니다.

C#

```
public class BaseClass
{
    private string _name = "Name-BaseClass";
    private string _id = "ID-BaseClass";

    public string Name
    {
        get { return _name; }
        set { }
    }

    public string Id
    {
        get { return _id; }
        set { }
    }
}

public class DerivedClass : BaseClass
{
    private string _name = "Name-DerivedClass";
    private string _id = "ID-DerivedClass";

    new public string Name
    {
        get
        {
            return _name;
        }

        // Using "protected" would make the set accessor not accessible.
        set
        {
            _name = value;
        }
    }

    // Using private on the following property hides it in the Main Class.
    // Any assignment to the property will use Id in BaseClass.
    new private string Id
    {
        get
        {
            return _id;
        }
        set
        {
            _id = value;
        }
    }
}
```

```

class MainClass
{
    static void Main()
    {
        BaseClass b1 = new BaseClass();
        DerivedClass d1 = new DerivedClass();

        b1.Name = "Mary";
        d1.Name = "John";

        b1.Id = "Mary123";
        d1.Id = "John123"; // The BaseClass.Id property is called.

        System.Console.WriteLine("Base: {0}, {1}", b1.Name, b1.Id);
        System.Console.WriteLine("Derived: {0}, {1}", d1.Name, d1.Id);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
   Base: Name-BaseClass, ID-BaseClass
   Derived: John, ID-BaseClass
*/

```

## 설명

`new private string Id` 선언을 `new public string Id`로 바꿀 경우 다음과 같이 출력됩니다.

Name and ID in the base class: Name-BaseClass, ID-BaseClass  
 Name and ID in the derived class: John, John123

## 참고 항목

- 속성
- 인덱서
- 액세스 한정자
- 초기화 전용 속성
- 필수 속성

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 읽기/쓰기 속성 선언 및 사용 방법(C# 프로그래밍 가이드)

아티클 • 2024. 03. 09.

속성은 개체 데이터에 대한 액세스가 보호, 제어, 확인되지 않을 위험 없이 공용 데이터 멤버의 편리함을 제공합니다. 속성은 기본 데이터 멤버에서 값을 할당하고 검색하는 특수 메서드인 접근자를 선언합니다. `set` 접근자를 통해 데이터 멤버를 할당할 수 있으며, `get` 접근자는 데이터 멤버 값을 검색합니다.

이 샘플에서는 `Name(string)` 및 `Age(int)`의 두 속성이 있는 `Person` 클래스를 보여 줍니다. 두 속성 모두 `get` 및 `set` 접근자를 제공하므로 읽기/쓰기 속성으로 간주됩니다.

## 예시

C#

```
class Person
{
    private string _name = "N/A";
    private int _age = 0;

    // Declare a Name property of type string:
    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }

    // Declare an Age property of type int:
    public int Age
    {
        get
        {
            return _age;
        }

        set
        {
            _age = value;
        }
    }
}
```

```
public override string ToString()
{
    return "Name = " + Name + ", Age = " + Age;
}
}

public class Wrapper
{
    private string _name = "N/A";
    public string Name
    {
        get
        {
            return _name;
        }
        private set
        {
            _name = value;
        }
    }
}

class TestPerson
{
    static void Main()
    {
        // Create a new Person object:
        Person person = new Person();

        // Print out the name and the age associated with the person:
        Console.WriteLine("Person details - {0}", person);

        // Set some values on the person object:
        person.Name = "Joe";
        person.Age = 99;
        Console.WriteLine("Person details - {0}", person);

        // Increment the Age property:
        person.Age += 1;
        Console.WriteLine("Person details - {0}", person);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   Person details - Name = N/A, Age = 0
   Person details - Name = Joe, Age = 99
   Person details - Name = Joe, Age = 100
*/
```

# 강력한 프로그래밍

이전 예제에서 `Name` 및 `Age` 속성은 `public`이며, `get` 및 `set` 접근자를 모두 포함합니다. 공용 접근자를 사용하면 모든 개체가 이러한 속성을 읽고 쓸 수 있습니다. 그러나 때로는 접근자 중 하나를 제외하는 것이 좋습니다. 속성을 읽기 전용으로 만들려면 `set` 접근자를 생략할 수 있습니다.

C#

```
public string Name
{
    get
    {
        return _name;
    }
    private set
    {
        _name = value;
    }
}
```

또는 하나의 접근자를 공개적으로 노출하고 다른 접근자를 `private` 또는 `protected`로 설정할 수 있습니다. 자세한 내용은 [비대칭 접근자 접근성](#)을 참조하세요.

속성이 선언되면 클래스의 필드로 사용할 수 있습니다. 속성은 다음 문과 같이 속성 값을 가져오고 설정할 때 자연스러운 구문을 허용합니다.

C#

```
person.Name = "Joe";
person.Age = 99;
```

속성 `set` 메서드에서는 특수한 `value` 변수를 사용할 수 있습니다. 이 변수에는 사용자가 지정한 값이 포함됩니다. 예를 들면 다음과 같습니다.

C#

```
_name = value;
```

`Person` 개체의 `Age` 속성을 증가하기 위한 정리된 구문은 다음과 같습니다.

C#

```
person.Age += 1;
```

개별 `set` 및 `get` 메서드를 사용하여 속성을 모델링한 경우 동등한 코드가 다음과 같이 표시될 수 있습니다.

C#

```
person.SetAge(person.GetAge() + 1);
```

다음 예제에서는 `ToString` 메서드가 재정의되었습니다.

C#

```
public override string ToString()
{
    return "Name = " + Name + ", Age = " + Age;
}
```

`ToString`은 프로그램에서 명시적으로 사용되지 않습니다. 기본적으로 `WriteLine` 호출에 의해 호출됩니다.

## 참고 항목

- 속성
- C# 형식 시스템

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

[설명서 문제 열기](#)

[제품 사용자 의견 제공](#)

# 자동으로 구현된 속성(C# 프로그래밍 가이드)

아티클 • 2024. 03. 10.

자동 구현 속성을 사용하면 속성 접근자에 추가 논리가 필요하지 않을 때 속성 선언이 더 간결해집니다. 이를 통해 클라이언트 코드에서 개체를 만들 수도 있습니다. 다음 예제와 같이 속성을 선언할 때 컴파일러는 속성의 `get` 및 `set` 접근자를 통해서만 액세스할 수 있는 전용 익명 지원 필드를 만듭니다. `init` 접근자는 자동 구현 속성으로 선언될 수도 있습니다.

## 예시

다음 예제에서는 일부 자동 구현 속성이 있는 간단한 클래스를 보여 줍니다.

C#

```
// This class is mutable. Its data can be modified from
// outside the class.
public class Customer
{
    // Auto-implemented properties for trivial get and set
    public double TotalPurchases { get; set; }
    public string Name { get; set; }
    public int CustomerId { get; set; }

    // Constructor
    public Customer(double purchases, string name, int id)
    {
        TotalPurchases = purchases;
        Name = name;
        CustomerId = id;
    }

    // Methods
    public string GetContactInfo() { return "ContactInfo"; }
    public string GetTransactionHistory() { return "History"; }

    // .. Additional methods, events, etc.
}

class Program
{
    static void Main()
    {
        // Initialize a new object.
        Customer cust1 = new Customer(4987.63, "Northwind", 90108);
```

```
// Modify a property.  
    cust1.TotalPurchases += 499.99;  
}  
}
```

인터페이스에서는 자동 구현 속성을 선언할 수 없습니다. 자동 구현 속성은 private 인스턴스 지원 필드를 선언하는데, 인터페이스는 인스턴스 필드를 선언할 수 없기 때문입니다. 인터페이스에서 본문을 정의하지 않고 속성을 선언하면 해당 인터페이스를 구현하는 각 형식에 의해 구현되어야 하는 접근자와 함께 속성이 선언됩니다.

필드와 유사하게 자동 구현 속성을 초기화할 수 있습니다.

C#

```
public string FirstName { get; set; } = "Jane";
```

앞의 예제에 표시된 클래스는 변경할 수 있습니다. 클라이언트 코드에서는 개체가 만들어진 후에 개체의 값을 변경할 수 있습니다. 데이터 및 중요 동작(메서드)을 포함하는 복잡한 클래스에는 public 속성이 필요한 경우가 많습니다. 그러나 값 세트(데이터)만 캡슐화하고 동작이 적거나 없는 작은 클래스나 구조체의 경우 다음 옵션 중 하나를 사용하여 개체를 변경할 수 없게 해야 합니다.

- `get` 접근자만 선언합니다(생성자를 제외한 모든 위치에서 변경할 수 없음).
- `get` 접근자와 `init` 접근자를 선언합니다(개체 생성 중을 제외하고 모든 위치에서 변경할 수 없음).
- `set` 접근자를 [프라이빗](#)으로 선언합니다(소비자가 변경할 수 없음).

자세한 내용은 [자동으로 구현된 속성을 사용하여 간단한 클래스를 구현하는 방법](#)을 참조하세요.

## 참고 항목

- [자동 구현 속성 사용\(스타일 규칙 IDE0032\)](#)
- [속성](#)
- [한정자](#)

 GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

수도 있습니다. 자세한 내용은  
참여자 가이드를 참조하세요.

☞ 설명서 문제 열기

☞ 제품 사용자 의견 제공

# 자동으로 구현된 속성을 사용하여 간단한 클래스를 구현하는 방법(C# 프로그래밍 가이드)

아티클 • 2024. 03. 10.

이 예제에서는 자동 구현 속성 집합을 캡슐화하는 데만 사용되는 변경할 수 없는 간단한 클래스를 만드는 방법을 보여 줍니다. 참조 형식 의미 체계를 사용해야 하는 경우 구조체 대신 이러한 종류의 구문을 사용합니다.

다음과 같은 방법으로 변경할 수 없는 속성을 만들 수 있습니다.

- `get` 접근자만 선언하여 형식의 생성자를 제외한 어떤 위치에서도 속성을 변경할 수 없도록 만듭니다.
- `set` 접근자 대신 `init` 접근자를 선언합니다. 이렇게 하면 생성자에서만 또는 [개체 초기화자](#)를 사용하여 속성을 설정할 수 있습니다.
- `set` 접근자를 [프라이빗](#)으로 선언합니다. 속성은 형식 내에서 설정할 수 있지만 소비자는 변경할 수 없습니다.

속성 선언에 `required` 한정자를 추가하여 호출자가 새 개체 초기화의 일부로 속성을 설정하도록 할 수 있습니다.

다음 예제는 `get` 접근자만 있는 속성이 `get` 및 `private` 집합이 있는 속성과 어떻게 다른지 보여 줍니다.

C#

```
class Contact
{
    public string Name { get; }
    public string Address { get; private set; }

    public Contact(string contactName, string contactAddress)
    {
        // Both properties are accessible in the constructor.
        Name = contactName;
        Address = contactAddress;
    }

    // Name isn't assignable here. This will generate a compile error.
    //public void ChangeName(string newName) => Name = newName;

    // Address is assignable here.
    public void ChangeAddress(string newAddress) => Address = newAddress;
}
```

# 예시

다음 예제에서는 자동 구현 속성을 갖는 변경할 수 없는 클래스를 구현하는 두 가지 방법을 보여 줍니다. 각 방법에서 속성 중 하나는 private `set`으로 선언하고 다른 하나는 `get`으로만 선언합니다. 첫 번째 클래스는 생성자만 사용하여 속성을 초기화하고 두 번째 클래스는 생성자를 호출하는 정적 팩터리 메서드를 사용합니다.

C#

```
// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// constructor to initialize its properties.
class Contact
{
    // Read-only property.
    public string Name { get; }

    // Read-write property with a private set accessor.
    public string Address { get; private set; }

    // Public constructor.
    public Contact(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }
}

// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// static method and private constructor to initialize its properties.
public class Contact2
{
    // Read-write property with a private set accessor.
    public string Name { get; private set; }

    // Read-only property.
    public string Address { get; }

    // Private constructor.
    private Contact2(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }

    // Public factory method.
    public static Contact2 CreateContact(string name, string address)
    {
        return new Contact2(name, address);
    }
}
```

```

public class Program
{
    static void Main()
    {
        // Some simple data sources.
        string[] names = ["Terry Adams", "Fadi Fakhouri", "Hanying Feng",
                          "Cesar Garcia", "Debra Garcia"];
        string[] addresses = ["123 Main St.", "345 Cypress Ave.", "678 1st
Ave",
                               "12 108th St.", "89 E. 42nd St."];

        // Simple query to demonstrate object creation in select clause.
        // Create Contact objects by using a constructor.
        var query1 = from i in Enumerable.Range(0, 5)
                     select new Contact(names[i], addresses[i]);

        // List elements cannot be modified by client code.
        var list = query1.ToList();
        foreach (var contact in list)
        {
            Console.WriteLine("{0}, {1}", contact.Name, contact.Address);
        }

        // Create Contact2 objects by using a static factory method.
        var query2 = from i in Enumerable.Range(0, 5)
                     select Contact2.CreateContact(names[i],
addresses[i]);

        // Console output is identical to query1.
        var list2 = query2.ToList();

        // List elements cannot be modified by client code.
        // CS0272:
        // list2[0].Name = "Eugene Zabokritski";
    }
}

/* Output:
 Terry Adams, 123 Main St.
 Fadi Fakhouri, 345 Cypress Ave.
 Hanying Feng, 678 1st Ave
 Cesar Garcia, 12 108th St.
 Debra Garcia, 89 E. 42nd St.
*/

```

컴파일러는 각 자동 구현 속성에 대해 지원 필드를 만듭니다. 필드는 소스 코드에서 직접 액세스할 수 없습니다.

## 참고 항목

- 속성
- struct
- 개체 이니셜라이저 및 컬렉션 이니셜라이저

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 메서드(C# 프로그래밍 가이드)

아티클 • 2023. 04. 07.

메서드는 일련의 문을 포함하는 코드 블록입니다. 프로그램을 통해 메서드를 호출하고 필요한 메서드 인수를 지정하여 문을 실행합니다. C#에서는 실행된 모든 명령이 메서드의 컨텍스트에서 수행됩니다.

**Main** 메서드는 모든 C# 애플리케이션의 진입점이고 프로그램이 시작될 때 CLR(공용 언어 런타임)에서 호출됩니다. **최상위 문**을 사용하는 애플리케이션에서 **Main** 메서드는 컴파일러에 의해 생성되며 모든 최상위 문을 포함합니다.

## ① 참고

이 항목에서는 명명된 메서드에 대해 설명합니다. 무명 기능에 대한 자세한 내용은 [람다 식](#)을 참조하세요.

## 메서드 시그니처

메서드는 [클래스](#), [구조체](#) 또는 [인터페이스](#)에서 액세스 수준(예: `public` 또는 `private`), 선택적 한정자(예: `abstract` 또는 `sealed`), 반환 값, 메서드 이름 및 기타 메서드 매개 변수를 지정하여 선언합니다. 이들 파트는 함께 메서드 서명을 구성합니다.

## ① 중요

메서드의 반환 값은 메서드 오버로드를 위한 메서드 서명의 파트가 아닙니다. 그러나 대리자와 대리자가 가리키는 메서드 간의 호환성을 결정할 경우에는 메서드 서명의 부분입니다.

메서드 매개 변수는 괄호로 묶고 쉼표로 구분합니다. 빈 괄호는 메서드에 매개 변수가 필요하지 않음을 나타냅니다. 이 클래스에는 다음 네 개의 메서드가 있습니다.

C#

```
abstract class Motorcycle
{
    // Anyone can call this.
    public void StartEngine() /* Method statements here */

    // Only derived classes can call this.
    protected void AddGas(int gallons) /* Method statements here */
}
```

```

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) { /* Method statements
here */ return 1; }

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}

```

## 메서드 액세스

개체에 대한 메서드 호출은 필드 액세스와 비슷합니다. 개체 이름 뒤에 마침표, 메서드 이름 및 괄호를 추가합니다. 인수는 괄호 안에 나열되고 쉼표로 구분합니다. `Motorcycle` 클래스의 메서드는 다음 예제와 같이 호출될 수 있습니다.

C#

```

class TestMotorcycle : Motorcycle
{
    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {
        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}

```

## 메서드 매개 변수 및 인수

메서드 정의는 필요한 모든 매개 변수의 이름 및 형식을 지정합니다. 호출하는 코드에서 메서드를 호출할 때 해당 코드는 각 매개 변수에 대한 인수라는 구체적인 값을 제공합니다. 인수는 매개 변수 형식과 호환되어야 하지만 호출하는 코드에 사용된 인수 이름(있는 경우)은 메서드에 정의된 명명된 매개 변수와 동일할 필요가 없습니다. 예를 들어:

C#

```

public void Caller()
{
    int numA = 4;
    // Call with an int variable.
    int productA = Square(numA);

    int numB = 32;
    // Call with another int variable.
    int productB = Square(numB);

    // Call with an integer literal.
    int productC = Square(12);

    // Call with an expression that evaluates to int.
    productC = Square(productA * 3);
}

int Square(int i)
{
    // Store input argument in a local variable.
    int input = i;
    return input * input;
}

```

## 참조로 전달할 것인지, 아니면 값으로 전달할 것인지 선택

기본적으로 값 형식의 인스턴스가 메서드에 전달될 때 인스턴스 자체가 아닌 해당 복사본이 전달됩니다. 따라서 인수에 대한 변경 내용은 호출하는 메서드의 원래 인스턴스에 영향을 주지 않습니다. 참조를 통해 값 형식 인스턴스를 전달하려면 `ref` 키워드를 사용합니다. 자세한 내용은 [값 형식 매개 변수 전달을 참조하세요](#).

참조 형식의 개체가 메서드에 전달될 때 개체에 대한 참조가 전달됩니다. 즉, 메서드는 개체 자체가 아니라 개체의 위치를 나타내는 인수를 수신합니다. 이 참조를 사용하여 개체의 멤버를 변경하면 개체를 값으로 전달하더라도 변경 내용은 호출하는 메서드의 인수에 반영됩니다.

다음 예제와 같이 `class` 키워드를 사용하여 참조 형식을 만듭니다.

C#

```

public class SampleRefType
{
    public int value;
}

```

이제 이 형식에 기반을 둔 개체를 메서드에 전달하면 개체에 대한 참조가 전달됩니다. 다음 예제에서는 `SampleRefType` 형식의 개체를 `ModifyObject` 메서드에 전달합니다.

```
C#  
  
public static void TestRefType()  
{  
    SampleRefType rt = new SampleRefType();  
    rt.value = 44;  
    ModifyObject(rt);  
    Console.WriteLine(rt.value);  
}  
  
static void ModifyObject(SampleRefType obj)  
{  
    obj.value = 33;  
}
```

이 예제는 인수를 값으로 메서드에 전달한다는 점에서 기본적으로 이전 예제와 같은 작업을 수행합니다. 그러나 참조 형식이 사용되므로 결과가 다릅니다. `ModifyObject`에서 매개 변수 `value`의 `obj` 필드에 대해 수정한 내용으로 인해 `value` 메서드에서 `rt` 인수의 `TestRefType` 필드도 변경됩니다. `TestRefType` 메서드는 출력으로 33을 표시합니다.

참조 형식을 참조 및 값으로 전달하는 방법에 대한 자세한 내용은 [참조-형식 매개 변수 전달 및 참조 형식](#)을 참조하세요.

## 반환 값

메서드는 호출자에 값을 반환할 수 있습니다. 메서드 이름 앞에 나열된 반환 형식이 `void`가 아니면 메서드는 `return 문`을 사용하여 값을 반환할 수 있습니다. `return` 키워드에 이어 반환 형식과 일치하는 값을 포함하는 문은 메서드 호출자에 값을 반환합니다.

호출자에게 값으로 또는 [참조로](#) 값을 반환할 수 있습니다. `ref` 키워드가 메서드 시그니처에 사용되고 각 `return` 키워드 뒤에 오면 값이 호출자에게 참조로 반환됩니다. 예를 들어 다음 메서드 시그니처 및 `return` 문은 메서드가 호출자를 참조하여 `estDistance`라는 변수를 반환함을 나타냅니다.

```
C#  
  
public ref double GetEstimatedDistance()  
{  
    return ref estDistance;  
}
```

`return` 키워드는 메서드 실행을 중지합니다. 반환 형식이 `void`이면 값이 없는 `return` 문을 사용하여 메서드 실행을 중지할 수 있습니다. `return` 키워드를 사용하지 않으면 메서드는 코드 블록 끝에 도달할 때 실행을 중지합니다. `return` 키워드를 사용하여 값을 반환하려면 `void`가 아닌 반환 값을 포함한 메서드가 필요합니다. 예를 들어 이들 두 메서드에서는 `return` 키워드를 사용하여 정수를 반환합니다.

C#

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

메서드에서 반환된 값을 사용하려면 호출하는 메서드에서 같은 형식의 값으로 충분한 모든 경우에 메서드 호출 자체를 사용하면 됩니다. 반환 값을 변수에 할당할 수도 있습니다. 예를 들어 다음 두 코드 예제에서는 같은 목표를 달성합니다.

C#

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

C#

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

지역 변수(이 경우 `result`)를 사용하여 값을 저장하는 것은 선택 사항입니다. 코드의 가독성에 도움이 될 수 있고 전체 메서드 범위에 대해 인수의 원래 값을 저장해야 할 경우 필요할 수도 있습니다.

메서드에서 참조로 반환된 값을 사용하려면 해당 값을 수정하려는 경우 [참조 지역](#) 변수를 선언해야 합니다. 예를 들어 `Planet.GetEstimatedDistance` 메서드가 `Double` 값을 참조로 반환하는 경우 다음과 같은 코드를 사용하여 참조 지역 변수로 정의할 수 있습니다.

C#

```
ref double distance = ref Planet.GetEstimatedDistance();
```

호출하는 함수가 배열을 `M`에 전달한 경우에는 배열 내용을 수정하는 `M` 메서드에서 다차원 배열을 반환할 필요가 없습니다. 좋은 스타일이나 값의 기능 흐름을 위해 `M`의 결과 배열을 반환할 수 있지만, C#에서는 모든 참조 형식을 값으로 전달하고 배열 참조의 값이 배열에 대한 포인터이기 때문에 필요하지 않습니다. 다음 예제와 같이 `M` 메서드의 배열 내용에 대한 변경 사항은 배열에 대한 참조가 있는 모든 코드에서 관찰할 수 있습니다.

C#

```
static void Main(string[] args)
{
    int[,] matrix = new int[2, 2];
    FillMatrix(matrix);
    // matrix is now full of -1
}

public static void FillMatrix(int[,] matrix)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
    {
        for (int j = 0; j < matrix.GetLength(1); j++)
        {
            matrix[i, j] = -1;
        }
    }
}
```

## 비동기 메서드

비동기 기능을 사용하면 명시적 콜백을 사용하거나 수동으로 여러 메서드 또는 람다 식에 코드를 분할하지 않고도 비동기 메서드를 호출할 수 있습니다.

메서드에 `async` 한정자를 표시하면 메서드에서 `await` 연산자를 사용할 수 있습니다. 컨트롤이 비동기 메서드의 `await` 식에 도달하면 컨트롤이 호출자로 돌아가고 대기 중인 작업이 완료될 때까지 메서드의 진행이 일시 중단됩니다. 작업이 완료되면 메서드가 실행이 다시 시작될 수 있습니다.

### ① 참고

비동기 메서드는 아직 완료되지 않은 첫 번째 대기된 개체를 검색할 때나 비동기 메서드의 끝에 도달할 때 중에서 먼저 발생하는 시점에 호출자에게 반환됩니다.

비동기 메서드는 일반적으로 반환 형식이 `Task<TResult>`, `Task`, `IAsyncEnumerable<T>` 또는 `void`입니다. `void` 반환 형식은 기본적으로 `void` 반환 형식이 필요할 때 이벤트 처리 기를 정의하는 데 사용됩니다. `void`를 반환하는 비동기 메서드는 대기할 수 없고 `void`를 반환하는 메서드의 호출자는 메서드가 `throw`하는 예외를 `catch`할 수 없습니다. 비동기 메서드에는 [작업과 유사한 반환 형식](#)이 있을 수 있습니다.

다음 예제에서 `DelayAsync` 는 반환 형식이 `Task<TResult>`인 비동기 메서드입니다.

`DelayAsync` 에는 정수를 반환하는 `return` 문이 포함됩니다. 따라서 `DelayAsync` 의 메서드 선언의 반환 형식은 `Task<int>`여야 합니다. 반환 형식이 `Task<int>`이므로 `await` 의 `DoSomethingAsync` 식 계산에서 다음 문과 같이 정수가 생성됩니다. `int result = await delayTask.`

`Main` 메서드는 반환 형식이 `Task`인 비동기 메서드의 한 가지 예입니다. `DoSomethingAsync` 메서드로 이동하며, 해당 메서드는 한 줄로 표현되므로 `async` 및 `await` 키워드를 생략할 수 있습니다. `DoSomethingAsync` 는 비동기 메서드이므로 다음 문과 같이 `DoSomethingAsync` 호출에 대한 작업을 기다려야 합니다. `await DoSomethingAsync();`.

```
C#  
  
class Program  
{  
    static Task Main() => DoSomethingAsync();  
  
    static async Task DoSomethingAsync()  
    {  
        Task<int> delayTask = DelayAsync();  
        int result = await delayTask;  
  
        // The previous two statements may be combined into  
        // the following statement.  
        //int result = await DelayAsync();  
  
        Console.WriteLine($"Result: {result}");  
    }  
  
    static async Task<int> DelayAsync()  
    {  
        await Task.Delay(100);  
        return 5;  
    }  
}  
// Example output:  
// Result: 5
```

비동기 메서드는 모든 `ref` 또는 `out` 매개 변수를 선언할 수 없지만, 이러한 매개 변수가 있는 메서드를 호출할 수는 있습니다.

비동기 메서드에 관한 자세한 내용은 [async 및 await를 사용한 비동기 프로그래밍](#) 및 [비동기 반환 형식](#)을 참조하세요.

## 식 본문 정의

일반적으로 식의 결과와 함께 바로 반환되거나 단일 문이 메서드 본문으로 포함된 메서드 정의가 있습니다. `=>`를 사용하여 해당 메서드를 속성을 정의하기 위한 구문 바로 가기는 다음과 같습니다.

C#

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

메서드가 `void` 를 반환하거나 비동기 메서드이면 메서드 본문은 문식이어야 합니다(람다에서와 같음). 속성 및 인덱서의 경우 읽기 전용이어야 하며, `get` 접근자 키워드를 사용하지 않습니다.

## Iterators

반복기는 배열 목록과 같은 컬렉션에 대해 사용자 지정 반복을 수행합니다. 반복기는 `yield return` 문을 사용하여 각 요소를 따로따로 반환할 수 있습니다. `yield return` 문에 도달하면 코드의 현재 위치가 기억됩니다. 다음에 반복기가 호출되면 해당 위치에서 실행이 다시 시작됩니다.

`foreach` 문을 사용하여 클라이언트 코드에서 반복기를 호출합니다.

반복기의 반환 형식은 `IEnumerable`, `IEnumerable<T>`, `IAsyncEnumerable<T>`, `IEnumerator` 또는 `IEnumerator<T>` 일 수 있습니다.

자세한 내용은 [반복기](#)를 참조하세요.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)을 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

# 참고 항목

- C# 프로그래밍 가이드
- C# 형식 시스템
- 액세스 한정자
- 정적 클래스 및 정적 클래스 멤버
- 상속
- 추상/봉인된 클래스 및 클래스 멤버
- params
- out
- ref
- 메서드 매개 변수

# 로컬 함수(C# 프로그래밍 가이드)

아티클 • 2023. 04. 08.

로컬 함수는 다른 멤버에 중첩된 형식의 메서드입니다. 포함하는 멤버에서만 호출할 수 있습니다. 로컬 함수는 다음에서 선언하고 호출할 수 있습니다.

- 메서드, 특히 반복기 메서드 및 비동기 메서드
- 생성자
- 속성 접근자
- 이벤트 접근자
- 무명 메서드
- 람다 식
- 종료자
- 다른 로컬 함수

그러나 식 본문 멤버 내에서는 로컬 함수를 선언할 수 없습니다.

## ① 참고

로컬 함수에서도 지원하는 기능을 람다 식으로 구현할 수 있는 경우도 있습니다. 비교를 보려면 [로컬 함수 및 람다 식](#)을 참조하세요.

로컬 함수는 코드의 의도를 명확하게 합니다. 코드를 읽는 모든 사용자가 포함하는 메서드를 통해서만 메서드를 호출할 수 있음을 알 수 있습니다. 팀 프로젝트의 경우 로컬 함수를 사용하면 다른 개발자가 실수로 클래스 또는 구조체의 다른 곳에서 직접 메서드를 호출하는 경우를 방지할 수도 있습니다.

## 로컬 함수 구문

로컬 함수는 포함하는 멤버 내의 중첩 메서드로 정의됩니다. 해당 정의는 다음 구문을 사용합니다.

C#

```
<modifiers> <return-type> <method-name> <parameter-list>
```

로컬 함수에 다음 한정자를 사용할 수 있습니다.

- `async`
- `unsafe`

- `static` 정적 로컬 함수는 지역 변수 또는 instance 상태를 캡처할 수 없습니다.
- `extern` 외부 로컬 함수는 이어야 `static` 합니다.

해당 메서드 매개 변수를 비롯하여 포함하는 멤버에 정의된 모든 지역 변수는 정적이지 않은 로컬 함수에서 액세스할 수 있습니다.

메서드 정의와 달리 로컬 함수 정의에는 멤버 액세스 한정자를 포함할 수 없습니다. 모든 로컬 함수는 `private`이므로 `private` 키워드 등의 액세스 한정자를 포함하면 컴파일러 오류 CS0106, "이 항목의 'private' 한정자가 유효하지 않습니다."가 생성됩니다.

다음 예제에서는 `GetText` 메서드에 대해 `private`인 로컬 함수 `AppendPathSeparator`를 정의합니다.

C#

```
private static string GetText(string path, string filename)
{
    var reader = File.OpenText(${AppendPathSeparator(path)}{filename}");
    var text = reader.ReadToEnd();
    return text;

    string AppendPathSeparator(string filepath)
    {
        return filepath.EndsWith(@"\") ? filepath : filepath + @"\";
    }
}
```

C# 9.0부터 다음 예제와 같이 로컬 함수, 매개 변수, 형식 매개 변수에 특성을 적용할 수 있습니다.

C#

```
#nullable enable
private static void Process(string?[] lines, string mark)
{
    foreach (var line in lines)
    {
        if (IsValid(line))
        {
            // Processing logic...
        }
    }

    bool IsValid([NotNullWhen(true)] string? line)
    {
        return !string.IsNullOrEmpty(line) && line.Length >= mark.Length;
    }
}
```

이전 예제에서는 [특수 특성을](#) 사용하여 null 허용 컨텍스트에서 컴파일러의 정적 분석을 지원합니다.

## 로컬 함수 및 예외

로컬 함수의 유용한 기능 중 하나는 예외가 즉시 나타나도록 할 수 있다는 것입니다. 반복기 메서드의 경우 반환된 시퀀스가 열거될 때만 예외가 표시되고 반복기를 검색할 때는 예외가 표시되지 않습니다. 비동기 메서드의 경우 비동기 메서드에서 throw된 예외는 반환된 작업을 대기할 때 관찰됩니다.

다음 예제에서는 지정한 범위의 홀수를 열거하는 `OddSequence` 메서드를 정의합니다. 100보다 큰 숫자를 `OddSequence` 열거자 메서드에 전달하기 때문에 메서드가 `ArgumentOutOfRangeException`을 throw합니다. 예제의 출력에서 볼 수 있듯이 예외는 숫자를 반복하는 경우에만 나타나고 열거자를 검색할 때는 나타나지 않습니다.

C#

```
public class IteratorWithoutLocalExample
{
    public static void Main()
    {
        IEnumerable<int> xs = OddSequence(50, 110);
        Console.WriteLine("Retrieved enumerator...");

        foreach (var x in xs) // line 11
        {
            Console.Write($"{x} ");
        }
    }

    public static IEnumerable<int> OddSequence(int start, int end)
    {
        if (start < 0 || start > 99)
            throw new ArgumentOutOfRangeException(nameof(start), "start must be
between 0 and 99.");
        if (end > 100)
            throw new ArgumentOutOfRangeException(nameof(end), "end must be
less than or equal to 100.");
        if (start >= end)
            throw new ArgumentException("start must be less than end.");

        for (int i = start; i <= end; i++)
        {
            if (i % 2 == 1)
                yield return i;
        }
    }
}

// The example displays the output like this:
```

```
//  
//    Retrieved enumerator...  
//    Unhandled exception. System.ArgumentOutOfRangeException: end must be  
less than or equal to 100. (Parameter 'end')  
//        at IteratorWithoutLocalExample.OddSequence(Int32 start, Int32  
end)+MoveNext() in IteratorWithoutLocal.cs:line 22  
//        at IteratorWithoutLocalExample.Main() in IteratorWithoutLocal.cs:line  
11
```

로컬 함수에 반복기 논리를 추가하는 경우 다음 예제와 같이 열거자를 검색할 때 인수 유효성 검사 예외가 throw됩니다.

C#

```
public class IteratorWithLocalExample  
{  
    public static void Main()  
    {  
        IEnumerable<int> xs = OddSequence(50, 110); // line 8  
        Console.WriteLine("Retrieved enumerator...");  
  
        foreach (var x in xs)  
        {  
            Console.Write($"{x} ");  
        }  
    }  
  
    public static IEnumerable<int> OddSequence(int start, int end)  
    {  
        if (start < 0 || start > 99)  
            throw new ArgumentOutOfRangeException(nameof(start), "start must be  
between 0 and 99.");  
        if (end > 100)  
            throw new ArgumentOutOfRangeException(nameof(end), "end must be  
less than or equal to 100.");  
        if (start >= end)  
            throw new ArgumentException("start must be less than end.");  
  
        return GetOddSequenceEnumerator();  
  
        IEnumerable<int> GetOddSequenceEnumerator()  
        {  
            for (int i = start; i <= end; i++)  
            {  
                if (i % 2 == 1)  
                    yield return i;  
            }  
        }  
    }  
    // The example displays the output like this:  
    //  
    //    Unhandled exception. System.ArgumentOutOfRangeException: end must be
```

```
less than or equal to 100. (Parameter 'end')
//   at IteratorWithLocalExample.OddSequence(Int32 start, Int32 end) in
IteratorWithLocal.cs:line 22
//   at IteratorWithLocalExample.Main() in IteratorWithLocal.cs:line 8
```

## 로컬 함수 및 람다 식

얼핏 보기에도 로컬 함수와 [람다 식](#)은 매우 유사합니다. 대부분의 경우 람다 식과 로컬 함수 사용 중에서 선택하는 것은 [스타일과 개인 기본 설정의 문제](#)입니다. 그러나 하나 또는 다른 것을 사용할 수 있는 것에 알고 있어야 하는 실제 차이점이 있습니다.

계속 알고리즘의 로컬 함수 및 람다 식 구현 간의 차이점을 살펴보겠습니다. 로컬 함수를 사용하는 버전은 다음과 같습니다.

C#

```
public static int LocalFunctionFactorial(int n)
{
    return nthFactorial(n);

    int nthFactorial(int number) => number < 2
        ? 1
        : number * nthFactorial(number - 1);
}
```

이 버전은 람다 식을 사용합니다.

C#

```
public static int LambdaFactorial(int n)
{
    Func<int, int> nthFactorial = default(Func<int, int>);

    nthFactorial = number => number < 2
        ? 1
        : number * nthFactorial(number - 1);

    return nthFactorial(n);
}
```

## 이름 지정

로컬 함수는 메서드와 같이 명시적으로 이름이 지정됩니다. 람다 식은 무명 메서드이며 일반적으로 `Action` 또는 `Func` 형식인 `delegate` 형식의 변수에 할당해야 합니다. 로컬 함

수를 선언하는 경우 프로세스는 일반 메서드를 작성하는 것과 유사하며, 반환 형식 및 함수 시그니처를 선언합니다.

## 함수 시그니처 및 람다 식 형식

람다 식은 인수 및 반환 형식을 결정하기 위해 할당되는 `Action`/`Func` 변수의 형식을 사용합니다. 로컬 함수에서 구문은 일반적인 메서드를 작성하는 것과 매우 유사하기 때문에 인수 형식 및 반환 형식이 이미 함수 선언에 포함되어 있습니다.

C# 10부터, 일부 람다 식은 컴파일러가 람다 식의 반환 형식과 매개 변수 형식을 유추할 수 있도록 해 주는 '자연 형식'을 갖습니다.

## 한정된 할당

람다 식은 런타임에 선언되고 할당되는 개체입니다. 람다 식을 사용하려면 명확하게 할당해야 합니다. 할당될 `Action`/`Func` 변수를 선언하고 람다 식을 할당해야 합니다.

`LambdaFactorial`은 정의하기 전에 먼저 람다 식 `nthFactorial`을 선언하고 초기화해야 합니다. 이렇게 하지 않으면 `nthFactorial`을 할당하기 전에 참조하여 컴파일 시간 오류가 발생합니다.

로컬 함수는 컴파일 시간에 정의됩니다. 변수에 할당되지 않기 때문에 범위에 있는 모든 코드 위치에서 참조할 수 있습니다. 첫 번째 예제 `LocalFunctionFactorial`에서는 `return` 문 위 또는 아래에 로컬 함수를 선언하고 컴파일러 오류를 트리거하지 않을 수 있습니다.

이러한 차이점은 로컬 함수를 사용하여 재귀 알고리즘을 쉽게 만들 수 있음을 의미합니다. 자신을 호출하는 로컬 함수를 선언하고 정의할 수 있습니다. 람다 식을 동일한 람다 식을 참조하는 본문에 다시 할당하려면 선언하고, 기본 값을 할당해야 합니다.

## 대리자로 구현

람다 식은 선언될 때 대리자로 변환됩니다. 로컬 함수는 기존 메서드 '또는' 대리자로 작성할 수 있다는 점에서 더욱 유연합니다. 로컬 함수는 대리자로 **사용되는** 경우에만 대리자로 변환됩니다.

로컬 함수를 선언하고 메서드처럼 호출하여 참조하는 경우 대리자로 변환되지 않습니다.

## 변수 캡처

**한정된 할당** 규칙은 로컬 함수 또는 람다 식에서 캡처되는 변수에도 영향을 줍니다. 컴파일러는 로컬 함수가 포함된 범위에서 캡처된 변수를 한정적으로 할당할 수 있도록 하는 정적 분석을 수행할 수 있습니다. 다음 예제를 고려해 보세요.

C#

```
int M()
{
    int y;
    LocalFunction();
    return y;

    void LocalFunction() => y = 0;
}
```

컴파일러는 `LocalFunction`에서 호출될 때 `y`를 한정적으로 할당하는지 확인할 수 있습니다. `LocalFunction`은 `return` 문 전에 호출되므로 `y`는 `return` 문에서 한정적으로 할당됩니다.

로컬 함수가 바깥쪽 범위에서 변수를 캡처하는 경우 로컬 함수는 대리자 형식으로 구현됩니다.

## 힙 할당

해당 용도에 따라 로컬 함수는 람다 식에 항상 필요한 힙 할당을 피할 수 있습니다. 로컬 함수가 대리자로 변환되지 않고 로컬 함수에 의해 캡처된 변수가 대리자로 변환된 다른 람다 식 또는 로컬 함수에 의해 캡처되지 않는 경우 컴파일러는 힙 할당을 피할 수 있습니다.

다음 비동기 예제를 살펴보세요.

C#

```
public async Task<string> PerformLongRunningWorkLambda(string address, int
index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required",
paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index),
message: "The index must be non-negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name",
paramName: nameof(name));

    Func<Task<string>> longRunningWorkImplementation = async () =>
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}.
Enjoy.";
    };
}
```

```
        return await longRunningWorkImplementation();
    }
```

이 람다 식의 클로저에는 `address`, `index` 및 `name` 변수가 포함됩니다. 로컬 함수의 경우 클로저를 구현하는 개체는 `struct` 형식일 수 있습니다. 해당 구조체 형식은 로컬 함수에 참조로 전달됩니다. 구현에서 이러한 차이점은 할당에 저장됩니다.

람다 식에 필요한 인스턴스화는 추가 메모리 할당을 의미하며, 시간이 중요한 코드 경로에서 성능에 영향을 줄 수 있습니다. 로컬 함수는 이러한 오버헤드를 유발하지 않습니다. 위 예제에서 로컬 함수 버전은 람다 식 버전보다 할당 수가 2개 더 적습니다.

로컬 함수가 대리자로 변환되지 않고 로컬 함수에 의해 캡처된 변수가 대리자로 변환된 다른 람다 또는 로컬 함수에 의해 캡처되지 않는 경우 로컬 함수를 `static` 로컬 함수로 선언하여 힙에 할당되지 않도록 보장할 수 있습니다.

### 💡 팁

로컬 함수가 항상 `static`으로 표시되도록 .NET 코드 스타일 규칙 **IDE0062**를 사용하도록 설정합니다.

### ⓘ 참고

이 메서드에 해당하는 로컬 함수는 클로저에 클래스도 사용합니다. 로컬 함수의 클로저가 `class` 또는 `struct`로 구현되는지 여부는 구현 세부 정보입니다. 로컬 함수는 `struct`를 사용할 수 있는 반면, 람다는 항상 `class`를 사용합니다.

C#

```
public async Task<string> PerformLongRunningWork(string address, int index,
string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required",
paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index),
message: "The index must be non-negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name",
paramName: nameof(name));

    return await longRunningWorkImplementation();

async Task<string> longRunningWorkImplementation()
```

```
{  
    var interimResult = await FirstWork(address);  
    var secondResult = await SecondStep(index, name);  
    return $"The results are {interimResult} and {secondResult}.  
Enjoy.";  
}  
}
```

## yield 키워드 사용

이 샘플에서 설명하지 않은 한 가지 최종 장점은 `yield return` 구문을 사용해서 로컬 함수를 반복기로 구현하여 값 시퀀스를 생성할 수 있다는 것입니다.

C#

```
public IEnumerable<string> SequenceToLowercase(IEnumerable<string> input)  
{  
    if (!input.Any())  
    {  
        throw new ArgumentException("There are no items to convert to  
lowercase.");  
    }  
  
    return LowercaseIterator();  
  
    IEnumerable<string> LowercaseIterator()  
    {  
        foreach (var output in input.Select(item => item.ToLower()))  
        {  
            yield return output;  
        }  
    }  
}
```

`yield return` 문은 람다 식에 허용되지 않습니다. 자세한 내용은 [컴파일러 오류 CS1621](#)을 참조하세요.

로컬 함수는 람다 식과 중복되는 것처럼 보일 수도 있지만, 실제로 다른 용도로 다르게 사용됩니다. 로컬 함수는 다른 메서드의 컨텍스트에서만 호출되는 함수를 작성하려는 경우에 더 효율적입니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 로컬 함수 선언](#) 섹션을 참조하세요.

## 추가 정보

- 람다 대신 로컬 함수 사용(스타일 규칙 IDE0039)
- 메서드

# 암시적 형식 지역 변수(C# 프로그래밍 가이드)

아티클 • 2023. 03. 14.

명시적 형식을 제공하지 않고 지역 변수를 선언할 수 있습니다. `var` 키워드는 초기화 문의 오른쪽에 있는 식에서 변수의 형식을 유추하도록 컴파일러에 지시합니다. 유추된 형식은 기본 제공 형식, 무명 형식, 사용자 정의 형식 또는 .NET 클래스 라이브러리에 정의된 형식일 수 있습니다. `var`을 사용하여 배열을 초기화하는 방법에 대한 자세한 내용은 [암시적으로 형식화된 배열을 참조하세요](#).

다음 예제에서는 `var`을 사용하여 지역 변수를 선언하는 다양한 방법을 보여 줍니다.

C#

```
// i is compiled as an int
var i = 5;

// s is compiled as a string
var s = "Hello";

// a is compiled as int[]
var a = new[] { 0, 1, 2 };

// expr is compiled as IEnumerable<Customer>
// or perhaps IQueryable<Customer>
var expr =
    from c in customers
    where c.City == "London"
    select c;

// anon is compiled as an anonymous type
var anon = new { Name = "Terry", Age = 34 };

// list is compiled as List<int>
var list = new List<int>();
```

`var` 키워드는 "variant"를 의미하지 않고 변수가 느슨하게 형식화되었거나 런타임에 바인딩되었음을 나타내지도 않습니다. 단지 컴파일러가 가장 적절한 형식을 결정하고 할당함을 의미합니다.

`var` 키워드는 다음과 같은 컨텍스트에서 사용할 수 있습니다.

- 앞의 예제와 같이 지역 변수(메서드 범위에서 선언된 변수)에 대해 사용
- `for` 초기화 문에서 사용

C#

```
for (var x = 1; x < 10; x++)
```

- `foreach` 초기화 문에서 사용

C#

```
foreach (var item in list) {...}
```

- `using` 문에서 사용

C#

```
using (var file = new StreamReader("C:\\myfile.txt")) {...}
```

자세한 내용은 [쿼리 식에서 암시적 형식 지역 변수 및 배열을 사용하는 방법](#)을 참조하세요.

## var 및 무명 형식

대부분의 경우 `var` 사용은 선택 사항이며 단지 편리한 구문을 위해 사용됩니다. 그러나 변수가 무명 형식을 사용하여 초기화된 경우 나중에 개체의 속성에 액세스해야 하면 변수를 `var`로 선언해야 합니다. 이것이 LINQ 쿼리 식의 일반적인 시나리오입니다. 자세한 내용은 [무명 형식](#)을 참조하세요.

소스 코드의 관점에서 무명 형식에는 이름이 없습니다. 따라서 쿼리 변수가 `var`로 초기화된 경우 반환된 개체 시퀀스의 속성에 액세스하는 유일한 방법은 `var`을 `foreach` 문의 반복 변수 형식으로 사용하는 것입니다.

C#

```
class ImplicitlyTypedLocals2
{
    static void Main()
    {
        string[] words = { "aPPLE", "BLUeBeRrY", "cHeRry" };

        // If a query produces a sequence of anonymous types,
        // then use var in the foreach statement to access the properties.
        var upperLowerWords =
            from w in words
            select new { Upper = w.ToUpper(), Lower = w.ToLower() };

        // Execute the query
    }
}
```

```

        foreach (var ul in upperLowerWords)
        {
            Console.WriteLine("Uppercase: {0}, Lowercase: {1}", ul.Upper,
ul.Lower);
        }
    }
/* Outputs:
   Uppercase: APPLE, Lowercase: apple
   Uppercase: BLUEBERRY, Lowercase: blueberry
   Uppercase: CHERRY, Lowercase: cherry
*/

```

## 설명

암시적 형식 변수 선언에는 다음과 같은 제한 사항이 적용됩니다.

- 지역 변수가 동일한 문에서 선언 및 초기화된 경우에만 `var`을 사용할 수 있습니다. 변수를 `null`이나 메서드 그룹 또는 익명 함수로 초기화할 수는 없습니다.
- 클래스 범위의 필드에는 `var`을 사용할 수 없습니다.
- `var`을 사용하여 선언된 변수는 초기화 식에 사용할 수 없습니다. 즉, 이 식(`int i = (i = 20);`)은 유효하지만, 이 식(`var i = (i = 20);`)은 컴파일 시간 오류를 생성합니다.
- 동일한 문에서 여러 개의 암시적 형식 변수를 초기화할 수 없습니다.
- `var`라는 형식이 범위 내에 있으면 `var` 키워드가 해당 형식 이름으로 확인되고 암시적 형식 지역 변수 선언의 일부로 처리되지 않습니다.

`var` 키워드를 사용한 암시적 형식화는 로컬 메서드 범위의 변수에만 적용할 수 있습니다. C# 컴파일러가 코드를 처리하면서 논리적 패러독스를 만나게 되므로 암시적 형식 지정은 클래스 필드에 사용할 수 없습니다. 컴파일러는 필드 형식을 알아야 하나 할당 식을 분석할 때까지 형식을 결정할 수 없고 형식을 모르면 식을 평가할 수 없습니다. 다음 코드를 살펴보세요.

C#

```
private var bookTitles;
```

`bookTitles`는 `var` 형식의 클래스 필드입니다. 이 필드는 평가할 식이 없으므로 어떤 형식의 `bookTitles`가 될지 컴파일러가 추론하는 것이 불가능합니다. 또한 필드에 식을 추가(로컬 변수에서처럼)하는 것으로는 부족합니다.

C#

```
private var bookTitles = new List<string>();
```

컴파일러에서 코드 컴파일 중 필드를 만나면 연결된 식을 처리하기 전에 각 필드의 형식을 기록합니다. 컴파일러가 `bookTitles` 구문 분석을 시도하는 동일한 패리독스를 만나면 필드 형식을 알아야 하지만 컴파일러는 일반적으로 식을 분석하여 `var` 형식을 판단합니다. 이는 앞의 형식을 알지 못하면 불가능합니다.

`var`은 생성된 쿼리 변수 형식을 정확하게 확인하기 어려운 쿼리 식에서도 유용할 수 있습니다. 그룹화 및 정렬 작업에서 이러한 경우가 발생할 수 있습니다.

또한 `var` 키워드는 변수의 특정 형식이 키보드에서 입력하기 번거롭거나, 명확하거나, 코드의 가독성에 도움이 되지 않는 경우에 유용할 수 있습니다. 이런 방식으로 `var`이 유용한 한 가지 예로 그룹 작업에 사용되는 경우 등의 중첩된 제네릭 형식이 있습니다. 다음 쿼리에서 쿼리 변수의 형식은 `IEnumerable<IGrouping<string, Student>>`입니다. 사용자나 코드를 유지 관리해야 하는 다른 사용자가 이 점을 이해하기만 하면 편리하고 간단하도록 암시적 형식을 사용하는 데 문제가 없습니다.

C#

```
// Same as previous example except we use the entire last name as a key.  
// Query variable is an IEnumerable<IGrouping<string, Student>>  
var studentQuery3 =  
    from student in students  
    group student by student.Last;
```

`var`을 사용하면 코드를 간소화하는 데 도움이 되지만, 필요한 경우나 코드를 더 쉽게 읽도록 만들 때로 사용을 제한해야 합니다. `var`을 제대로 사용해야 하는 경우에 대한 자세한 내용은 C# 코딩 지침 문서의 [암시적 형식 지역 변수](#) 섹션을 참조하세요.

## 참조

- [C# 참조](#)
- [암시적으로 형식화된 배열](#)
- [쿼리 식에서 암시적으로 형식화된 지역 변수 및 배열 사용 방법](#)
- [익명 형식](#)
- [개체 이니셜라이저 및 컬렉션 이니셜라이저](#)
- [var](#)
- [C#의 LINQ](#)
- [LINQ\(Language-Integrated Query\)](#)
- [반복 문](#)

- using 문

# 쿼리 식에서 암시적으로 형식화된 지역 변수 및 배열을 사용하는 방법(C# 프로그래밍 가이드)

아티클 • 2024. 03. 12.

컴파일러에서 지역 변수의 형식을 확인하려는 경우 항상 암시적으로 형식화된 지역 변수를 사용할 수 있습니다. 쿼리 식에 자주 사용되는 무명 형식을 저장하려면 암시적으로 형식화된 지역 변수를 사용해야 합니다. 다음 예제에서는 쿼리에서 암시적으로 형식화된 지역 변수의 선택적 및 필수 사용을 보여 줍니다.

암시적으로 형식화된 지역 변수는 `var` 상황별 키워드를 사용하여 선언됩니다. 자세한 내용은 [암시적으로 형식화된 지역 변수 및 암시적으로 형식화된 배열](#)을 참조하세요.

## 예제

다음 예제에서는 `var` 키워드가 필요한 일반적인 시나리오로, 무명 형식의 시퀀스를 생성하는 쿼리 식을 보여 줍니다. 이 시나리오에서는 무명 형식의 형식 이름에 대한 액세스 권한이 없기 때문에 `foreach` 문의 쿼리 변수와 반복 변수가 모두 `var`을 사용하여 암시적으로 형식화되어야 합니다. 무명 형식에 대한 자세한 내용은 [무명 형식](#)을 참조하세요.

```
C#  
  
private static void QueryNames(char firstLetter)  
{  
    // Create the query. Use of var is required because  
    // the query produces a sequence of anonymous types:  
    // System.Collections.Generic.IEnumerable<????>.  
    var studentQuery =  
        from student in students  
        where student.FirstName[0] == firstLetter  
        select new { student.FirstName, student.LastName };  
  
    // Execute the query and display the results.  
    foreach (var anonType in studentQuery)  
    {  
        Console.WriteLine("First = {0}, Last = {1}", anonType.FirstName,  
            anonType.LastName);  
    }  
}
```

다음 예제에서는 유사하지만 `var` 사용이 선택 사항인 상황에서 `var` 키워드를 사용합니다. `student.LastName`이 문자열이기 때문에 쿼리를 실행하면 문자열 시퀀스가 반환됩니다.

다. 따라서 `queryId`의 형식을 `var` 대신

`System.Collections.Generic.IEnumerable<string>`로 선언할 수 있습니다. `var` 키워드는 편의상 사용됩니다. 예제에서 `foreach` 문의 반복 변수는 명시적으로 문자열로 형식화되었지만, 대신 `var`을 사용하여 선언할 수도 있습니다. 반복 변수의 형식이 무명 형식이 아니기 때문에 `var` 사용은 요구 사항이 아니라 옵션입니다. `var` 자체는 형식이 아니라 형식을 유추하고 할당하도록 컴파일러에 지시하는 명령입니다.

C#

```
// Variable queryId could be declared by using
// System.Collections.Generic.IEnumerable<string>
// instead of var.
var queryId =
    from student in students
    where student.Id > 111
    select student.LastName;

// Variable str could be declared by using var instead of string.
foreach (string str in queryId)
{
    Console.WriteLine("Last name: {0}", str);
}
```

## 참고 항목

- 확장명 메서드
- LINQ(Language-Integrated Query)
- C#의 LINQ

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

#### 설명서 문제 열기

#### 제품 사용자 의견 제공

# 확장명 메서드(C# 프로그래밍 가이드)

아티클 • 2024. 03. 15.

확장명 메서드를 사용하면 새 파생 형식을 만들거나 다시 컴파일하거나 원래 형식을 수정하지 않고도 기존 형식에 메서드를 "추가"할 수 있습니다. 확장 메서드는 정적 메서드이지만 확장 형식의 인스턴스 메서드인 것처럼 호출됩니다. C#, F# 및 Visual Basic에서 작성된 클라이언트 코드의 경우 확장명 메서드를 호출하는 것과 형식에 정의된 메서드를 호출하는 데는 명백한 차이가 없습니다.

가장 일반적인 확장명 메서드는 쿼리 기능을 기준 [System.Collections.IEnumerable](#) 및 [System.Collections.Generic.IEnumerable<T>](#) 형식에 추가하는 LINQ 표준 쿼리 연산자입니다. 표준 쿼리 연산자를 사용하려면 `using System.Linq` 지시문을 사용해서 먼저 범위를 지정합니다. 그러면 [IQueryable<T>](#)을 구현하는 모든 형식에 [GroupBy](#), [OrderBy](#), [Average](#) 등의 인스턴스 메서드가 있는 것처럼 나타납니다. [List<T>](#) 또는 [Array](#)와 같은 [IEnumerable<T>](#) 형식의 인스턴스 뒤에 "dot"를 입력하면 IntelliSense 문 완성에서 이러한 추가 메서드를 볼 수 있습니다.

## OrderBy 예제

다음 예제에서는 정수 배열에서 표준 쿼리 연산자 `orderBy`를 호출하는 방법을 보여 줍니다. 괄호 안의 식은 람다 식입니다. 많은 표준 쿼리 연산자가 람다 식을 매개 변수로 사용하지만 확장명 메서드에 대한 요구 사항은 아닙니다. 자세한 내용은 [람다 식을 참조하세요](#).

```
C#  
  
class ExtensionMethods2  
{  
  
    static void Main()  
    {  
        int[] ints = [10, 45, 15, 39, 21, 26];  
        var result = ints.OrderBy(g => g);  
        foreach (var i in result)  
        {  
            System.Console.Write(i + " ");  
        }  
    }  
    //Output: 10 15 21 26 39 45
```

확장명 메서드는 정적 메서드로 정의되지만 인스턴스 메서드 구문을 사용하여 호출됩니다. 첫 번째 매개 변수는 메서드가 작동하는 형식을 지정합니다. 매개 변수는 이 [한정자를](#)

[따릅니다](#). 확장 메서드는 `using` 지시문을 사용하여 명시적으로 네임스페이스를 소스 코드로 가져오는 경우에만 범위에 있습니다.

다음 예제에서는 `System.String` 클래스에 대해 정의된 확장 메서드를 보여 줍니다. 이 확장 메서드는 제네릭이 아닌 비중첩 정적 클래스 내부에서 정의됩니다.

C#

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this string str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                            StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

`WordCount` 지시문을 사용하여 `using` 확장 메서드를 범위로 가져올 수 있습니다.

C#

```
using ExtensionMethods;
```

또한 다음 구문을 사용하여 애플리케이션에서 확장 메서드를 호출할 수 있습니다.

C#

```
string s = "Hello Extension Methods";
int i = s.WordCount();
```

코드에서 인스턴스 메서드 구문을 사용하여 확장 메서드를 호출합니다. 컴파일러에서 생성된 IL(중간 언어)이 코드를 정적 메서드 호출로 변환합니다. 캡슐화의 원칙은 실제로 위반되지 않습니다. 확장 메서드는 확장 중인 형식의 프라이빗 변수에 액세스할 수 없습니다.

`MyExtensions` 클래스와 `static` 메서드는 모두 `static`이며, 다른 모든 `WordCount` 멤버에 액세스하듯 액세스할 수 있습니다. `WordCount` 메서드는 다음과 같이 다른 `static` 메서드를 호출하듯 호출할 수 있습니다.

C#

```
string s = "Hello Extension Methods";
```

```
int i = MyExtensions.WordCount(s);
```

위의 C# 코드에서:

- 값이 "Hello Extension Methods" 인 s라는 새 string을 선언하고 할당합니다.
- 지정된 인수를 호출 MyExtensions.WordCount 합니다 s.

자세한 내용은 [사용자 지정 확장 메서드를 구현 및 호출하는 방법](#)을 참조하세요.

일반적으로 확장명 메서드를 직접 구현하는 것보다 호출하는 경우가 훨씬 많습니다. 확장 메서드는 인스턴스 메서드 구문을 사용하여 호출되므로 특별한 지식이 없어도 클라이언트 코드에서 확장 메서드를 사용할 수 있습니다. 특정 형식의 확장 메서드를 사용하려면 해당 메서드가 정의된 네임스페이스에 대해 using 지시문을 추가합니다. 예를 들어 표준 쿼리 연산자를 사용하려면 다음 using 지시문을 코드에 추가합니다.

C#

```
using System.Linq;
```

System.Core.dll에 대한 참조를 추가해야 할 수도 있습니다. 이제 표준 쿼리 연산자가 대부분의 `IEnumerable<T>` 형식에 사용할 수 있는 추가 메서드로 IntelliSense에 표시됩니다.

## 컴파일 타임에 확장 메서드 바인딩

확장 메서드를 사용하여 클래스 또는 인터페이스를 확장할 수 있지만 재정의할 수는 없습니다. 이름과 시그니처가 인터페이스 또는 클래스 메서드와 동일한 확장 메서드는 호출되지 않습니다. 컴파일 시간에 확장 메서드는 항상 형식 자체에서 정의된 인스턴스 메서드보다 우선 순위가 낮습니다. 즉, 형식에 `Process(int i)`라는 메서드가 있고 동일한 시그니처를 가진 확장 메서드가 있는 경우 컴파일러는 항상 인스턴스 메서드에 바인딩합니다. 컴파일러는 메서드 호출을 발견할 경우 먼저 형식의 인스턴스 메서드에서 일치 항목을 찾습니다. 일치하는 항목이 없으면 형식에 대해 정의된 확장 메서드를 검색하고 찾은 첫 번째 확장 메서드에 바인딩합니다.

## 예시

다음 예제에서는 C# 컴파일러가 메서드 호출을 형식의 인스턴스 메서드 또는 확장명 메서드에 바인딩할 것인지 결정할 때 따르는 규칙을 보여 줍니다. 정적 클래스 Extensions는 `IMyInterface`를 구현하는 모든 형식에 대해 정의된 확장 메서드를 포함합니다. A, B 및 C 클래스는 모두 인터페이스를 구현합니다.

**MethodB** 확장 메서드는 이름과 시그니처가 클래스에서 이미 구현된 메서드와 정확하게 일치하므로 호출되지 않습니다.

일치하는 시그니처를 가진 인스턴스 메서드를 찾을 수 없으면 컴파일러는 일치하는 확장 명 메서드(있는 경우)에 바인딩합니다.

C#

```
// Define an interface named IMyInterface.
namespace DefineIMyInterface
{
    public interface IMyInterface
    {
        // Any class that implements IMyInterface must define a method
        // that matches the following signature.
        void MethodB();
    }
}

// Define extension methods for IMyInterface.
namespace Extensions
{
    using System;
    using DefineIMyInterface;

    // The following extension methods can be accessed by instances of any
    // class that implements IMyInterface.
    public static class Extension
    {
        public static void MethodA(this IMyInterface myInterface, int i)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, int i)");
        }

        public static void MethodA(this IMyInterface myInterface, string s)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, string
s)");
        }

        // This method is never called in ExtensionMethodsDemo1, because
each
        // of the three classes A, B, and C implements a method named
MethodB
        // that has a matching signature.
        public static void MethodB(this IMyInterface myInterface)
        {
            Console.WriteLine
                ("Extension.MethodB(this IMyInterface myInterface)");
        }
    }
}
```

```
}
```

```
// Define three classes that implement IMyInterface, and then use them to
// test
// the extension methods.
```

```
namespace ExtensionMethodsDemo1
{
    using System;
    using Extensions;
    using DefineIMyInterface;
```

```
    class A : IMyInterface
    {
        public void MethodB() { Console.WriteLine("A.MethodB()"); }
    }
```

```
    class B : IMyInterface
    {
        public void MethodB() { Console.WriteLine("B.MethodB()"); }
        public void MethodA(int i) { Console.WriteLine("B.MethodA(int i)"); }
    }
```

```
    class C : IMyInterface
    {
        public void MethodB() { Console.WriteLine("C.MethodB()"); }
        public void MethodA(object obj)
        {
            Console.WriteLine("C.MethodA(object obj)");
        }
    }
```

```
    class ExtMethodDemo
    {
        static void Main(string[] args)
        {
            // Declare an instance of class A, class B, and class C.
            A a = new A();
            B b = new B();
            C c = new C();

            // For a, b, and c, call the following methods:
            //     -- MethodA with an int argument
            //     -- MethodA with a string argument
            //     -- MethodB with no argument.

            // A contains no MethodA, so each call to MethodA resolves to
            // the extension method that has a matching signature.
            a.MethodA(1);           // Extension.MethodA(IMyInterface, int)
            a.MethodA("hello");     // Extension.MethodA(IMyInterface,
string)

            // A has a method that matches the signature of the following
call
            // to MethodB.
```

```

    a.MethodB();           // A.MethodB()

    // B has methods that match the signatures of the following
    // method calls.
    b.MethodA(1);         // B.MethodA(int)
    b.MethodB();          // B.MethodB()

    // B has no matching method for the following call, but
    // class Extension does.
    b.MethodA("hello");   // Extension.MethodA(IMyInterface,
string)

    // C contains an instance method that matches each of the
following
    // method calls.
    c.MethodA(1);         // C.MethodA(object)
    c.MethodA("hello");   // C.MethodA(object)
    c.MethodB();          // C.MethodB()
}
}
}

/* Output:
Extension.MethodA(this IMyInterface myInterface, int i)
Extension.MethodA(this IMyInterface myInterface, string s)
A.MethodB()
B.MethodA(int i)
B.MethodB()
Extension.MethodA(this IMyInterface myInterface, string s)
C.MethodA(object obj)
C.MethodA(object obj)
C.MethodB()
*/

```

## 일반적인 사용 패턴

### 컬렉션 기능

과거에는 지정된 형식에 대한 [System.Collections.Generic.IEnumerable<T>](#) 인터페이스를 구현하고 해당 형식의 컬렉션에 작동하는 기능을 포함하는 "컬렉션 클래스"를 만드는 것이 일반적이었습니다. 이 형식의 컬렉션 개체를 만들어도 아무런 문제가 없지만 [System.Collections.Generic.IEnumerable<T>](#) 확장을 사용하여 동일한 기능을 구현할 수 있습니다. 확장은 해당 형식에 대한 [System.Collections.Generic.IEnumerable<T>](#)를 구현하는 [System.Array](#) 또는 [System.Collections.Generic.List<T>](#) 같은 모든 컬렉션에서 기능을 호출할 수 있다는 장점이 있습니다. Int32 배열을 사용하는 해당 예제는 [이 문서의 앞부분](#)에 있습니다.

### 레이어 관련 기능

양파형 아키텍처 또는 다른 계층화된 애플리케이션 디자인을 사용하는 경우 애플리케이션 경계에서 통신하는 데 사용할 수 있는 도메인 엔터티 또는 데이터 전송 개체의 집합을 사용하는 것이 일반적입니다. 이러한 개체는 일반적으로 아무 기능이 없거나 애플리케이션의 모든 레이어에 적용되는 기능만 포함합니다. 확장 메서드를 사용하여 다른 레이어에서 필요하지 않은 메서드를 사용하여 개체를 로드하지 않고 각 애플리케이션 레이어와 관련된 기능을 추가할 수 있습니다.

C#

```
public class DomainEntity
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

static class DomainEntityExtensions
{
    static string FullName(this DomainEntity value)
        => $"{value.FirstName} {value.LastName}";
}
```

## 미리 정의된 형식 확장

재사용 가능한 기능을 만들어야 할 때 새 개체를 만드는 대신 기존 형식(예: .NET 또는 CLR 형식)을 확장하는 경우가 많습니다. 예를 들어 확장 메서드를 사용하지 않는 경우 SQL Server에 대해 쿼리를 실행하는 작업을 수행하기 위해 코드의 여러 위치에서 호출될 수 있는 `Engine` 또는 `Query` 클래스를 만들 수 있습니다. 그러나 대신 확장 메서드를 사용하여 `System.Data.SqlClient.SqlConnection` 클래스를 확장하면 SQL Server에 연결된 모든 위치에서 해당 쿼리를 수행할 수 있습니다. 다른 예로는 `System.String` 클래스에 공통 기능 추가, `System.IO.Stream` 개체의 데이터 처리 기능 확장, 특정 오류 처리 기능을 위한 `System.Exception` 개체를 들 수 있습니다. 이러한 사용 사례 유형은 상상력과 판단력에 의해서만 제한됩니다.

미리 정의된 형식의 확장은 메서드에 값으로 전달되는 `struct` 형식에는 사용하기 어려울 수 있습니다. 구조체의 모든 변경 내용이 구조체의 복사본에 적용되기 때문입니다. 이러한 변경 내용은 확장 메서드가 만들어진 이후에는 표시되지 않습니다. 첫 번째 인수에 `ref` 한정자를 추가하여 `ref` 확장 메서드로 만들 수 있습니다. `ref` 키워드는 의미 체계상의 차이 없이 `this` 키워드 앞이나 뒤에 나타날 수 있습니다. `ref` 한정자를 추가하면 첫 번째 인수가 참조로 전달됨을 나타냅니다. 이렇게 하면 확장 중인 구조체의 상태를 변경하는 확장 메서드를 작성할 수 있습니다(프라이빗 멤버에 액세스할 수 없음). 구조체로 제한된 값 형식 또는 제네릭 형식(자세한 내용은 [struct 제약 조건](#) 참조)만 `ref` 확장 메서드의 첫 번째 매개 변수로 허용됩니다. 다음 예에서는 결과를 다시 할당하거나 `ref` 키워드

가 있는 함수를 통해 전달할 필요 없이 `ref` 확장 메서드를 사용하여 기본 제공 형식을 직접 수정하는 방법을 보여 줍니다.

C#

```
public static class IntExtensions
{
    public static void Increment(this int number)
        => number++;

    // Take note of the extra ref keyword here
    public static void RefIncrement(this ref int number)
        => number++;

}

public static class IntProgram
{
    public static void Test()
    {
        int x = 1;

        // Takes x by value leading to the extension method
        // Increment modifying its own copy, leaving x unchanged
        x.Increment();
        Console.WriteLine($"x is now {x}"); // x is now 1

        // Takes x by reference leading to the extension method
        // RefIncrement changing the value of x directly
        x.RefIncrement();
        Console.WriteLine($"x is now {x}"); // x is now 2
    }
}
```

다음 예에서는 사용자 정의 구조체 형식에 대한 `ref` 확장 메서드를 보여 줍니다.

C#

```
public struct Account
{
    public uint id;
    public float balance;

    private int secret;
}

public static class AccountExtensions
{
    // ref keyword can also appear before the this keyword
    public static void Deposit(ref this Account account, float amount)
    {
        account.balance += amount;
    }
}
```

```

        // The following line results in an error as an extension
        // method is not allowed to access private members
        // account.secret = 1; // CS0122
    }

}

public static class AccountProgram
{
    public static void Test()
    {
        Account account = new()
        {
            id = 1,
            balance = 100f
        };

        Console.WriteLine($"I have ${account.balance}"); // I have $100

        account.Deposit(50f);
        Console.WriteLine($"I have ${account.balance}"); // I have $150
    }
}

```

## 일반 지침

개체의 코드를 수정하거나 적절하고 가능할 때마다 새 형식을 파생하는 것을 여전히 선호할 수 있지만 확장 메서드는 .NET 에코시스템 전체에서 재사용 가능한 기능을 만들기 위한 중요한 옵션이 됩니다. 사용자가 원래 소스를 제어하지 않는 경우, 파생 개체가 부적절하거나 불가능한 경우 또는 기능을 적용 가능한 범위 이상으로 노출하지 않아야 하는 경우에는 확장 메서드를 선택하는 것이 좋습니다.

파생 형식에 대한 자세한 내용은 [상속](#)을 참조하세요.

기존 메서드를 사용하여 소스 코드를 제어할 수 없는 형식을 확장하는 경우 형식의 구현이 변경되어 확장명 메서드가 손상될 수도 있습니다.

지정된 형식에 대해 확장 메서드를 구현하는 경우 다음 사항에 유의하세요.

- 형식에 정의된 메서드와 동일한 시그니처가 있는 경우 확장 메서드가 호출되지 않습니다.
- 확장 메서드는 네임스페이스 수준에서 범위로 가져옵니다. 예를 들어 `Extensions`라는 단일 네임스페이스에 확장 메서드를 포함하는 여러 개의 정적 클래스가 있는 경우 `using Extensions;` 지시문을 통해 모두 범위로 가져옵니다.

구현된 클래스 라이브러리의 경우 어셈블리의 버전 번호가 증가되는 것을 방지하기 위해 확장 메서드를 사용해서는 안 됩니다. 소스 코드를 소유하고 있는 라이브러리에 중요 기

능을 추가하려는 경우 어셈블리 버전 관리를 위한 .NET 지침을 따르세요. 자세한 내용은 어셈블리 버전 관리를 참조하세요.

## 참고 항목

- 병렬 프로그래밍 샘플(많은 예제 확장 메서드 포함)
- 람다 식
- 표준 쿼리 연산자 개요
- 인스턴스 매개 변수의 변환 규칙 및 그에 따른 영향
- 언어 간 확장 메서드 상호 운용성
- 확장 메서드 및 대리자 변환
- 확장 메서드 바인딩 및 오류 보고

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 사용자 지정 확장명 메서드 구현 및 호출 방법(C# 프로그래밍 가이드)

아티클 • 2023. 05. 10.

이 항목에서는 모든 .NET 형식에 대한 사용자 고유의 확장 메서드를 구현하는 방법을 보여 줍니다. 클라이언트 코드는 확장 메서드를 포함하는 DLL에 대한 참조를 추가하고 확장 메서드가 정의된 네임스페이스를 지정하는 `using` 지시문을 추가하여 확장 메서드를 사용할 수 있습니다.

## 확장 메서드를 정의하고 호출하려면

1. 확장 메서드가 포함될 정적 [클래스](#)를 정의합니다.

클래스가 클라이언트 코드에 표시되어야 합니다. 액세스 가능성 규칙에 대한 자세한 내용은 [액세스 한정자](#)를 참조하세요.

2. 최소한 포함하는 클래스와 동일한 표시 유형으로 확장 메서드를 정적 메서드로 구현합니다.
3. 메서드의 첫 번째 매개 변수는 메서드가 작동하는 형식을 지정합니다. `this` 한정자가 앞에 와야 합니다.
4. 호출 코드에 `using` 지시문을 추가하여 확장 메서드 클래스를 포함하는 [네임스페이스](#)를 지정합니다.
5. 형식의 인스턴스 메서드인 것처럼 메서드를 호출합니다.

연산자가 적용되는 형식을 나타내기 때문에 첫 번째 매개 변수는 호출 코드에서 지정되지 않고 컴파일러가 개체 형식을 이미 알고 있습니다. 매개 변수 2 ~ `n`에 대한 인수만 제공하면 됩니다.

## 예제

다음 예제에서는 `CustomExtensions.StringExtension` 클래스에 `WordCount`라는 확장 메서드를 구현합니다. 이 메서드는 첫 번째 매개 변수로 지정된 `String` 클래스에 대해 작동합니다. `CustomExtensions` 네임스페이스를 애플리케이션 네임스페이스로 가져오고, `Main` 메서드 내부에서 메서드가 호출됩니다.

C#

```

using System.Linq;
using System.Text;
using System;

namespace CustomExtensions
{
    // Extension methods must be defined in a static class.
    public static class StringExtension
    {
        // This is the extension method.
        // The first parameter takes the "this" modifier
        // and specifies the type for which the method is defined.
        public static int WordCount(this string str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}

namespace Extension_Methods_Simple
{
    // Import the extension method namespace.
    using CustomExtensions;
    class Program
    {
        static void Main(string[] args)
        {
            string s = "The quick brown fox jumped over the lazy dog.";
            // Call the method as if it were an
            // instance method on the type. Note that the first
            // parameter is not specified by the calling code.
            int i = s.WordCount();
            System.Console.WriteLine("Word count of s is {0}", i);
        }
    }
}

```

## .NET 보안

확장 메서드는 특정 보안 취약성은 없습니다. 형식 자체에서 정의된 인스턴스 또는 정적 메서드를 기준으로 모든 이름 충돌이 해결되기 때문에 확장 메서드를 사용하여 형식의 기존 메서드를 가장할 수는 없습니다. 확장 메서드는 확장된 클래스의 개인 데이터에 액세스할 수 없습니다.

## 참조

- C# 프로그래밍 가이드
- 확장명 메서드

- LINQ(Language-Integrated Query)
- 정적 클래스 및 정적 클래스 멤버
- protected
- internal
- public
- this
- namespace

# 새 열거형 메서드를 만드는 방법(C# 프로그래밍 가이드)

아티클 • 2024. 03. 12.

확장 메서드를 사용하여 특정 열거형 형식과 관련된 기능을 추가할 수 있습니다.

## 예시

다음 예제에서 `Grades` 열거형은 학생이 클래스에서 받을 수 있는 문자 성적을 나타냅니다. 해당 형식의 각 인스턴스가 이제 합격 성적을 나타내는지 여부를 "알 수 있도록" `Passing`이라는 확장 메서드가 `Grades` 형식에 추가됩니다.

C#

```
using System;

namespace EnumExtension
{
    // Define an extension method in a non-nested static class.
    public static class Extensions
    {
        public static Grades minPassing = Grades.D;
        public static bool Passing(this Grades grade)
        {
            return grade >= minPassing;
        }
    }

    public enum Grades { F = 0, D=1, C=2, B=3, A=4 };
    class Program
    {
        static void Main(string[] args)
        {
            Grades g1 = Grades.D;
            Grades g2 = Grades.F;
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is not");

            Extensions.minPassing = Grades.C;
            Console.WriteLine("\r\nRaising the bar!\r\n");
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is not");
        }
    }
}
```

```
}

/* Output:
   First is a passing grade.
   Second is not a passing grade.

   Raising the bar!

   First is not a passing grade.
   Second is not a passing grade.
*/
```

또한 `Extensions` 클래스에는 동적으로 업데이트되는 정적 변수가 포함되며, 확장 메서드의 반환 값이 해당 변수의 현재 값을 반영합니다. 이는 확장 메서드가 정의된 정적 클래스에서 내부적으로 직접 호출됨을 보여 줍니다.

## 참고 항목

- [확장명 메서드](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 명명된 인수와 선택적 인수(C# 프로그래밍 가이드)

아티클 • 2024. 03. 20.

명명된 인수를 사용하면 인수를 매개 변수 목록 내의 해당 위치가 아닌 해당 이름과 일치시켜 매개 변수에 대한 인수를 지정할 수 있습니다. 선택적 인수를 사용하면 일부 매개 변수에 대한 인수를 생략할 수 있습니다. 두 기법 모두 메서드, 인덱서, 생성자 및 대리자에 사용할 수 있습니다.

명명된 인수와 선택적 인수를 사용하는 경우 매개 변수 목록이 아니라 인수 목록에 표시되는 순서대로 인수가 평가됩니다.

명명된 매개 변수와 선택적 매개 변수를 사용하여 선택한 매개 변수에 대한 인수를 제공할 수 있습니다. 이 기능 덕분에 Microsoft Office 자동화 API와 같은 COM 인터페이스에 대한 호출이 매우 간단해집니다.

## 명명된 인수

명명된 인수를 사용하면 인수 순서를 호출된 메서드의 매개 변수 목록에 있는 매개 변수 순서와 일치시킬 필요가 없습니다. 각 매개 변수의 인수는 매개 변수 이름으로 지정할 수 있습니다. 예를 들어 함수에 정의된 순서의 위치로 인수를 보내서 주문 세부 정보(예: 판매자 이름, 주문 번호 및 제품 이름)를 호출할 수 있습니다.

C#

```
PrintOrderDetails("Gift Shop", 31, "Red Mug");
```

매개 변수의 순서를 기억하지 못하지만 해당 이름을 알고 있는 경우 임의의 순서로 인수를 보낼 수 있습니다.

C#

```
PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);
```

또한 명명된 인수는 각 인수가 무엇을 나타내는지를 식별하여 코드의 가독성을 향상합니다. 아래 예제 메서드에서 `sellerName`은 null 또는 공백일 수 없습니다. `sellerName` 및 `productName`은 모두 문자열 형식이므로, 위치로 인수를 보내는 대신, 명명된 인수를 사용하여 두 코드를 구분하고 코드를 읽는 사람의 혼동을 줄일 수 있습니다.

위치 인수와 함께 사용된 명명된 인수는

- 그 다음에 위치 인수가 없거나

C#

```
PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
```

- 올바른 위치에 사용됩니다. 아래 예제에서 `orderNum` 매개 변수는 올바른 위치에 있지만 명시적으로 명명되지 않습니다.

C#

```
PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");
```

잘못된 순서의 명명된 인수 다음에 오는 위치 인수는 유효하지 않습니다.

C#

```
// This generates CS1738: Named argument specifications must appear after  
// all fixed arguments have been specified.  
PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
```

## 예시

다음 코드는 몇 가지 추가 코드와 함께 이 섹션의 예제를 구현합니다.

C#

```
class NamedExample  
{  
    static void Main(string[] args)  
    {  
        // The method can be called in the normal way, by using positional  
        // arguments.  
        PrintOrderDetails("Gift Shop", 31, "Red Mug");  
  
        // Named arguments can be supplied for the parameters in any order.  
        PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName:  
        "Gift Shop");  
        PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop",  
        orderNum: 31);  
  
        // Named arguments mixed with positional arguments are valid  
        // as long as they are used in their correct position.  
        PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");  
        PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red
```

```

    "Mug");
    PrintOrderDetails("Gift Shop", orderNum: 31, "Red Mug");

        // However, mixed arguments are invalid if used out-of-order.
        // The following statements will cause a compiler error.
        // PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
        // PrintOrderDetails(31, sellerName: "Gift Shop", "Red Mug");
        // PrintOrderDetails(31, "Red Mug", sellerName: "Gift Shop");
}

static void PrintOrderDetails(string sellerName, int orderNum, string
productName)
{
    if (string.IsNullOrWhiteSpace(sellerName))
    {
        throw new ArgumentException(message: "Seller name cannot be null
or empty.", paramName: nameof(sellerName));
    }

    Console.WriteLine($"Seller: {sellerName}, Order #: {orderNum},
Product: {productName}");
}
}

```

## 선택형 인수

메서드, 생성자, 인덱서 또는 대리자의 정의에서 해당 매개 변수를 필수 또는 선택 사항으로 지정할 수 있습니다. 호출 시 모든 필수 매개 변수에 대한 인수를 제공해야 하지만 선택적 매개 변수에 대한 인수는 생략할 수 있습니다.

각 선택적 매개 변수에는 해당 정의의 일부로 기본값이 있습니다. 해당 매개 변수에 대한 인수가 전송되지 않은 경우 기본값이 사용됩니다. 기본값은 다음 유형의 식 중 하나여야 합니다.

- 상수 식
- `new ValType()` 형태의 식. 여기서 `ValType`은 `enum` 또는 `struct`와 같은 값 형식입니다.
- `default(ValType)` 형태의 식. 여기서 `ValType`은 값 형식입니다.

선택적 매개 변수는 매개 변수 목록의 끝에서 모든 필수 매개 변수 다음에 정의됩니다. 호출자가 연속된 선택적 매개 변수 중 하나에 대한 인수를 제공하는 경우 이전의 모든 선택적 매개 변수에 대한 인수를 제공해야 합니다. 인수 목록에서 쉼표로 구분된 간격은 지원되지 않습니다. 예를 들어 다음 코드에서 인스턴스 메서드 `ExampleMethod`는 필수 매개 변수 하나와 선택적 매개 변수 두 개로 정의됩니다.

```
public void ExampleMethod(int required, string optionalstr = "default string",
    int optionalint = 10)
```

다음과 같은 `ExampleMethod` 호출에서는 세 번째 매개 변수에 대한 인수가 제공되었지만 두 번째 매개 변수에 대한 인수는 제공되지 않았기 때문에 컴파일러 오류가 발생합니다.

C#

```
//anExample.ExampleMethod(3, ,4);
```

그러나 세 번째 매개 변수의 이름을 알고 있으면 명명된 인수를 사용하여 작업을 수행할 수 있습니다.

C#

```
anExample.ExampleMethod(3, optionalint: 4);
```

IntelliSense는 다음 그림과 같이 대괄호를 사용하여 선택적 매개 변수를 나타냅니다.

```
anExample.ExampleMethod(
    void ExampleClass.ExampleMethod(int required,
        [string optionalstr = "default string"],
        [int optionalint = 10])
```

## ① 참고

.NET [OptionalAttribute](#) 클래스를 사용하여 선택적 매개 변수를 선언할 수도 있습니다. `OptionalAttribute` 매개 변수는 기본값이 필요하지 않습니다. 그러나 기본값을 원할 경우 [DefaultParameterValueAttribute](#) 클래스를 살펴봅니다.

## 예시

다음 예제에서는 `ExampleClass`에 대한 생성자에 선택 사항인 매개 변수 하나가 있습니다. 인스턴스 메서드 `ExampleMethod`에는 `required`라는 필수 매개 변수 하나와 `optionalstr` 및 `optionalint`라는 선택적 매개 변수 두 개가 있습니다. `Main`의 코드는 생성자와 메서드를 호출할 수 있는 여러 방법을 보여 줍니다.

C#

```
namespace OptionalNamespace
{
```

```
class OptionalExample
{
    static void Main(string[] args)
    {
        // Instance anExample does not send an argument for the
constructor's
        // optional parameter.
        ExampleClass anExample = new ExampleClass();
        anExample.ExampleMethod(1, "One", 1);
        anExample.ExampleMethod(2, "Two");
        anExample.ExampleMethod(3);

        // Instance anotherExample sends an argument for the
constructor's
        // optional parameter.
        ExampleClass anotherExample = new ExampleClass("Provided name");
        anotherExample.ExampleMethod(1, "One", 1);
        anotherExample.ExampleMethod(2, "Two");
        anotherExample.ExampleMethod(3);

        // The following statements produce compiler errors.

        // An argument must be supplied for the first parameter, and it
        // must be an integer.
        //anExample.ExampleMethod("One", 1);
        //anExample.ExampleMethod();

        // You cannot leave a gap in the provided arguments.
        //anExample.ExampleMethod(3, ,4);
        //anExample.ExampleMethod(3, 4);

        // You can use a named parameter to make the previous
        // statement work.
        anExample.ExampleMethod(3, optionalint: 4);
    }
}

class ExampleClass
{
    private string _name;

    // Because the parameter for the constructor, name, has a default
    // value assigned to it, it is optional.
    public ExampleClass(string name = "Default name")
    {
        _name = name;
    }

    // The first parameter, required, has no default value assigned
    // to it. Therefore, it is not optional. Both optionalstr and
    // optionalint have default values assigned to them. They are
optional.
    public void ExampleMethod(int required, string optionalstr =
"default string",
                            int optionalint = 10)
```

```

    {
        Console.WriteLine(
            $"({_name}): {required}, {optionalstr}, and {optionalint}.");
    }
}

// The output from this example is the following:
// Default name: 1, One, and 1.
// Default name: 2, Two, and 10.
// Default name: 3, default string, and 10.
// Provided name: 1, One, and 1.
// Provided name: 2, Two, and 10.
// Provided name: 3, default string, and 10.
// Default name: 3, default string, and 4.
}

```

위의 코드는 선택적 매개 변수가 올바르게 적용되지 않는 몇 가지 예제를 보여 줍니다. 첫 번째 예제는 필수 항목인 첫 번째 매개 변수에 인수를 제공해야 함을 보여 줍니다.

## 호출자 정보 특성

**호출자 정보 특성**(예: [CallerFilePathAttribute](#), [CallerLineNumberAttribute](#), [CallerMemberNameAttribute](#) 및 [CallerArgumentExpressionAttribute](#))은 메서드에 대한 호출자에 대한 정보를 가져오는 데 사용됩니다. 이러한 특성은 디버깅 중이거나 메서드 호출에 대한 정보를 기록해야 할 때 특히 유용합니다.

이러한 특성은 컴파일러에서 제공하는 기본값을 사용하는 선택적 매개 변수입니다. 호출자는 이러한 매개 변수에 대한 값을 명시적으로 제공하지 않아야 합니다.

## COM 인터페이스

동적 개체에 대한 지원과 더불어 명명된 인수와 선택적 인수는 Office 자동화 API와 같은 COM API와의 상호 운용성을 크게 향상합니다.

예를 들어 [AutoFormat](#) Microsoft Office Excel [Range](#) 인터페이스의 메서드에는 모두 선택 사항인 매개 변수 7개가 있습니다. 해당 매개 변수는 다음 그림에 표시됩니다.

```

excelApp.get_Range("A1", "B4").AutoFormat(
    dynamic Range.AutoFormat([Excel.XlRangeAutoFormat Format = 1],
    [object Number = Type.Missing], [object Font = Type.Missing],
    [object Alignment = Type.Missing], [object Border = Type.Missing],
    [object Pattern = Type.Missing], [object Width = Type.Missing])
)

```

그러나 명명된 인수와 선택적 인수를 사용하면 [AutoFormat](#)에 대한 호출을 크게 간소화할 수 있습니다. 명명된 인수와 선택적 인수를 사용하면 매개 변수의 기본값을 변경하지 않

으려는 경우 선택적 매개 변수에 대한 인수를 생략할 수 있습니다. 다음 호출에서는 7개 매개 변수 중 하나에 대한 값만 지정되었습니다.

C#

```
var excelApp = new Microsoft.Office.Interop.Excel.Application();
excelApp.Workbooks.Add();
excelApp.Visible = true;

var myFormat =
    Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFormatAccounting
    1;

excelApp.Range["A1", "B4"].AutoFormat( Format: myFormat );
```

자세한 내용 및 예제는 [Office 프로그래밍에 명명된 인수와 선택적 인수를 사용하는 방법](#) 및 [C# 기능을 사용하여 Office interop 개체에 액세스하는 방법](#)을 참조하세요.

## 오버로드 해결

명명된 인수 및 선택적 인수를 사용하면 다음과 같은 방법으로 오버로드 확인에 영향을 줍니다.

- 메서드, 인덱서 또는 생성자는 해당 매개 변수가 각각 선택 사항이거나 이름 또는 위치로 호출하는 문의 단일 인수에 해당하고 이 인수를 매개 변수의 형식으로 변환할 수 있는 경우 실행 후보가 됩니다.
- 둘 이상의 인증서가 있으면 기본 설정 변환에 대한 오버로드 확인 규칙이 명시적으로 지정된 인수에 적용됩니다. 선택적 매개 변수에 대해 생략된 인수는 무시됩니다.
- 두 후보가 똑같이 정상이라고 판단되는 경우 기본적으로 호출에서 인수가 생략된 선택적 매개 변수가 없는 후보가 설정됩니다. 오버로드 확인은 일반적으로 매개 변수가 더 적은 후보를 선호합니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)을 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.



GitHub에서 Microsoft와 공동 작업



.NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

제공해 주세요.

💡 설명서 문제 열기

↗️ 제품 사용자 의견 제공

# 생성자(C# 프로그래밍 가이드)

아티클 • 2024. 03. 10.

클래스 또는 구조체의 인스턴스가 만들어질 때마다 해당 생성자가 호출됩니다. 클래스 또는 구조체에는 서로 다른 인수를 사용하는 여러 생성자가 있을 수 있습니다. 프로그래머는 생성자를 통해 기본값을 설정하고, 인스턴스화를 제한하며, 유연하고 읽기 쉬운 코드를 작성할 수 있습니다. 자세한 내용 및 예제는 [인스턴스 생성자](#) 및 [생성자 사용](#)을 참조하세요.

새 인스턴스 초기화에는 몇 가지 작업이 포함됩니다. 해당 작업은 다음 순서로 수행됩니다.

1. 인스턴스 필드는 0으로 설정됩니다. 이는 일반적으로 런타임에 의해 수행됩니다.
2. 필드 이니셜라이저가 실행됩니다. 가장 많이 파생된 형식의 필드 이니셜라이저가 실행됩니다.
3. 기본 형식 필드 이니셜라이저가 실행됩니다. 직접 베이스로 시작하여 각 기본 형식을 거쳐 `System.Object`까지의 필드 이니셜라이저입니다.
4. 기본 인스턴스 생성자가 실행됩니다. `Object.Object`로 시작하여 각 기본 클래스를 거쳐 직접 기본 클래스에 이르는 모든 인스턴스 생성자입니다.
5. 인스턴스 생성자가 실행됩니다. 해당 형식의 인스턴스 생성자가 실행됩니다.
6. 개체 이니셜라이저가 실행됩니다. 식에 개체 이니셜라이저가 포함된 경우 해당 이니셜라이저는 인스턴스 생성자가 실행된 후에 실행됩니다. 개체 이니셜라이저는 텍스트 순서로 실행됩니다.

이전 작업은 새 인스턴스가 초기화될 때 발생합니다. `struct`의 새 인스턴스가 `default` 값으로 설정되면 모든 인스턴스 필드가 0으로 설정됩니다.

정적 생성자가 실행되지 않은 경우 인스턴스 생성자 작업이 발생하기 전에 정적 생성자가 실행됩니다.

## 생성자 구문

생성자는 이름이 해당 형식의 이름과 동일한 메서드입니다. 메서드 서명에는 선택적 [액세스 한정자](#), 메서드 이름 및 매개 변수 목록만 포함됩니다. 반환 형식은 포함되지 않습니다. 다음 예제에서는 `Person`이라는 클래스에 대한 생성자를 보여 줍니다.

C#

```
public class Person
{
    private string last;
    private string first;
```

```

public Person(string lastName, string firstName)
{
    last = lastName;
    first = firstName;
}

// Remaining implementation of Person class.
}

```

생성자를 단일 문으로 구현할 수 있는 경우 [식 본문 정의](#)를 사용할 수 있습니다. 다음 예제에서는 생성자에 *name*이라는 단일 문자열 매개 변수가 있는 `Location` 클래스를 정의합니다. 식 본문 정의에서 `locationName` 필드에 인수를 할당합니다.

C#

```

public class Location
{
    private string locationName;

    public Location(string name) => Name = name;

    public string Name
    {
        get => locationName;
        set => locationName = value;
    }
}

```

## 정적 생성자

앞의 예제에서는 새 개체를 만드는 인스턴스 생성자를 모두 보여 주었습니다. 클래스 또는 구조체에 형식의 정적 멤버를 초기화하는 정적 생성자도 있을 수 있습니다. 정적 생성자에는 매개 변수가 없습니다. 정적 필드를 초기화하는 정적 생성자를 제공하지 않으면 C# 컴파일러는 정적 필드를 [C# 형식의 기본값](#)에 나열된 기본값으로 초기화합니다.

다음 예제에서는 정적 생성자를 사용하여 정적 필드를 초기화합니다.

C#

```

public class Adult : Person
{
    private static int minimumAge;

    public Adult(string lastName, string firstName) : base(lastName,
firstName)
    { }
}

```

```
    static Adult()
    {
        minimumAge = 18;
    }

    // Remaining implementation of Adult class.
}
```

다음 예제와 같이 식 본문 정의를 사용하여 정적 생성자를 정의할 수도 있습니다.

C#

```
public class Child : Person
{
    private static int maximumAge;

    public Child(string lastName, string firstName) : base(lastName,
firstName)
    { }

    static Child() => maximumAge = 18;

    // Remaining implementation of Child class.
}
```

자세한 내용 및 예제는 [정적 생성자를 참조하세요](#).

## 섹션 내용

- 생성자 사용
- 인스턴스 생성자
- 전용 생성자
- 정적 생성자
- 복사 생성자 작성 방법

## 참고 항목

- C# 형식 시스템
- 종료자
- static
- 이니셜라이저가 생성자와 반대 순서로 실행되는 이유는 무엇인가요? 1부

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 생성자 사용(C# 프로그래밍 가이드)

아티클 • 2023. 05. 26.

클래스 또는 구조체가 인스턴스화되면 해당 생성자가 호출됩니다. 생성자는 클래스 또는 구조체와 이름이 같으며 일반적으로 새 개체의 데이터 멤버를 초기화합니다.

다음 예제에서는 간단한 생성자를 사용하여 `Taxi`란 이름의 클래스를 정의합니다. 그런 다음 `new` 연산자를 사용하여 이 클래스를 인스턴스화합니다. 새 개체에 메모리가 할당된 직후 `new` 연산자가 `Taxi` 생성자를 호출합니다.

C#

```
public class Taxi
{
    public bool IsInitialized;

    public Taxi()
    {
        IsInitialized = true;
    }
}

class TestTaxi
{
    static void Main()
    {
        Taxi t = new Taxi();
        Console.WriteLine(t.IsInitialized);
    }
}
```

매개 변수가 없는 생성자를 **매개 변수 없는 생성자**라고 합니다. `new` 연산자를 사용하여 개체가 인스턴스화되고 `new`에 제공된 인수가 없을 때마다 매개 변수가 없는 생성자가 호출됩니다. C# 12에는 기본 생성자가 도입되었습니다. 기본 생성자는 새 개체를 초기화하기 위해 제공해야 하는 매개 변수를 지정합니다. 자세한 내용은 [인스턴스 생성자](#)를 참조하세요.

클래스가 **정적**이 아닌 경우 생성자가 없는 클래스에는 클래스 인스턴스화를 사용할 수 있도록 C# 컴파일러에서 공용 매개 변수 없는 생성자가 제공됩니다. 자세한 내용은 [static 클래스 및 static 클래스 멤버](#)를 참조하세요.

다음과 같이 생성자를 비공개로 설정하여 클래스의 인스턴스화를 방지할 수 있습니다.

C#

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E; //2.71828...
}
```

자세한 내용은 [전용 생성자를](#) 참조하세요.

**구조체** 형식의 생성자는 클래스 생성자와 유사합니다. 구조체 형식이 `new`로 인스턴스화 되면 생성자가 호출됩니다. 가 `struct` 해당 `default` 값으로 설정되면 런타임은 구조체의 모든 메모리를 0으로 초기화합니다. C# 10 `structs` 이전에는 컴파일러에서 자동으로 제공되므로 명시적 매개 변수 없는 생성자를 포함할 수 없습니다. 자세한 내용은 [구조체 형식](#) 문서의 [구조체 초기화 및 기본값](#) 섹션을 참조하세요.

다음 코드에서는 예 매개 `Int32` 변수가 없는 생성자를 사용하여 정수가 초기화되도록 합니다.

C#

```
int i = new int();
Console.WriteLine(i);
```

그러나 다음 코드는 `new`를 사용하지 않고 초기화되지 않은 개체를 사용하려고 하기 때문에 컴파일러 오류를 발생합니다.

C#

```
int i;
Console.WriteLine(i);
```

또는 `structs` 기반의 개체(모든 기본 제공 숫자 형식 포함)를 초기화하거나 할당한 후 다음 예제와 같이 사용할 수 있습니다.

C#

```
int a = 44; // Initialize the value type...
int b;
b = 33;      // Or assign it before using it.
Console.WriteLine("{0}, {1}", a, b);
```

클래스와 구조체는 모두 기본 생성자를 포함하여 매개 변수를 사용하는 [생성자를](#) 정의할 수 있습니다. 매개 변수를 사용하는 생성자는 `new` 문 또는 `base` 문을 통해 호출해야 합니다.

다. 클래스 및 구조체는 여러 생성자를 정의할 수도 있으며 매개 변수가 없는 생성자를 정의하는 데 둘 다 필요하지 않습니다. 예를 들어:

C#

```
public class Employee
{
    public int Salary;

    public Employee() { }

    public Employee(int annualSalary)
    {
        Salary = annualSalary;
    }

    public Employee(int weeklySalary, int numberOfWeeks)
    {
        Salary = weeklySalary * numberOfWeeks;
    }
}
```

다음 문 중 하나를 사용하여 이 클래스를 만들 수 있습니다.

C#

```
Employee e1 = new Employee(30000);
Employee e2 = new Employee(500, 52);
```

생성자는 `base` 키워드를 사용하여 기본 클래스의 생성자를 호출할 수 있습니다. 예를 들어:

C#

```
public class Manager : Employee
{
    public Manager(int annualSalary)
        : base(annualSalary)
    {
        //Add further instructions here.
    }
}
```

이 예제에서는 생성자의 블록이 실행되기 전에 기본 클래스의 생성자가 호출됩니다.

`base` 키워드는 매개 변수와 함께 또는 매개 변수 없이 사용할 수 있습니다. 생성자에 대한 매개 변수는 `base`에 대한 매개 변수로 또는 식의 일부로 사용할 수 있습니다. 자세한 내용은 [base](#)를 참조하세요.

파생 클래스에서 기본 클래스 생성자가 키워드(keyword) 사용하여 `base` 명시적으로 호출되지 않으면 매개 변수가 없는 생성자가 있는 경우 암시적으로 호출됩니다. 다음 생성자 선언은 사실상 동일합니다.

C#

```
public Manager(int initialData)
{
    //Add further instructions here.
}
```

C#

```
public Manager(int initialData)
    : base()
{
    //Add further instructions here.
}
```

기본 클래스가 매개 변수 없는 생성자를 제공하지 않는 경우 파생 클래스는 를 사용하여 `base` 기본 생성자를 명시적으로 호출해야 합니다.

생성자는 `this` 키워드를 사용해 동일한 개체의 다른 생성자를 호출할 수 있습니다. `base` 와 마찬가지로 `this`도 매개 변수 유무와 상관없이 사용할 수 있으며, 생성자에 대한 매개 변수는 `this`에 대한 매개 변수로 또는 식의 일부로 사용할 수 있습니다. 예를 들어, 이전 예제의 두 번째 생성자는 `this`를 사용해 다시 작성할 수 있습니다.

C#

```
public Employee(int weeklySalary, int numberOfWeeks)
    : this(weeklySalary * numberOfWeeks)
{ }
```

이전 예제에서 `this` 키워드를 사용하면 이 생성자가 호출됩니다.

C#

```
public Employee(int annualSalary)
{
    Salary = annualSalary;
}
```

생성자는 `public`, `private`, `protected`, `internal`, `protected internal` 또는 `private protected`로 표시될 수 있습니다. 이러한 액세스 한정자는 클래스의 사용자가 클래스를 생성하는 방법

을 정의합니다. 자세한 내용은 [액세스 한정자](#)를 참조하세요.

`static` 키워드를 사용하여 생성자를 정적으로 선언할 수 있습니다. 정적 생성자는 정적 필드에 액세스하기 직전에 자동으로 호출되며 정적 클래스 멤버를 초기화하는 데 사용됩니다. 자세한 내용은 [정적 생성자](#)를 참조하세요.

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 인스턴스 생성자](#) 및 [정적 생성자](#)를 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [C# 형식 시스템](#)
- [생성자](#)
- [종료자](#)

# 인스턴스 생성자(C# 프로그래밍 가이드)

아티클 • 2024. 05. 29.

`new` 식으로 형식의 새 인스턴스를 만들 때 실행되는 코드를 지정하기 위해 인스턴스 생성자를 선언합니다. 정적 클래스 또는 비정적 클래스의 정적 변수를 초기화하려면 정적 생성자를 정의할 수 있습니다.

다음 예제와 같이 한 형식으로 여러 인스턴스 생성자를 선언할 수 있습니다.

```
C#  
  
class Coords  
{  
    public Coords()  
        : this(0, 0)  
    {    }  
  
    public Coords(int x, int y)  
    {  
        X = x;  
        Y = y;  
    }  
  
    public int X { get; set; }  
    public int Y { get; set; }  
  
    public override string ToString() => $"({X},{Y})";  
}  
  
class Example  
{  
    static void Main()  
    {  
        var p1 = new Coords();  
        Console.WriteLine($"Coords #1 at {p1}");  
        // Output: Coords #1 at (0,0)  
  
        var p2 = new Coords(5, 3);  
        Console.WriteLine($"Coords #2 at {p2}");  
        // Output: Coords #2 at (5,3)  
    }  
}
```

앞의 예제에서 첫 번째 매개 변수가 없는 생성자는 두 인수가 모두 `0`인 두 번째 생성자를 호출합니다. 이를 위해 `this` 키워드를 사용합니다.

파생 클래스에서 인스턴스 생성자를 선언하는 경우 기본 클래스의 생성자를 호출할 수 있습니다. 이렇게 하려면 다음 예제와 같이 `base` 키워드를 사용합니다.

C#

```
abstract class Shape
{
    public const double pi = Math.PI;
    protected double x, y;

    public Shape(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    public abstract double Area();
}

class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    { }

    public override double Area() => pi * x * x;
}

class Cylinder : Circle
{
    public Cylinder(double radius, double height)
        : base(radius)
    {
        y = height;
    }

    public override double Area() => (2 * base.Area()) + (2 * pi * x * y);
}

class Example
{
    static void Main()
    {
        double radius = 2.5;
        double height = 3.0;

        var ring = new Circle(radius);
        Console.WriteLine($"Area of the circle = {ring.Area():F2}");
        // Output: Area of the circle = 19.63

        var tube = new Cylinder(radius, height);
        Console.WriteLine($"Area of the cylinder = {tube.Area():F2}");
        // Output: Area of the cylinder = 86.39
    }
}
```

# 매개 변수 없는 생성자

'클래스'에 명시적 인스턴스 생성자가 없는 경우 C#은 다음 예제와 같이 해당 클래스의 인스턴스를 인스턴스화하기 위해 사용할 수 있는 매개 변수가 없는 생성자를 제공합니다.

C#

```
public class Person
{
    public int age;
    public string name = "unknown";
}

class Example
{
    static void Main()
    {
        var person = new Person();
        Console.WriteLine($"Name: {person.name}, Age: {person.age}");
        // Output: Name: unknown, Age: 0
    }
}
```

이 생성자는 해당하는 이니셜라이저에 따라 인스턴스 필드와 속성을 초기화합니다. 필드 또는 속성에 이니셜라이저가 없는 경우 해당 값은 필드 또는 속성 형식의 [기본값](#)으로 설정됩니다. 클래스에서 하나 이상의 인스턴스 생성자를 선언하는 경우 C#은 매개 변수가 없는 생성자를 제공하지 않습니다.

구조체 형식은 항상 매개 변수가 없는 생성자를 제공합니다. 매개 변수 없는 생성자는 형식의 기본값을 생성하는 암시적 매개 변수 없는 생성자이거나 명시적으로 선언된 매개 변수 없는 생성자입니다. 자세한 내용은 [구조체 형식](#) 문서의 [구조체 초기화 및 기본값](#) 섹션을 참조하세요.

## 기본 생성자

C# 12부터 클래스 및 구조체에서 기본 생성자를 선언할 수 있습니다. 형식 이름 뒤에 매개 변수를 괄호 안에 넣습니다.

C#

```
public class NamedItem(string name)
{
    public string Name => name;
}
```

기본 생성자에 대한 매개 변수는 선언 형식의 전체 본문 범위에 있습니다. 속성이나 필드를 초기화할 수 있습니다. 메서드나 로컬 함수에서 변수로 사용될 수 있습니다. 기본 생성자에 전달될 수 있습니다.

기본 생성자는 이러한 매개 변수가 해당 형식의 모든 인스턴스에 필요함을 나타냅니다. 명시적으로 작성된 모든 생성자는 기본 생성자를 호출하려면 `this(...)` 이니셜라이저 구문을 사용해야 합니다. 이렇게 하면 기본 생성자 매개 변수가 모든 생성자에 의해 명확히 할당됩니다. `record class` 형식을 포함한 모든 `class` 형식의 경우 기본 생성자가 있으면 암시적 매개 변수 없는 생성자가 생성되지 않습니다. `record struct` 형식을 포함한 모든 `struct` 형식의 경우 암시적 매개 변수 없는 생성자가 항상 내보내지고 기본 생성자 매개 변수를 포함한 모든 필드를 항상 0비트 패턴으로 초기화합니다. 매개 변수가 없는 명시적 생성자를 작성하는 경우 기본 생성자를 호출해야 합니다. 이 경우 기본 생성자 매개 변수에 대해 다른 값을 지정할 수 있습니다. 다음 코드는 기본 생성자의 예를 보여 줍니다.

C#

```
// name isn't captured in Widget.
// width, height, and depth are captured as private fields
public class Widget(string name, int width, int height, int depth) :
    NamedItem(name)
{
    public Widget() : this("N/A", 1,1,1) {} // unnamed unit cube

    public int WidthInCM => width;
    public int HeightInCM => height;
    public int DepthInCM => depth;

    public int Volume => width * height * depth;
}
```

특성에 `[method: MyAttribute]` 대상을 지정하여 합성된 기본 생성자 메서드에 특성을 추가할 수 있습니다.

C#

```
[method: MyAttribute]
public class TaggedWidget(string name)
{
    // details elided
}
```

`[method]` 대상을 지정하지 않으면 특성은 메서드가 아닌 클래스에 배치됩니다.

`class` 및 `struct` 형식에서 기본 생성자 매개 변수는 형식 본문 어디에서나 사용할 수 있습니다. 매개 변수는 캡처된 프라이빗 필드로 구현될 수 있습니다. 매개 변수에 대한 유일한 참조가 이니셜라이저 및 생성자 호출인 경우 해당 매개 변수는 프라이빗 필드에 캡처

되지 않습니다. 해당 형식의 다른 멤버에 사용하면 컴파일러가 전용 필드의 매개 변수를 캡처하게 됩니다.

형식에 `record` 한정자가 포함된 경우 컴파일러는 대신 기본 생성자 매개 변수와 동일한 이름을 가진 공용 속성을 합성합니다. `record class` 형식의 경우 기본 생성자 매개 변수가 기본 생성자와 동일한 이름을 사용하는 경우 해당 속성은 기본 `record class` 형식의 공용 속성입니다. 파생된 `record class` 형식에서는 중복되지 않습니다. 이러한 속성은 `record` 형식이 아닌 경우 생성되지 않습니다.

## 참고 항목

- [클래스, 구조체, 레코드](#)
- [생성자](#)
- [종료자](#)
- [base](#)
- [this](#)
- [기본 생성자 기능 사양](#)

### ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

💡 설명서 문제 열기

↗️ 제품 사용자 의견 제공

# 전용 생성자(C# 프로그래밍 가이드)

아티클 • 2024. 03. 03.

전용 생성자는 특수 인스턴스 생성자입니다. 일반적으로 정적 멤버만 포함하는 클래스에서 사용됩니다. 클래스에 하나 이상의 private 생성자가 있고 public 생성자가 없는 경우 중첩 클래스를 제외한 다른 클래스는 이 클래스의 인스턴스를 만들 수 없습니다. 예시:

```
C#  
  
class NLog  
{  
    // Private Constructor:  
    private NLog() { }  
  
    public static double e = Math.E; //2.71828...  
}
```

빈 생성자를 선언하면 매개 변수 없는 생성자가 자동으로 생성되지 않습니다. 액세스 한정자를 생성자와 함께 사용하지 않는 경우 기본적으로 여전히 private입니다. 그러나 일반적으로 private 한정자는 명시적으로 사용되어 클래스를 인스턴스화할 수 없음을 명확하게 합니다.

private 생성자는 Math 클래스 등의 인스턴스 필드 또는 메서드가 없는 경우 또는 메서드를 호출하여 클래스 인스턴스를 가져오는 경우 클래스 인스턴스를 만들 수 없도록 하는데 사용됩니다. 클래스의 모든 메서드가 정적인 경우 전체 클래스를 정적 클래스로 만드는 것이 좋습니다. 자세한 내용은 [정적 클래스 및 정적 클래스 멤버](#)를 참조하세요.

## 예시

다음은 private 생성자를 사용하는 클래스의 예입니다.

```
C#  
  
public class Counter  
{  
    private Counter() { }  
  
    public static int currentCount;  
  
    public static int IncrementCount()  
    {  
        return ++currentCount;  
    }  
}
```

```
class TestCounter
{
    static void Main()
    {
        // If you uncomment the following statement, it will generate
        // an error because the constructor is inaccessible:
        // Counter aCounter = new Counter(); // Error

        Counter.currentCount = 100;
        Counter.IncrementCount();
        Console.WriteLine("New count: {0}", Counter.currentCount);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: New count: 101
```

예제에서 다음 문의 주석 처리를 제거하는 경우 보호 수준 때문에 생성자에 액세스할 수 없어 오류가 생성됩니다.

C#

```
// Counter aCounter = new Counter(); // Error
```

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [C# 형식 시스템](#)
- [생성자](#)
- [종료자](#)
- [private](#)
- [public](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

#### 설명서 문제 열기

#### 제품 사용자 의견 제공

# 정적 생성자(C# 프로그래밍 가이드)

아티클 • 2023. 10. 24.

정적 생성자는 정적 데이터를 초기화하거나 한 번만 수행해야 하는 특정 작업을 수행하는데 사용됩니다. 첫 번째 인스턴스가 만들어지거나 정적 멤버가 참조되기 전에 자동으로 호출됩니다. 정적 생성자는 한 번에 호출됩니다.

```
C#  
  
class SimpleClass  
{  
    // Static variable that must be initialized at run time.  
    static readonly long baseline;  
  
    // Static constructor is called at most one time, before any  
    // instance constructor is invoked or member is accessed.  
    static SimpleClass()  
    {  
        baseline = DateTime.Now.Ticks;  
    }  
}
```

정적 초기화의 일부인 몇 가지 작업이 있습니다. 이러한 작업은 다음 순서로 수행됩니다.

1. 정적 필드는 0으로 설정됩니다. 이 작업은 일반적으로 런타임에 의해 수행됩니다.
2. 정적 필드 이니셜라이저가 실행됩니다. 가장 파생된 형식 실행의 정적 필드 이니셜라이저입니다.
3. 기본 형식 정적 필드 이니셜라이저가 실행됩니다. 정적 필드 이니셜라이저는 각 기본 형식 [System.Object](#)을 통해 직접 밑부터 시작하여 .
4. 기본 정적 생성자가 실행됩니다. 각 기본 클래스를 통해 직접 기본 클래스로 [Object.Object](#) 시작하는 모든 정적 생성자입니다.
5. 정적 생성자가 실행됩니다. 형식에 대한 정적 생성자가 실행됩니다.

모듈 이 [니셜라이저](#) 는 정적 생성자에 대한 대안이 될 수 있습니다. 자세한 내용은 모듈 이니셜라이저 사양 [을 참조하세요](#).

## 설명

정적 생성자에는 다음과 같은 속성이 있습니다.

- 정적 생성자는 액세스 한정자를 사용하거나 매개 변수를 갖지 않습니다.
- 클래스 또는 구조체에는 한 개의 정적 생성자만 사용할 수 있습니다.
- 정적 생성자는 상속하거나 오버로드할 수 없습니다.

- 정적 생성자는 직접 호출할 수 없으며, CLR(공용 언어 런타임)을 통해서만 호출할 수 있습니다. 자동으로 호출됩니다.
- 사용자는 프로그램에서 정적 생성자가 실행되는 시기를 제어할 수 없습니다.
- 정적 생성자는 자동으로 호출되며 첫 번째 인스턴스가 생성되거나 (기본 클래스가 아닌) 해당 클래스에서 선언된 정적 멤버가 참조되기 전에 [클래스](#)를 초기화합니다. 정적 생성자는 인스턴스 생성자보다 먼저 실행됩니다. 정적 필드 변수 이니셜라이저가 정적 생성자의 클래스에 있는 경우 클래스 선언에 나타나는 텍스트 순서대로 실행됩니다. 이니셜라이저는 정적 생성자가 실행되기 직전에 실행됩니다.
- 정적 필드를 초기화하는 정적 생성자를 제공하지 않으면 모든 정적 필드가 [C# 형식의 기본값](#)에 나열된 기본값으로 초기화됩니다.
- 정적 생성자가 예외를 throw하는 경우 런타임에서 생성자를 다시 호출하지 않으며 애플리케이션 도메인의 수명 동안 형식이 초기화되지 않은 상태로 유지됩니다. 가장 일반적으로, 정적 생성자가 형식을 인스턴스화할 수 없는 경우 또는 정적 생성자 내에서 발생하는 처리되지 않은 예외에 대해 [TypeInitializationException](#) 예외가 throw됩니다. 소스 코드에서 명시적으로 정의되지 않은 정적 생성자의 경우 문제 해결을 위해 IL(중간 언어) 코드를 검사해야 할 수 있습니다.
- 정적 생성자가 있으면 [BeforeFieldInit](#) 형식 특성을 추가할 수 없습니다. 이 때문에 런타임 최적화가 제한됩니다.
- `static readonly`로 선언된 필드는 해당 선언의 일부로 또는 정적 생성자에서만 할당할 수 있습니다. 명시적 정적 생성자가 필요하지 않은 경우 런타임 최적화 향상을 위해 정적 생성자를 통하지 않고 선언에서 정적 필드를 초기화합니다.
- 런타임은 단일 애플리케이션 도메인에서 정적 생성자를 1번 이하로 호출합니다. 이 호출은 클래스의 특정 형식에 따라 잠금 영역에서 수행됩니다. 정적 생성자의 본문에는 추가 잠금 메커니즘이 필요하지 않습니다. 교착 상태 위험을 방지하려면 정적 생성자와 이니셜라이저에서 현재 스레드를 차단하지 않습니다. 예를 들어 작업, 스레드, 대기 핸들 또는 이벤트를 기다리지 않고 잠금을 획득하지 않으며 병렬 루프, `Parallel.Invoke`, 병렬 LINQ 쿼리와 같은 병렬 작업 차단을 실행하지 않습니다.

### ① 참고

직접 액세스할 수 없더라도, 명시적 정적 생성자가 있을 경우 초기화 예외 문제 해결에 도움이 되도록 문서화해야 합니다.

## 사용

- 정적 생성자는 일반적으로 클래스가 로그 파일을 사용하고, 생성자를 사용하여 이 파일에 항목을 쓰는 경우에 사용됩니다.
- 정적 생성자는 생성자가 `LoadLibrary` 메서드를 호출할 수 있을 때 비관리 코드에 대한 래퍼 클래스를 만드는 경우에도 유용합니다.

- 또한 정적 생성자를 사용하면 형식-매개 변수 제약 조건을 통해 컴파일 시간에 검사 할 수 없는 형식 매개 변수에 런타임 검사를 편리하게 적용할 수 있습니다.

## 예시

이 예제에서 `Bus` 클래스에는 정적 생성자가 있습니다. `Bus`의 첫 번째 인스턴스를 만들 때 (`bus1`) 정적 생성자가 호출되어 클래스를 초기화합니다. 샘플 출력은 `Bus`의 두 인스턴스가 생성된 경우에도 정적 생성자가 한 번만 실행되고, 인스턴스 생성자를 실행하기 전에 실행되는지 확인합니다.

```
C#  
  
public class Bus  
{  
    // Static variable used by all Bus instances.  
    // Represents the time the first bus of the day starts its route.  
    protected static readonly DateTime globalStartTime;  
  
    // Property for the number of each bus.  
    protected int RouteNumber { get; set; }  
  
    // Static constructor to initialize the static variable.  
    // It is invoked before the first instance constructor is run.  
    static Bus()  
    {  
        globalStartTime = DateTime.Now;  
  
        // The following statement produces the first line of output,  
        // and the line occurs only once.  
        Console.WriteLine("Static constructor sets global start time to  
{0}",  
            globalStartTime.ToString());  
    }  
  
    // Instance constructor.  
    public Bus(int routeNum)  
    {  
        RouteNumber = routeNum;  
        Console.WriteLine("Bus #{0} is created.", RouteNumber);  
    }  
  
    // Instance method.  
    public void Drive()  
    {  
        TimeSpan elapsedTime = DateTime.Now - globalStartTime;  
  
        // For demonstration purposes we treat milliseconds as minutes to  
        // simulate  
        // actual bus times. Do not do this in your actual bus schedule  
        // program!  
        Console.WriteLine("{0} is starting its route {1:N2} minutes after  
        // the global start time.", RouteNumber, elapsedTime.TotalMinutes);  
    }  
}
```

```

        global startTime {2}." ,
            this.RouteNumber,
            elapsedTime.Milliseconds,
            globalStartTime.ToShortTimeString());
    }
}

class TestBus
{
    static void Main()
    {
        // The creation of this instance activates the static constructor.
        Bus bus1 = new Bus(71);

        // Create a second bus.
        Bus bus2 = new Bus(72);

        // Send bus1 on its way.
        bus1.Drive();

        // Wait for bus2 to warm up.
        System.Threading.Thread.Sleep(25);

        // Send bus2 on its way.
        bus2.Drive();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Sample output:
   Static constructor sets global start time to 3:57:08 PM.
   Bus #71 is created.
   Bus #72 is created.
   71 is starting its route 6.00 minutes after global start time 3:57 PM.
   72 is starting its route 31.00 minutes after global start time 3:57 PM.
*/

```

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 정적 생성자 섹션](#)을 참조하세요.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [C# 형식 시스템](#)
- [생성자](#)
- [정적 클래스 및 정적 클래스 멤버](#)

- 종료자
- 생성자 디자인 지침
- 보안 경고 - CA2121: 정적 생성자는 프라이빗이어야 합니다.
- 모듈 아티클레이저

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET feedback

The .NET documentation is open source. Provide feedback here.

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# 복사 생성자를 작성하는 방법(C# 프로그래밍 가이드)

아티클 • 2024. 03. 12.

C# [레코드](#)는 개체의 복사 생성자를 제공하지만 클래스의 경우 직접 작성해야 합니다.

## ① 중요

클래스 계층 구문의 모든 파생 형식에 대해 작동하는 복사 생성자를 작성하는 것은 어려울 수 있습니다. 클래스가 `sealed` 가 아닌 경우 컴파일러 합성 복사 생성자를 사용하려면 `record class` 형식의 계층 구문을 만드는 것을 적극 고려해야 합니다.

## 예시

다음 예제에서 `Person` [클래스](#)는 `Person` 인스턴스를 해당 인수로 사용하는 복사 생성자를 정의합니다. 인수의 속성 값이 `Person`의 새 인스턴스 속성에 할당됩니다. 코드에는 복사 하려는 인스턴스의 `Name` 및 `Age` 속성을 클래스의 인스턴스 생성자에 보내는 대체 복사 생성자가 포함되어 있습니다. `Person` 클래스는 `sealed` 이므로 기본 클래스만 복사하여 오류를 일으킬 수 있는 파생 형식을 선언할 수 없습니다.

C#

```
public sealed class Person
{
    // Copy constructor.
    public Person(Person previousPerson)
    {
        Name = previousPerson.Name;
        Age = previousPerson.Age;
    }

    //// Alternate copy constructor calls the instance constructor.
    //public Person(Person previousPerson)
    //    : this(previousPerson.Name, previousPerson.Age)
    //{
    //}

    // Instance constructor.
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}
```

```

public int Age { get; set; }

public string Name { get; set; }

public string Details()
{
    return Name + " is " + Age.ToString();
}

class TestPerson
{
    static void Main()
    {
        // Create a Person object by using the instance constructor.
        Person person1 = new Person("George", 40);

        // Create another Person object, copying person1.
        Person person2 = new Person(person1);

        // Change each person's age.
        person1.Age = 39;
        person2.Age = 41;

        // Change person2's name.
        person2.Name = "Charles";

        // Show details to verify that the name and age fields are distinct.
        Console.WriteLine(person1.Details());
        Console.WriteLine(person2.Details());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// George is 39
// Charles is 41

```

## 참고 항목

- ICloneable
- 레코드
- C# 형식 시스템
- 생성자
- 종료자

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 종료자(C# 프로그래밍 가이드)

아티클 • 2024. 03. 12.

종료자(이전에는 소멸자라고 함)는 가비지 수집기에서 클래스 인스턴스를 수집할 때 필요한 최종 정리를 수행하는 데 사용됩니다. 대부분의 경우

`System.Runtime.InteropServices.SafeHandle` 또는 파생 클래스를 사용하여 관리되지 않는 핸들을 래핑하면 종료자를 작성하지 않아도 됩니다.

## 설명

- 종료자는 구조체에서 정의할 수 없으며, 클래스에서만 사용됩니다.
- 클래스에는 종료자가 하나만 있을 수 있습니다.
- 종료자는 상속하거나 오버로드할 수 없습니다.
- 종료자를 호출할 수 없습니다. 자동으로 호출됩니다.
- 종료자는 한정자를 사용하거나 매개 변수를 갖지 않습니다.

예를 들어 다음은 `Car` 클래스에 대한 종료자의 선언입니다.

```
C#  
  
class Car  
{  
    ~Car() // finalizer  
    {  
        // cleanup statements...  
    }  
}
```

다음 예제와 같이 종료자를 식 본문 정의로 구현할 수도 있습니다.

```
C#  
  
public class Destroyer  
{  
    public override string ToString() => GetType().Name;  
  
    ~Destroyer() => Console.WriteLine($"The {ToString()} finalizer is  
executing.");  
}
```

종료자는 개체의 기본 클래스에서 `Finalize`를 암시적으로 호출합니다. 따라서 종료자 호출은 다음 코드로 암시적으로 변환됩니다.

```
C#
```

```
protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}
```

이 설계에서는 `Finalize` 메서드가 상속 체인의 모든 인스턴스에 대해 최다 파생에서 최소 파생까지 재귀적으로 호출됩니다.

### ① 참고

빈 종료자는 사용할 수 없습니다. 클래스에 종료자가 포함되어 있으면 `Finalize` 큐에서 항목이 생성됩니다. 이 큐는 가비지 수집기에 의해 처리됩니다. GC는 큐를 처리할 때 각 종료자를 호출합니다. 빈 종료자를 포함하는 불필요한 종료자, 기본 클래스 종료자만 호출하는 종료자 또는 조건부로 내보낸 메서드만 호출하는 종료자는 불필요한 성능 손실을 발생시킵니다.

종료자가 호출되는 시기는 프로그래머가 제어할 수 없고, 가비지 수집기에 의해 결정됩니다. 가비지 수집기는 애플리케이션에서 더 이상 사용되지 않는 개체를 확인합니다. 개체를 종료할 수 있는 것으로 판단되면 종료자(있는 경우)를 호출하고 개체를 저장하는 데 사용된 메모리를 회수합니다. `Collect`를 호출하여 강제로 가비지를 수집할 수 있지만 대체로 성능 이슈가 발생할 수 있으므로 이 호출을 피해야 합니다.

### ① 참고

종료자가 애플리케이션 종료의 일부로 실행되는지 여부는 각 [.NET의 구현](#)에 따라 다릅니다. 애플리케이션이 종료될 때, 정리가 억제되지 않은 경우(예: 라이브러리 메서드 `GC.SuppressFinalize` 호출) .NET Framework는 아직 가비지가 수집되지 않은 개체에 대해 종료자를 호출하기 위해 모든 합리적인 노력을 기울입니다. .NET 5(.NET Core 포함) 이상 버전은 애플리케이션 종료 과정에서 종료자를 호출하지 않습니다. 자세한 내용은 GitHub 이슈 [dotnet/csharpstandard #291](#)을 참조하세요.

애플리케이션이 있을 때 안정적으로 정리를 수행해야 하는 경우

`System.AppDomain.ProcessExit` 이벤트에 대한 처리기를 등록합니다. 해당 처리기는 애플리케이션이 종료되기 전에 정리가 필요한 모든 개체에 대해 `IDisposable.Dispose()`(또는 `IAsyncDisposable.DisposeAsync()`)이 호출되었는지 확인합니다. `Finalize`를 직접 호출할

수 없고 종료 전에 가비지 수집기에서 모든 종료자를 호출하도록 보장할 수 없으므로 `Dispose` 또는 `DisposeAsync`를 사용하여 리소스가 해제되도록 해야 합니다.

## 종료자를 사용하여 리소스 해제

일반적으로 C#에서는 개발자 측이 가비지 수집을 사용하는 런타임을 대상으로 하지 않는 언어만큼 많은 메모리 관리를 수행할 필요가 없습니다. 이는 .NET 가비지 수집기에서 개체에 대한 메모리 할당 및 해제를 암시적으로 관리하기 때문입니다. 그러나 애플리케이션에서 창, 파일 및 네트워크 연결 등의 관리되지 않는 리소스를 캡슐화하는 경우 종료자를 사용하여 해당 리소스를 해제해야 합니다. 개체를 종료할 수 있으면 가비지 수집기에서 개체의 `Finalize` 메서드를 실행합니다.

## 리소스의 명시적 해제

애플리케이션에서 비용이 많이 드는 외부 리소스를 사용하는 경우 가비지 수집기에서 개체를 해제하기 전에 리소스를 명시적으로 해제하는 방법을 제공하는 것이 좋습니다. 해당 리소스를 해제하려면 [IDisposable](#) 인터페이스에서 개체에 필요한 정리를 수행하는 `Dispose` 메서드를 구현합니다. 이렇게 하면 애플리케이션의 성능을 상당히 향상시킬 수 있습니다. 이렇게 리소스를 명시적으로 제어하는 경우에도 종료자는 `Dispose` 메서드 호출에 실패할 경우 리소스를 정리하는 안전한 방법이 됩니다.

리소스 정리에 대한 자세한 내용은 다음 문서를 참조하세요.

- 관리되지 않는 리소스 정리
- `Dispose` 메서드 구현
- `DisposeAsync` 메서드 구현
- `using` 문

## 예시

다음 예제에서는 상속 체인을 구성하는 세 가지 클래스를 만듭니다. `First` 클래스는 기본 클래스이고, `Second`는 `First`에서 파생되며, `Third`는 `Second`에서 파생됩니다. 세 클래스 모두 종료자가 있습니다. `Main`에서 최종 파생 클래스의 인스턴스가 만들어집니다. 이 코드의 출력은 애플리케이션이 대상으로 하는 .NET의 구현에 따라 달라집니다.

- .NET Framework: 애플리케이션이 종료될 때 세 클래스에 대한 종료자가 가장 많이 파생된 것에서 가장 적게 파생된 것의 순서로 자동으로 호출되는 것을 출력에서 보여 줍니다.
- .NET 5(.NET Core 포함) 이상 버전: 이 .NET 구현은 애플리케이션이 종료될 때 종료자를 호출하지 않으므로 출력이 없습니다.

C#

```
class First
{
    ~First()
    {
        System.Diagnostics.Trace.WriteLine("First's finalizer is called.");
    }
}

class Second : First
{
    ~Second()
    {
        System.Diagnostics.Trace.WriteLine("Second's finalizer is called.");
    }
}

class Third : Second
{
    ~Third()
    {
        System.Diagnostics.Trace.WriteLine("Third's finalizer is called.");
    }
}

/*
Test with code like the following:
Third t = new Third();
t = null;

When objects are finalized, the output would be:
Third's finalizer is called.
Second's finalizer is called.
First's finalizer is called.
*/
```

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 종료자 섹션](#)을 참조하세요.

## 참고 항목

- [IDisposable](#)
- [생성자](#)
- [가비지 수집](#)

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 개체 및 컬렉션 이니셜라이저(C# 프로그래밍 가이드)

아티클 • 2024. 05. 27.

C#을 사용하면 개체 또는 컬렉션을 인스턴스화하고 단일 명령문에서 멤버 할당을 수행할 수 있습니다.

## 개체 이니셜라이저

개체 이니셜라이저를 사용하면 명시적으로 생성자를 호출한 다음 할당문 줄을 추가하지 않고도 생성 시 개체의 모든 액세스 가능한 필드나 속성에 값을 할당할 수 있습니다. 개체 이니셜라이저 구문을 사용하면 생성자의 인수를 지정하거나 인수(및 괄호 구문)를 생략할 수 있습니다. 다음 예제에서는 명명된 형식인 `Cat`로 개체 이니셜라이저를 사용하는 방법과 매개 변수가 없는 생성자를 호출하는 방법을 보여 줍니다. `Cat` 클래스의 자동 구현된 속성을 확인합니다. 자세한 내용은 [자동으로 구현된 속성을 참조하세요](#).

C#

```
public class Cat
{
    // Auto-implemented properties.
    public int Age { get; set; }
    public string? Name { get; set; }

    public Cat()
    {
    }

    public Cat(string name)
    {
        this.Name = name;
    }
}
```

C#

```
Cat cat = new Cat { Age = 10, Name = "Fluffy" };
Cat sameCat = new Cat("Fluffy"){ Age = 10 };
```

개체 이니셜라이저 구문을 사용하여 인스턴스를 만들 수 있으며, 만들고 나면 새로 만든 개체와 할당된 해당 속성이 할당의 변수에 할당됩니다.

개체 이니셜라이저는 필드 및 속성을 할당하는 것 외에도 인덱서를 설정할 수 있습니다. 기본적인 `Matrix` 클래스를 예로 들어 보겠습니다.

C#

```
public class Matrix
{
    private double[,] storage = new double[3, 3];

    public double this[int row, int column]
    {
        // The embedded array will throw out of range exceptions as
        appropriate.
        get { return storage[row, column]; }
        set { storage[row, column] = value; }
    }
}
```

다음 코드를 사용하여 `matrix`라는 ID를 초기화할 수 있습니다.

C#

```
var identity = new Matrix
{
    [0, 0] = 1.0,
    [0, 1] = 0.0,
    [0, 2] = 0.0,

    [1, 0] = 0.0,
    [1, 1] = 1.0,
    [1, 2] = 0.0,

    [2, 0] = 0.0,
    [2, 1] = 0.0,
    [2, 2] = 1.0,
};
```

액세스 가능한 setter가 포함된 액세스 가능한 인덱서는 인수의 개수나 형식에 관계없이 객체 이니셜라이저에서 식 중 하나로 사용할 수 있습니다. 인덱스 인수는 할당의 왼쪽에 있으며, 값은 식의 오른쪽에 있습니다. 예를 들어 다음 이니셜라이저는 `IndexersExample`에 적절한 인덱서가 있는 경우 모두 유효합니다.

C#

```
var thing = new IndexersExample
{
    name = "object one",
    [1] = '1',
    [2] = '4',
```

```
[3] = '9',
Size = Math.PI,
['C',4] = "Middle C"
}
```

이전 코드를 컴파일하려면 `IndexersExample` 형식에 다음 멤버가 있어야 합니다.

C#

```
public string name;
public double Size { set { ... }; }
public char this[int i] { set { ... }; }
public string this[char c, int i] { set { ... }; }
```

## 익명 형식의 개체 이니셜라이저

개체 이니셜라이저는 모든 컨텍스트에서 사용할 수 있지만 특히 LINQ 쿼리 식에 유용합니다. 쿼리 식은 [무명 형식](#)을 자주 사용합니다. 이 형식은 다음 선언에 표시된 바와 같이 개체 이니셜라이저를 사용하는 경우에만 초기화될 수 있습니다.

C#

```
var pet = new { Age = 10, Name = "Fluffy" };
```

무명 형식을 사용하면 LINQ 쿼리 식의 `select` 절에서 원래 시퀀스의 개체를 값과 모양이 원본과 다를 수 있는 개체로 변환할 수 있습니다. 각 개체의 일부 정보만 시퀀스에 저장하려는 경우가 있을 수 있습니다. 다음 예제에서 제품 개체(`p`)는 많은 필드와 메서드를 포함하며, 제품 이름과 단위 가격이 들어 있는 개체 시퀀스만 만들려 한다고 가정합니다.

C#

```
var productInfos =
    from p in products
    select new { p.ProductName, p.UnitPrice };
```

이 쿼리를 실행하면 다음 예제와 같이 `productInfos` 문에서 액세스할 수 있는 개체 시퀀스가 `foreach` 변수에 포함됩니다.

C#

```
foreach(var p in productInfos){...}
```

새 익명 형식의 각 개체에는 원래 개체의 속성이나 필드와 동일한 이름을 받는 두 개의 public 속성이 있습니다. 익명 형식을 만들 때 필드 이름을 바꿀 수도 있습니다. 다음 예제에서는 `UnitPrice` 필드의 이름을 `Price`로 바꿉니다.

C#

```
select new {p.ProductName, Price = p.UnitPrice};
```

## required 한정자가 있는 개체 이니셜라이저

`required` 키워드를 사용하여 호출자가 개체 이니셜라이저를 사용하여 속성 또는 필드의 값을 설정하도록 강제합니다. 필수 속성은 생성자 매개 변수로 설정할 필요가 없습니다. 컴파일러는 모든 호출자가 해당 값을 초기화하도록 합니다.

C#

```
public class Pet
{
    public required int Age;
    public string Name;
}

// `Age` field is necessary to be initialized.
// You don't need to initialize `Name` property
var pet = new Pet() { Age = 10};

// Compiler error:
// Error CS9035 Required member 'Pet.Age' must be set in the object
// initializer or attribute constructor.
// var pet = new Pet();
```

특히 관리할 필드나 속성이 여러 개 있고 생성자에 모두 포함하지 않으려는 경우 개체가 제대로 초기화되도록 보장하는 것이 일반적입니다.

## init 접근자가 있는 개체 이니셜라이저

`init` 접근자를 사용하면 누구도 디자인한 개체를 변경하지 못하도록 제한할 수 있습니다. 속성 값의 설정을 제한하는 데 도움이 됩니다.

C#

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; init; }
```

```

}

// The `LastName` property can be set only during initialization. It CAN'T
be modified afterwards.
// The `FirstName` property can be modified after initialization.
var pet = new Person() { FirstName = "Joe", LastName = "Doe"};

// You can assign the FirstName property to a different value.
pet.FirstName = "Jane";

// Compiler error:
// Error CS8852 Init - only property or indexer 'Person.LastName' can only
be assigned in an object initializer,
//           or on 'this' or 'base' in an instance constructor or an
'init' accessor.
// pet.LastName = "Kowalski";

```

필수 초기화 전용 속성은 변경할 수 없는 구조를 지원하는 동시에 해당 형식의 사용자에 대한 자연스러운 구문을 허용합니다.

## 클래스 형식 속성이 있는 개체 이니셜라이저

개체를 초기화할 때는 클래스 형식 속성에 대한 의미를 고려해야 합니다.

C#

```

public class HowToClassTypedInitializer
{
    public class EmbeddedClassTypeA
    {
        public int I { get; set; }
        public bool B { get; set; }
        public string S { get; set; }
        public EmbeddedClassTypeB ClassB { get; set; }

        public override string ToString() => $"{I}|{B}|{S}|||{ClassB}";

        public EmbeddedClassTypeA()
        {
            Console.WriteLine($"Entering EmbeddedClassTypeA constructor.
Values are: {this}");
            I = 3;
            B = true;
            S = "abc";
            ClassB = new() { BB = true, BI = 43 };
            Console.WriteLine($"Exiting EmbeddedClassTypeA constructor.
Values are: {this}");
        }
    }

    public class EmbeddedClassTypeB

```

```

{
    public int BI { get; set; }
    public bool BB { get; set; }
    public string BS { get; set; }

    public override string ToString() => $"{BI}|{BB}|{BS}";

    public EmbeddedClassTypeB()
    {
        Console.WriteLine($"Entering EmbeddedClassTypeB constructor.
Values are: {this}");
        BI = 23;
        BB = false;
        BS = "BBBabc";
        Console.WriteLine($"Exiting EmbeddedClassTypeB constructor.
Values are: {this}");
    }
}

public static void Main()
{
    var a = new EmbeddedClassTypeA
    {
        I = 103,
        B = false,
        ClassB = { BI = 100003 }
    };
    Console.WriteLine($"After initializing EmbeddedClassTypeA: {a}");

    var a2 = new EmbeddedClassTypeA
    {
        I = 103,
        B = false,
        ClassB = new() { BI = 100003 } //New instance
    };
    Console.WriteLine($"After initializing EmbeddedClassTypeA a2:
{a2}");
}

// Output:
//Entering EmbeddedClassTypeA constructor Values are: 0|False|||
//Entering EmbeddedClassTypeB constructor Values are: 0|False|
//Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc)
//Exiting EmbeddedClassTypeA constructor Values are:
3|True|abc|||43|True|BBBabc)
//After initializing EmbeddedClassTypeA:
103|False|abc|||100003|True|BBBabc
//Entering EmbeddedClassTypeA constructor Values are: 0|False|||
//Entering EmbeddedClassTypeB constructor Values are: 0|False|
//Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc)
//Exiting EmbeddedClassTypeA constructor Values are:
3|True|abc|||43|True|BBBabc)
//Entering EmbeddedClassTypeB constructor Values are: 0|False|
//Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc)
//After initializing EmbeddedClassTypeA a2:

```

```
103|False|abc|||100003|False|BBBabc  
}
```

다음 예에서는 ClassB의 경우 초기화 프로세스에서 원래 인스턴스의 다른 값을 보존하면서 특정 값을 업데이트하는 방법을 보여 줍니다. 이니셜라이저는 현재 인스턴스를 재사용합니다. ClassB의 값은 다음과 같습니다. 100003(여기서 할당한 새 값), true(EmbeddedClassTypeA의 초기화에서 유지됨), BBBabc(EmbeddedClassTypeB에서 변경되지 않은 기본값)

## 컬렉션 이니셜라이저

컬렉션 이니셜라이저를 사용하면 `IEnumerable`을 구현하고 적절한 시그니처가 있는 `Add`를 인스턴스 메서드 또는 확장 메서드로 포함하는 컬렉션 형식을 초기화할 때 하나 이상의 요소 이니셜라이저를 지정할 수 있습니다. 요소 이니셜라이저는 값, 식 또는 개체 이니셜라이저일 수 있습니다. 컬렉션 이니셜라이저를 사용하면 호출을 여러 번 지정할 필요가 없습니다. 컴파일러가 호출을 자동으로 추가합니다.

다음 예제에서는 두 개의 단순한 컬렉션 이니셜라이저를 보여줍니다.

C#

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };  
List<int> digits2 = new List<int> { 0 + 1, 12 % 3, MakeInt() };
```

다음 컬렉션 이니셜라이저는 개체 이니셜라이저를 사용하여 앞의 예제에서 정의된 `Cat` 클래스의 개체를 초기화합니다. 개별 개체 이니셜라이저는 괄호로 묶이고 쉼표로 구분됩니다.

C#

```
List<Cat> cats = new List<Cat>  
{  
    new Cat{ Name = "Sylvester", Age=8 },  
    new Cat{ Name = "Whiskers", Age=2 },  
    new Cat{ Name = "Sasha", Age=14 }  
};
```

컬렉션의 `Add` 메서드에서 허용하는 경우 `null`을 컬렉션 이니셜라이저의 요소로 지정할 수 있습니다.

C#

```
List<Cat?> moreCats = new List<Cat?>  
{
```

```
new Cat{ Name = "Furrytail", Age=5 },
new Cat{ Name = "Peaches", Age=4 },
null
};
```

컬렉션이 읽기/쓰기 인덱싱을 지원하는 경우 인덱싱된 요소를 지정할 수 있습니다.

C#

```
var numbers = new Dictionary<int, string>
{
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};
```

이전 샘플에서는 `Item[TKey]`을 호출하여 값을 설정하는 코드를 생성합니다. 다음 구문을 사용하여 사전 및 기타 연관 컨테이너를 초기화할 수도 있습니다. 괄호와 할당이 있는 인덱서 구문 대신 여러 값이 있는 개체를 사용합니다.

C#

```
var moreNumbers = new Dictionary<int, string>
{
    {19, "nineteen" },
    {23, "twenty-three" },
    {42, "forty-two" }
};
```

이 이니셜라이저 예제에서는 `Add(TKey, TValue)`를 호출하여 사전에 세 가지 항목을 추가합니다. 연관 컬렉션을 초기화하는 이러한 두 가지 방법은 컴파일러가 생성하는 메서드 호출 때문에 약간 다르게 작동합니다. 두 가지 방법 모두 `Dictionary` 클래스와 함께 작동합니다. 다른 형식은 공용 API에 따라 어느 한 쪽만 지원할 수 있습니다.

## 컬렉션 읽기 전용 속성 초기화를 사용하는 개체 이니셜라이저

일부 클래스에는 다음과 같은 경우 `CatOwner`의 `Cats` 속성이 읽기 전용인 컬렉션 속성이 있을 수 있습니다.

C#

```
public class CatOwner
{
```

```
    public IList<Cat> Cats { get; } = new List<Cat>();  
}
```

속성에 새 목록을 할당할 수 없으므로 지금까지 설명한 컬렉션 이니셜라이저 구문은 사용할 수 없습니다.

C#

```
CatOwner owner = new CatOwner  
{  
    Cats = new List<Cat>  
    {  
        new Cat{ Name = "Sylvester", Age=8 },  
        new Cat{ Name = "Whiskers", Age=2 },  
        new Cat{ Name = "Sasha", Age=14 }  
    }  
};
```

그러나 다음에 표시된 것처럼 목록 만들기(`new List<Cat>`)를 생략하여 초기화 구문을 사용해 새 항목을 `Cats`에 추가할 수 있습니다.

C#

```
CatOwner owner = new CatOwner  
{  
    Cats =  
    {  
        new Cat{ Name = "Sylvester", Age=8 },  
        new Cat{ Name = "Whiskers", Age=2 },  
        new Cat{ Name = "Sasha", Age=14 }  
    }  
};
```

추가될 항목 세트는 중괄호로 묶여 표시됩니다. 위의 코드는 다음을 작성하는 것과 동일합니다.

C#

```
CatOwner owner = new ();  
owner.Cats.Add(new Cat{ Name = "Sylvester", Age=8 });  
owner.Cats.Add(new Cat{ Name = "Whiskers", Age=2 });  
owner.Cats.Add(new Cat{ Name = "Sasha", Age=14 });
```

## 예제

다음 예제에서는 개체 및 컬렉션 이니셜라이저의 개념을 결합합니다.

C#

```
public class InitializationSample
{
    public class Cat
    {
        // Auto-implemented properties.
        public int Age { get; set; }
        public string? Name { get; set; }

        public Cat() { }

        public Cat(string name)
        {
            Name = name;
        }
    }

    public static void Main()
    {
        Cat cat = new Cat { Age = 10, Name = "Fluffy" };
        Cat sameCat = new Cat("Fluffy"){ Age = 10 };

        List<Cat> cats = new List<Cat>
        {
            new Cat { Name = "Sylvester", Age = 8 },
            new Cat { Name = "Whiskers", Age = 2 },
            new Cat { Name = "Sasha", Age = 14 }
        };

        List<Cat?> moreCats = new List<Cat?>
        {
            new Cat { Name = "Furrytail", Age = 5 },
            new Cat { Name = "Peaches", Age = 4 },
            null
        };

        // Display results.
        System.Console.WriteLine(cat.Name);

        foreach (Cat c in cats)
        {
            System.Console.WriteLine(c.Name);
        }

        foreach (Cat? c in moreCats)
        {
            if (c != null)
            {
                System.Console.WriteLine(c.Name);
            }
            else
            {
                System.Console.WriteLine("List element has null value.");
            }
        }
    }
}
```

```

        }
    }
}

// Output:
//Fluffy
//Sylvester
//Whiskers
//Sasha
//Furrytail
//Peaches
//List element has null value.
}

```

다음 예제에서는 [IEnumerable](#)을 구현하고 여러 매개 변수가 있는 `Add` 메서드를 포함하는 개체를 보여 줍니다. `Add` 메서드의 서명에 해당하는 목록의 항목당 여러 요소가 있는 컬렉션 이니셜라이저를 사용합니다.

C#

```

public class FullExample
{
    class FormattedAddresses : IEnumerable<string>
    {
        private List<string> internalList = new List<string>();
        public IEnumerator<string> GetEnumerator() =>
internalList.GetEnumerator();

        System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator() =>
internalList.GetEnumerator();

        public void Add(string firstname, string lastname,
            string street, string city,
            string state, string zipcode) => internalList.Add($"""
{firstname} {lastname}
{street}
{city}, {state} {zipcode}
""");

    }

    public static void Main()
    {
        FormattedAddresses addresses = new FormattedAddresses()
        {
            {"John", "Doe", "123 Street", "Topeka", "KS", "00000" },
            {"Jane", "Smith", "456 Street", "Topeka", "KS", "00000" }
        };

        Console.WriteLine("Address Entries:");

        foreach (string addressEntry in addresses)

```

```

    {
        Console.WriteLine("\r\n" + addressEntry);
    }
}

/*
 * Prints:

Address Entries:

John Doe
123 Street
Topeka, KS 00000

Jane Smith
456 Street
Topeka, KS 00000
*/
}

```

다음 예제에 표시된 것처럼 `Add` 메서드는 `params` 키워드를 사용하여 가변적인 인수 개수를 사용할 수 있습니다. 이 예제에서는 인덱스를 사용하여 컬렉션을 초기화하기 위한 인덱서의 사용자 지정 구현도 보여줍니다. C# 13부터 `params` 매개 변수는 배열로 제한되지 않습니다. 컬렉션 형식 또는 인터페이스일 수 있습니다.

C#

```

public class DictionaryExample
{
    class RudimentaryMultiValuedDictionary<TKey, TValue> :
    IEnumerable<KeyValuePair<TKey, List<TValue>>> where TKey : notnull
    {
        private Dictionary<TKey, List<TValue>> internalDictionary = new
        Dictionary<TKey, List<TValue>>();

        public IEnumerator<KeyValuePair<TKey, List<TValue>>> GetEnumerator()
        => internalDictionary.GetEnumerator();

        System.Collections.IEnumerator
        System.Collections.IEnumerable.GetEnumerator() =>
        internalDictionary.GetEnumerator();

        public List<TValue> this[TKey key]
        {
            get => internalDictionary[key];
            set => Add(key, value);
        }

        public void Add(TKey key, params TValue[] values) => Add(key,
        (IEnumerable<TValue>)values);

        public void Add(TKey key, IEnumerable<TValue> values)
    }
}

```

```

        {
            if (!internalDictionary.TryGetValue(key, out List< TValue >?
storedValues))
            {
                internalDictionary.Add(key, storedValues = new List< TValue >
());
            }
            storedValues.AddRange(values);
        }
    }

    public static void Main()
{
    RudimentaryMultiValuedDictionary< string, string >
rudimentaryMultiValuedDictionary1
    = new RudimentaryMultiValuedDictionary< string, string >()
    {
        {"Group1", "Bob", "John", "Mary" },
        {"Group2", "Eric", "Emily", "Debbie", "Jesse" }
    };
    RudimentaryMultiValuedDictionary< string, string >
rudimentaryMultiValuedDictionary2
    = new RudimentaryMultiValuedDictionary< string, string >()
    {
        ["Group1"] = new List< string >() { "Bob", "John", "Mary" },
        ["Group2"] = new List< string >() { "Eric", "Emily", "Debbie",
"Jesse" }
    };
    RudimentaryMultiValuedDictionary< string, string >
rudimentaryMultiValuedDictionary3
    = new RudimentaryMultiValuedDictionary< string, string >()
    {
        {"Group1", new string []{ "Bob", "John", "Mary" } },
        {"Group2", new string[]{ "Eric", "Emily", "Debbie", "Jesse"
} }
    };
}

Console.WriteLine("Using first multi-valued dictionary created with
a collection initializer:");

    foreach (KeyValuePair< string, List< string >> group in
rudimentaryMultiValuedDictionary1)
    {
        Console.WriteLine($"\\r\\nMembers of group {group.Key}: ");

        foreach (string member in group.Value)
        {
            Console.WriteLine(member);
        }
    }

Console.WriteLine("\\r\\nUsing second multi-valued dictionary created
with a collection initializer using indexing:");

    foreach (KeyValuePair< string, List< string >> group in

```

```

rudimentaryMultiValuedDictionary2)
{
    Console.WriteLine($"\\r\\nMembers of group {group.Key}: ");

    foreach (string member in group.Value)
    {
        Console.WriteLine(member);
    }
}
Console.WriteLine("\\r\\nUsing third multi-valued dictionary created
with a collection initializer using indexing:");

foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary3)
{
    Console.WriteLine($"\\r\\nMembers of group {group.Key}: ");

    foreach (string member in group.Value)
    {
        Console.WriteLine(member);
    }
}
}

/*
 * Prints:

    Using first multi-valued dictionary created with a collection
initializer:

    Members of group Group1:
    Bob
    John
    Mary

    Members of group Group2:
    Eric
    Emily
    Debbie
    Jesse

    Using second multi-valued dictionary created with a collection
initializer using indexing:

    Members of group Group1:
    Bob
    John
    Mary

    Members of group Group2:
    Eric
    Emily
    Debbie
    Jesse

```

```
Using third multi-valued dictionary created with a collection
initializer using indexing:
```

```
    Members of group Group1:
    Bob
    John
    Mary

    Members of group Group2:
    Eric
    Emily
    Debbie
    Jesse
    */
}
```

## 참고 항목

- 개체 이니셜라이저 사용(스타일 규칙 IDE0017)
- 컬렉션 이니셜라이저 사용(스타일 규칙 IDE0028)
- C#의 LINQ
- 익명 형식

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# 개체 이니셜라이저를 사용하여 개체를 초기화하는 방법(C# 프로그래밍 가이드)

아티클 • 2024. 05. 15.

개체 이니셜라이저를 사용하여 형식에 대한 생성자를 명시적으로 호출하지 않고 선언적 방식으로 형식 개체를 초기화할 수 있습니다.

다음 예제에서는 명명된 개체와 함께 개체 이니셜라이저를 사용하는 방법을 보여 줍니다. 컴파일러는 먼저 매개 변수 없는 인스턴스 생성자에 액세스한 다음 멤버 초기화를 처리하여 개체 이니셜라이저를 처리합니다. 따라서 클래스에서 매개 변수가 없는 생성자가 `private`으로 선언된 경우 공용 액세스가 필요한 개체 이니셜라이저는 실패합니다.

무명 형식을 정의하는 경우 개체 이니셜라이저를 사용해야 합니다. 자세한 내용은 [쿼리에서 요소 속성의 하위 집합을 반환하는 방법](#)을 참조하세요.

## 예시

다음 예제에서는 개체 이니셜라이저를 사용하여 새 `StudentName` 형식을 초기화하는 방법을 보여 줍니다. 이 예제에서는 `StudentName` 형식의 속성을 설정합니다.

```
C#  
  
public class HowToObjectInitializers  
{  
    public static void Main()  
    {  
        // Declare a StudentName by using the constructor that has two  
        // parameters.  
        StudentName student1 = new StudentName("Craig", "Playstead");  
  
        // Make the same declaration by using an object initializer and  
        // sending  
        // arguments for the first and last names. The parameterless  
        // constructor is  
        // invoked in processing this declaration, not the constructor that  
        // has  
        // two parameters.  
        StudentName student2 = new StudentName  
        {  
            FirstName = "Craig",  
            LastName = "Playstead"  
        };  
  
        // Declare a StudentName by using an object initializer and sending  
        // an argument for only the ID property. No corresponding  
        // constructor is
```

```

        // necessary. Only the parameterless constructor is used to process
object
        // initializers.
StudentName student3 = new StudentName
{
    ID = 183
};

// Declare a StudentName by using an object initializer and sending
// arguments for all three properties. No corresponding constructor
is
        // defined in the class.
StudentName student4 = new StudentName
{
    FirstName = "Craig",
    LastName = "Playstead",
    ID = 116
};

Console.WriteLine(student1.ToString());
Console.WriteLine(student2.ToString());
Console.WriteLine(student3.ToString());
Console.WriteLine(student4.ToString());
}

// Output:
// Craig 0
// Craig 0
// 183
// Craig 116

public class StudentName
{
    // This constructor has no parameters. The parameterless constructor
    // is invoked in the processing of object initializers.
    // You can test this by changing the access modifier from public to
    // private. The declarations in Main that use object initializers
will
    // fail.
    public StudentName() { }

    // The following constructor has parameters for two of the three
    // properties.
    public StudentName(string first, string last)
    {
        FirstName = first;
        LastName = last;
    }

    // Properties.
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
    public int ID { get; set; }

    public override string ToString() => FirstName + " " + ID;
}

```

```
    }  
}
```

개체 이니셜라이저를 사용하여 개체의 이니셜라이저를 설정할 수 있습니다. 다음 예제에서는 인덱서를 사용하여 여러 포지션의 선수를 가져오고 설정하는 `BaseballTeam` 클래스를 정의합니다. 이 이니셜라이저는 포지션을 가리키는 약어 또는 각 포지션에 사용되는 번호를 기준으로 선수에게 애구 득점표를 할당할 수 있습니다.

C#

```
public class HowToIndexInitializer  
{  
    public class BaseballTeam  
    {  
        private string[] players = new string[9];  
        private readonly List<string> positionAbbreviations = new  
List<string>  
        {  
            "P", "C", "1B", "2B", "3B", "SS", "LF", "CF", "RF"  
        };  
  
        public string this[int position]  
        {  
            // Baseball positions are 1 - 9.  
            get { return players[position-1]; }  
            set { players[position-1] = value; }  
        }  
        public string this[string position]  
        {  
            get { return players[positionAbbreviations.IndexOf(position)]; }  
            set { players[positionAbbreviations.IndexOf(position)] = value; }  
        }  
    }  
  
    public static void Main()  
    {  
        var team = new BaseballTeam  
        {  
            ["RF"] = "Mookie Betts",  
            [4] = "Jose Altuve",  
            ["CF"] = "Mike Trout"  
        };  
  
        Console.WriteLine(team["2B"]);  
    }  
}
```

다음 예제에서는 매개 변수를 사용 또는 사용하지 않고 생성자를 사용하여 생성자 및 멤버 초기화의 실행 순서를 보여 줍니다.

## 참고 항목

- 개체 이니셜라이저 및 컬렉션 이니셜라이저

### ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

💡 설명서 문제 열기

✍️ 제품 사용자 의견 제공

# 컬렉션 이니셜라이저를 사용하여 사전을 초기화하는 방법(C# 프로그래밍 가이드)

아티클 • 2024. 05. 25.

`Dictionary< TKey, TValue >`에는 키/값 쌍의 컬렉션이 있습니다. 해당 `Add` 메서드는 두 개의 매개 변수를 사용하며, 하나는 키에, 다른 하나는 값에 사용됩니다.

`Dictionary< TKey, TValue >` 또는 여러 매개 변수를 사용하는 `Add` 메서드를 초기화하는 한 가지 방법은 다음 예제와 같이 각 매개 변수 집합을 중괄호로 묶는 것입니다. 또한 다음 예제와 같이 인덱스 이니셜라이저를 사용하는 방법도 있습니다.

## ① 참고

컬렉션을 초기화하는 이 두 가지 방법의 주요 차이점은 중복된 키가 있는 경우입니다. 예를 들면 다음과 같습니다.

C#

```
{ 111, new StudentName { FirstName="Sachin", LastName="Karnik", ID=211
} },
{ 111, new StudentName { FirstName="Dina", LastName="Salimzianova",
ID=317 } },
```

`Add` 메서드는 `ArgumentException`: 'An item with the same key has already been added. Key: 111' 을 throw하는 반면, 예의 두 번째 부분인 공용 읽기/쓰기 인덱서 메서드는 동일한 키로 기존 항목을 자동으로 덮어씁니다.

## 예시

다음 코드 예제에서 `Dictionary< TKey, TValue >`는 `StudentName` 유형의 인스턴스를 사용하여 초기화됩니다. 첫 번째 초기화에서는 `Add` 메서드와 두 인수를 사용합니다. 컴파일러는 각 `int` 키 및 `StudentName` 값 쌍에 `Add`에 대한 호출을 생성합니다. 두 번째 초기화에서는 `Dictionary` 클래스의 공용 읽기/쓰기 인덱서 메서드를 사용합니다.

C#

```
public class HowToDictionaryInitializer
{
    class StudentName
    {
        public string? FirstName { get; set; }
        public string? LastName { get; set; }
```

```

        public int ID { get; set; }

    }

    public static void Main()
    {
        var students = new Dictionary<int, StudentName>()
        {
            { 111, new StudentName { FirstName="Sachin", LastName="Karnik",
ID=211 } },
            { 112, new StudentName { FirstName="Dina",
LastName="Salimzianova", ID=317 } },
            { 113, new StudentName { FirstName="Andy", LastName="Ruth",
ID=198 } }
        };

        foreach(var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is
{students[index].FirstName} {students[index].LastName}");
        }
        Console.WriteLine();

        var students2 = new Dictionary<int, StudentName>()
        {
            [111] = new StudentName { FirstName="Sachin", LastName="Karnik",
ID=211 },
            [112] = new StudentName { FirstName="Dina",
LastName="Salimzianova", ID=317 } ,
            [113] = new StudentName { FirstName="Andy", LastName="Ruth",
ID=198 }
        };

        foreach (var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is
{students2[index].FirstName} {students2[index].LastName}");
        }
    }
}

```

첫 번째 선언에서 컬렉션의 각 요소에는 두 쌍의 중괄호가 있습니다. 안쪽 중괄호는 `StudentName`에 대한 개체 이니셜라이저를 묶고, 바깥쪽 중괄호는 `students Dictionary<TKey,TValue>`에 추가할 키/값 쌍에 대한 이니셜라이저를 묶습니다. 마지막으로, 사전에 대한 전체 컬렉션 이니셜라이저가 중괄호로 묶여 있습니다. 두 번째 초기화에서 할당의 왼쪽은 키이고, 오른쪽은 값이며 `StudentName`에 개체 이니셜라이저를 사용합니다.

## 참고 항목

- 개체 이니셜라이저 및 컬렉션 이니셜라이저

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 중첩 형식(C# 프로그래밍 가이드)

아티클 • 2023. 04. 07.

클래스, 구조체 또는 인터페이스 내에서 선언된 형식을 중첩 형식이라고 합니다. 예

C#

```
public class Container
{
    class Nested
    {
        Nested() { }
    }
}
```

외부 형식이 클래스, 인터페이스 또는 구조체 중 무엇인지에 관계없이, 중첩 형식은 기본적으로 `private`으로 설정됩니다. 즉, 포함하는 형식에서만 액세스할 수 있습니다. 앞의 예제에서 `Nested` 클래스는 외부 형식에서 액세스할 수 없습니다.

다음과 같이 `액세스 한정자`를 지정하여 중첩 형식의 접근성을 정의할 수도 있습니다.

- **클래스의** 중첩 형식은 `public`, `protected`, `internal`, `protected internal`, `private` 또는 `private protected`일 수 있습니다.

그러나 `sealed` 클래스 내에서 `protected`, `protected internal` 또는 `private protected` 중첩 클래스를 정의하면 컴파일러 경고 [CS0628](#), “`sealed` 클래스에 새 `protected` 멤버가 선언되었습니다.”가 생성됩니다.

또한 중첩 형식을 외부에서 볼 수 있도록 하는 것은 코드 품질 규칙 [CA1034](#)(“중첩 형식은 노출되면 안됨”)를 위반한다는 점에 유의하세요.

- **구조체의** 중첩 형식은 `public`, `internal` 또는 `private`일 수 있습니다.

다음 예제에서는 `Nested` 클래스를 `public`으로 설정합니다.

C#

```
public class Container
{
    public class Nested
    {
        Nested() { }
    }
}
```

중첩 형식(내부 형식)은 이 형식을 포함하고 있는 형식(외부 형식)에 액세스할 수 있습니다. 포함하는 형식에 액세스하려면 중첩 형식의 생성자에 인수로 전달합니다. 예를 들어:

```
C#  
  
public class Container  
{  
    public class Nested  
    {  
        private Container? parent;  
  
        public Nested()  
        {  
        }  
        public Nested(Container parent)  
        {  
            this.parent = parent;  
        }  
    }  
}
```

중첩 형식은 이 형식을 포함하는 형식에 액세스할 수 있는 모든 멤버에 액세스할 수 있습니다. 또한 상속 및 보호된 멤버를 포함하여 외부 형식의 개인 및 보호된 멤버에 액세스할 수 있습니다.

위 선언에서 `Nested` 클래스의 전체 이름은 `Container.Nested`입니다. 이 이름은 다음과 같이 중첩된 클래스의 새 인스턴스를 만드는 데 사용됩니다.

```
C#  
  
Container.Nested nest = new Container.Nested();
```

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [C# 형식 시스템](#)
- [액세스 한정자](#)
- [생성자](#)
- [CA1034 규칙](#)

# Partial 클래스 및 메서드(C# 프로그래밍 가이드)

아티클 • 2024. 03. 16.

클래스, 구조체, 인터페이스 또는 메서드의 정의를 둘 이상의 소스 파일에 분할할 수 있습니다. 각 소스 파일에는 형식 또는 메서드 정의 섹션이 있으며 모든 부분은 애플리케이션 이 컴파일될 때 결합됩니다.

## partial 클래스

클래스 정의를 분할하는 것이 바람직한 몇 가지 상황이 있습니다.

- 별도의 파일에 클래스를 선언하면 여러 프로그래머가 동시에 작업할 수 있습니다.
- 자동으로 생성된 원본을 포함하는 소스 파일을 다시 만들 필요 없이 클래스에 코드를 추가할 수 있습니다. Visual Studio에서는 Windows Forms, 웹 서비스 랙퍼 코드 등에 만들 때 이 방식을 사용합니다. Visual Studio에서 만든 파일을 수정하지 않고도 이러한 클래스를 사용하는 코드를 만들 수 있습니다.
- 원본 생성기는 클래스에서 추가 기능을 생성할 수 있습니다.

클래스 정의를 분할하려면 다음과 같이 `partial` 키워드 한정자를 사용합니다.

```
C#  
  
public partial class Employee  
{  
    public void DoWork()  
    {  
    }  
}  
  
public partial class Employee  
{  
    public void GoToLunch()  
    {  
    }  
}
```

`partial` 키워드는 클래스, 구조체 또는 인터페이스의 다른 부분을 네임스페이스에서 정의할 수 있음을 나타냅니다. 모든 부분은 `partial` 키워드를 사용해야 합니다. 최종 형식을 생성하려면 컴파일 시간에 모든 부분을 사용할 수 있어야 합니다. 모든 부분에 `public`, `private` 등의 동일한 액세스 가능성이 있어야 합니다.

부분이 abstract로 선언된 경우 전체 형식이 abstract로 간주됩니다. 부분이 sealed로 선언된 경우 전체 형식이 sealed로 간주됩니다. 부분이 기본 형식을 선언하는 경우 전체 형식이 해당 클래스를 상속합니다.

기본 클래스를 지정하는 부분은 모두 일치해야 하지만 기본 클래스를 생략하는 부분도 여전히 기본 형식을 상속합니다. 부분에서 다른 기본 인터페이스를 지정할 수 있으며, 최종 형식은 모든 partial 선언에 나열된 모든 인터페이스를 구현합니다. 부분 정의에 선언된 클래스, 구조체 또는 인터페이스 멤버는 다른 모든 부분에서 사용할 수 있습니다. 최종 형식은 컴파일 시간의 모든 부분 조합입니다.

### ① 참고

대리자 또는 열거형 선언에서는 partial 한정자를 사용할 수 없습니다.

다음 예제에서는 중첩된 형식이 부분 형식이 아니더라도 부분적일 수 있음을 보여 주는 예제입니다.

C#

```
class Container
{
    partial class Nested
    {
        void Test() { }
    }

    partial class Nested
    {
        void Test2() { }
    }
}
```

컴파일 시간에 부분 형식(Partial Type) 정의의 특성이 병합됩니다. 예를 들어 다음 선언을 살펴보세요.

C#

```
[SerializableAttribute]
partial class Moon { }

[ObsoleteAttribute]
partial class Moon { }
```

다음 선언과 동일합니다.

C#

```
[SerializableAttribute]  
[ObsoleteAttribute]  
class Moon { }
```

다음은 모든 부분 형식(Partial Type) 정의에서 병합됩니다.

- XML 주석
- interfaces
- 제네릭 형식 매개 변수 특성
- 클래스 특성
- 구성원

예를 들어 다음 선언을 살펴보세요.

C#

```
partial class Earth : Planet, IRotate { }  
partial class Earth : IRevolve { }
```

다음 선언과 동일합니다.

C#

```
class Earth : Planet, IRotate, IRevolve { }
```

## 제한 사항

부분 클래스 정의를 사용할 때 따라야 할 몇 가지 규칙이 있습니다.

- 동일한 형식의 일부로 작성된 모든 부분 형식(Partial Type) 정의를 `partial`로 수정해야 합니다. 예를 들어 다음 클래스 선언은 오류를 생성합니다.

C#

```
public partial class A { }  
//public class A { } // Error, must also be marked partial
```

- `partial` 한정자는 키워드(keyword) `class struct interface` 바로 앞에 나타날 수 있습니다.
- 다음 예제와 같이 부분 형식(Partial Type) 정의에 중첩된 부분 형식(Partial Type)을 사용할 수 있습니다.

C#

```
partial class ClassWithNestedClass
{
    partial class NestedClass { }
}

partial class ClassWithNestedClass
{
    partial class NestedClass { }
}
```

- 동일한 형식의 일부로 작성된 모든 부분 형식(Partial Type) 정의는 동일한 어셈블리와 동일한 모듈(.exe 또는 .dll 파일)에서 정의해야 합니다. 부분 정의는 여러 모듈에 걸쳐 있지 않습니다.
- 모든 부분 형식(Partial Type) 정의에서 클래스 이름 및 제네릭 형식 매개 변수가 일치해야 합니다. 제네릭 형식은 부분일 수 있습니다. 각 부분 선언에서 동일한 매개 변수 이름을 동일한 순서로 사용해야 합니다.
- 부분 형식 정의에 대한 다음 키워드(keyword) 선택 사항이지만 한 부분 형식 정의에 있는 경우 동일한 형식의 다른 부분 정의에 지정된 키워드(keyword) 충돌할 수 없습니다.
  - public
  - private
  - protected
  - internal
  - abstract
  - sealed
  - 기본 클래스
  - new 한정자(중첩된 부분)
  - 제네릭 제약 조건

자세한 내용은 [형식 매개 변수에 대한 제약 조건](#)을 참조하세요.

## 예제

다음 예제에서는 `Coords` 클래스의 생성자 및 필드가 하나의 partial 클래스 정의에서 선언되고 `PrintCoords` 멤버가 다른 partial 클래스 정의에서 선언됩니다.

C#

```
public partial class Coords
{
    private int x;
    private int y;
```

```

public Coords(int x, int y)
{
    this.x = x;
    this.y = y;
}

public partial class Coords
{
    public void PrintCoords()
    {
        Console.WriteLine("Coords: {0},{1}", x, y);
    }
}

class TestCoords
{
    static void Main()
    {
        Coords myCoords = new Coords(10, 15);
        myCoords.PrintCoords();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Coords: 10,15

```

다음 예제에서는 partial 구조체와 인터페이스도 개발할 수 있음을 보여 줍니다.

```

C#

partial interface ITest
{
    void Interface_Test();
}

partial interface ITest
{
    void Interface_Test2();
}

partial struct S1
{
    void Struct_Test() { }
}

partial struct S1
{
    void Struct_Test2() { }
}

```

# 부분 메서드

partial 클래스 또는 구조체는 partial 메서드를 포함할 수 있습니다. 클래스의 한 부분에는 메서드의 시그니처가 포함되어 있습니다. 구현은 동일한 부분 또는 다른 부분에서 정의될 수 있습니다.

서명이 다음 규칙을 준수하는 경우 부분 메서드에 구현이 필요하지 않습니다.

- 선언에는 액세스 한정자가 포함되지 않습니다. 메서드는 `private` 기본적으로 액세스 권한이 있습니다.
- 반환 형식은 .입니다 `void`.
- 매개 변수에는 한정자가 `out` 없습니다.
- 메서드 선언에는 다음 한정자를 포함할 수 없습니다.
  - `virtual`
  - `override`
  - `sealed`
  - `신규`
  - `extern`

메서드와 메서드에 대한 모든 호출은 구현이 없을 때 컴파일 시간에 제거됩니다.

해당 일부 제한 사항을 따르지 않는 모든 메서드(`public virtual partial void` 메서드)는 구현을 제공해야 합니다. 해당 구현은 원본 생성기에서 제공할 수 있습니다.

부분 메서드를 사용하면 클래스의 한 부분의 구현자가 메서드를 선언할 수 있습니다. 클래스의 다른 부분의 구현자는 해당 메서드를 정의할 수 있습니다. 이 분리가 유용한 두 가지 시나리오, 즉 상용구 코드를 생성하는 템플릿과 소스 생성기가 있습니다.

- **템플릿 코드:** 생성된 코드에서 메서드를 호출할 수 있도록 템플릿에서 메서드 이름 및 시그니처를 예약합니다. 이러한 메서드는 개발자가 메서드를 구현할지 여부를 결정할 수 있도록 지원하는 제한 사항을 따릅니다. 메서드가 구현되지 않은 경우 컴파일러는 메서드 시그니처와 메서드에 대한 모든 호출을 제거합니다. 호출의 인수 평가에서 발생하는 모든 결과를 포함하여 메서드 호출은 런타임에 영향을 주지 않습니다. 따라서 partial 클래스의 모든 코드는 구현이 제공되지 않더라도 partial 메서드를 자유롭게 사용할 수 있습니다. 메서드가 호출되었지만 구현되지 않은 경우 컴파일 시간 또는 런타임 오류가 발생하지 않습니다.
- **소스 생성기:** 소스 생성기는 메서드에 대한 구현을 제공합니다. 휴먼 개발자는 메서드 선언을 추가할 수 있습니다(소스 생성기에서 읽은 특성이 포함되는 경우가 많음). 개발자는 이러한 메서드를 호출하는 코드를 작성할 수 있습니다. 소스 생성기는 컴파일 중에 실행되고 구현을 제공합니다. 이 시나리오에서는 구현되지 않을 수 있는 부분 메서드에 대한 제한을 따르지 않는 경우가 많습니다.

```
// Definition in file1.cs
partial void OnNameChanged();

// Implementation in file2.cs
partial void OnNameChanged()
{
    // method body
}
```

- 부분 메서드 선언은 상황별 키워드 `partial`로 시작해야 합니다.
- 부분 형식(Partial Type)의 두 부분에 있는 부분 메서드 시그니처가 일치해야 합니다.
- 부분 메서드(Partial Method)는 `static` 및 `unsafe` 한정자를 사용할 수 없습니다.
- 부분 메서드(Partial Method)는 제네릭일 수 있습니다. 제약 조건은 메서드 선언 정의 및 구현에서 동일해야 합니다. 매개 변수 및 형식 매개 변수 이름은 구현 선언에서 정의 선언과 동일할 필요가 없습니다.
- 부분 메서드에 대한 대리자를 정의하고 구현할 수 있지만 구현이 없는 부분 메서드에는 대리자를 만들 수 없습니다.

## C# 언어 사양

자세한 내용은 C# 언어 사양의 Partial 형식 및 Partial 메서드를 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다. 부분 메서드에 대한 추가 기능은 가능 사양에 정의되어 있습니다.

## 참고 항목

- [클래스](#)
- [구조체 형식](#)
- [인터페이스](#)
- [partial\(형식\)](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.



### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

#### 설명서 문제 열기

#### 제품 사용자 의견 제공

# 쿼리에서 요소 속성의 하위 집합을 반환하는 방법(C# 프로그래밍 가이드)

아티클 • 2024. 03. 12.

이러한 조건이 둘 다 적용되는 경우 쿼리 식에서 무명 형식을 사용합니다.

- 각 소스 요소의 속성 중 일부만 반환하려고 합니다.
- 쿼리가 실행되는 메서드의 범위 외부에 쿼리 결과를 저장할 필요는 없습니다.

각 소스 요소에서 하나의 속성 또는 필드만 반환하려는 경우 `select` 절에 점 연산자만 사용할 수 있습니다. 예를 들어 각 `student`의 `ID`만 반환하려면 `select` 절을 다음과 같이 작성합니다.

C#

```
select student.ID;
```

## 예시

다음 예제에서는 무명 형식을 사용하여 지정된 조건과 일치하는 각 소스 요소의 속성 하위 집합만 반환하는 방법을 보여 줍니다.

C#

```
private static void QueryByScore()
{
    // Create the query. var is required because
    // the query produces a sequence of anonymous types.
    var queryHighScores =
        from student in students
        where student.ExamScores[0] > 95
        select new { student.FirstName, student.LastName };

    // Execute the query.
    foreach (var obj in queryHighScores)
    {
        // The anonymous type's properties were not named. Therefore
        // they have the same names as the Student properties.
        Console.WriteLine(obj.FirstName + ", " + obj.LastName);
    }
}
/* Output:
Adams, Terry
Fakhouri, Fadi
```

Garcia, Cesar  
Omelchenko, Svetlana  
Zabokritski, Eugene  
\*/

이름이 지정되지 않은 경우 무명 형식은 소스 요소의 이름을 해당 속성에 사용합니다. 무명 형식의 속성에 새 이름을 지정하려면 `select` 문을 다음과 같이 작성합니다.

C#

```
select new { First = student.FirstName, Last = student.LastName };
```

앞의 예제에서 이 작업을 수행하는 경우 `Console.WriteLine` 문도 변경되어야 합니다.

C#

```
Console.WriteLine(student.First + " " + student.Last);
```

## 코드 컴파일

이 코드를 실행하려면 `System.Linq`에 대한 `using` 지시문을 통해 클래스를 복사하여 C# 콘솔 애플리케이션 프로젝트에 붙여넣습니다.

## 참고 항목

- 익명 형식
- C#의 LINQ

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 명시적 인터페이스 구현(C# 프로그래밍 가이드)

아티클 • 2023. 04. 07.

클래스가 시그니처가 동일한 멤버를 포함하는 두 인터페이스를 구현하는 경우, 해당 멤버를 클래스에 구현하면 양쪽 인터페이스 모두가 해당 멤버를 구현에 사용합니다. 다음 예제에서 `Paint`에 대한 모든 호출은 같은 메서드를 호출합니다. 이 첫 번째 샘플에서는 다음 형식을 정의합니다.

C#

```
public interface IControl
{
    void Paint();
}

public interface ISurface
{
    void Paint();
}

public class SampleClass : IControl, ISurface
{
    // Both ISurface.Paint and IControl.Paint call this method.
    public void Paint()
    {
        Console.WriteLine("Paint method in SampleClass");
    }
}
```

다음 샘플에서는 다음 메서드를 호출합니다.

C#

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
sample.Paint();
control.Paint();
surface.Paint();

// Output:
// Paint method in SampleClass
// Paint method in SampleClass
// Paint method in SampleClass
```

그러나 두 인터페이스에 대해 모두 동일한 구현을 호출하지 않으려고 할 수 있습니다. 사용 중인 인터페이스에 따라 다른 구현을 호출하기 위해 인터페이스 멤버를 명시적으로 구현할 수 있습니다. 명시적 인터페이스 구현은 지정된 인터페이스를 통해서만 호출되는 클래스 멤버입니다. 클래스 이름 앞에 인터페이스 이름과 마침표를 추가하여 클래스 멤버의 이름을 지정합니다. 예를 들어:

C#

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {
        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint()
    {
        System.Console.WriteLine("ISurface.Paint");
    }
}
```

클래스 멤버 `IControl.Paint`는 `IControl` 인터페이스를 통해서만 사용할 수 있고 `ISurface.Paint`는 `ISurface`를 통해서만 사용할 수 있습니다. 두 메서드 구현은 서로 별개이며 어느 쪽도 클래스에서 직접 사용할 수 없습니다. 예를 들어:

C#

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
//sample.Paint(); // Compiler error.
control.Paint(); // Calls IControl.Paint on SampleClass.
surface.Paint(); // Calls ISurface.Paint on SampleClass.

// Output:
// IControl.Paint
// ISurface.Paint
```

명시적 구현은 두 인터페이스가 속성이나 메서드와 같이 동일한 이름을 가진 서로 다른 멤버를 선언하는 사례를 해결하는 데도 사용됩니다. 두 인터페이스를 모두 구현하려면 클래스는 `P` 속성이나 `P` 메서드 중 하나 또는 둘 다에 대해 명시적 구현을 사용하여 컴파일러 오류를 방지해야 합니다. 예를 들어:

C#

```

interface ILeft
{
    int P { get; }
}
interface IRight
{
    int P();
}

class Middle : ILeft, IRight
{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}

```

명시적 인터페이스 구현은 정의되어 있는 형식의 멤버로 액세스할 수 없기 때문에 해당 구현에는 액세스 한정자가 없습니다. 대신 인터페이스 인스턴스를 통해 호출되는 경우에만 액세스할 수 있습니다. 명시적 인터페이스 구현의 액세스 한정자를 지정하면 컴파일러 오류 CS0106이 발생합니다. 자세한 내용은 [interface\(C# 참조\)](#)를 참조하세요.

인터페이스에 선언된 멤버에 대한 구현을 정의할 수 있습니다. 클래스가 인터페이스로부터 메서드 구현을 상속하는 경우, 해당 메서드는 인터페이스 형식의 참조를 통해서만 액세스할 수 있습니다. 상속된 멤버는 public 인터페이스의 일부로 표시되지 않습니다. 다음 샘플은 인터페이스에 메서드에 대한 기본 구현을 정의합니다.

C#

```

public interface IControl
{
    void Paint() => Console.WriteLine("Default Paint method");
}
public class SampleClass : IControl
{
    // Paint() is inherited from IControl.
}

```

다음 샘플은 기본 구현을 호출합니다.

C#

```

var sample = new SampleClass();
//sample.Paint(); // "Paint" isn't accessible.
var control = sample as IControl;
control.Paint();

```

`IControl` 인터페이스를 구현하는 모든 클래스는 기본 `Paint` 메서드를 public 메서드로 또는 명시적 인터페이스 구현으로 재정의할 수 있습니다.

## 참고 항목

- C# 프로그래밍 가이드
- 개체 지향 프로그래밍
- 인터페이스
- 상속

# 인터페이스 멤버를 명시적으로 구현하는 방법(C# 프로그래밍 가이드)

아티클 • 2024. 03. 12.

이 예제에서는 `interface IDimensions` 및 `Box` 클래스를 선언합니다. 이 클래스는 인터페이스 멤버 `GetLength` 및 `GetWidth`를 명시적으로 구현합니다. 멤버는 인터페이스 인스턴스 `dimensions`를 통해 액세스합니다.

## 예시

C#

```
interface IDimensions
{
    float GetLength();
    float GetWidth();
}

class Box : IDimensions
{
    float lengthInches;
    float widthInches;

    Box(float length, float width)
    {
        lengthInches = length;
        widthInches = width;
    }
    // Explicit interface member implementation:
    float IDimensions.GetLength()
    {
        return lengthInches;
    }
    // Explicit interface member implementation:
    float IDimensions.GetWidth()
    {
        return widthInches;
    }

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new Box(30.0f, 20.0f);

        // Declare an interface instance dimensions:
        IDimensions dimensions = box1;

        // The following commented lines would produce compilation
//        Console.WriteLine("Length: " + box1.GetLength());
//        Console.WriteLine("Width: " + box1.GetWidth());
    }
}
```

```

        // errors because they try to access an explicitly implemented
        // interface member from a class instance:
        //System.Console.WriteLine("Length: {0}", box1.GetLength());
        //System.Console.WriteLine("Width: {0}", box1.GetWidth());

        // Print out the dimensions of the box by calling the methods
        // from an instance of the interface:
        System.Console.WriteLine("Length: {0}", dimensions.GetLength());
        System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
    }
}

/* Output:
   Length: 30
   Width: 20
*/

```

## 강력한 프로그래밍

- `Main` 메서드의 다음 줄은 컴파일 오류를 생성하므로 주석으로 처리되었습니다. 명시적으로 구현된 인터페이스 멤버는 `class` 인스턴스에서 액세스할 수 없습니다.

C#

```

//System.Console.WriteLine("Length: {0}", box1.GetLength());
//System.Console.WriteLine("Width: {0}", box1.GetWidth());

```

- 또한 `Main` 메서드의 다음 줄은 메서드가 인터페이스 인스턴스에서 호출되기 때문에 상자 크기를 성공적으로 출력합니다.

C#

```

System.Console.WriteLine("Length: {0}", dimensions.GetLength());
System.Console.WriteLine("Width: {0}", dimensions.GetWidth());

```

## 참고 항목

- [개체 지향 프로그래밍](#)
- [인터페이스](#)
- [두 인터페이스의 멤버를 명시적으로 구현하는 방법](#)

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 두 인터페이스의 멤버를 명시적으로 구현하는 방법(C# 프로그래밍 가이드)

아티클 • 2024. 03. 12.

명시적 인터페이스 구현을 통해 프로그래머는 멤버 이름이 같고 각 인터페이스 멤버에 별도 구현을 제공하는 두 인터페이스를 구현할 수도 있습니다. 이 예제에서는 미터 단위와 인치 단위 모두로 상자 크기를 표시합니다. 상자 `class`는 서로 다른 측정 시스템을 나타내는 두 인터페이스 `IEnglishDimensions` 및 `IMetricDimensions`를 구현합니다. 두 인터페이스에 동일한 멤버 이름 `Length`와 `Width`가 있습니다.

## 예시

C#

```
// Declare the English units interface:  
interface IEnglishDimensions  
{  
    float Length();  
    float Width();  
}  
  
// Declare the metric units interface:  
interface IMetricDimensions  
{  
    float Length();  
    float Width();  
}  
  
// Declare the Box class that implements the two interfaces:  
// IEnglishDimensions and IMetricDimensions:  
class Box : IEnglishDimensions, IMetricDimensions  
{  
    float lengthInches;  
    float widthInches;  
  
    public Box(float lengthInches, float widthInches)  
    {  
        this.lengthInches = lengthInches;  
        this.widthInches = widthInches;  
    }  
  
    // Explicitly implement the members of IEnglishDimensions:  
    float IEnglishDimensions.Length() => lengthInches;  
  
    float IEnglishDimensions.Width() => widthInches;  
  
    // Explicitly implement the members of IMetricDimensions:
```

```

float IMetricDimensions.Length() => lengthInches * 2.54f;

float IMetricDimensions.Width() => widthInches * 2.54f;

static void Main()
{
    // Declare a class instance box1:
    Box box1 = new Box(30.0f, 20.0f);

    // Declare an instance of the English units interface:
    IEnglishDimensions eDimensions = box1;

    // Declare an instance of the metric units interface:
    IMetricDimensions mDimensions = box1;

    // Print dimensions in English units:
    System.Console.WriteLine("Length(in): {0}", eDimensions.Length());
    System.Console.WriteLine("Width (in): {0}", eDimensions.Width());

    // Print dimensions in metric units:
    System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
    System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
}

/* Output:
   Length(in): 30
   Width (in): 20
   Length(cm): 76.2
   Width (cm): 50.8
*/

```

## 강력한 프로그래밍

기본 측정값을 인치 단위로 설정하려면 일반적으로 Length 및 Width 메서드를 구현하고, IMetricDimensions 인터페이스에서 Length 및 Width 메서드를 명시적으로 구현합니다.

C#

```

// Normal implementation:
public float Length() => lengthInches;
public float Width() => widthInches;

// Explicit implementation:
float IMetricDimensions.Length() => lengthInches * 2.54f;
float IMetricDimensions.Width() => widthInches * 2.54f;

```

이 경우 클래스 인스턴스에서 인치 단위에 액세스하고 인터페이스 인스턴스에서 미터 단위에 액세스할 수 있습니다.

C#

```
public static void Test()
{
    Box box1 = new Box(30.0f, 20.0f);
    IMetricDimensions mDimensions = box1;

    System.Console.WriteLine("Length(in): {0}", box1.Length());
    System.Console.WriteLine("Width (in): {0}", box1.Width());
    System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
    System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
}
```

## 참고 항목

- 개체 지향 프로그래밍
- 인터페이스
- 인터페이스 멤버를 명시적으로 구현하는 방법

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

[설명서 문제 열기](#)

[제품 사용자 의견 제공](#)

# 대리자(C# 프로그래밍 가이드)

아티클 • 2024. 04. 11.

대리자는 특정 매개 변수 목록 및 반환 형식이 있는 메서드에 대한 참조를 나타내는 형식입니다. 대리자를 인스턴스화하면 모든 메서드가 있는 인스턴스를 호환되는 시그니처 및 반환 형식에 연결할 수 있습니다. 대리자 인스턴스를 통해 메서드를 호출할 수 있습니다.

대리자는 메서드를 다른 메서드에 인수로 전달하는 데 사용됩니다. 이벤트 처리기는 대리자를 통해 호출되는 메서드라고 할 수 있습니다. 사용자 지정 메서드를 만들면 Windows 컨트롤 같은 클래스가 특정 이벤트가 발생했을 때 해당 메서드를 호출할 수 있습니다. 다음 예제에서는 대리자 선언을 보여 줍니다.

C#

```
public delegate int PerformCalculation(int x, int y);
```

액세스 가능한 클래스 또는 대리자 형식과 일치하는 구조의 모든 메서드는 대리자에 할당할 수 있습니다. 메서드는 정적 메서드이거나 인스턴스 메서드일 수 있습니다. 이러한 유연성은 프로그래밍 방식으로 메서드 호출을 변경하거나 기존 클래스에 새 코드를 삽입할 수 있음을 의미합니다.

## ① 참고

메서드 오버로드의 컨텍스트에서는 메서드 시그니처에 반환 값이 포함되지 않지만 대리자 컨텍스트에서는 시그니처에 반환 값이 포함됩니다. 즉 메서드의 반환 형식이 대리자의 반환 형식과 같아야 합니다.

대리자에서는 이와 같이 메서드를 매개 변수로 취급할 수 있으므로 대리자는 콜백 메서드 정의에 이상적입니다. 애플리케이션에서 두 개체를 비교하는 메서드를 작성할 수 있습니다. 이 메서드는 정렬 알고리즘의 대리자에서 사용할 수 있습니다. 비교 코드는 라이브러리와 별개이므로 정렬 메서드가 더욱 일반적일 수 있습니다.

함수 포인터는 호출 규칙을 더욱 세부적으로 제어해야 하는 유사한 시나리오용으로 C# 9에 추가되었습니다. 대리자와 연결된 코드는 대리자 형식에 추가된 가상 메서드를 사용하여 호출됩니다. 함수 포인터를 사용하여 다른 규칙을 지정할 수 있습니다.

## 대리자 개요

대리자에는 다음과 같은 속성이 있습니다.

- 대리자는 C++ 함수 포인터와 유사하지만 C++ 함수 포인터와 달리 멤버 함수에 대해 완전히 개체 지향입니다. 대리자는 개체 인스턴스 및 메서드를 모두 캡슐화합니다.
- 대리자를 통해 메서드를 매개 변수로 전달할 수 있습니다.
- 대리자를 사용하여 콜백 메서드를 정의할 수 있습니다.
- 여러 대리자를 연결할 수 있습니다. 예를 들어 단일 이벤트에 대해 여러 메서드를 호출할 수 있습니다.
- 메서드는 대리자 형식과 정확히 일치하지 않아도 됩니다. 자세한 내용은 [대리자의 가변성 사용](#)을 참조하세요.
- 람다 식은 인라인 코드 블록을 작성하는 더욱 간단한 방법입니다. 특정 컨텍스트에서는 람다 식이 대리자 형식으로 컴파일됩니다. 람다 식에 대한 자세한 내용은 [람다 식](#)을 참조하세요.

## 섹션 내용

- [대리자 사용](#)
- [인터페이스\(C# 프로그래밍 가이드\)](#) 대신 대리자를 사용하는 경우
- 명명된 메서드 및 무명 메서드가 있는 대리자
- [대리자의 가변성 사용](#)
- [대리자를 결합하는 방법\(멀티캐스트 대리자\)](#)
- [대리자를 선언, 인스턴스화, 사용하는 방법](#)

## C# 언어 사양

자세한 내용은 [대리자](#)에 [C# 언어 사양](#)합니다. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

## 중요 설명서 장

- [대리자, Events, and Lambda Expressions](#)에 [C# 3.0 Cookbook, Third Edition: 250 개 이상의 솔루션에 대한 C# 3.0 프로그래머](#)
- [C# 3.0 학습: C# 3.0의 기본 사항의 대리자 및 이벤트](#)

## 참고 항목

- [Delegate](#)
- [C# 프로그래밍 가이드](#)
- [이벤트](#)

# 피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공 ↗](#)

# 대리자 사용(C# 프로그래밍 가이드)

아티클 • 2024. 03. 12.

대리자는 C 및 C++의 함수 포인터처럼 메서드를 안전하게 캡슐화하는 형식입니다. 함수 포인터와는 달리 대리자는 개체 지향적이고 형식이 안전하며 보안이 유지됩니다. 대리자의 형식은 대리자의 이름으로 정의됩니다. 다음 예제에서는 `string`을 인수로 사용하고 `void`를 반환하는 메서드를 캡슐화할 수 있는 `Callback` 대리자를 선언합니다.

C#

```
public delegate void Callback(string message);
```

대리자 개체는 일반적으로 대리자가 래핑할 메서드의 이름을 제공하거나 [람다 식](#)을 사용하여 생성합니다. 이러한 방식으로 대리자가 인스턴스화되면 호출될 수 있습니다. 대리자를 호출하면 대리자 인스턴스에 연결된 메서드가 호출됩니다. 호출자가 대리자에게 전달한 매개 변수가 메서드로 전달되며 메서드의 반환 값(있는 경우)이 대리자에 의해 호출자로 반환됩니다. 예시:

C#

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}
```

C#

```
// Instantiate the delegate.
Callback handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

대리자 형식은 .NET의 `Delegate` 클래스에서 파생되며, [봉인](#)되어 있으므로 `Delegate`에서만 파생될 수 없으며 해당 클래스에서 사용자 지정 클래스를 파생할 수도 없습니다. 인스턴스화된 대리자는 개체이므로 인수로 전달되거나 속성에 할당될 수 있습니다. 따라서 메서드가 대리자를 매개 변수로 허용하고 나중에 대리자를 호출할 수 있습니다. 비동기 콜백이라는 이러한 방식은 긴 프로세스 완료 시 호출자에게 알림을 제공하는 일반적인 방법입니다. 이러한 방식으로 대리자를 사용하면 대리자를 사용하는 코드가 사용 중인 메서드의 구현을 확인하지 않아도 됩니다. 이 기능은 캡슐화 인터페이스에서 제공하는 기능과 비슷합니다.

콜백은 사용자 지정 비교 메서드를 정의하고 해당 대리자를 정렬 메서드로 전달할 때도 일반적으로 사용됩니다. 이 경우 호출자의 코드를 정렬 알고리즘의 일부분으로 포함할 수 있습니다. 다음 예제 메서드는 `De1` 형식을 매개 변수로 사용합니다.

C#

```
public static void MethodWithCallback(int param1, int param2, Callback
callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

위에서 작성된 대리자를 해당 메서드로 전달할 수 있습니다.

C#

```
MethodWithCallback(1, 2, handler);
```

그러면 콘솔에 다음 출력이 표시됩니다.

콘솔

```
The number is: 3
```

대리자를 추상화로 사용하는 경우 `MethodWithCallback`이 콘솔을 직접 호출할 필요가 없으며 콘솔을 호출하기 위해 이 메서드를 지정하지 않아도 됩니다. `MethodWithCallback`은 단순히 문자열을 준비하여 다른 메서드로 전달할 뿐입니다. 위임된 메서드는 매개 변수를 필요한 수만큼 사용할 수 있으므로 이러한 방식은 특히 유용합니다.

인스턴스 메서드를 래핑하기 위한 대리를 생성할 때 해당 대리는 인스턴스와 메서드를 모두 참조합니다. 대리는 래핑 대상 메서드 이외의 인스턴스 형식은 알 수 없으므로 해당 개체에 대리자 서명과 일치하는 메서드가 있으면 모든 개체 형식을 참조할 수 있습니다. 정적 메서드를 래핑하기 위해 생성하는 대리는 메서드만 참조합니다. 다음의 선언을 살펴보세요.

C#

```
public class MethodClass
{
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

이제는 위에 나와 있는 정적 `DelegateMethod`와 함께 3개 메서드를 `d1` 인스턴스로 래핑 할 수 있습니다.

대리자는 호출 시 둘 이상의 메서드를 호출할 수 있습니다. 이러한 호출을 멀티캐스트라고 합니다. 대리자의 메서드 목록(호출 목록)에 메서드를 더 추가하려는 경우 더하기 또는 더하기 대입 연산자('+' 또는 '+=')를 사용하여 대리자만 두 개 더 추가하면 됩니다. 예시:

C#

```
var obj = new MethodClass();
Callback d1 = obj.Method1;
Callback d2 = obj.Method2;
Callback d3 = DelegateMethod;

//Both types of assignment are valid.
Callback allMethodsDelegate = d1 + d2;
allMethodsDelegate += d3;
```

이 시점에서 `allMethodsDelegate`의 호출 목록에는 `Method1`, `Method2`, `DelegateMethod`의 3개 메서드가 포함되어 있습니다. 원래 대리자 3개(`d1`, `d2`, `d3`)는 그대로 유지됩니다.

`allMethodsDelegate`를 호출하면 3개 메서드가 모두 순서대로 호출됩니다. 대리자가 참조 매개 변수를 사용하는 경우 참조는 각 3개 메서드에 순서대로 전달되며 메서드 하나의 변경 내용은 다음 메서드에도 표시됩니다. 메서드 중 하나라도 메서드 내에서 catch되지 않은 예외를 throw하면 해당 예외가 대리자의 호출자에게 해당 예외가 전달되며 호출 목록의 후속 메서드는 호출되지 않습니다. 반환 값 및/또는 out 매개 변수를 포함하는 대리자는 마지막으로 호출한 메서드의 반환 값과 매개 변수를 반환합니다. 호출 목록에서 메서드를 제거하려면 **빼기 또는 빼기 대입 연산자**(`-` 또는 `-=`)를 사용합니다. 예시:

C#

```
//remove Method1
allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2
Callback oneMethodDelegate = allMethodsDelegate - d2;
```

대리자 형식은 `System.Delegate`에서 파생되므로 해당 클래스로 정의되는 메서드와 속성을 대리자에서 호출할 수 있습니다. 예를 들어 대리자의 호출 목록에 있는 메서드 수를 확인하려는 경우 다음과 같이 코드를 작성하면 됩니다.

C#

```
int invocationCount = d1.GetInvocationList().GetLength(0);
```

호출 목록에 메서드가 둘 이상 포함된 대리자는 [MulticastDelegate](#)의 하위 클래스인 [System.Delegate](#)에서 파생됩니다. 두 클래스가 모두 [GetInvocationList](#)를 지원하므로 위의 코드는 두 경우에 모두 사용 가능합니다.

멀티캐스트 대리자는 이벤트 처리에서 광범위하게 사용됩니다. 이벤트 소스 개체는 해당 이벤트를 받도록 등록된 받는 사람 개체에 이벤트 알림을 보냅니다. 이벤트를 등록하기 위해 받는 사람은 이벤트를 처리하도록 설계된 메서드를 만든 다음 해당 메서드의 대리자를 만들어 이벤트 소스로 전달합니다. 소스는 이벤트 발생 시 대리자를 호출합니다. 그러면 대리자가 받는 사람에 대해 이벤트 처리 메서드를 호출하여 이벤트 데이터를 전달합니다. 지정된 이벤트의 대리자 형식은 이벤트 소스에 의해 정의됩니다. 자세한 내용은 [이벤트](#)를 참조하세요.

컴파일 시간에 할당된 서로 다른 형식의 두 대리자를 비교하면 컴파일 오류가 발생합니다. 대리자 인스턴스가 정적 [System.Delegate](#) 형식이면 비교는 허용되지만 런타임에서 `false`가 반환됩니다. 다음은 그 예입니다.

C#

```
delegate void Callback1();
delegate void Callback2();

static void method(Callback1 d, Callback2 e, System.Delegate f)
{
    // Compile-time error.
    //Console.WriteLine(d == e);

    // OK at compile-time. False if the run-time type of f
    // is not the same as that of d.
    Console.WriteLine(d == f);
}
```

## 참고 항목

- [대리자](#)
- [대리자의 가변성 사용](#)
- [대리자의 가변성](#)
- [Func 및 Action 제네릭 대리자에 가변성 사용](#)
- [이벤트](#)

 GitHub에서 Microsoft와 공동 작업

.NET

.NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

제공해 주세요.

💡 설명서 문제 열기

↗️ 제품 사용자 의견 제공

# 명명된 메서드와 무명 메서드의 대리자 비교(C# 프로그래밍 가이드)

아티클 • 2024. 03. 12.

대리자는 명명된 메서드에 연결할 수 있습니다. 명명된 메서드를 사용하여 대리자를 인스턴스화하면 메서드가 매개 변수로 전달됩니다. 예를 들면 다음과 같습니다.

C#

```
// Declare a delegate.  
delegate void WorkCallback(int x);  
  
// Define a named method.  
void DoWork(int k) { /* ... */ }  
  
// Instantiate the delegate using the method as a parameter.  
WorkCallback d = obj.DoWork;
```

이 코드는 명명된 메서드를 사용하여 호출됩니다. 명명된 메서드를 사용하여 생성된 대리자는 정적 메서드 또는 인스턴스 메서드를 캡슐화할 수 있습니다. 명명된 메서드는 이전 버전의 C#에서 대리자를 인스턴스화할 수 있는 유일한 방법입니다. 그러나 새 메서드 생성이 불필요한 오버헤드인 경우 C#에서 대리자를 인스턴스화하고 호출 시 대리자에서 처리할 코드 블록을 즉시 지정할 수 있습니다. 블록에는 람다 식 또는 무명 메서드가 포함될 수 있습니다.

대리자 매개 변수로 전달하는 메서드에는 대리자 선언과 동일한 시그니처가 있어야 합니다. 대리자 인스턴스는 정적 또는 인스턴스 메서드를 캡슐화할 수 있습니다.

## ① 참고

대리자는 out 매개 변수를 사용할 수 있지만, 멀티캐스트 이벤트 대리자의 경우 호출 될 대리자를 알 수 없기 때문에 사용하지 않는 것이 좋습니다.

C# 10부터, 하나의 오버로드가 있는 메서드 그룹은 '자연 형식'을 갖습니다. 즉, 컴파일러는 대리자 형식의 반환 형식과 매개 변수 형식을 유추할 수 있습니다.

C#

```
var read = Console.Read; // Just one overload; Func<int> inferred  
var write = Console.Write; // ERROR: Multiple overloads, can't choose
```

# 예제

다음은 대리자를 선언하고 사용하는 간단한 예제입니다. 대리자 `De1` 및 연결된 메서드 `MultiplyNumbers`에 동일한 시그니처가 있습니다.

C#

```
// Declare a delegate
delegate void MultiplyCallback(int i, double j);

class MathClass
{
    static void Main()
    {
        MathClass m = new MathClass();

        // Delegate instantiation using "MultiplyNumbers"
        MultiplyCallback d = m.MultiplyNumbers;

        // Invoke the delegate object.
        Console.WriteLine("Invoking the delegate using 'MultiplyNumbers':");
        for (int i = 1; i <= 5; i++)
        {
            d(i, 2);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // Declare the associated method.
    void MultiplyNumbers(int m, double n)
    {
        Console.Write(m * n + " ");
    }
}
/* Output:
   Invoking the delegate using 'MultiplyNumbers':
   2 4 6 8 10
*/
```

다음 예제에서는 한 대리자가 정적 메서드와 인스턴스 메서드 모두에 매핑되며 각각의 특정 정보를 반환합니다.

C#

```
// Declare a delegate
delegate void Callback();

class SampleClass
```

```

{
    public void InstanceMethod()
    {
        Console.WriteLine("A message from the instance method.");
    }

    static public void StaticMethod()
    {
        Console.WriteLine("A message from the static method.");
    }
}

class TestSampleClass
{
    static void Main()
    {
        var sc = new SampleClass();

        // Map the delegate to the instance method:
        Callback d = sc.InstanceMethod;
        d();

        // Map to the static method:
        d = SampleClass.StaticMethod;
        d();
    }
}
/* Output:
   A message from the instance method.
   A message from the static method.
*/

```

## 참고 항목

- 대리자
- 대리자를 결합하는 방법(멀티캐스트 대리자)
- 이벤트

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.



### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

#### 설명서 문제 열기

#### 제품 사용자 의견 제공

# 대리자를 결합하는 방법(멀티캐스트 대리자)(C# 프로그래밍 가이드)

아티클 • 2024. 03. 12.

이 예제에서는 멀티캐스트 대리자를 만드는 방법을 보여 줍니다. [대리자](#) 개체의 유용한 속성은 `+` 연산자를 사용하여 하나의 대리자 인스턴스에 여러 개체를 할당할 수 있다는 것입니다. 멀티캐스트 대리자는 할당된 대리자 목록을 포함합니다. 멀티캐스트 대리자가 호출되면 목록에 있는 대리자가 순서대로 호출됩니다. 같은 형식의 대리자만 결합할 수 있습니다.

- 연산자는 멀티캐스트 대리자에서 구성 요소 대리자를 제거하는 데 사용할 수 있습니다.

## 예시

C#

```
using System;

// Define a custom delegate that has a string parameter and returns void.
delegate void CustomCallback(string s);

class TestClass
{
    // Define two methods that have the same signature as CustomCallback.
    static void Hello(string s)
    {
        Console.WriteLine($"  Hello, {s}!");
    }

    static void Goodbye(string s)
    {
        Console.WriteLine($"  Goodbye, {s}!");
    }

    static void Main()
    {
        // Declare instances of the custom delegate.
        CustomCallback hiDel, byeDel, multiDel, multiMinusHiDel;

        // In this example, you can omit the custom delegate if you
        // want to and use Action<string> instead.
        //Action<string> hiDel, byeDel, multiDel, multiMinusHiDel;

        // Initialize the delegate object hiDel that references the
        // method Hello.
```

```

        hiDel = Hello;

        // Initialize the delegate object byeDel that references the
        // method Goodbye.
        byeDel = Goodbye;

        // The two delegates, hiDel and byeDel, are combined to
        // form multiDel.
        multiDel = hiDel + byeDel;

        // Remove hiDel from the multicast delegate, leaving byeDel,
        // which calls only the method Goodbye.
        multiMinusHiDel = multiDel - hiDel;

        Console.WriteLine("Invoking delegate hiDel:");
        hiDel("A");
        Console.WriteLine("Invoking delegate byeDel:");
        byeDel("B");
        Console.WriteLine("Invoking delegate multiDel:");
        multiDel("C");
        Console.WriteLine("Invoking delegate multiMinusHiDel:");
        multiMinusHiDel("D");
    }
}

/* Output:
Invoking delegate hiDel:
Hello, A!
Invoking delegate byeDel:
Goodbye, B!
Invoking delegate multiDel:
Hello, C!
Goodbye, C!
Invoking delegate multiMinusHiDel:
Goodbye, D!
*/

```

## 참고 항목

- MulticastDelegate
- 이벤트

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할



### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

### 설명서 문제 열기

수도 있습니다. 자세한 내용은  
참여자 가이드를 참조하세요.

☞ 제품 사용자 의견 제공

# 대리자를 선언, 인스턴스화, 사용하는 방법(C# 프로그래밍 가이드)

아티클 • 2023. 04. 07.

다음과 같은 메서드 중 하나라도 사용하여 대리자를 선언할 수 있습니다.

- 대리자 형식을 선언하고 그와 일치하는 서명이 있는 메서드를 선언합니다.

C#

```
// Declare a delegate.  
delegate void Del(string str);  
  
// Declare a method with the same signature as the delegate.  
static void Notify(string name)  
{  
    Console.WriteLine($"Notification received for: {name}");  
}
```

C#

```
// Create an instance of the delegate.  
Del del1 = new Del(Notify);
```

- 대리자 형식에 메서드 그룹을 할당합니다.

C#

```
// C# 2.0 provides a simpler way to declare an instance of Del.  
Del del2 = Notify;
```

- 익명 메서드를 선언합니다.

C#

```
// Instantiate Del by using an anonymous method.  
Del del3 = delegate(string name)  
    { Console.WriteLine($"Notification received for: {name}"); };
```

- 다음의 람다식을 사용합니다.

C#

```
// Instantiate Del by using a lambda expression.  
Del del4 = name => { Console.WriteLine($"Notification received for:  
{name}"); };
```

자세한 내용은 [람다 식](#)을 참조하세요.

다음 예제에서는 대리자를 선언, 인스턴스화 및 사용하는 방법을 보여 줍니다. `BookDB` 클래스는 책 데이터베이스를 유지 관리하는 서점 데이터베이스를 캡슐화합니다. 그리고 데이터베이스의 모든 문고판 책을 찾아 각 책에 대해 대리자를 호출하는 `ProcessPaperbackBooks` 메서드를 표시합니다. 이때 사용되는 `delegate` 형식의 이름은 `ProcessBookCallback`입니다. `Test` 클래스는 이 클래스를 사용하여 문고판 책의 제목과 평균 가격을 인쇄합니다.

대리자를 사용하면 서점 데이터베이스와 클라이언트 코드 간에 기능을 효율적으로 구분할 수 있습니다. 클라이언트 코드는 책이 저장되는 방식이나 서점 코드가 문고판 책을 찾는 방식을 알 수 없습니다. 서점 코드는 발견된 문고판 책에 대해 수행되는 처리를 알 수 없습니다.

## 예제

C#

```
// A set of classes for handling a bookstore:  
namespace Bookstore  
{  
    using System.Collections;  
  
    // Describes a book in the book list:  
    public struct Book  
    {  
        public string Title;           // Title of the book.  
        public string Author;          // Author of the book.  
        public decimal Price;          // Price of the book.  
        public bool Paperback;         // Is it paperback?  
  
        public Book(string title, string author, decimal price, bool  
paperBack)  
        {  
            Title = title;  
            Author = author;  
            Price = price;  
            Paperback = paperBack;  
        }  
    }  
  
    // Declare a delegate type for processing a book:  
    public delegate void ProcessBookCallback(Book book);
```

```
// Maintains a book database.
public class BookDB
{
    // List of all books in the database:
    ArrayList list = new ArrayList();

    // Add a book to the database:
    public void AddBook(string title, string author, decimal price, bool
paperBack)
    {
        list.Add(new Book(title, author, price, paperBack));
    }

    // Call a passed-in delegate on each paperback book to process it:
    public void ProcessPaperbackBooks(ProcessBookCallback processBook)
    {
        foreach (Book b in list)
        {
            if (b.Paperback)
                // Calling the delegate:
                processBook(b);
        }
    }
}

// Using the Bookstore classes:
namespace BookTestClient
{
    using Bookstore;

    // Class to total and average prices of books:
    class PriceTotaller
    {
        int countBooks = 0;
        decimal priceBooks = 0.0m;

        internal void AddBookToTotal(Book book)
        {
            countBooks += 1;
            priceBooks += book.Price;
        }

        internal decimal AveragePrice()
        {
            return priceBooks / countBooks;
        }
    }

    // Class to test the book database:
    class Test
    {
        // Print the title of the book.
        static void PrintTitle(Book b)
```

```

    {
        Console.WriteLine($"    {b.Title}");
    }

    // Execution starts here.
    static void Main()
    {
        BookDB bookDB = new BookDB();

        // Initialize the database with some books:
        AddBooks(bookDB);

        // Print all the titles of paperbacks:
        Console.WriteLine("Paperback Book Titles:");

        // Create a new delegate object associated with the static
        // method Test.PrintTitle:
        bookDB.ProcessPaperbackBooks(PrintTitle);

        // Get the average price of a paperback by using
        // a PriceTotaller object:
        PriceTotaller totaller = new PriceTotaller();

        // Create a new delegate object associated with the nonstatic
        // method AddBookToTotal on the object totaller:
        bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);

        Console.WriteLine("Average Paperback Book Price: ${0:#.##}",
            totaller.AveragePrice());
    }

    // Initialize the book database with some test books:
    static void AddBooks(BookDB bookDB)
    {
        bookDB.AddBook("The C Programming Language", "Brian W. Kernighan
and Dennis M. Ritchie", 19.95m, true);
        bookDB.AddBook("The Unicode Standard 2.0", "The Unicode
Consortium", 39.95m, true);
        bookDB.AddBook("The MS-DOS Encyclopedia", "Ray Duncan", 129.95m,
false);
        bookDB.AddBook("Dogbert's Clues for the Clueless", "Scott
Adams", 12.00m, true);
    }
}

/* Output:
Paperback Book Titles:
The C Programming Language
The Unicode Standard 2.0
Dogbert's Clues for the Clueless
Average Paperback Book Price: $23.97
*/

```

# 강력한 프로그래밍

- 대리자 선언

다음 문은 새 대리자 형식을 선언합니다.

```
C#
```

```
public delegate void ProcessBookCallback(Book book);
```

각 대리자 형식은 인수의 수와 형식 및 캡슐화 가능한 메서드의 형식과 반환 값을 설명합니다. 새 인수 형식 또는 반환 값 형식 집합이 필요할 때마다 새 대리자 형식을 선언해야 합니다.

- 대리자 인스턴스화

대리자 형식을 선언한 후에는 대리자 객체를 생성하여 특정 메서드와 연결해야 합니다. 이전 예제의 경우 다음 예제와 같이 `PrintTitle` 메서드를 `ProcessPaperbackBooks` 메서드에 전달하여 이 작업을 수행합니다.

```
C#
```

```
bookDB.ProcessPaperbackBooks(PrintTitle);
```

이렇게 하면 정적 메서드 `Test.PrintTitle`에 연결된 새 대리자 객체가 생성됩니다. 마찬가지로 객체 `totaller`의 비정적 메서드 `AddBookToTotal`도 다음 예제와 같이 전달됩니다.

```
C#
```

```
bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);
```

두 경우 모두 새 대리자 객체가 `ProcessPaperbackBooks` 메서드로 전달됩니다.

대리자를 만든 후에도 대리자가 연결된 메서드는 변경되지 않습니다. 대리자 객체는 변경이 불가능합니다.

- 대리자 호출

생성된 대리자 객체는 대개 대리자를 호출하는 다른 코드로 전달됩니다. 대리자 객체의 이름 뒤에 대리자로 전달할 인수를 괄호로 묶어 추가한 형식을 사용하여 대리자 객체를 호출합니다. 아래에 대리자 호출의 예가 나와 있습니다.

```
C#
```

```
processBook(b);
```

대리자는 이 예제와 같이 동기적으로 호출할 수도 있고 `BeginInvoke` 및 `EndInvoke` 메서드를 사용하여 비동기적으로 호출할 수도 있습니다.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [이벤트](#)
- [대리자](#)

# 문자열 및 문자열 리터럴

아티클 • 2024. 04. 11.

문자열은 값이 텍스트인 [String](#) 형식의 개체입니다. 내부적으로 텍스트는 [Char](#) 개체의 순차적 읽기 전용 컬렉션으로 저장됩니다. C# 문자열의 끝에 null 종료 문자가 없습니다. 따라서 C# 문자열에는 임의 개수의 포함된 null 문자('\'0')가 포함될 수 있습니다. 문자열의 [Length](#) 속성은 유니코드 문자 수가 아닌 포함된 [Char](#) 개체 수를 나타냅니다. 문자열에서 개별 유니코드 코드 포인트에 액세스하려면 [StringInfo](#) 개체를 사용합니다.

## 문자열과 System.String

C#에서 `string` 키워드는 [String](#)의 별칭입니다. 따라서 `String` 및 `string` 는 없이도 `using System;` 작동하므로 제공된 별칭을 `string` 사용하는 것이 좋습니다. [String](#) 클래스는 문자열을 안전하게 작성, 조작 및 비교할 수 있도록 다양한 메서드를 제공합니다. 또한 C# 언어는 일반적인 문자열 작업을 간소화하기 위해 일부 연산자를 오버로드합니다. 키워드에 대한 자세한 내용은 [string](#)을 참조하세요. 형식 및 메서드에 대한 자세한 내용은 [String](#)을 참조하세요.

## 문자열 선언 및 초기화

다음 예제에서와 같이 다양한 방법으로 문자열을 선언하고 초기화할 수 있습니다.

C#

```
// Declare without initializing.
string message1;

// Initialize to null.
string message2 = null;

// Initialize as an empty string.
// Use the Empty constant instead of the literal "".
string message3 = System.String.Empty;

// Initialize with a regular string literal.
string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";

// Initialize with a verbatim string literal.
string newPath = @"c:\\Program Files\\Microsoft Visual Studio 9.0";

// Use System.String if you prefer.
System.String greeting = "Hello World!";

// In local variables (i.e. within a method body)
// you can use implicit typing.
```

```

var temp = "I'm still a strongly-typed System.String!";

// Use a const string to prevent 'message4' from
// being used to store another string value.
const string message4 = "You can't get rid of me!";

// Use the String constructor only when creating
// a string from a char*, char[], or sbyte*. See
// System.String documentation for details.
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);

```

chars 배열을 사용하여 문자열을 초기화하는 경우를 제외하고 새 연산자를 사용하여 문자열 개체를 만들지 않습니다.

문자열 길이가 0인 새 `String` 개체를 만들려면 `Empty` 상수 값이 포함된 문자열을 초기화하세요. 빈 문자열을 문자열 리터럴로 나타내면 ""로 표시됩니다. `null` 대신 `Empty` 값이 포함된 문자열을 초기화하면 `NullReferenceException` 발생을 줄일 수 있습니다. 액세스하기 전에 문자열의 값을 확인하려면 정적 `IsNullOrEmpty(String)` 메서드를 사용하세요.

## 문자열의 불변성

문자열 개체는 **변경할 수 없습니다**. 문자열 개체를 만든 후에는 변경할 수 없습니다. 실제로 문자열을 수정하는 것으로 나타나는 모든 `String` 메서드 및 C# 연산자는 새로운 문자열 개체에 결과를 반환합니다. 다음 예제에서 `s1` 및 `s2`의 콘텐츠는 단일 문자열을 형성하도록 연결되며, 두 개의 원본 문자열은 변경되지 않습니다. `+=` 연산자는 결합된 콘텐츠를 포함하는 새 문자열을 만듭니다. 새 개체는 `s1` 변수에 할당되며, 참조를 유지하는 변수가 없으므로 `s1`에 할당된 원래 개체는 가비지 수집을 위해 해제됩니다.

C#

```

string s1 = "A string is more ";
string s2 = "than the sum of its chars.";

// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.

```

문자열 "수정"은 실제로 새 문자열을 만드는 것이므로 문자열에 대한 참조를 만들 때 주 의해야 합니다. 문자열에 대한 참조를 만든 후 원래 문자열을 "수정"하더라도 참조는 문

자열을 수정할 때 만든 새 개체가 아니라 원래 개체를 계속 가리킵니다. 이 동작은 다음 코드에서 볼 수 있습니다.

```
C#  
  
string str1 = "Hello ";  
string str2 = str1;  
str1 += "World";  
  
System.Console.WriteLine(str2);  
//Output: Hello
```

원래 문자열에 대한 검색 및 바꾸기 작업과 같이, 수정을 기반으로 하는 새 문자열 작성 방법에 대한 자세한 내용은 [문자열 콘텐츠 수정 방법](#)을 참조하세요.

## 따옴표 붙은 문자열 리터럴

따옴표 붙은 문자열 리터럴은 동일한 줄에 작은따옴표 문자(“)로 시작되고 끝납니다. 따옴표 붙은 문자열 리터럴은 한 줄에 맞고 이 [스케이프 시퀀스](#)를 포함하지 않는 문자열에 가장 적합합니다. 따옴표 붙은 문자열 리터럴은 다음 예제와 같이 이스케이프 문자를 포함해야 합니다.

```
C#  
  
string columns = "Column 1\tColumn 2\tColumn 3";  
//Output: Column 1           Column 2           Column 3  
  
string rows = "Row 1\r\nRow 2\r\nRow 3";  
/* Output:  
   Row 1  
   Row 2  
   Row 3  
 */  
  
string title = "\"The \u00C6olean Harp\", by Samuel Taylor Coleridge";  
//Output: "The Äolean Harp", by Samuel Taylor Coleridge
```

## 축자 문자열 리터럴

축자 문자열 리터럴은 여러 줄 문자열, 백슬래시 문자가 포함된 문자열 또는 포함된 큰따옴표에 더 편리합니다. 축자 문자열은 문자열 텍스트의 일부로 새 줄 문자를 유지합니다. 축자 문자열 내에 따옴표를 포함하려면 큰따옴표를 사용하세요. 다음 예제에서는 몇 가지 일반적인 축자 문자열에 대한 사용을 보여 줍니다.

```
C#
```

```

string filePath = @"C:\Users\scoleridge\Documents\";
//Output: C:\Users\scoleridge\Documents\

string text = @"My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...";
/* Output:
My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...
```

```

string quote = @"Her name was ""Sara."";
//Output: Her name was "Sara."

```

## 원시 문자열 리터럴

C# 11부터 원시 문자열 리터럴을 사용하여 여러 줄인 문자열을 더 쉽게 만들거나 이스케이프 시퀀스가 필요한 문자를 사용할 수 있습니다. 원시 문자열 리터럴은 이스케이프 시퀀스를 사용할 필요가 없습니다. 공백 서식을 포함하여 문자열을 출력에 표시할 방법을 작성할 수 있습니다. 원시 문자열 리터럴:

- 세 개 이상의 큰따옴표("") 시퀀스로 시작하고 끝납니다. 세 개 이상의 반복된 따옴표 문자가 포함된 문자열 리터럴을 지원하기 위해 시퀀스를 시작하고 종료할 수 있습니다.
- 한 줄 원시 문자열 리터럴에는 같은 줄에 여는 따옴표 문자와 닫는 따옴표 문자가 필요합니다.
- 여러 줄 원시 문자열 리터럴에는 자체 줄에 여는 따옴표 문자와 닫는 따옴표 문자가 모두 필요합니다.
- 여러 줄 원시 문자열 리터럴에서는 닫는 따옴표 왼쪽의 공백이 모두 제거됩니다.

다음 예제에서는 이러한 규칙을 보여 줍니다.

C#

```

string singleLine = """Friends say "hello" as they pass by."";
string multiLine = """
    Hello World!" is typically the first program someone writes.
    """;
string embeddedXML = """
    <element attr = "content">
        <body style="normal">
            Here is the main text
        </body>
        <footer>
            Excerpts from "An amazing story"
        </footer>

```

```

</element >
""";
// The line "<element attr = "content">" starts in the first column.
// All whitespace left of that column is removed from the string.

string rawStringLiteralDelimiter = """
    Raw string literals are delimited
    by a string of at least three double quotes,
    like this: """
""";

```

다음 예제에서는 이러한 규칙에 따라 보고된 컴파일러 오류를 보여 줍니다.

C#

```

// CS8997: Unterminated raw string literal.
var multiLineStart = """This
    is the beginning of a string
""";

// CS9000: Raw string literal delimiter must be on its own line.
var multiLineEnd = """
    This is the beginning of a string """;

// CS8999: Line does not start with the same whitespace as the closing line
// of the raw string literal
var noOutdenting = """
    A line of text.
Trying to outdent the second line.
""";

```

여러 줄 원시 문자열 리터럴에는 자체 줄에 여는 따옴표 시퀀스와 닫는 따옴표 시퀀스가 필요하기 때문에 처음 두 예제는 유효하지 않습니다. 세 번째 예제는 텍스트가 닫는 따옴표 시퀀스에서 들여쓰기되어 있기 때문에 유효하지 않습니다.

따옴표 붙은 문자열 리터럴 또는 축자 문자열 리터럴을 사용할 때 이 [스케이프 시퀀스](#)가 필요한 문자를 포함하는 텍스트를 생성할 때 원시 문자열 리터럴을 고려해야 합니다. 원시 문자열 리터럴은 출력 텍스트와 더 유사하므로 사용자와 다른 사용자가 더 쉽게 읽을 수 있습니다. 예를 들어 형식이 지정된 JSON 문자열을 포함하는 다음 코드를 고려합니다.

C#

```

string jsonString = """
{
    "Date": "2019-08-01T00:00:00-07:00",
    "TemperatureCelsius": 25,
    "Summary": "Hot",
    "DatesAvailable": [
        "2019-08-01T00:00:00-07:00",
        "2019-08-02T00:00:00-07:00"
    ]
}

```

```

        ],
    "TemperatureRanges": {
        "Cold": {
            "High": 20,
            "Low": -10
        },
        "Hot": {
            "High": 60,
            "Low": 20
        }
    },
    "SummaryWords": [
        "Cool",
        "Windy",
        "Humid"
    ]
}
"""
;
```

이 새 기능을 사용하지 않는 JSON serialization 샘플의 해당 텍스트와 해당 텍스트를 비교합니다.

## 문자열 이스케이프 시퀀스

 테이블 확장

이스케이 프 시퀀스	문자 이름	유니코드 인코딩
\'	작은따옴표	0x0027
\"	큰따옴표	0x0022
\	백슬래시	0x005C
\0	Null	0x0000
\a	경고	0x0007
\b	백스페이스	0x0008
\f	폼 피드	0x000C
\n	줄 바꿈	0x000A
\r	캐리지 리턴	0x000D
\t	가로 탭	0x0009
\v	세로 탭	0x000B

이스케이프 시퀀스	문자 이름	유니코드 인코딩
\u	유니코드 이스케이프 시퀀스(UTF-16)	\uHHHH (범위: 0000-FFFF; 예제: \u00E7 "ç" =)
\U	유니코드 이스케이프 시퀀스(UTF-32)	\U00HHHHHH (range: 000000 - 10FFFF; example: \U0001F47D = "👀")
\x	길이가 변하는 경우를 제외하고 "\u"와 유사한 유니코드 이스케이프 시퀀스합니다.	\xH[H][H][H] (범위: 0-FFFF; 예: \x0E7 나 \x0E7 또는 \xE7 "ç" =)

### ⚠ 경고

\x 이스케이프 시퀀스를 사용하고 4자리 미만의 16진수를 지정하는 경우, 이스케이프 시퀀스 바로 뒤에 있는 문자가 유효한 16진수(예: 0-9, A-F 및 a-f)이면 이스케이프 시퀀스의 일부로 해석됩니다. 예를 들어 은 \xA1 코드 포인트 U+00A1인 "\\"를 생성합니다. 그러나 다음 문자가 "A" 또는 "a"인 경우 이스케이프 시퀀스는 대신로 \xA1A 해석되고 코드 포인트 U+0A1A인 ""을 생성합니다. 이러한 경우 4자리 16진수(예: \x00A1)를 모두 지정하면 잘못된 해석을 방지할 수 있습니다.

### ① 참고

컴파일 시 축자 문자열은 모두 동일한 이스케이프 시퀀스가 포함된 일반 문자열로 변환됩니다. 따라서 디버거 조사식 창에서 축자 문자열을 확인할 경우 소스 코드의 축자 버전이 아니라 컴파일러에 의해 추가된 이스케이프 문자가 나타납니다. 예를 들어 축자 문자열 @"C:\files.txt" 은 조사식 창에 "C:\files.txt"로 표시됩니다.

## 형식 문자열

형식 문자열은 콘텐츠가 런타임에 동적으로 결정되는 문자열입니다. 형식 문자열은 문자열 내의 중괄호 안에 '보간된 식'이나 자리 표시자를 포함하여 만들어집니다. 중괄호 ({...}) 안의 모든 내용은 런타임에 하나의 값으로 확인되고 형식화된 문자열로 출력됩니다. 형식 문자열을 만드는 두 가지 방법은 문자열 보간 및 복합 형식 지정입니다.

## 문자열 보간

C# 6.0 이상에서 사용 가능한 '[보간된 문자열](#)'은 \$ 특수 문자로 식별되고 중괄호 안에 보간된 식을 포함합니다. 문자열 보간을 처음으로 사용하는 경우 빠른 개요는 [문자열 보간 - C# 대화형 자습서](#)를 참조하세요.

코드의 가독성과 유지 관리를 개선하려면 문자열 보간을 사용합니다. 문자열 보간은 `String.Format` 메서드와 동일한 결과를 제공하지만 더 편리하고 인라인 명확성이 향상됩니다.

C#

```
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, published: 1761);
Console.WriteLine($"{jh.firstName} {jh.lastName} was an African American poet born in {jh.born}.");
Console.WriteLine($"He was first published in {jh.published} at the age of {jh.published - jh.born}.");
Console.WriteLine($"He'd be over {Math.Round((2018d - jh.born) / 100d) * 100d} years old today.");

// Output:
// Jupiter Hammon was an African American poet born in 1711.
// He was first published in 1761 at the age of 50.
// He'd be over 300 years old today.
```

C# 10부터 자리 표시자에 사용되는 모든 식이 상수 문자열인 경우 문자열 보간을 사용하여 상수 문자열을 초기화할 수 있습니다.

C# 11부터 원시 문자열 리터럴을 문자열 보간과 결합할 수 있습니다. 세 개 이상의 연속 큰따옴표로 서식 문자열을 시작하고 종료합니다. 출력 문자열에 또는 문자가 `{` 포함되어야 하는 경우 추가 `$` 문자를 사용하여 보간을 시작하고 종료하는 문자 수를 `}` 지정할 `{` 수 `}` 있습니다. 출력에 더 적 `{`거나 `}` 문자의 모든 시퀀스가 포함됩니다. 다음 예제에서는 해당 기능을 사용하여 원점에서 점의 거리를 표시하고 중괄호 안에 점을 배치하는 방법을 보여 줍니다.

C#

```
int X = 2;
int Y = 3;

var pointMessage = $$""The point {{X}}, {{Y}} is {{Math.Sqrt(X * X + Y * Y)}} from the origin.""";

Console.WriteLine(pointMessage);
// Output:
// The point {2, 3} is 3.605551275463989 from the origin.
```

## 복합 형식 지정

`String.Format`은 중괄호 안에 자리 표시자를 활용하여 형식 문자열을 만듭니다. 이 예제는 위에서 사용한 문자열 보간 방법과 유사한 출력을 생성합니다.

C#

```
var pw = (firstName: "Phillis", lastName: "Wheatley", born: 1753, published: 1773);
Console.WriteLine("{0} {1} was an African American poet born in {2}.",
    pw.firstName, pw.lastName, pw.born);
Console.WriteLine("She was first published in {0} at the age of {1}.",
    pw.published, pw.published - pw.born);
Console.WriteLine("She'd be over {0} years old today.", Math.Round((2018d - pw.born) / 100d) * 100d);

// Output:
// Phillis Wheatley was an African American poet born in 1753.
// She was first published in 1773 at the age of 20.
// She'd be over 300 years old today.
```

.NET 형식의 서식 지정에 대한 자세한 내용은 [.NET의 형식 서식 지정을](#) 참조하세요.

## 부분 문자열

부분 문자열은 문자열에 포함된 임의의 문자 시퀀스입니다. 원래 문자열 일부에서 새 문자열을 만들려면 [Substring](#) 메서드를 사용하세요. [IndexOf](#) 메서드를 사용하면 부분 문자열 항목을 하나 이상 검색할 수 있습니다. 지정된 부분 문자열의 모든 항목을 새 문자열로 바꾸려면 [Replace](#) 메서드를 사용하세요. 메서드 [Replace](#)와 [Substring](#) 마찬가지로는 실제로 새 문자열을 반환하고 원래 문자열을 수정하지 않습니다. 자세한 내용은 [문자열 검색 방법](#) 및 [문자열 내용 수정 방법](#)을 참조하세요.

C#

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2));
// Output: "C#"

System.Console.WriteLine(s3.Replace("C#", "Basic"));
// Output: "Visual Basic Express"

// Index values are zero-based
int index = s3.IndexOf("C");
// index = 7
```

## 개별 문자 액세스

다음 예제와 같이 인덱스 값이 있는 배열 표기법을 사용하여 개별 문자에 대한 읽기 전용 액세스 권한을 얻을 수 있습니다.

C#

```

string s5 = "Printing backwards";

for (int i = 0; i < s5.Length; i++)
{
    System.Console.Write(s5[s5.Length - i - 1]);
}
// Output: "sdrawkcab gnitnirP"

```

메서드가 문자열의 [String](#) 개별 문자를 수정해야 하는 기능을 제공하지 않는 경우 개체를 [StringBuilder](#) 사용하여 개별 문자 "현재 위치"를 수정한 다음 메서드를 사용하여 [StringBuilder](#) 결과를 저장할 새 문자열을 만들 수 있습니다. 다음 예제에서는 특정 방식으로 원래 문자열을 수정한 다음 나중에 사용할 수 있도록 결과를 저장해야 한다고 가정합니다.

C#

```

string question = "hOW DOES mICROSOFT wORD DEAL WITH THE cAPS LOCK KEY?";
System.Text.StringBuilder sb = new System.Text.StringBuilder(question);

for (int j = 0; j < sb.Length; j++)
{
    if (System.Char.IsLower(sb[j]) == true)
        sb[j] = System.Char.ToUpper(sb[j]);
    else if (System.Char.IsUpper(sb[j]) == true)
        sb[j] = System.Char.ToLower(sb[j]);
}
// Store the new string.
string corrected = sb.ToString();
System.Console.WriteLine(corrected);
// Output: How does Microsoft Word deal with the Caps Lock key?

```

## Null 문자열 및 빈 문자열

빈 문자열은 문자가 포함되지 않은 [System.String](#) 개체의 인스턴스입니다. 빈 문자열은 빈 텍스트 필드를 나타내는 다양한 프로그래밍 시나리오에서 자주 사용됩니다. 메서드는 유 효한 [System.String](#) 개체이므로 빈 문자열에서 호출할 수 있습니다. 빈 문자열은 다음과 같이 초기화됩니다.

C#

```

string s = String.Empty;

```

반면 null 문자열은 개체의 [System.String](#) 인스턴스를 참조하지 않으며 null 문자열에서 메서드를 호출하려고 하면 [NullReferenceException](#) 발생합니다. 그러나 다른 문자열과 연

결 및 비교 작업에서는 null 문자열을 사용할 수 있습니다. 다음 예제에서는 null 문자열에 대한 참조로 인해 예외가 throw되지 않는 몇 가지 사례를 보여 줍니다.

C#

```
string str = "hello";
string nullStr = null;
string emptyStr = String.Empty;

string tempStr = str + nullStr;
// Output of the following line: hello
Console.WriteLine(tempStr);

bool b = (emptyStr == nullStr);
// Output of the following line: False
Console.WriteLine(b);

// The following line creates a new empty string.
string newStr = emptyStr + nullStr;

// Null strings and empty strings behave differently. The following
// two lines display 0.
Console.WriteLine(emptyStr.Length);
Console.WriteLine(newStr.Length);
// The following line raises a NullReferenceException.
//Console.WriteLine(nullStr.Length);

// The null character can be displayed and counted, like other chars.
string s1 = "\x0" + "abc";
string s2 = "abc" + "\x0";
// Output of the following line: * abc*
Console.WriteLine("*" + s1 + "*");
// Output of the following line: *abc *
Console.WriteLine("*" + s2 + "*");
// Output of the following line: 4
Console.WriteLine(s2.Length);
```

## 빠른 문자열 만들기를 위해 `StringBuilder` 사용

.NET의 문자열 작업은 고도로 최적화되어 있으며 대부분의 경우 성능에 큰 영향을 주지 않습니다. 그러나 수백 번 또는 수천 번 실행하는 타이트 루프와 같은 일부 시나리오에서는 문자열 작업이 성능에 영향을 미칠 수 있습니다. 프로그램이 여러 문자열 조작을 수행하는 경우에는 `StringBuilder` 클래스에서 개선된 성능을 제공하는 문자열 버퍼를 만듭니다. `StringBuilder` 또한 문자열을 사용하면 기본 제공 문자열 데이터 형식에서 지원하지 않는 개별 문자를 다시 할당할 수 있습니다. 예를 들어 이 코드는 새 문자열을 만들지 않고 문자열의 콘텐츠를 변경합니다.

C#

```
System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal  
pet");  
sb[0] = 'C';  
System.Console.WriteLine(sb.ToString());  
//Outputs Cat: the ideal pet
```

이 예제에서 [StringBuilder](#) 개체는 숫자 형식 집합에서 문자열을 만드는 데 사용됩니다.

C#

```
var sb = new StringBuilder();  
  
// Create a string composed of numbers 0 - 9  
for (int i = 0; i < 10; i++)  
{  
    sb.Append(i.ToString());  
}  
Console.WriteLine(sb); // displays 0123456789  
  
// Copy one character of the string (not possible with a System.String)  
sb[0] = sb[9];  
  
Console.WriteLine(sb); // displays 9123456789
```

## 문자열, 확장 메서드 및 LINQ

[String](#) 형식이 [IEnumerable<T>](#)을 구현하므로 문자열에서 [Enumerable](#) 클래스에 정의된 확장 메서드를 사용할 수 있습니다. 시각적 혼란을 방지하기 위해 이러한 메서드는 형식에 대한 [String](#) IntelliSense에서 제외되지만 그럼에도 불구하고 사용할 수 있습니다. 문자열에 LINQ 쿼리 식을 사용할 수도 있습니다. 자세한 내용은 [LINQ 및 문자열](#)을 참조하세요.

## 관련 문서

- [문자열 내용을 수정하는 방법](#): [문자열](#)을 변환하고 문자열 내용을 수정하는 기술을 보여 줍니다.
- [문자열을 비교하는 방법](#): 서수 및 문화권별 문자열 비교를 수행하는 방법을 보여 줍니다.
- [여러 문자열을 연결하는 방법](#): [여러 문자열을 하나로 조인하는 다양한 방법](#)을 보여 줍니다.
- [String.Split을 사용하여 문자열을 구문 분석하는 방법](#): 메서드를 사용하여 [String.Split](#) 문자열을 구문 분석하는 방법을 보여 주는 코드 예제를 포함합니다.

- [문자열을 검색하는 방법](#): 문자열에서 특정 텍스트 또는 패턴 검색을 사용하는 방법을 설명합니다.
  - [문자열이 숫자 값을 나타내는지 여부를 확인하는 방법](#): 문자열을 안전하게 구문 분석하여 유효한 숫자 값이 있는지 확인하는 방법을 보여 줍니다.
  - [문자열 보간](#): 문자열 서식을 지정하는 편리한 구문을 제공하는 문자열 보간 기능을 설명합니다.
  - [기본 문자열 작업](#): 및 메서드를 사용하여 `System.String``System.Text.StringBuilder` 기본 문자열 작업을 수행하는 아티클에 대한 링크를 제공합니다.
  - [문자열 구문 분석](#): .NET 기본 형식의 문자열 표현을 해당 형식의 인스턴스로 변환하는 방법을 설명합니다.
  - [.NET에서 날짜 및 시간 문자열 구문 분석](#): "01/24/2008"과 같은 문자열을 개체로 `System.DateTime` 변환하는 방법을 보여 줍니다.
  - [문자열 비교](#): 문자열을 비교하는 방법에 대한 정보를 포함하고 C# 및 Visual Basic의 예제를 제공합니다.
  - [StringBuilder 클래스 사용](#): [클래스](#)를 사용하여 `StringBuilder` 동적 문자열 개체를 만들고 수정하는 방법을 설명합니다.
  - [LINQ 및 문자열](#): LINQ 쿼리를 사용하여 다양한 문자열 작업을 수행하는 방법에 대한 정보를 제공합니다.
- 

## 피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) ↗

# 문자열이 숫자 값을 나타내는지 확인하는 방법(C# 프로그래밍 가이드)

아티클 • 2024. 04. 11.

문자열이 지정된 숫자 형식의 유효한 표현인지 확인하려면 모든 기본 숫자 형식 및 `DateTime`, `IPAddress` 등의 형식에 의해서도 구현되는 정적 `TryParse` 메서드를 사용합니다. 다음 예제에서는 "108"이 유효한 `int`인지 확인하는 방법을 보여 줍니다.

C#

```
int i = 0;
string s = "108";
bool result = int.TryParse(s, out i); //i now = 108
```

문자열이 숫자가 아닌 문자를 포함하거나, 숫자 값이 지정한 특정 형식에 비해 너무 크거나 너무 작은 경우 `TryParse`는 `false`를 반환하고 `out` 매개 변수를 0으로 설정합니다. 그렇지 않으면 `true`를 반환하고 `out` 매개 변수를 문자열의 숫자 값으로 설정합니다.

## ① 참고

문자열이 숫자만 포함해도 사용하는 `TryParse` 메서드의 형식에 유효하지 않을 수도 있습니다. 예를 들어 "256"은 `byte`에 유효한 값이 아니지만 `int`에는 유효합니다. "98.6"은 `int`에 유효한 값이 아니지만 유효한 `decimal`입니다.

## 예시

다음 예제에서는 `long`, `byte` 및 `decimal` 값의 문자열 표현과 함께 `TryParse`를 사용하는 방법을 보여 줍니다.

C#

```
string numString = "1287543"; //"1287543.0" will return false for a long
long number1 = 0;
bool canConvert = long.TryParse(numString, out number1);
if (canConvert == true)
    Console.WriteLine("number1 now = {0}", number1);
else
    Console.WriteLine("numString is not a valid long");

byte number2 = 0;
numString = "255"; // A value of 256 will return false
```

```
canConvert = byte.TryParse(numString, out number2);
if (canConvert == true)
    Console.WriteLine("number2 now = {0}", number2);
else
    Console.WriteLine("numString is not a valid byte");

decimal number3 = 0;
numString = "27.3"; // "27" is also a valid decimal
canConvert = decimal.TryParse(numString, out number3);
if (canConvert == true)
    Console.WriteLine("number3 now = {0}", number3);
else
    Console.WriteLine("number3 is not a valid decimal");
```

## 강력한 프로그래밍

또한 기본 숫자 형식은 문자열이 유효한 숫자가 아닌 경우 예외를 throw하는 `Parse` 정적 메서드를 구현합니다. 일반적으로 숫자가 유효하지 않은 경우 단순히 `false`를 반환하는 `TryParse` 가 더 효율적입니다.

## .NET 보안

항상 `TryParse` 또는 `Parse` 메서드를 사용하여 텍스트 상자, 콤보 상자 등의 컨트롤에서 들어오는 사용자 입력의 유효성을 검사합니다.

## 참고 항목

- 바이트 배열을 int로 변환하는 방법
- 문자열을 숫자로 변환하는 방법
- 16진수 문자열과 숫자 형식 간에 변환하는 방법
- 숫자 문자열 구문 분석
- 형식 서식 지정

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

☞ 설명서 문제 열기

☞ 제품 사용자 의견 제공

# 인덱서(C# 프로그래밍 가이드)

아티클 • 2024. 04. 11.

인덱서에서는 클래스나 구조체의 인스턴스를 배열처럼 인덱싱할 수 있습니다. 인덱싱 값은 형식이나 인스턴스 멤버를 명시적으로 지정하지 않고도 설정하거나 검색할 수 있습니다. 인덱서는 해당 접근자가 매개 변수를 사용한다는 점을 제외하면 [속성](#)과 유사합니다.

다음 예제에서는 간단한 `get` 및 `set` 접근자 메서드를 사용해서 값을 할당하거나 검색하는 제네릭 클래스를 정의합니다. `Program` 클래스는 이 클래스의 인스턴스를 만들어 문자열을 저장합니다.

```
C#  
  
using System;  
  
class SampleCollection<T>  
{  
    // Declare an array to store the data elements.  
    private T[] arr = new T[100];  
  
    // Define the indexer to allow client code to use [] notation.  
    public T this[int i]  
    {  
        get { return arr[i]; }  
        set { arr[i] = value; }  
    }  
}  
  
class Program  
{  
    static void Main()  
    {  
        var stringCollection = new SampleCollection<string>();  
        stringCollection[0] = "Hello, World";  
        Console.WriteLine(stringCollection[0]);  
    }  
}  
// The example displays the following output:  
//      Hello, World.
```

## ① 참고

추가 예제는 [관련 섹션](#)을 참조하세요.

## 식 본문 정의

인덱서의 get 또는 set 접근자는 값을 반환하거나 설정하는 단일 문으로 구성되는 것이 일반적입니다. 식 본문이 있는 멤버는 이 시나리오를 지원하기 위해 간단한 구문을 제공합니다. C# 6부터 읽기 전용 인덱서는 다음 예제와 같이 식 본문이 있는 멤버로 구현할 수 있습니다.

```
C#  
  
using System;  
  
class SampleCollection<T>  
{  
    // Declare an array to store the data elements.  
    private T[] arr = new T[100];  
    int nextIndex = 0;  
  
    // Define the indexer to allow client code to use [] notation.  
    public T this[int i] => arr[i];  
  
    public void Add(T value)  
    {  
        if (nextIndex >= arr.Length)  
            throw new IndexOutOfRangeException($"The collection can hold only  
{arr.Length} elements.");  
        arr[nextIndex++] = value;  
    }  
}  
  
class Program  
{  
    static void Main()  
    {  
        var stringCollection = new SampleCollection<string>();  
        stringCollection.Add("Hello, World");  
        System.Console.WriteLine(stringCollection[0]);  
    }  
}  
// The example displays the following output:  
//     Hello, World.
```

=>에서 식 본문을 도입하며 `get` 키워드는 사용되지 않습니다.

C# 7.0부터 get 및 set 접근자 모두를 식 본문 멤버로 구현할 수 있습니다. 이 경우 `get` 및 `set` 키워드를 둘 다 사용해야 합니다. 예를 들어:

```
C#  
  
using System;  
  
class SampleCollection<T>  
{  
    // Declare an array to store the data elements.
```

```

private T[] arr = new T[100];

// Define the indexer to allow client code to use [] notation.
public T this[int i]
{
    get => arr[i];
    set => arr[i] = value;
}
}

class Program
{
    static void Main()
    {
        var stringCollection = new SampleCollection<string>();
        stringCollection[0] = "Hello, World.";
        Console.WriteLine(stringCollection[0]);
    }
}
// The example displays the following output:
//      Hello, World.

```

## 인덱서 개요

- 인덱서를 사용하면 배열과 유사한 방식으로 개체를 인덱싱할 수 있습니다.
- `get` 접근자는 값을 반환합니다. `set` 접근자는 값을 할당합니다.
- `this` 키워드는 인덱서를 정의하는 데 사용됩니다.
- `value` 키워드는 `set` 접근자가 할당하는 값을 정의하는 데 사용됩니다.
- 인덱서는 정수 값으로 인덱싱될 필요가 없으며, 특정 조회 메커니즘을 정의하는 방법을 결정해야 합니다.
- 인덱서는 오버로드될 수 있습니다.
- 예를 들어 인덱서는 2차원 배열에 액세스하는 경우 둘 이상의 정식 매개 변수를 사용할 수 있습니다.

## 관련 단원

- [인덱서 사용](#)
- [인터페이스의 인덱서](#)
- [속성 및 인덱서 비교](#)

- 접근자 접근성 제한

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 인덱서](#)를 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [속성](#)

---

## 피드백

이 페이지가 도움이 되었나요?

Yes

No

[제품 사용자 의견 제공](#)

# 인덱서 사용(C# 프로그래밍 가이드)

아티클 • 2023. 04. 07.

인덱서는 클라이언트 애플리케이션이 배열처럼 액세스할 수 있는 `class`, `struct`, `interface` 를 만들 수 있게 해주는 편리한 구문입니다. 컴파일러는 `Item` 속성(또는 `IndexerNameAttribute`가 있는 경우 달리 명명된 속성) 및 적절한 접근자 메서드를 생성합니다. 인덱서는 내부 컬렉션 또는 배열을 캡슐화하는 데 주로 사용되는 형식에서 자주 구현됩니다. 예를 들어 24시간 동안 10회 기록된 화씨온도를 나타내는 `TempRecord` 클래스가 있다고 가정합니다. 이 클래스에는 온도 값을 저장할 `float[]` 형식의 `temps` 배열이 포함되어 있습니다. 이 클래스에서 인덱서를 구현하면 클라이언트가 `TempRecord` 인스턴스의 온도에 `float temp = tempRecord.temps[4]` 대신 `float temp = tempRecord[4]`로 액세스할 수 있습니다. 인덱서 표기법은 클라이언트 애플리케이션에 대한 구문을 간소화할 뿐 아니라 클래스와 해당 용도를 다른 개발자가 이해하기 쉽게 만듭니다.

클래스 또는 구조체에서 인덱서를 선언하려면 다음 예제에 표시된 대로 `this` 키워드를 사용합니다.

C#

```
// Indexer declaration
public int this[int index]
{
    // get and set accessors
}
```

## ① 중요

인덱서를 선언하면 개체에 `Item`이라는 속성이 자동으로 생성됩니다. `Item` 속성은 **멤버 액세스 식** 인스턴스에서 직접 액세스할 수 없습니다. 또한 인덱서를 사용하여 `Item` 속성을 개체에 추가하는 경우 **CS0102 컴파일러 오류**가 표시됩니다. 이 오류를 방지하려면 아래에 설명된 대로 `IndexerNameAttribute`를 사용하여 인덱서 이름을 바꿉니다.

## 설명

인덱서의 형식과 해당 매개 변수의 형식은 최소한 인덱서 자체만큼 액세스 가능해야 합니다. 접근성 수준에 대한 자세한 내용은 [액세스 한정자](#)를 참조하세요.

인터페이스와 함께 인덱서를 사용하는 방법에 대한 자세한 내용은 [인터페이스 인덱서](#)를 참조하세요.

인덱서의 시그니처는 정식 매개 변수의 형식 및 개수로 구성됩니다. 정식 매개 변수의 이름이나 인덱서 형식은 포함되지 않습니다. 동일한 클래스에서 둘 이상의 인덱서를 선언하는 경우 다른 시그니처가 있어야 합니다.

인덱서는 변수로 분류되지 않습니다. 따라서 인덱서 값이 참조가 아니면 참조([ref](#) 또는 [out](#) 매개 변수)로 인덱서 값을 전달할 수 없습니다(즉, 참조로 반환됨).

다른 언어에서 사용할 수 있는 이름을 인덱서에 제공하려면 다음 예제에 표시된 대로 [System.Runtime.CompilerServices.IndexerNameAttribute](#)를 사용합니다.

C#

```
// Indexer declaration
[System.Runtime.CompilerServices.IndexerName("TheItem")]
public int this[int index]
{
    // get and set accessors
}
```

이 인덱서는 인덱서 이름 특성으로 재정의되므로 `TheItem`이라는 이름을 갖게 됩니다. 기본적으로 인덱서 이름은 `Item`입니다.

## 예제 1

다음 예제에서는 전용 배열 필드, `temps`, 인덱서를 선언하는 방법을 보여 줍니다. 인덱서를 사용하면 `tempRecord[i]` 인스턴스에 직접 액세스할 수 있습니다. 인덱서를 사용하지 않으려면 배열을 [public](#) 멤버로 선언하고 해당 멤버인 `tempRecord.temps[i]`에 직접 액세스합니다.

C#

```
public class TempRecord
{
    // Array of temperature values
    float[] temps = new float[10]
    {
        56.2F, 56.7F, 56.5F, 56.9F, 58.8F,
        61.3F, 65.9F, 62.1F, 59.2F, 57.5F
    };

    // To enable client code to validate input
    // when accessing your indexer.
    public int Length => temps.Length;

    // Indexer declaration.
    // If index is out of range, the temps array will throw the exception.
    public float this[int index]
```

```

    {
        get => temps[index];
        set => temps[index] = value;
    }
}

```

인덱서의 액세스가 `Console.WriteLine` 문 등에서 평가될 때 `get` 접근자가 호출됩니다. 따라서 `get` 접근자가 없으면 컴파일 시간 오류가 발생합니다.

C#

```

class Program
{
    static void Main()
    {
        var tempRecord = new TempRecord();

        // Use the indexer's set accessor
        tempRecord[3] = 58.3F;
        tempRecord[5] = 60.1F;

        // Use the indexer's get accessor
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine($"Element #{i} = {tempRecord[i]}");
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
    /* Output:
        Element #0 = 56.2
        Element #1 = 56.7
        Element #2 = 56.5
        Element #3 = 58.3
        Element #4 = 58.8
        Element #5 = 60.1
        Element #6 = 65.9
        Element #7 = 62.1
        Element #8 = 59.2
        Element #9 = 57.5
    */
}

```

## 다른 값을 사용하여 인덱싱

C#은 인덱스 매개 변수 형식을 정수로 제한되지 않습니다. 예를 들어 인덱서와 함께 문자열을 사용하면 유용할 수 있습니다. 이러한 인덱서는 컬렉션에서 문자열을 검색하고 적절

한 값을 반환하여 구현할 수 있습니다. 접근자를 오버로드할 수 있기 때문에 문자열 및 정수 버전을 함께 사용할 수 있습니다.

## 예제 2

다음 예제에서는 요일을 저장하는 클래스를 선언합니다. `get` 접근자는 문자열인 요일 이름을 사용하고 해당 정수를 반환합니다. 예를 들어 “일요일”이면 0을 반환하고, “월요일”이면 1을 반환합니다.

C#

```
// Using a string as an indexer value
class DayCollection
{
    string[] days = { "Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat" };

    // Indexer with only a get accessor with the expression-bodied
    definition:
    public int this[string day] => FindDayIndex(day);

    private int FindDayIndex(string day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }

        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be in the form
\"Sun\", \"Mon\", etc");
    }
}
```

## 예제 2 사용

C#

```
class Program
{
    static void Main(string[] args)
    {
        var week = new DayCollection();
        Console.WriteLine(week[ "Fri"]);
```

```

try
{
    Console.WriteLine(week["Made-up day"]);
}
catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine($"Not supported input: {e.Message}");
}
}
// Output:
// 5
// Not supported input: Day Made-up day is not supported.
// Day input must be in the form "Sun", "Mon", etc (Parameter 'day')
}

```

## 예제 3

다음 예제에서는 `System.DayOfWeek` 열거형을 사용하여 요일을 저장하는 클래스를 선언합니다. `get` 접근자는 요일 값인 `DayOfWeek`를 사용하고 해당 정수를 반환합니다. 예를 들어 `DayOfWeek.Sunday`는 0을 반환하고 `DayOfWeek.Monday`는 1을 반환합니다.

C#

```

using Day = System.DayOfWeek;

class DayOfWeekCollection
{
    Day[] days =
    {
        Day.Sunday, Day.Monday, Day.Tuesday, Day.Wednesday,
        Day.Thursday, Day.Friday, Day.Saturday
    };

    // Indexer with only a get accessor with the expression-bodied
    definition:
    public int this[Day day] => FindDayIndex(day);

    private int FindDayIndex(Day day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }
        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be a defined
            System.DayOfWeek value.");
    }
}

```

```
    }  
}
```

## 예제 3 사용

C#

```
class Program  
{  
    static void Main()  
    {  
        var week = new DayOfWeekCollection();  
        Console.WriteLine(week[DayOfWeek.Friday]);  
  
        try  
        {  
            Console.WriteLine(week[(DayOfWeek)43]);  
        }  
        catch (ArgumentOutOfRangeException e)  
        {  
            Console.WriteLine($"Not supported input: {e.Message}");  
        }  
    }  
    // Output:  
    // 5  
    // Not supported input: Day 43 is not supported.  
    // Day input must be a defined System.DayOfWeek value. (Parameter 'day')  
}
```

## 강력한 프로그래밍

인덱서의 보안과 안정성을 향상할 수 있는 다음 두 가지 방법이 있습니다.

- 클라이언트 코드에서 잘못된 인덱스 값을 전달할 가능성을 처리하는 일부 유형의 오류 처리 전략을 통합해야 합니다. 이 항목의 앞부분에 있는 첫 번째 예제에서 TempRecord 클래스는 클라이언트 코드에서 입력을 인덱서에 전달하기 전에 확인 할 수 있게 해주는 Length 속성을 제공합니다. 인덱서 자체 안에 오류 처리 코드를 넣을 수도 있습니다. 사용자를 위해 인덱서 접근자 내에서 throw하는 모든 예외를 문서화해야 합니다.
- get 및 set 접근자의 접근성을 적절하게 제한적으로 설정합니다. 특히 set 접근자의 경우 이 작업이 중요합니다. 자세한 내용은 [접근자 액세스 가능성 제한](#)을 참조하세요.

## 참조

- C# 프로그래밍 가이드
- 인덱서
- 속성

# 인터페이스의 인덱서(C# 프로그래밍 가이드)

아티클 • 2024. 03. 12.

`interface`에 인덱서를 선언할 수 있습니다. 인터페이스 인덱서 접근자와 `class` 인덱서 접근자 간에는 다음과 같은 차이점이 있습니다.

- 인터페이스 접근자는 한정자를 사용하지 않습니다.
- 일반적으로 인터페이스 접근자에는 본문이 없습니다.

접근자의 목적은 인덱서가 읽기/쓰기인지, 읽기 전용인지, 쓰기 전용인지를 나타내는 것입니다. 인터페이스에 정의된 인덱서에 대해 구현을 정의할 수 있긴 하나, 이는 드문 경우입니다. 인덱서는 일반적으로 API를 정의하여 데이터 필드에 액세스하는데, 데이터 필드는 인터페이스에서 정의할 수 없기 때문입니다.

다음은 인터페이스 인덱서 접근자의 예입니다.

```
C#  
  
public interface ISomeInterface  
{  
    //...  
  
    // Indexer declaration:  
    string this[int index]  
    {  
        get;  
        set;  
    }  
}
```

인덱서의 시그니처는 동일한 인터페이스에 선언된 다른 모든 인덱서의 시그니처와 달라야 합니다.

## 예시

다음 예제에서는 인터페이스 인덱서를 구현하는 방법을 보여 줍니다.

```
C#  
  
// Indexer on an interface:  
public interface IIndexInterface  
{  
    // Indexer declaration:  
}
```

```
int this[int index]
{
    get;
    set;
}
}

// Implementing the interface.
class IndexerClass : IIndexInterface
{
    private int[] arr = new int[100];
    public int this[int index]    // indexer declaration
    {
        // The arr object will throw IndexOutOfRangeException exception.
        get => arr[index];
        set => arr[index] = value;
    }
}
```

C#

```
IndexerClass test = new IndexerClass();
System.Random rand = System.Random.Shared;
// Call the indexer to initialize its elements.
for (int i = 0; i < 10; i++)
{
    test[i] = rand.Next();
}
for (int i = 0; i < 10; i++)
{
    System.Console.WriteLine($"Element #{i} = {test[i]}");
}

/* Sample output:
   Element #0 = 360877544
   Element #1 = 327058047
   Element #2 = 1913480832
   Element #3 = 1519039937
   Element #4 = 601472233
   Element #5 = 323352310
   Element #6 = 1422639981
   Element #7 = 1797892494
   Element #8 = 875761049
   Element #9 = 393083859
*/
```

앞의 예제에서는 인터페이스 멤버의 정규화된 이름을 사용하여 명시적 인터페이스 멤버 구현을 사용할 수 있습니다. 예를 들면 다음과 같습니다.

C#

```
string IIndexInterface.this[int index]
{
}
```

그러나 정규화된 이름은 클래스가 동일한 인덱서 시그니처로 둘 이상의 인터페이스를 구현할 때 모호성을 피하기 위해서만 필요합니다. 예를 들어 Employee 클래스가 두 인터페이스 ICitizen 및 IEmployee를 구현하고 두 인터페이스의 인덱서 시그니처가 같으면 명시적 인터페이스 멤버 구현이 필요합니다. 즉, 다음과 같은 인덱서 선언이 있다고 가정합니다.

C#

```
string IEmployee.this[int index]
{
}
```

이 선언은 IEmployee 인터페이스의 인덱서를 구현합니다. 또한 다음과 같은 선언이 있다고 가정합니다.

C#

```
string ICitizen.this[int index]
{
}
```

이 선언은 ICitizen 인터페이스의 인덱서를 구현합니다.

## 참고 항목

- [인덱서](#)
- [속성](#)
- [인터페이스](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

☞ 설명서 문제 열기

☞ 제품 사용자 의견 제공

# 속성 및 인덱서 비교(C# 프로그래밍 가이드)

아티클 • 2024. 03. 12.

인덱서는 속성과 비슷합니다. 다음 표에 나와 있는 차이점을 제외하면 속성 접근자에 대해 정의된 모든 규칙이 인덱서 접근자에도 적용됩니다.

[+] 테이블 확장

속성	인덱서
공용 데이터 멤버인 것처럼 메서드를 호출할 수 있게 합니다.	개체 자체에 배열 표기법을 사용하여 개체의 내부 컬렉션 요소에 액세스할 수 있게 합니다.
단순한 이름을 통해 액세스합니다.	인덱스를 통해 액세스합니다.
정적 또는 인스턴스 멤버일 수 있습니다.	인스턴스 멤버여야 합니다.
속성의 <code>get</code> 접근자에는 매개 변수가 없습니다.	인덱서의 <code>get</code> 접근자에는 인덱서와 동일한 형식 매개 변수 목록이 있습니다.
속성의 <code>set</code> 접근자에는 암시적 <code>value</code> 매개 변수가 포함되어 있습니다.	인덱서의 <code>set</code> 접근자에는 인덱서와 동일한 형식 매개 변수 목록이 있으며 <code>value</code> 매개 변수도 있습니다.
자동으로 구현된 속성을 사용하여 약식 구문을 지원합니다.	가져오기만 수행(Get only) 인덱서를 위한 식 본문 멤버를 지원합니다.

## 참고 항목

- [인덱서](#)
- [속성](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# 이벤트(C# 프로그래밍 가이드)

아티클 • 2024. 04. 11.

클래스나 개체에서는 특정 상황이 발생할 때 이벤트를 통해 다른 클래스나 개체에 이를 알려줄 수 있습니다. 이벤트를 보내거나 발생시키는 클래스를 게시자라고 하며 이벤트를 받거나 처리하는 클래스를 구독자라고 합니다.

일반적인 C# Windows Forms 또는 웹 애플리케이션, 단추 및 목록 상자 같은 컨트롤에 의해 발생하는 이벤트를 구독합니다. 컨트롤이 게시하는 이벤트를 찾아보고 처리할 이벤트를 선택하려면 Visual C# IDE(통합 개발 환경)를 사용할 수 있습니다. IDE는 빈 이벤트 처리기 메서드 및 이벤트를 구독하기 위한 코드를 자동으로 추가하는 편리한 방법을 제공합니다. 자세한 내용은 [이벤트를 구독 및 구독 취소하는 방법](#)을 참조하세요.

## 이벤트 개요

이벤트에는 다음과 같은 속성이 있습니다.

- 게시자는 이벤트 발생 시기를 결정합니다. 구독자는 이벤트에 대한 응답으로 수행 할 작업을 결정합니다.
- 한 이벤트에는 여러 구독자가 있을 수 있습니다. 구독자는 여러 게시자의 여러 이벤트를 처리할 수 있습니다.
- 구독자가 없는 이벤트는 발생하지 않습니다.
- 이벤트는 일반적으로 그래픽 사용자 인터페이스에서 단추 클릭이나 메뉴 선택 같은 사용자 작업을 표시하는 데 사용됩니다.
- 이벤트에 여러 구독자가 있는 경우 이벤트 처리기는 이벤트가 발생할 때 동기적으로 호출됩니다. 이벤트를 비동기적으로 호출하려면 [동기 메서드를 비동기 방식으로 호출](#)을 참조하세요.
- .NET 클래스 라이브러리에서 이벤트는 `EventHandler` 대리자 및 `EventArgs` 기본 클래스를 기반으로 합니다.

## 관련 단원

자세한 내용은 다음을 참조하세요.

- [이벤트를 구독 및 구독 취소하는 방법](#)
- [.NET 지침을 따르는 이벤트를 게시하는 방법](#)

- 파생 클래스에서 기본 클래스 이벤트를 발생하는 방법
- 인터페이스 이벤트를 구현하는 방법
- 사용자 지정 이벤트 접근자를 구현하는 방법

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 이벤트](#)를 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

## 중요 설명서 장

[대리자, Events, and Lambda Expressions](#) 에 C# 3.0 Cookbook, Third Edition: 250 개 이상의 솔루션에 대한 C# 3.0 프로그래머

[C# 3.0 학습: C# 3.0의 기본 사항](#)의 대리자 및 이벤트

## 참고 항목

- [EventHandler](#)
- [C# 프로그래밍 가이드](#)
- [대리자](#)
- [Windows Forms에서 이벤트 처리기 만들기](#)

---

## 피드백

이 페이지가 도움이 되었나요?

 Yes

 No

[제품 사용자 의견 제공](#) ↗

# 이벤트 구독 및 구독 취소 방법(C# 프로그래밍 가이드)

아티클 • 2023. 03. 10.

해당 이벤트가 발생할 때 호출되는 사용자 지정 코드를 작성하려는 경우 다른 클래스에 의해 게시되는 이벤트를 구독합니다. 예를 들어 사용자가 단추를 클릭할 때 애플리케이션에서 유용한 작업을 수행하도록 하려면 단추의 `click` 이벤트를 구독할 수 있습니다.

## Visual Studio IDE를 사용하여 이벤트를 구독하려면

- 속성 창이 표시되지 않는 경우 디자인 보기에서 이벤트 처리기를 만들려는 양식 또는 컨트롤을 마우스 오른쪽 단추로 클릭하고 속성을 선택합니다.
- 속성 창의 맨 위에서 이벤트 아이콘을 클릭합니다.
- 만들려는 이벤트(예: `Load` 이벤트)를 두 번 클릭합니다.

Visual C#에서 빈 이벤트 처리기 메서드를 만들고 코드에 추가합니다. 또는 코드 보기에서 수동으로 코드를 추가할 수 있습니다. 예를 들어 다음 코드 줄은 `Form` 클래스에서 `Load` 이벤트가 발생할 때 호출되는 이벤트 처리기 메서드를 선언합니다.

C#

```
private void Form1_Load(object sender, System.EventArgs e)
{
    // Add your form load event handling code here.
}
```

이벤트를 구독하는 데 필요한 코드 줄도 프로젝트의 `Form1.Designer.cs` 파일에 있는 `InitializeComponent` 메서드에 자동으로 생성됩니다. 해당 코드는 다음과 같습니다.

C#

```
this.Load += new System.EventHandler(this.Form1_Load);
```

## 프로그래밍 방식으로 이벤트를 구독하려면

- 이벤트의 대리자 시그니처와 일치하는 시그니처를 가진 이벤트 처리기 메서드를 정의합니다. 예를 들어 이벤트가 `EventHandler` 대리자 형식을 기반으로 하는 경우 다음 코드는 메서드 스텝을 나타냅니다.

C#

```
void HandleCustomEvent(object sender, CustomEventArgs a)
{
    // Do something useful here.
}
```

2. 더하기 대입 연산자(+=)를 사용하여 이벤트에 이벤트 처리기를 연결합니다. 다음 예제에서는 `publisher` 개체에 `RaiseCustomEvent`라는 이벤트가 있다고 가정합니다. 구독자 클래스가 해당 이벤트를 구독하려면 게시자 클래스에 대한 참조가 필요합니다.

C#

```
publisher.RaiseCustomEvent += HandleCustomEvent;
```

람다 식을 사용하여 이벤트 처리기를 지정할 수도 있습니다.

C#

```
public Form1()
{
    InitializeComponent();
    this.Click += (s,e) =>
    {
        MessageBox.Show(((MouseEventArgs)e).Location.ToString());
    };
}
```

## 무명 기능을 사용하여 이벤트를 구독

나중에 이벤트에서 구독을 취소할 필요가 없는 경우 더하기 대입 연산자(+=)를 사용하여 무명 메서드를 이벤트 처리기로서 연결할 수 있습니다. 다음 예제에서는 `publisher` 개체에 `RaiseCustomEvent`라는 이벤트가 있고 일종의 특수 이벤트 정보를 전달하도록 `CustomEventArgs` 클래스도 정의되었다고 가정합니다. 구독자 클래스가 해당 이벤트를 구독하려면 `publisher`에 대한 참조가 필요합니다.

C#

```
publisher.RaiseCustomEvent += (object o, CustomEventArgs e) =>
{
    string s = o.ToString() + " " + e.ToString();
    Console.WriteLine(s);
};
```

무명 함수를 사용하여 이벤트를 구독한 경우 이벤트 구독을 쉽게 취소할 수 없습니다. 이 시나리오에서 구독을 취소하려면 이벤트를 구독하고, 대리자 변수에 무명 기능을 저장한 다음 이벤트에 대리자를 추가하는 코드로 돌아가야 합니다. 코드의 뒷부분에서 이벤트 구독을 취소해야 하는 경우 무명 함수를 사용하여 이벤트를 구독하지 않는 것이 좋습니다. 무명 기능에 대한 자세한 내용은 [람다 식](#)을 참조하세요.

## 구독 해제

이벤트가 발생할 때 이벤트 처리기가 호출되지 않도록 하려면 이벤트 구독을 취소합니다. 리소스 누수를 방지하려면 구독자 개체를 삭제하기 전에 이벤트 구독을 취소해야 합니다. 이벤트 구독을 취소할 때까지 게시 개체에서 이벤트의 기반이 되는 멀티캐스트 대리자에 구독자의 이벤트 처리기를 캡슐화하는 대리자에 대한 참조가 있습니다. 게시 개체에 해당 참조가 있으면 하면 가비지 수집 시 구독자 개체가 삭제되지 않습니다.

### 이벤트 구독을 취소하려면

- 빼기 대입 연산자(`--=`)를 사용하여 이벤트 구독을 취소합니다.

```
C#
```

```
publisher.RaiseCustomEvent -= HandleCustomEvent;
```

모든 구독자가 이벤트 구독을 취소하면 게시자 클래스의 이벤트 인스턴스가 `null`로 설정됩니다.

## 참조

- [이벤트](#)
- [event](#)
- [.NET 지침을 따르는 이벤트를 게시하는 방법](#)
- [- 및 `-=` 연산자](#)
- [+ 및 `+ =` 연산자](#)

# .NET 지침을 따르는 이벤트를 게시하는 방법(C# 프로그래밍 가이드)

아티클 • 2023. 05. 09.

다음 절차에서는 표준 .NET 패턴을 따르는 이벤트를 클래스와 구조체에 추가하는 방법을 보여 줍니다. .NET 클래스 라이브러리의 모든 이벤트는 다음과 같이 정의된 `EventHandler` 대리자를 기반으로 합니다.

C#

```
public delegate void EventHandler(object sender, EventArgs e);
```

## ① 참고

.NET Framework 2.0에서는 이 대리자의 제네릭 버전인 `EventHandler<TEventArgs>` 가 도입되었습니다. 다음 예제에서는 두 버전을 사용하는 방법을 모두 보여 줍니다.

정의하는 클래스의 이벤트는 값을 반환하는 대리자를 포함하여 유효한 모든 대리자 형식을 기반으로 할 수 있지만, 다음 예제와 같이 `EventHandler`를 사용하여 .NET 패턴에 따라 이벤트를 만드는 것이 좋습니다.

이름 `EventHandler`는 이벤트를 실제로 처리하지는 않으므로 약간의 혼동을 일으킬 수 있습니다. `EventHandler` 및 제네릭 `EventHandler<TEventArgs>`는 대리자 형식입니다. 시그니처가 대리자 정의와 일치하는 메서드 또는 람다 식은 이벤트 처리기이며, 이벤트가 발생할 때 호출됩니다.

## EventHandler 패턴에 따라 이벤트 게시

1. 이벤트와 함께 사용자 지정 데이터를 보낼 필요가 없는 경우 이 단계를 건너뛰고 3a 단계로 이동합니다. 게시자 및 구독자 클래스 들 다에 표시되는 범위에서 사용자 지정 데이터에 대한 클래스를 선언합니다. 그런 다음 사용자 지정 이벤트 데이터를 저장하는 데 필요한 멤버를 추가합니다. 이 예제에서는 간단한 문자열이 반환됩니다.

C#

```
public class CustomEventArgs : EventArgs
{
    public CustomEventArgs(string message)
    {
        Message = message;
    }

    public string Message { get; }
```

```
    public string Message { get; set; }  
}
```

2. `EventHandler<TEventArgs>`의 제네릭 버전을 사용하는 경우 이 단계를 건너뜁니다. 게시 클래스에서 대리자를 선언합니다. `EventHandler`로 끝나는 이름을 지정합니다. 두 번째 매개 변수는 사용자 지정 `EventArgs` 형식을 지정합니다.

C#

```
public delegate void CustomEventHandler(object sender, CustomEventArgs  
args);
```

3. 다음 단계 중 하나를 사용하여 게시 클래스에서 이벤트를 선언합니다.

- a. 사용자 지정 `EventArgs` 클래스가 없는 경우 이벤트 유형은 제네릭이 아닌 `EventHandler` 대리자가 됩니다. C# 프로젝트를 만들 때 포함된 `System` 네임스페이스에서 이미 선언되었기 때문에 대리자를 선언할 필요는 없습니다. 게시자 클래스에 다음 코드를 추가합니다.

C#

```
public event EventHandler RaiseCustomEvent;
```

- b. 제네릭이 아닌 버전의 `EventHandler`를 사용 중이고 `EventArgs`에서 파생된 사용자 지정 클래스가 있는 경우 게시 클래스 내에서 이벤트를 선언하고 2단계의 대리자를 형식으로 사용합니다.

C#

```
public event CustomEventHandler RaiseCustomEvent;
```

- c. 제네릭 버전을 사용하는 경우에는 사용자 지정 대리자가 필요하지 않습니다. 대신, 게시 클래스에서 이벤트 유형을 `EventHandler<CustomEventArgs>`로 지정하고 고유한 클래스 이름을 꺽쇠 괄호로 묶어 대체합니다.

C#

```
public event EventHandler<CustomEventArgs> RaiseCustomEvent;
```

## 예제

다음 예제에서는 사용자 지정 EventArgs 클래스와 EventHandler<TEventArgs>를 이벤트 유형으로 사용하여 이전 단계를 보여 줍니다.

C#

```
using System;

namespace DotNetEvents
{
    // Define a class to hold custom event info
    public class CustomEventArgs : EventArgs
    {
        public CustomEventArgs(string message)
        {
            Message = message;
        }

        public string Message { get; set; }
    }

    // Class that publishes an event
    class Publisher
    {
        // Declare the event using EventHandler<T>
        public event EventHandler<CustomEventArgs> RaiseCustomEvent;

        public void DoSomething()
        {
            // Write some code that does something useful here
            // then raise the event. You can also raise an event
            // before you execute a block of code.
            OnRaiseCustomEvent(new CustomEventArgs("Event triggered"));
        }

        // Wrap event invocations inside a protected virtual method
        // to allow derived classes to override the event invocation
        behavior
        protected virtual void OnRaiseCustomEvent(CustomEventArgs e)
        {
            // Make a temporary copy of the event to avoid possibility of
            // a race condition if the last subscriber unsubscribes
            // immediately after the null check and before the event is
            raised.
            EventHandler<CustomEventArgs> raiseEvent = RaiseCustomEvent;

            // Event will be null if there are no subscribers
            if (raiseEvent != null)
            {
                // Format the string to send inside the CustomEventArgs
                parameter
                e.Message += $" at {DateTime.Now}";

                // Call to raise the event.
                raiseEvent(this, e);
            }
        }
    }
}
```

```

        }
    }

//Class that subscribes to an event
class Subscriber
{
    private readonly string _id;

    public Subscriber(string id, Publisher pub)
    {
        _id = id;

        // Subscribe to the event
        pub.RaiseCustomEvent += HandleCustomEvent;
    }

    // Define what actions to take when the event is raised.
    void HandleCustomEvent(object sender, CustomEventArgs e)
    {
        Console.WriteLine($"{_id} received this message: {e.Message}");
    }
}

class Program
{
    static void Main()
    {
        var pub = new Publisher();
        var sub1 = new Subscriber("sub1", pub);
        var sub2 = new Subscriber("sub2", pub);

        // Call the method that raises the event.
        pub.DoSomething();

        // Keep the console window open
        Console.WriteLine("Press any key to continue...");
        Console.ReadLine();
    }
}
}

```

## 참조

- Delegate
- C# 프로그래밍 가이드
- 이벤트
- 대리자

# 파생 클래스에서 기본 클래스 이벤트를 발생하는 방법(C# 프로그래밍 가이드)

아티클 • 2024. 03. 13.

다음 간단한 예제에서는 파생 클래스에서도 발생할 수 있도록 기본 클래스에서 이벤트를 선언하는 표준 방법을 보여 줍니다. 이 패턴은 .NET 클래스 라이브러리의 Windows Forms 클래스에서 광범위하게 사용됩니다.

다른 클래스의 기본 클래스로 사용할 수 있는 클래스를 만드는 경우 이벤트가 해당 이벤트를 선언한 클래스 내에서만 호출할 수 있는 특수 유형의 대리자라는 사실을 고려해야 합니다. 파생 클래스는 기본 클래스 내에서 선언된 이벤트를 직접 호출할 수 없습니다. 기본 클래스에서만 발생할 수 있는 이벤트를 원하는 경우도 있지만 대부분의 경우 파생 클래스가 기본 클래스 이벤트를 호출할 수 있도록 해야 합니다. 이렇게 하려면 이벤트를 래핑하는 기본 클래스에서 보호된 호출 메서드를 만듭니다. 이 호출 메서드를 호출하거나 재정의하면 파생 클래스에서 간접적으로 이벤트를 호출할 수 있습니다.

## ① 참고

기본 클래스에서 가상 이벤트를 선언하지 말고 파생 클래스에서 재정의합니다. C# 컴파일러는 이러한 이벤트를 올바르게 처리하지 않으며, 파생 이벤트의 구독자가 실제로 기본 클래스 이벤트를 구독할지 여부를 예측할 수 없습니다.

## 예시

C#

```
namespace BaseClassEvents
{
    // Special EventArgs class to hold info about Shapes.
    public class ShapeEventArgs : EventArgs
    {
        public ShapeEventArgs(double area)
        {
            NewArea = area;
        }

        public double NewArea { get; }
    }

    // Base class event publisher
    public abstract class Shape
    {
```

```
protected double _area;

public double Area
{
    get => _area;
    set => _area = value;
}

// The event. Note that by using the generic EventHandler<T> event
type
// we do not need to declare a separate delegate type.
public event EventHandler<ShapeEventArgs> ShapeChanged;

public abstract void Draw();

//The event-invoking method that derived classes can override.
protected virtual void OnShapeChanged(ShapeEventArgs e)
{
    // Safely raise the event for all subscribers
    ShapeChanged?.Invoke(this, e);
}

public class Circle : Shape
{
    private double _radius;

    public Circle(double radius)
    {
        _radius = radius;
        _area = 3.14 * _radius * _radius;
    }

    public void Update(double d)
    {
        _radius = d;
        _area = 3.14 * _radius * _radius;
        OnShapeChanged(new ShapeEventArgs(_area));
    }

    protected override void OnShapeChanged(ShapeEventArgs e)
    {
        // Do any circle-specific processing here.

        // Call the base class event invocation method.
        base.OnShapeChanged(e);
    }

    public override void Draw()
    {
        Console.WriteLine("Drawing a circle");
    }
}

public class Rectangle : Shape
```

```
{  
    private double _length;  
    private double _width;  
  
    public Rectangle(double length, double width)  
    {  
        _length = length;  
        _width = width;  
        _area = _length * _width;  
    }  
  
    public void Update(double length, double width)  
    {  
        _length = length;  
        _width = width;  
        _area = _length * _width;  
        OnShapeChanged(new ShapeEventArgs(_area));  
    }  
  
    protected override void OnShapeChanged(ShapeEventArgs e)  
    {  
        // Do any rectangle-specific processing here.  
  
        // Call the base class event invocation method.  
        base.OnShapeChanged(e);  
    }  
  
    public override void Draw()  
    {  
        Console.WriteLine("Drawing a rectangle");  
    }  
}  
  
// Represents the surface on which the shapes are drawn  
// Subscribes to shape events so that it knows  
// when to redraw a shape.  
public class ShapeContainer  
{  
    private readonly List<Shape> _list;  
  
    public ShapeContainer()  
    {  
        _list = new List<Shape>();  
    }  
  
    public void AddShape(Shape shape)  
    {  
        _list.Add(shape);  
  
        // Subscribe to the base class event.  
        shape.ShapeChanged += HandleShapeChanged;  
    }  
  
    // ...Other methods to draw, resize, etc.
```

```

        private void HandleShapeChanged(object sender, ShapeEventArgs e)
    {
        if (sender is Shape shape)
        {
            // Diagnostic message for demonstration purposes.
            Console.WriteLine($"Received event. Shape area is now
{e.NewArea}");

            // Redraw the shape here.
            shape.Draw();
        }
    }

    class Test
    {
        static void Main()
        {
            //Create the event publishers and subscriber
            var circle = new Circle(54);
            var rectangle = new Rectangle(12, 9);
            var container = new ShapeContainer();

            // Add the shapes to the container.
            container.AddShape(circle);
            container.AddShape(rectangle);

            // Cause some events to be raised.
            circle.Update(57);
            rectangle.Update(7, 7);

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }
    }
}

/* Output:
   Received event. Shape area is now 10201.86
   Drawing a circle
   Received event. Shape area is now 49
   Drawing a rectangle
*/

```

## 참고 항목

- 이벤트
- 대리자
- 액세스 한정자
- Windows Forms에서 이벤트 처리기 만들기

## GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 설명서 문제 열기

 제품 사용자 의견 제공

# 인터페이스 이벤트를 구현하는 방법(C# 프로그래밍 가이드)

아티클 • 2023. 03. 10.

인터페이스는 [이벤트](#)를 선언할 수 있습니다. 다음 예제에서는 클래스에서 인터페이스 이벤트를 구현하는 방법을 보여 줍니다. 기본적인 규칙은 다른 모든 인터페이스 메서드나 속성을 구현할 때의 규칙과 같습니다.

## 클래스에서 인터페이스 이벤트를 구현하려면

클래스에서 이벤트를 선언한 다음 적절한 영역에서 이벤트를 호출합니다.

C#

```
namespace ImplementInterfaceEvents
{
    public interface IDrawingObject
    {
        event EventHandler ShapeChanged;
    }
    public class MyEventArgs : EventArgs
    {
        // class members
    }
    public class Shape : IDrawingObject
    {
        public event EventHandler ShapeChanged;
        void ChangeShape()
        {
            // Do something here before the event...
            OnShapeChanged(new MyEventArgs(/*arguments*/));
            // or do something here after the event.
        }
        protected virtual void OnShapeChanged(MyEventArgs e)
        {
            ShapeChanged?.Invoke(this, e);
        }
    }
}
```

## 예제

다음 예제에서는 클래스가 둘 이상의 인터페이스에서 상속을 받으면 각 인터페이스에는 이름이 같은 이벤트가 있는 드문 상황을 처리하는 방법을 보여 줍니다. 이러한 상황에서는 이벤트 하나 이상에 대해 명시적 인터페이스 구현을 제공해야 합니다. 이벤트에 대한 명시적 인터페이스 구현을 쓸 때는 `add` 및 `remove` 이벤트 접근자도 써야 합니다. 일반적으로는 컴파일러가 이러한 접근자를 제공하지만 이 예제의 경우에는 컴파일러가 접근자를 제공할 수 없습니다.

고유한 접근자를 제공하면 클래스에서 두 이벤트를 같은 이벤트로 표시할지 아니면 다른 이벤트로 표시할지를 지정할 수 있습니다. 예를 들어 인터페이스 사양에 따라 이벤트가 서로 다른 시간에 발생해야 하는 경우에는 각 이벤트를 클래스의 개별 구현과 연결할 수 있습니다. 다음 예제에서는 구독자가 `IShape` 또는 `IDrawingObject`에 셰이프 참조를 캐스팅하여 수신할 `OnDraw` 이벤트를 결정합니다.

C#

```
namespace WrapTwoInterfaceEvents
{
    using System;

    public interface IDrawingObject
    {
        // Raise this event before drawing
        // the object.
        event EventHandler OnDraw;
    }

    public interface IShape
    {
        // Raise this event after drawing
        // the shape.
        event EventHandler OnDraw;
    }

    // Base class event publisher inherits two
    // interfaces, each with an OnDraw event
    public class Shape : IDrawingObject, IShape
    {
        // Create an event for each interface event
        event EventHandler PreDrawEvent;
        event EventHandler PostDrawEvent;

        object objectLock = new Object();

        // Explicit interface implementation required.
        // Associate IDrawingObject's event with
        // PreDrawEvent
        #region IDrawingObjectOnDraw
        event EventHandler IDrawingObject.OnDraw
        {
            add
            {
```

```

        lock (objectLock)
    {
        PreDrawEvent += value;
    }
}
remove
{
    lock (objectLock)
    {
        PreDrawEvent -= value;
    }
}
#endregion
// Explicit interface implementation required.
// Associate IShape's event with
// PostDrawEvent
event EventHandler IShape.OnDraw
{
    add
    {
        lock (objectLock)
        {
            PostDrawEvent += value;
        }
    }
    remove
    {
        lock (objectLock)
        {
            PostDrawEvent -= value;
        }
    }
}

// For the sake of simplicity this one method
// implements both interfaces.
public void Draw()
{
    // Raise IDrawingObject's event before the object is drawn.
    PreDrawEvent?.Invoke(this, EventArgs.Empty);

    Console.WriteLine("Drawing a shape.");

    // Raise IShape's event after the object is drawn.
    PostDrawEvent?.Invoke(this, EventArgs.Empty);
}
}

public class Subscriber1
{
    // References the shape object as an IDrawingObject
    public Subscriber1(Shape shape)
    {
        IDrawingObject d = (IDrawingObject)shape;
        d.OnDraw += d_OnDraw;
    }
}

```

```

        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub1 receives the IDrawingObject event.");
        }
    }
    // References the shape object as an IShape
    public class Subscriber2
    {
        public Subscriber2(Shape shape)
        {
            IShape d = (IShape)shape;
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub2 receives the IShape event.");
        }
    }

    public class Program
    {
        static void Main(string[] args)
        {
            Shape shape = new Shape();
            Subscriber1 sub = new Subscriber1(shape);
            Subscriber2 sub2 = new Subscriber2(shape);
            shape.Draw();

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
}

/* Output:
   Sub1 receives the IDrawingObject event.
   Drawing a shape.
   Sub2 receives the IShape event.
*/

```

## 참고 항목

- C# 프로그래밍 가이드
- 이벤트
- 대리자
- 명시적 인터페이스 구현
- 파생 클래스에서 기본 클래스 이벤트를 발생하는 방법

# 사용자 지정 이벤트 접근자를 구현하는 방법(C# 프로그래밍 가이드)

아티클 • 2024. 03. 03.

이벤트는 선언된 클래스 내에서만 호출할 수 있는 특수한 종류의 멀티캐스트 대리자입니다. 클라이언트 코드는 이벤트가 발생할 때 호출되어야 하는 메서드에 대한 참조를 제공하여 이벤트를 구독합니다. 이러한 메서드는 이벤트 접근자의 이름이 `add` 및 `remove`로 지정된다는 점을 제외하고 속성 접근자와 유사한 이벤트 접근자를 통해 대리자의 호출 목록에 추가됩니다. 대부분의 경우 사용자 지정 이벤트 접근자를 제공할 필요가 없습니다. 코드에서 사용자 지정 이벤트 접근자를 제공하지 않으면 컴파일러가 자동으로 추가합니다. 그러나 경우에 따라 사용자 지정 동작을 제공해야 할 수도 있습니다. [인터페이스 이벤트를 구현하는 방법](#) 항목에는 이러한 사례 중 하나가 나와 있습니다.

## 예시

다음 예제에서는 사용자 지정 `add` 및 `remove` 이벤트 접근자를 구현하는 방법을 보여 줍니다. 접근자 내의 모든 코드를 대체할 수 있지만 새 이벤트 처리기 메서드를 추가하거나 제거하기 전에 이벤트를 잠그는 것이 좋습니다.

```
C#  
  
event EventHandler IDrawingObject.OnDraw  
{  
    add  
    {  
        lock (objectLock)  
        {  
            PreDrawEvent += value;  
        }  
    }  
    remove  
    {  
        lock (objectLock)  
        {  
            PreDrawEvent -= value;  
        }  
    }  
}
```

## 참고 항목

- [이벤트](#)

- event

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

💡 설명서 문제 열기

↗️ 제품 사용자 의견 제공

# 제네릭 형식 매개 변수(C# 프로그래밍 가이드)

아티클 • 2024. 03. 09.

제네릭 형식 또는 메서드 정의에서 형식 매개 변수는 클라이언트가 제네릭 형식의 인스턴스를 만들 때 지정하는 특정 형식에 대한 자리 표시자입니다. [제네릭 소개](#)에 나열된 `GenericList<T>`와 같은 제네릭 클래스는 실제로 형식이 아니고 형식에 대한 청사진과 같으므로 있는 그대로 사용할 수는 없습니다. `GenericList<T>`를 사용하려면, 클라이언트 코드에서 꺼쇠 괄호 안에 형식 인수를 지정하여 생성된 형식을 선언하고 인스턴스화해야 합니다. 이 특정 클래스의 형식 인수는 컴파일러에서 인식하는 모든 형식이 될 수 있습니다. 만들 수 있는 생성된 형식 인스턴스의 수에는 제한이 없고, 각 인스턴스에서는 다음과 같이 서로 다른 형식 인수를 사용할 수 있습니다.

C#

```
GenericList<float> list1 = new GenericList<float>();
GenericList<ExampleClass> list2 = new GenericList<ExampleClass>();
GenericList<ExampleStruct> list3 = new GenericList<ExampleStruct>();
```

`GenericList<T>`의 각 인스턴스에서 클래스에 있는 모든 `T`는 런타임에 형식 인수로 대체됩니다. 이러한 대체를 통해 단일 클래스 정의를 사용하여 세 개의 형식이 안전한 효율적인 개체를 만들었습니다. CLR에서 이러한 대체를 수행하는 방식에 대한 자세한 내용은 [런타임의 제네릭](#)을 참조하세요.

[명명 규칙](#) 문서에서 제네릭 형식 매개 변수의 명명 규칙을 알아볼 수 있습니다.

## 참고 항목

- [System.Collections.Generic](#)
- [C# 프로그래밍 가이드](#)
- [제네릭](#)
- [C++ 템플릿과 C# 제네릭의 차이점](#)

GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

수도 있습니다. 자세한 내용은  
참여자 가이드를 참조하세요.

☞ 설명서 문제 열기

☞ 제품 사용자 의견 제공

# 형식 매개 변수에 대한 제약 조건(C# 프로그래밍 가이드)

아티클 • 2024. 03. 12.

제약 조건은 형식 인수에서 갖추고 있어야 하는 기능을 컴파일러에 알립니다. 제약 조건이 없으면 형식 인수가 어떤 형식이든 될 수 있습니다. 컴파일러는 모든 .NET 형식의 궁극적인 기본 클래스인 `System.Object`의 멤버만 가정할 수 있습니다. 자세한 내용은 [제약 조건을 사용하는 이유](#)를 참조하세요. 클라이언트 코드가 제약 조건을 충족하지 않는 형식을 사용하는 경우 컴파일러는 오류를 발생시킵니다. 제약 조건은 `where` 상황별 키워드를 사용하여 지정됩니다. 다음 표에는 다양한 형식의 제약 조건이 나열되어 있습니다.

## [+] 테이블 확장

제약 조건	설명
<code>where T : struct</code>	형식 인수는 형식을 포함하는 <code>record</code> <code>struct</code> <code>nullable</code> 이 아닌 <code>값 형식</code> 이어야 합니다. <code>Null</code> 허용 <code>값 형식</code> 에 대한 자세한 내용은 <a href="#">Null 허용 값 형식</a> 을 참조하세요. 모든 <code>값 형식</code> 에는 선언되거나 암시적으로 <code>struct</code> 액세스 가능한 매개 변수가 없는 생성자가 있으므로 제약 조건은 제약 조건을 의미 <code>new()</code> 하며 제약 조건과 <code>new()</code> 결합할 수 없습니다. <code>struct</code> 제약 조건을 <code>unmanaged</code> 제약 조건과 결합할 수 없습니다.
<code>where T : class</code>	형식 인수는 참조 형식이어야 합니다. 이 제약 조건은 모든 클래스, 인터페이스, 대리자 또는 배열 형식에도 적용됩니다. <code>null</code> 허용 컨텍스트에서 <code>T</code> 는 <code>null</code> 을 허용하지 않는 참조 형식이어야 합니다.
<code>where T : class?</code>	형식 인수는 <code>null</code> 을 허용하거나 <code>null</code> 을 허용하지 않는 참조 형식이어야 합니다. 이 제약 조건은 레코드를 포함하여 모든 클래스, 인터페이스, 대리자 또는 배열 형식에도 적용됩니다.
<code>where T : notnull</code>	형식 인수는 <code>nullable</code> 이 아닌 형식이어야 합니다. 인수는 <code>null</code> 을 허용하지 않는 참조 형식이거나 <code>null</code> 을 허용하지 않는 <code>값 형식</code> 일 수 있습니다.
<code>where T : unmanaged</code>	형식 인수는 <code>nullable</code> 이 아닌 <a href="#">비관리형 형식</a> 이어야 합니다. <code>unmanaged</code> 제약 조건은 <code>struct</code> 제약 조건을 나타내며 <code>struct</code> 또는 <code>new()</code> 제약 조건과 결합할 수 없습니다.
<code>where T : new()</code>	형식 인수에 매개 변수가 없는 <code>public</code> 생성자가 있어야 합니다. 다른 제약 조건과 함께 사용할 경우 <code>new()</code> 제약 조건을 마지막에 지정해야 합니다. <code>new()</code> 제약 조건은 <code>struct</code> 또는 <code>unmanaged</code> 제약 조건과 결합할 수 없습니다.
<code>where T : &lt; 기본 클래스 이름 &gt;</code>	형식 인수가 지정된 기본 클래스이거나 지정된 기본 클래스에서 파생되어야 합니다. <code>null</code> 허용 컨텍스트에서 <code>T</code> 는 지정된 기본 클래스에서 파생된 <code>null</code> 을 허용하지 않는 참조 형식이어야 합니다.
<code>where T : &lt; 기본 클래스 &gt;</code>	형식 인수가 지정된 기본 클래스이거나 지정된 기본 클래스에서 파생되어야 합니다. <code>nullable</code> 컨텍스트 <code>T</code> 에서 지정된 기본 클래스에서 파생된 <code>nullable</code> 또는

제약 조건	설명
<code>이름&gt;?</code>	<code>nullable</code> 이 아닌 형식일 수 있습니다.
<code>where T : &lt;인터페이스 이름&gt;</code>	형식 인수가 지정된 인터페이스이거나 지정된 인터페이스를 구현해야 합니다. 여러 인터페이스 제약 조건을 지정할 수 있습니다. 제약 인터페이스가 제네릭일 수도 있습니다. <code>null</code> 허용 컨텍스트에서 <code>T</code> 는 지정된 인터페이스를 구현하는 <code>null</code> 을 허용하지 않는 형식이어야 합니다.
<code>where T : &lt;인터페이스 이름&gt;?</code>	형식 인수가 지정된 인터페이스이거나 지정된 인터페이스를 구현해야 합니다. 여러 인터페이스 제약 조건을 지정할 수 있습니다. 제약 인터페이스가 제네릭일 수도 있습니다. <code>nullable</code> 컨텍스트 <code>T</code> 에서 <code>nullable</code> 참조 형식, <code>nullable</code> 이 아닌 참조 형식 또는 값 형식일 수 있습니다. <code>T</code> 은 <code>nullable</code> 값 형식일 수 없습니다.
<code>where T : U</code>	<code>T</code> 에 대해 제공되는 형식 인수는 <code>U</code> 에 대해 제공되는 인수이거나 이 인수에서 파생되어야 합니다. <code>nullable</code> 컨텍스트에서 <code>nullable</code> 이 아닌 참조 형식인 <code>T</code> 경우 <code>U</code> <code>nullable</code> 이 아닌 참조 형식이어야 합니다. <code>nullable</code> 참조 형식 <code>T</code> 인 경우 <code>U</code> <code>null</code> 을 허용하거나 <code>null</code> 을 허용하지 않을 수 있습니다.
<code>where T : default</code>	이 제약 조건은 메서드를 재정의하거나 명시적 인터페이스 구현을 제공할 때 비제한 형식 매개 변수를 지정해야 할 경우 모호성을 해결합니다. <code>default</code> 제약 조건은 <code>class</code> 또는 <code>struct</code> 제약 조건이 없는 기본 메서드를 의미합니다. 자세한 내용은 <a href="#">default 제약 조건</a> 사양 제안을 참조하세요.

일부 제약 조건은 상호 배타적이고 일부 제약 조건은 지정된 순서로 되어 있어야 합니다.

- `,, notnull class?` 및 `unmanaged` 제약 조건 중 `struct class` 하나만 적용할 수 있습니다. 이러한 제약 조건을 제공하는 경우 해당 형식 매개 변수에 대해 지정된 첫 번째 제약 조건이어야 합니다.
- 기본 클래스 제약 조건(`where T : Base` 또는 `where T : Base?`)을 제약 조건, `class` 또는 `class? notnull unmanaged` 제약 조건 `struct`과 결합할 수 없습니다.
- 두 가지 형식으로 최대 하나의 기본 클래스 제약 조건을 적용할 수 있습니다. `nullable` 기본 형식을 지원하려면 `.`를 사용합니다 `Base?.`.
- 인터페이스의 `nullable`이 아닌 형식과 `nullable` 형식의 이름을 모두 제약 조건으로 지정할 수는 없습니다.
- `new()` 제약 조건은 `struct` 또는 `unmanaged` 제약 조건과 결합할 수 없습니다. 제약 조건을 `new()` 지정하는 경우 해당 형식 매개 변수에 대한 마지막 제약 조건이어야 합니다.
- 제약 조건은 재정의 `default` 또는 명시적 인터페이스 구현에만 적용할 수 있습니다. 또는 제약 조건과 `struct class` 결합할 수 없습니다.

## 제약 조건을 사용하는 이유

제약 조건은 형식 매개 변수의 기능 및 기대치를 지정합니다. 해당 제약 조건을 선언하면 제약 형식의 작업 및 메서드 호출을 사용할 수 있습니다. 제네릭 클래스 또는 메서드가 단

순 할당 이외의 제네릭 멤버에 대한 작업을 사용하는 경우 형식 매개 변수에 제약 조건을 적용합니다. 여기에는 지원되지 않는 메서드 호출이 포함됩니다 [System.Object](#). 예를 들어 기본 클래스 제약 조건은 이 형식의 개체 또는 이 형식에서 파생된 개체만 해당 형식 인수를 대체할 수 있음을 컴파일러에 알려줍니다. 컴파일러에 이 보장이 있으면 해당 형식의 메서드가 제네릭 클래스에서 호출되도록 허용할 수 있습니다. 다음 코드 예제에서는 기본 클래스 제약 조건을 적용하여 [GenericList<T>](#) 클래스(제네릭 소개에 있음)에 추가할 수 있는 기능을 보여 줍니다.

C#

```
public class Employee
{
    public Employee(string name, int id) => (Name, ID) = (name, id);
    public string Name { get; set; }
    public int ID { get; set; }
}

public class GenericList<T> where T : Employee
{
    private class Node
    {
        public Node(T t) => (Next, Data) = (null, t);

        public Node? Next { get; set; }
        public T Data { get; set; }
    }

    private Node? head;

    public void AddHead(T t)
    {
        Node n = new Node(t) { Next = head };
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node? current = head;

        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    public T? FindFirstOccurrence(string s)
    {
        Node? current = head;
        T? t = null;

        while (current != null)
```

```

    {
        //The constraint enables access to the Name property.
        if (current.Data.Name == s)
        {
            t = current.Data;
            break;
        }
        else
        {
            current = current.Next;
        }
    }
    return t;
}
}

```

이 제약 조건을 통해 제네릭 클래스에서 `Employee.Name` 속성을 사용할 수 있습니다. 제약 조건은 `T` 형식의 모든 항목을 `Employee` 개체 또는 `Employee`에서 상속하는 개체 중 하나로 보장하도록 지정합니다.

동일한 형식 매개 변수에 여러 개의 제약 조건을 적용할 수 있으며, 제약 조건 자체가 다음과 같이 제네릭 형식일 수 있습니다.

C#

```

class EmployeeList<T> where T : Employee,
System.Collections.Generic.IList<T>, IDisposable, new()
{
    // ...
}

```

제약 조건을 `where T : class` 적용할 때 이러한 연산자는 값 같음이 아니라 참조 ID만 테스트하므로 형식 매개 변수에서 연 `!=` 산자와 연산자를 사용하지 마십시오 `==`. 이러한 연산자가 인수로 사용되는 형식에서 오버로드되는 경우에도 이 동작이 발생합니다. 다음 코드는 이 내용을 보여 줍니다. `String` 클래스가 `==` 연산자를 오버로드하지만 출력이 `false`입니다.

C#

```

public static void OpEqualsTest<T>(T s, T t) where T : class
{
    System.Console.WriteLine(s == t);
}

private static void TestStringEquality()
{
    string s1 = "target";
    System.Text.StringBuilder sb = new System.Text.StringBuilder("target");
}

```

```
        string s2 = sb.ToString();
        OpEqualsTest<string>(s1, s2);
    }
```

컴파일러에서 컴파일 시간에 `T`가 참조 형식이고 모든 참조 형식에 유효한 기본 연산자를 사용해야 한다는 것만 인식합니다. 값 같음을 테스트해야 하는 경우 또는 `where T : IComparable<T>` 제약 조건을 적용 `where T : IEquatable<T>` 하고 제네릭 클래스를 생성하는 데 사용되는 모든 클래스에서 인터페이스를 구현합니다.

## 여러 매개 변수 제한

다음 예제와 같이 여러 매개 변수에 제약 조건을 적용하고, 단일 매개 변수에 여러 제약 조건을 적용할 수 있습니다.

C#

```
class Base { }
class Test<T, U>
    where U : struct
    where T : Base, new()
{ }
```

## 바인딩되지 않은 형식 매개 변수

공용 클래스 `SampleClass<T>{}`의 `T`와 같이 제약 조건이 없는 형식 매개 변수를 바인딩되지 않은 형식 매개 변수라고 합니다. 바인딩되지 않은 형식 매개 변수에는 다음 규칙이 있습니다.

- `!=` 구체적인 형식 인수가 이러한 연산자를 지원한다는 보장은 없으므로 및 `==` 연산자를 사용할 수 없습니다.
- `System.Object`로/에서 변환하거나 임의의 인터페이스 형식으로 명시적으로 변환할 수 있습니다.
- `null`과 비교할 수 있습니다. 바인딩되지 않은 매개 변수를 비교 `null`하는 경우 형식 인수가 값 형식인 경우 비교는 항상 `false`를 반환합니다.

## 제약 조건으로 형식 매개 변수 사용

다음 예제와 같이 고유한 형식 매개 변수가 있는 멤버 함수가 해당 매개 변수를 포함 형식의 형식 매개 변수로 제약해야 하는 경우 제네릭 형식 매개 변수를 제약 조건으로 사용하면 유용합니다.

C#

```
public class List<T>
{
    public void Add<U>(List<U> items) where U : T {/*...*/}
}
```

앞의 예제에서 `T`는 `Add` 메서드 컨텍스트에서는 형식 제약 조건이고, `List` 클래스 컨텍스트에서는 바인딩되지 않은 형식 매개 변수입니다.

제네릭 클래스 정의에서 형식 매개 변수를 제약 조건으로 사용할 수도 있습니다. 형식 매개 변수는 다른 형식 매개 변수와 함께 꺪쇠괄호 안에 선언해야 합니다.

C#

```
//Type parameter V is used as a type constraint.
public class SampleClass<T, U, V> where T : V { }
```

컴파일러에서 형식 매개 변수가 `System.Object`에서 파생된다는 점을 제외하고는 형식 매개 변수에 대해 아무 것도 가정할 수 없기 때문에, 제네릭 클래스에서 형식 매개 변수를 제약 조건으로 사용하는 경우는 제한됩니다. 두 형식 매개 변수 사이의 상속 관계를 적용하려는 시나리오에서 제네릭 클래스에 형식 매개 변수를 제약 조건으로 사용합니다.

## notnull 제약 조건

`notnull` 제약 조건을 사용하여 형식 인수가 `null`을 허용하지 않는 값 형식 또는 `null`을 허용하지 않는 참조 형식이어야 함을 지정할 수 있습니다. 대부분 다른 제약 조건과 달리 형식 인수가 `notnull` 제약 조건을 위반하면 컴파일러는 오류 대신 경고를 생성합니다.

`notnull` 제약 조건은 `null` 허용 컨텍스트에서 사용되는 경우에만 영향을 미칩니다. `null` 허용 인식 불가능한 컨텍스트에서 `notnull` 제약 조건을 추가하면 컴파일러는 제약 조건 위반에 대한 경고 또는 오류를 생성하지 않습니다.

## class 제약 조건

`null` 허용 컨텍스트의 `class` 제약 조건은 형식 인수가 `null`을 허용하지 않는 참조 형식이어야 함을 지정합니다. `null` 허용 컨텍스트에서 형식 인수가 `null` 허용 참조 형식이면 컴파일러는 경고를 생성합니다.

## default 제약 조건

nullable 참조 형식 추가로 제네릭 형식 또는 메서드에서 `T?` 사용이 복잡해집니다. `T?`는 `struct` 또는 `class` 제약 조건과 함께 사용할 수 있지만 둘 중 하나가 있어야 합니다. `class` 제약 조건을 사용한 경우 `T?`는 `T`의 nullable 참조 형식을 나타냅니다. 제약 조건이 적용되지 않을 때 `T?`를 사용할 수 있습니다. 이 경우 값 형식 및 참조 형식에 대해 `T?`는 `T?`로 해석됩니다. 그러나 `T`가 `Nullable<T>`의 인스턴스인 경우 `T?`는 `T`와 동일합니다. 즉, `T??`가 되지 않습니다.

이제 `T?`를 `class` 또는 `struct` 제약 조건 없이 사용할 수 있으므로 재정의 또는 명시적 인터페이스 구현에서 모호성이 발생할 수 있습니다. 두 경우 모두에서 재정의는 제약 조건을 포함하지 않지만 기본 클래스에서 상속합니다. 기본 클래스가 `class` 또는 `struct` 제약 조건을 적용하지 않는 경우 파생 클래스는 둘 중 어떤 제약 조건도 없이 기본 메서드에 재정의가 적용되도록 지정해야 합니다. 파생 메서드는 제약 조건을 `default` 적용합니다. `default` 제약 조건은 `class` 또는 `struct` 제약 조건을 ‘모두’ 명확하게 지정하지 않습니다.

## 관리되지 않는 제약 조건

`unmanaged` 제약 조건을 사용하여 형식 매개 변수가 `null`을 허용하지 않는 [관리되지 않는 형식](#)이어야 함을 지정할 수 있습니다. `unmanaged` 제약 조건을 사용하면 다음 예제와 같이 메모리 블록으로 조작할 수 있는 형식을 사용하도록 재사용 가능한 루틴을 작성할 수 있습니다.

C#

```
unsafe public static byte[] ToByteArray<T>(this T argument) where T : unmanaged
{
    var size = sizeof(T);
    var result = new Byte[size];
    Byte* p = (byte*)&argument;
    for (var i = 0; i < size; i++)
        result[i] = *p++;
    return result;
}
```

앞의 메서드는 기본 제공 형식으로 알려지지 않은 형식에서 `sizeof` 연산자를 사용하므로 `unsafe` 컨텍스트에서 컴파일해야 합니다. `unmanaged` 제약 조건이 없으면 `sizeof` 연산자를 사용할 수 없습니다.

`unmanaged` 제약 조건은 `struct` 제약 조건을 나타내며 함께 사용할 수 없습니다. `struct` 제약 조건은 `new()` 제약 조건을 나타내며 `unmanaged` 제약 조건은 `new()` 제약 조건과 결합할 수 없습니다.

# 대리자 제약 조건

기본 클래스 제약 조건으로 `System.Delegate` 또는 `System.MulticastDelegate`를 사용할 수 있습니다. CLR에서는 항상 이 제약 조건을 허용했지만, C# 언어에서는 이 제약 조건을 허용하지 않았습니다. `System.Delegate` 제약 조건을 사용하면 형식이 안전한 방식으로 대리자에서 작동하는 코드를 작성할 수 있습니다. 다음 코드는 두 대리자가 동일한 형식인 경우 이를 결합하는 확장 메서드를 정의합니다.

C#

```
public static TDelegate? TypeSafeCombine<TDelegate>(this TDelegate source,  
TDelegate target)  
    where TDelegate : System.Delegate  
    => Delegate.Combine(source, target) as TDelegate;
```

위의 메서드를 사용하여 동일한 형식의 대리자를 결합할 수 있습니다.

C#

```
Action first = () => Console.WriteLine("this");  
Action second = () => Console.WriteLine("that");  
  
var combined = first.TypeSafeCombine(second);  
combined!();  
  
Func<bool> test = () => true;  
// Combine signature ensures combined delegates must  
// have the same type.  
//var badCombined = first.TypeSafeCombine(test);
```

마지막 줄의 주석 처리를 제거하면 컴파일되지 않습니다. `first` 및 `test`는 모두 대리자 형식이지만 서로 다른 대리자 형식입니다.

# 열거형 제약 조건

`System.Enum` 형식을 기본 클래스 제약 조건으로 지정할 수도 있습니다. CLR에서는 항상 이 제약 조건을 허용했지만, C# 언어에서는 이 제약 조건을 허용하지 않았습니다.

`System.Enum`을 사용하는 제네릭은 `System.Enum`의 정적 메서드를 사용하여 결과를 캐시하기 위해 형식이 안전한 프로그래밍을 제공합니다. 다음 샘플에서는 열거형 형식에 유 효한 값을 모두 찾은 다음, 해당 값을 문자열 표현에 매핑하는 사전을 작성합니다.

C#

```
public static Dictionary<int, string> EnumNamedValues<T>() where T :  
System.Enum
```

```

{
    var result = new Dictionary<int, string>();
    var values = Enum.GetValues(typeof(T));

    foreach (int item in values)
        result.Add(item, Enum.GetName(typeof(T), item)!);
    return result;
}

```

`Enum.GetValues` 및 `Enum.GetName`은 성능에 영향을 미치는 리플렉션을 사용합니다. 리플렉션이 필요한 호출을 반복하는 대신, `EnumNamedValues`를 호출하여 캐시되고 다시 사용되는 컬렉션을 작성할 수 있습니다.

다음 샘플과 같이 이 메서드는 열거형을 만들고 해당 값과 이름의 사전을 작성하는 데 사용할 수 있습니다.

C#

```

enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}

```

C#

```

var map = EnumNamedValues<Rainbow>();

foreach (var pair in map)
    Console.WriteLine($"{pair.Key}:\t{pair.Value}");

```

## 형식 인수는 선언된 인터페이스를 구현함

일부 시나리오에서는 형식 매개 변수에 제공된 인수가 해당 인터페이스를 구현해야 합니다. 예시:

C#

```

public interface IAdditionSubtraction<T> where T : IAdditionSubtraction<T>
{
    public abstract static T operator +(T left, T right);
}

```

```
    public abstract static T operator -(T left, T right);  
}
```

이 패턴을 사용하면 C# 컴파일러가 오버로드된 연산자나 `static virtual` 또는 `static abstract` 메서드에 대한 포함 형식을 결정할 수 있습니다. 포함 형식에 더하기 및 빼기 연산자를 정의할 수 있도록 구문을 제공합니다. 이 제약 조건이 없으면 형식 매개 변수가 아닌 매개 변수와 인수를 인터페이스로 선언해야 합니다.

C#

```
public interface IAdditionSubtraction<T> where T : IAdditionSubtraction<T>  
{  
    public abstract static IAdditionSubtraction<T> operator +(  
        IAdditionSubtraction<T> left,  
        IAdditionSubtraction<T> right);  
  
    public abstract static IAdditionSubtraction<T> operator -(  
        IAdditionSubtraction<T> left,  
        IAdditionSubtraction<T> right);  
}
```

앞의 구문에서는 구현자가 해당 메서드에 대해 [명시적 인터페이스 구현](#)을 사용해야 합니다. 추가 제약 조건 사용 약관을 제공하면 인터페이스가 형식 매개 변수 측면에서 연산자를 정의할 수 있습니다. 인터페이스를 구현하는 형식은 인터페이스 메서드를 암시적으로 구현할 수 있습니다.

## 참고 항목

- [System.Collections.Generic](#)
- [제네릭 소개](#)
- [제네릭 클래스](#)
- [new 제약 조건](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# 제네릭 클래스(C# 프로그래밍 가이드)

아티클 • 2024. 03. 12.

제네릭 클래스는 특정 데이터 형식과 관련이 없는 작업을 캡슐화합니다. 제네릭 클래스는 연결된 목록, 해시 테이블, 스택, 큐, 트리 등의 컬렉션에 가장 일반적으로 사용됩니다. 컬렉션에서 항목을 추가하고 제거하는 등의 작업은 저장되는 데이터의 형식과 관계없이 기본적으로 동일한 방식으로 수행됩니다.

컬렉션 클래스를 필요로 하는 대부분의 시나리오에서는 .NET 클래스 라이브러리에서 제공하는 컬렉션 클래스를 사용하는 것이 좋습니다. 이러한 클래스 사용에 대한 자세한 내용은 [.NET의 제네릭 컬렉션](#)을 참조하세요.

일반적으로 기존의 구체적인 클래스로 시작하여 일반성과 편의성의 균형이 맞을 때까지 형식을 하나씩 형식 매개 변수로 변경하여 제네릭 클래스를 만듭니다. 고유한 제네릭 클래스를 만들 때 중요한 고려 사항은 다음과 같습니다.

- 형식 매개 변수로 일반화할 형식

일반적으로 매개 변수화할 수 있는 형식이 많을수록 코드의 유용성과 재사용 가능성이 향상됩니다. 그러나 지나친 일반화는 다른 개발자가 읽거나 이해하기 어려운 코드를 만들어낼 소지가 있습니다.

- 형식 매개 변수에 적용할 제약 조건([형식 매개 변수에 대한 제약 조건](#) 참조)

필요한 형식을 처리할 수 있는 범위 내에서 최대한 많은 제약 조건을 적용하는 것이 좋습니다. 예를 들어 제네릭 클래스를 참조 형식으로만 사용하려는 경우에는 클래스 제약 조건을 적용합니다. 이렇게 하면 클래스를 값 형식으로 잘못 사용하는 것을 막을 수 있고, `as` 연산자를 `T`에 적용하여 null 값 여부를 확인할 수 있습니다.

- 제네릭 동작을 기본 클래스와 서브클래스로 분할할지 여부

제네릭 클래스는 기본 클래스가 될 수 있으므로 제네릭이 아닌 클래스에 적용되는 디자인 고려 사항이 동일하게 적용됩니다. 자세한 내용은 이 항목의 뒷부분에서 설명하는 제네릭 기본 클래스에서 상속하는 데 대한 규칙을 참조하세요.

- 제네릭 인터페이스를 하나 이상 구현할지 여부

예를 들어 제네릭 기반 컬렉션에 항목을 만드는 데 사용될 클래스를 디자인할 경우 클래스 형식이 `T`인 `IComparable<T>`와 같은 인터페이스를 구현해야 할 수 있습니다.

간단한 제네릭 클래스의 예제를 보려면 [제네릭 소개](#)를 참조하세요.

형식 매개 변수와 제약 조건에 대한 규칙은 제네릭 클래스 동작, 특히 상속과 멤버 접근성에 몇 가지 영향을 줍니다. 계속하려면 몇 가지 용어를 이해하고 있어야 합니다. 제네릭 클래스 `Node<T>`, 의 경우 클라이언트 코드는 형식 인수를 지정하여 클래스를 참조하여 닫힌 생성된 형식을 만들거나 형식 `Node<int>` 매개 변수를 지정하지 않은 상태로 두어(예: 제네릭 기본 클래스를 지정할 때) 열린 생성된 형식(`Node<T>`)을 만들 수 있습니다. 제네릭 클래스는 구체적인 클래스, 폐쇄형 생성 클래스 또는 개방형 생성 기본 클래스에서 상속할 수 있습니다.

C#

```
class BaseNode { }
class BaseNodeGeneric<T> { }

// concrete type
class NodeConcrete<T> : BaseNode { }

//closed constructed type
class NodeClosed<T> : BaseNodeGeneric<int> { }

//open constructed type
class NodeOpen<T> : BaseNodeGeneric<T> { }
```

제네릭이 아닌 구체적인 클래스는 폐쇄형 생성 기본 클래스에서는 상속할 수 있지만 개방형 생성 클래스 또는 형식 매개 변수에서는 상속할 수 없습니다. 이는 런타임에 클라이언트 코드에서 기본 클래스를 인스턴스화할 때 필요한 형식 인수를 제공할 수 없기 때문입니다.

C#

```
//No error
class Node1 : BaseNodeGeneric<int> { }

//Generates an error
//class Node2 : BaseNodeGeneric<T> {}

//Generates an error
//class Node3 : T {}
```

개방형 생성 형식에서 상속하는 제네릭 클래스에서는 다음 코드와 같이 상속하는 클래스에서 공유하지 않는 모든 기본 클래스 형식 매개 변수에 대해 형식 인수를 제공해야 합니다.

C#

```
class BaseNodeMultiple<T, U> { }

//No error
```

```
class Node4<T> : BaseNodeMultiple<T, int> { }

//No error
class Node5<T, U> : BaseNodeMultiple<T, U> { }

//Generates an error
//class Node6<T> : BaseNodeMultiple<T, U> {}
```

개방형 생성 형식에서 상속하는 제네릭 클래스에서는 기본 형식에 대한 제약 조건을 포함하거나 암시하는 제약 조건을 지정해야 합니다.

C#

```
class NodeItem<T> where T : System.IComparable<T>, new() { }
class SpecialNodeItem<T> : NodeItem<T> where T : System.IComparable<T>, new() { }
```

제네릭 형식은 다음과 같이 여러 형식 매개 변수와 제약 조건을 사용할 수 있습니다.

C#

```
class SuperKeyType<K, V, U>
    where U : System.IComparable<U>
    where V : new()
{ }
```

개방형 생성 형식 및 폐쇄형 생성 형식은 메서드 매개 변수로 사용할 수 있습니다.

C#

```
void Swap<T>(List<T> list1, List<T> list2)
{
    //code to swap items
}

void Swap(List<int> list1, List<int> list2)
{
    //code to swap items
}
```

제네릭 클래스에서 인터페이스를 구현하면 이 클래스의 모든 인스턴스를 해당 인터페이스에 캐스팅할 수 있습니다.

제네릭 클래스는 고정적입니다. 즉, 입력 매개 변수에서 `List<BaseClass>`를 지정하면 `List<DerivedClass>`를 제공하려고 할 때 컴파일 시간 오류가 발생합니다.

# 참고 항목

- System.Collections.Generic
- 제네릭
- 열거자의 상태 저장
- 상속 퍼즐, 1부

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

意见反馈

제품 사용자 의견 제공

# 제네릭 인터페이스(C# 프로그래밍 가이드)

아티클 • 2024. 03. 09.

제네릭 컬렉션 클래스에 대한 인터페이스 또는 컬렉션의 항목을 나타내는 제네릭 클래스에 대한 인터페이스를 정의하는 것이 대개 유용합니다. 값 형식에 대한 boxing 및 unboxing 작업을 피하려면 제네릭 클래스에서 [IComparable<T>](#)와 같은 [제네릭 인터페이스](#)를 사용하는 것이 좋습니다. .NET 클래스 라이브러리에는 [System.Collections.Generic](#) 네임스페이스의 컬렉션 클래스에 사용할 제네릭 인터페이스가 여러 개 정의되어 있습니다. 이러한 인터페이스에 대한 자세한 내용은 [제네릭 인터페이스](#)를 참조하세요.

인터페이스를 형식 매개 변수에 대한 제약 조건으로 지정한 경우 이 인터페이스를 구현하는 형식만 사용할 수 있습니다. 다음 코드 예제는 [GenericList<T>](#) 클래스에서 파생되는 [SortedList<T>](#) 클래스를 보여 줍니다. 자세한 내용은 [제네릭 소개](#)를 참조하세요.

[SortedList<T>](#) 는 `where T : IComparable<T>` 제약 조건을 추가합니다. 이 제약 조건을 사용하면 [SortedList<T>](#)의 [BubbleSort](#) 메서드가 목록 요소에 제네릭 [CompareTo](#) 메서드를 사용할 수 있습니다. 이 예제에서 목록 요소는 [IComparable<Person>](#)을 구현하는 단순 클래스인 [Person](#)입니다.

C#

```
//Type parameter T in angle brackets.
public class GenericList<T> : System.Collections.Generic.IEnumerable<T>
{
    protected Node head;
    protected Node current = null;

    // Nested class is also generic on T
    protected class Node
    {
        public Node next;
        private T data; //T as private member datatype

        public Node(T t) //T used in non-generic constructor
        {
            next = null;
            data = t;
        }

        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        public T Data //T as return type of property
    }
}
```

```

        {
            get { return data; }
            set { data = value; }
        }
    }

    public GenericList() //constructor
    {
        head = null;
    }

    public void AddHead(T t) //T as method parameter type
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    // Implementation of the iterator
    public System.Collections.Generic.IEnumerator<T> GetEnumerator()
    {
        Node current = head;
        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    // IEnumerable<T> inherits from IEnumerable, therefore this class
    // must implement both the generic and non-generic versions of
    // GetEnumerator. In most cases, the non-generic method can
    // simply call the generic method.
    System.Collections.IEnumerator
    System.Collections.IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}

public class SortedList<T> : GenericList<T> where T : System.IComparable<T>
{
    // A simple, unoptimized sort algorithm that
    // orders list elements from lowest to highest:

    public void BubbleSort()
    {
        if (null == head || null == head.Next)
        {
            return;
        }
        bool swapped;

        do
        {

```

```

        Node previous = null;
        Node current = head;
        swapped = false;

        while (current.next != null)
        {
            // Because we need to call this method, the SortedList
            // class is constrained on IComparable<T>
            if (current.Data.CompareTo(current.next.Data) > 0)
            {
                Node tmp = current.next;
                current.next = current.next.next;
                tmp.next = current;

                if (previous == null)
                {
                    head = tmp;
                }
                else
                {
                    previous.next = tmp;
                }
                previous = tmp;
                swapped = true;
            }
            else
            {
                previous = current;
                current = current.next;
            }
        }
    } while (swapped);
}

// A simple class that implements IComparable<T> using itself as the
// type argument. This is a common design pattern in objects that
// are stored in generic lists.
public class Person : System.IComparable<Person>
{
    string name;
    int age;

    public Person(string s, int i)
    {
        name = s;
        age = i;
    }

    // This will cause list elements to be sorted on age values.
    public int CompareTo(Person p)
    {
        return age - p.age;
    }
}

```

```
public override string ToString()
{
    return name + ":" + age;
}

// Must implement Equals.
public bool Equals(Person p)
{
    return (this.age == p.age);
}
}

public class Program
{
    public static void Main()
    {
        //Declare and instantiate a new generic SortedList class.
        //Person is the type argument.
        SortedList<Person> list = new SortedList<Person>();

        //Create name and age values to initialize Person objects.
        string[] names =
        [
            "Franscoise",
            "Bill",
            "Li",
            "Sandra",
            "Gunnar",
            "Alok",
            "Hiroyuki",
            "Maria",
            "Alessandro",
            "Raul"
        ];

        int[] ages = [45, 19, 28, 23, 18, 9, 108, 72, 30, 35];

        //Populate the list.
        for (int x = 0; x < 10; x++)
        {
            list.AddHead(new Person(names[x], ages[x]));
        }

        //Print out unsorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
        System.Console.WriteLine("Done with unsorted list");

        //Sort the list.
        list.BubbleSort();

        //Print out sorted list.
        foreach (Person p in list)
```

```

    {
        System.Console.WriteLine(p.ToString());
    }
    System.Console.WriteLine("Done with sorted list");
}
}

```

단일 형식에 다음과 같이 여러 인터페이스를 제약 조건으로 지정할 수 있습니다.

C#

```

class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{
}

```

하나의 인터페이스는 다음과 같은 두 개 이상의 형식 매개 변수를 정의할 수 있습니다.

C#

```

interface IDictionary<K, V>
{
}

```

클래스에 적용되는 상속 규칙은 인터페이스에도 적용됩니다.

C#

```

interface IMonth<T> { }

interface IJanuary : IMonth<int> { } //No error
interface IFebruary<T> : IMonth<int> { } //No error
interface IMarch<T> : IMonth<T> { }      //No error
                                         //interface IApril<T> : IMonth<T, U>
{} //Error

```

제네릭 인터페이스가 공변(covariant)인 경우, 제네릭 인터페이스는 제네릭이 아닌 인터페이스에서 상속할 수 있습니다. 즉, 제네릭 인터페이스는 형식 매개 변수만 반환 값으로 사용합니다. .NET 클래스 라이브러리에서 `IEnumerable<T>`은 `IEnumerable`에서 상속받습니다. `IEnumerable<T>`이 `GetEnumerator`의 반환 값과 `Current` 속성 getter에 `T`만 사용하기 때문입니다.

구체적인 클래스는 다음과 같이 폐쇄형으로 생성된 인터페이스를 구현할 수 있습니다.

C#

```

interface IBaseInterface<T> { }

```

```
class SampleClass : IBaseInterface<string> { }
```

클래스 매개 변수 목록이 다음과 같이 인터페이스에 필요한 모든 인수를 제공하는 경우에 한해 제네릭 클래스는 제네릭 인터페이스 또는 폐쇄형으로 생성된 인터페이스를 구현할 수 있습니다.

C#

```
interface IBaseInterface1<T> { }
interface IBaseInterface2<T, U> { }

class SampleClass1<T> : IBaseInterface1<T> { }           //No error
class SampleClass2<T> : IBaseInterface2<T, string> { } //No error
```

제네릭 클래스, 제네릭 구조체 또는 제네릭 인터페이스 내의 메서드에는 메서드 오버로드를 제어하는 규칙이 동일하게 적용됩니다. 자세한 내용은 [제네릭 메서드](#)를 참조하세요.

C# 11부터 인터페이스는 `static abstract` 또는 `static virtual` 멤버를 선언할 수 있습니다. `static abstract` 또는 `static virtual` 멤버를 선언하는 인터페이스는 거의 항상 제네릭 인터페이스입니다. 컴파일러는 컴파일 시간에 `static virtual` 및 `static abstract` 메서드에 대한 호출을 확인해야 합니다. 인터페이스에 선언된 `static virtual` 및 `static abstract` 메서드에는 클래스에 선언된 `virtual` 또는 `abstract` 메서드와 유사한 런타임 디스패치 메커니즘이 없습니다. 대신 컴파일러는 컴파일 시간에 사용 가능한 형식 정보를 사용합니다. 이러한 멤버는 일반적으로 제네릭 인터페이스에서 선언됩니다. 또한 `static virtual` 또는 `static abstract` 메서드를 선언하는 대부분의 인터페이스는 형식 매개 변수 중 하나가 [선언된 인터페이스를 구현](#)해야 한다고 선언합니다. 그런 다음 컴파일러는 제공된 형식 인수를 사용하여 선언된 멤버의 형식을 확인합니다.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [제네릭 소개](#)
- [interface](#)
- [제네릭](#)

GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

수도 있습니다. 자세한 내용은  
참여자 가이드를 참조하세요.

☞ 설명서 문제 열기

☞ 제품 사용자 의견 제공

# 제네릭 메서드(C# 프로그래밍 가이드)

아티클 • 2024. 03. 28.

제네릭 메서드는 다음과 같은 형식 매개 변수를 사용하여 선언된 메서드입니다.

C#

```
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

다음 코드 예제는 형식 인수에 `int`를 사용하여 메서드를 호출하는 한 가지 방법을 보여줍니다.

C#

```
public static void TestSwap()
{
    int a = 1;
    int b = 2;

    Swap<int>(ref a, ref b);
    System.Console.WriteLine(a + " " + b);
}
```

형식 인수를 생략하고 컴파일러에서 이를 자동으로 유추하도록 할 수도 있습니다. `Swap`에 대한 다음 호출은 위 예제의 호출과 동일한 작업을 수행합니다.

C#

```
Swap(ref a, ref b);
```

정적 메서드와 인스턴스 메서드에는 형식 유추와 동일한 규칙이 적용됩니다. 컴파일러는 전달한 메서드 인수에 따라 형식 매개 변수를 유추할 수 있지만, 제약 조건이나 반환 값만으로는 형식 매개 변수를 유추할 수 없습니다. 따라서 매개 변수가 없는 메서드에 대해서는 형식 유추가 실행되지 않습니다. 형식 유추는 컴파일러에서 오버로드된 메서드 시그니처를 확인하려고 하기 전에 컴파일 시간에 진행됩니다. 컴파일러는 동일한 이름을 공유하는 모든 제네릭 메서드에 형식 유추 논리를 적용합니다. 오버로드 확인 단계에서 컴파일러는 형식 유추에 성공한 제네릭 메서드만 포함합니다.

제네릭 클래스 내에서 제네릭이 아닌 메서드는 다음과 같은 클래스 수준의 형식 매개 변수에 액세스할 수 있습니다.

C#

```
class SampleClass<T>
{
    void Swap(ref T lhs, ref T rhs) { }
}
```

포함하는 클래스와 동일한 형식 매개 변수를 사용하는 제네릭 메서드를 정의하면 컴파일러에서 CS0693 경고가 발생합니다. 메서드 범위 내에서 내부 `T`에 제공된 인수가 외부 `T`에 제공된 인수를 숨기기 때문입니다. 클래스를 인스턴스화할 때 제공한 형식 인수가 아닌 다른 형식 인수를 사용하여 제네릭 클래스 메서드를 호출할 수 있으려면 다음 예제의 `GenericList2<T>`에서와 같이 메서드의 형식 매개 변수에 다른 식별자를 제공해 보세요.

C#

```
class GenericList<T>
{
    // CS0693.
    void SampleMethod<T>() { }

    class GenericList2<T>
    {
        // No warning.
        void SampleMethod<U>() { }
    }
}
```

메서드에서 형식 매개 변수에 대한 더 구체적인 작업을 수행하려면 제약 조건을 사용합니다. `SwapIfGreater<T>`라는 이 버전의 `Swap<T>`은 `IComparable<T>`을 구현하는 형식 인수와 함께 사용할 수 있습니다.

C#

```
void SwapIfGreater<T>(ref T lhs, ref T rhs) where T : System.IComparable<T>
{
    T temp;
    if (lhs.CompareTo(rhs) > 0)
    {
        temp = lhs;
        lhs = rhs;
        rhs = temp;
    }
}
```

제네릭 메서드는 몇몇 형식 매개 변수에 오버로드될 수 있습니다. 예를 들어 다음 메서드는 모두 동일한 클래스에 지정할 수 있습니다.

C#

```
void DoWork() { }
void DoWork<T>() { }
void DoWork<T, U>() { }
```

형식 매개 변수를 메서드의 반환 형식으로 사용할 수도 있습니다. 다음 코드 예제에서는 형식 `T`의 배열을 반환하는 메서드를 보여줍니다.

C#

```
T[] Swap<T>(T a, T b)
{
    return [b, a];
}
```

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)을 참조하세요.

## 참고 항목

- [System.Collections.Generic](#)
- [제네릭 소개](#)
- [메서드](#)

### GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

### .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

[설명서 문제 열기](#)

[제품 사용자 의견 제공](#)

# 제네릭 및 배열(C# 프로그래밍 가이드)

아티클 • 2024. 03. 12.

하한이 0인 1차원 배열은 자동으로 `IList<T>`를 구현합니다. 이러한 특성으로 인해 동일한 코드를 사용하여 배열 및 기타 컬렉션 형식에서 반복할 수 있는 제네릭 메서드를 만들 수 있습니다. 이 기술은 주로 컬렉션에서 데이터를 읽는 데 유용합니다. `IList<T>` 인터페이스는 배열에서 요소를 추가하거나 제거하는 데 사용할 수 없습니다. 이 컨텍스트의 배열에서 `RemoveAt`과 같은 `IList<T>` 메서드를 호출하려는 경우 예외가 throw됩니다.

다음 코드 예제는 `IList<T>` 입력 매개 변수를 사용하는 단일 제네릭 메서드가 목록과 배열(여기에서는 정수 배열) 모두를 통해 반복하는 방법을 보여 줍니다.

C#

```
class Program
{
    static void Main()
    {
        int[] arr = [0, 1, 2, 3, 4];
        List<int> list = new List<int>();

        for (int x = 5; x < 10; x++)
        {
            list.Add(x);
        }

        ProcessItems<int>(arr);
        ProcessItems<int>(list);
    }

    static void ProcessItems<T>(IList<T> coll)
    {
        // IsReadOnly returns True for the array and False for the List.
        System.Console.WriteLine
            ("IsReadOnly returns {0} for this collection.",
            coll.IsReadOnly);

        // The following statement causes a run-time exception for the
        // array, but not for the List.
        //coll.RemoveAt(4);

        foreach (T item in coll)
        {
            System.Console.Write(item?.ToString() + " ");
        }
        System.Console.WriteLine();
    }
}
```

# 참고 항목

- [System.Collections.Generic](#)
- [제네릭](#)
- [배열](#)
- [제네릭](#)

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

意见反馈

제품 사용자 의견 제공

# 제네릭 대리자(C# 프로그래밍 가이드)

아티클 • 2024. 03. 12.

대리자는 자체 형식 매개 변수를 정의할 수 있습니다. 제네릭 대리자를 참조하는 코드는 다음 예제와 같이 제네릭 클래스를 인스턴스화하거나 제네릭 메서드를 호출하는 것과 같은 방법으로 형식 인수를 지정하여 폐쇄형 생성 형식을 만들 수 있습니다.

C#

```
public delegate void Del<T>(T item);
public static void Notify(int i) { }

Del<int> m1 = new Del<int>(Notify);
```

C# 버전 2.0에는 메서드 그룹 변환이라는 새로운 기능이 있습니다. 이 기능은 제네릭 대리자 형식은 물론 구체적인 대리자 형식에도 적용되며, 이 기능을 사용하여 위 코드 줄을 다음과 같이 간단한 구문으로 작성할 수 있습니다.

C#

```
Del<int> m2 = Notify;
```

제네릭 클래스 내에 정의된 대리자는 클래스 메서드와 같은 방식으로 제네릭 클래스 형식 매개 변수를 사용할 수 있습니다.

C#

```
class Stack<T>
{
    public delegate void StackDelegate(T[] items);
}
```

대리자를 참조하는 코드는 포함 클래스의 형식 인수를 다음과 같이 지정해야 합니다.

C#

```
private static void DoWork(float[] items) { }

public static void TestStack()
{
    Stack<float> s = new Stack<float>();
    Stack<float>.StackDelegate d = DoWork;
}
```

제네릭 대리자는 sender 인수를 강력하게 형식화할 수 있고 더 이상 sender 인수와 [Object](#) 간에 캐스팅하지 않아도 되기 때문에 일반적인 디자인 패턴을 기반으로 하는 이벤트를 정의할 때 특히 유용합니다.

C#

```
delegate void StackEventHandler<T, U>(T sender, U eventArgs);

class Stack<T>
{
    public class StackEventArgs : System.EventArgs { }
    public event StackEventHandler<Stack<T>, StackEventArgs>? StackEvent;

    protected virtual void OnStackChanged(StackEventArgs a)
    {
        if (StackEvent is not null)
            StackEvent(this, a);
    }
}

class SampleClass
{
    public void HandleStackChange<T>(Stack<T> stack, Stack<T>.StackEventArgs args) { }
}

public static void Test()
{
    Stack<double> s = new Stack<double>();
    SampleClass o = new SampleClass();
    s.StackEvent += o.HandleStackChange;
}
```

## 참고 항목

- [System.Collections.Generic](#)
- [제네릭 소개](#)
- [제네릭 메서드](#)
- [제네릭 클래스](#)
- [제네릭 인터페이스](#)
- [대리자](#)
- [제네릭](#)

## 동작

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

 [설명서 문제 열기](#)

 [제품 사용자 의견 제공](#)

# C++ 템플릿과 C# 제네릭의 차이점(C# 프로그래밍 가이드)

아티클 • 2023. 04. 07.

C# 제네릭 및 C++ 템플릿은 둘 다 매개 변수가 있는 형식을 지원하는 언어 기능입니다. 그러나 둘 사이에는 많은 차이가 있습니다. 구문 수준에서 C# 제네릭은 매개 변수가 있는 형식에 대한 더 간단한 접근 방식으로, C++ 템플릿의 복잡함이 없습니다. 또한 C#은 C++ 템플릿에서 제공하는 기능 중 일부를 제공하지 않습니다. 구현 수준에서 주요 차이점은 C# 제네릭 형식 자체가 런타임에 수행되어 인스턴스화된 개체에 대해 제네릭 형식 정보가 유지된다는 것입니다. 자세한 내용은 [런타임의 제네릭을 참조하세요](#).

다음은 C# 제네릭 및 C++ 템플릿 사이의 주요 차이점입니다.

- C# 제네릭은 C++ 템플릿과 동일한 수준의 유연성을 제공하지 않습니다. 예를 들어 C# 제네릭 클래스에서 산술 연산자는 호출할 수 없지만 사용자 정의 연산자는 호출할 수 있습니다.
- C#에서는 `template C<int i> {}` 같은 비형식 템플릿 매개 변수를 허용하지 않습니다.
- C#은 명시적 특수화 즉, 특정 형식에 대한 템플릿의 사용자 지정 구현을 지원하지 않습니다.
- C#은 부분 특수화 즉, 형식 인수의 하위 집합에 대한 사용자 지정 구현을 지원하지 않습니다.
- C#에서는 형식 매개 변수를 제네릭 형식에 대한 기본 클래스로 사용할 수 없습니다.
- C#에서는 형식 매개 변수가 기본 형식을 사용할 수 없습니다.
- C#에서 제네릭 형식 매개 변수 자체는 제네릭이 될 수 없지만 생성된 형식은 제네릭으로 사용할 수 있습니다. C++에서는 템플릿 매개 변수를 허용합니다.
- C++에서는 템플릿의 일부 형식 매개 변수에 적합하지 않아 형식 매개 변수로 사용되는 특정 형식을 확인하는 코드를 허용합니다. C#에서는 제약 조건을 충족하는 모든 형식에서 작동하는 방식으로 작성할 코드가 클래스에 필요합니다. 예를 들어 C++에서는 산술 연산자 `+` 및 `-`를 사용하는 함수를 형식 매개 변수의 개체에서 작성하여 이러한 연산자를 지원하지 않는 형식으로 템플릿을 인스턴스화할 때 오류를 생성할 수 있습니다. C#에서는 이를 허용하지 않습니다. 허용되는 유일한 언어 구문은 제약 조건에서 추론할 수 있는 구문입니다.

# 참고 항목

- C# 프로그래밍 가이드
- 제네릭 소개
- 템플릿

# 런타임의 제네릭(C# 프로그래밍 가이드)

아티클 • 2024. 04. 23.

제네릭 형식이나 메서드가 CIL(공용 중간 언어)로 컴파일되면 형식 매개 변수가 있는 것으로 식별하는 메타데이터가 포함됩니다. 제네릭 형식의 CIL이 사용되는 방식은 제공된 형식 매개 변수가 값 형식인지 참조 형식인지에 따라 다릅니다.

값 형식을 매개 변수로 사용하여 제네릭 형식을 처음 생성할 경우 런타임에서는 제공된 매개 변수를 CIL의 해당 위치에 대체하여 특수화된 제네릭 형식을 만듭니다. 고유한 값 형식이 매개 변수로 사용될 때마다 특수화된 제네릭 형식이 만들어집니다.

예를 들어 프로그램 코드에서 다음과 같이 정수로 구성된 스택을 선언한다고 가정합니다.

C#

```
Stack<int>? stack;
```

이 시점에서 런타임은 매개 변수를 정수로 적절히 대체하여 특수화된 버전의 `Stack<T>` 클래스를 생성합니다. 이제부터 프로그램 코드에서 정수 스택을 사용하면 런타임은 생성된 특수화 `Stack<T>` 클래스를 다시 사용합니다. 다음 예제에서는 `Stack<int>` 코드의 단일 인스턴스를 공유하는 정수 스택의 두 인스턴스를 만듭니다.

C#

```
Stack<int> stackOne = new Stack<int>();
Stack<int> stackTwo = new Stack<int>();
```

그러나 코드의 다른 지점에서 `long`과 같이 값 형식이 다르거나, 사용자 정의된 구조체를 매개 변수로 사용하는 다른 `Stack<T>` 클래스를 만들었다고 가정해 보겠습니다. 이 경우 런타임에서는 제네릭 형식의 다른 버전을 생성하여 CIL의 적절한 위치에 `long`을 대체합니다. 특수화된 각 제네릭 클래스에는 기본적으로 값 형식이 포함되므로 변환은 더 이상 필요하지 않습니다.

참조 형식에 대해서는 제네릭의 작동 방식이 조금 다릅니다. 참조 형식을 사용하여 제네릭 형식이 처음 생성될 때 런타임에서는 CIL의 매개 변수를 개체 참조로 대체하여 특수화된 제네릭 형식을 만듭니다. 이후 참조 형식과 관계없이 참조 형식을 매개 변수로 사용하여 생성된 형식이 인스턴스화될 때마다 런타임에서는 이전에 만든 특수화된 버전의 제네릭 형식을 다시 사용합니다. 이는 모든 참조의 크기가 동일하기 때문에 가능합니다.

예를 들어 `Customer` 클래스와 `Order` 클래스라는 두 참조 형식이 있고 `Customer` 형식의 스택을 만들었다고 가정합니다.

```
C#
```

```
class Customer { }  
class Order { }
```

```
C#
```

```
Stack<Customer> customers;
```

이 시점에서 런타임은 데이터를 저장하는 대신 이후에 채워질 개체 참조를 저장하는 특수화된 버전의 `Stack<T>` 클래스를 생성합니다. 다음 코드 줄에서 `Order`라는 다른 참조 형식의 스택을 만든다고 가정해 봅니다.

```
C#
```

```
Stack<Order> orders = new Stack<Order>();
```

값 형식과는 달리 `Order` 형식에 대한 또 다른 특수화된 버전의 `Stack<T>` 클래스는 만들 어지지 않습니다. 대신 특수화된 버전의 `Stack<T>` 클래스 인스턴스가 만들어지고 `orders` 변수가 이 인스턴스를 참조하도록 설정됩니다. 이후에 `Customer` 형식의 스택을 만드는 코드 줄이 나타난다고 가정해 봅니다.

```
C#
```

```
customers = new Stack<Customer>();
```

`Order` 형식을 사용하여 만든 `Stack<T>` 클래스를 사용한 경우와 마찬가지로 특수화된 `Stack<T>` 클래스의 다른 인스턴스가 생성됩니다. 여기에 포함된 포인터는 `Customer` 형식과 크기가 같은 메모리 영역을 참조하도록 설정됩니다. 참조 형식의 수는 프로그램마다 크게 다를 수 있으므로, 제네릭을 C# 방식으로 구현하면 컴파일러가 참조 형식의 제네릭 클래스에 대해 만드는 특수화된 클래스의 수가 1개로 줄어들어 코드가 매우 간결해집니다.

그뿐만 아니라, 값 형식 또는 참조 형식 매개 변수를 사용하여 제네릭 C# 클래스가 인스턴스화되면 리플렉션이 이를 런타임에 쿼리할 수 있고 실제 형식과 형식 매개 변수를 모두 확인할 수 있습니다.

## 참고 항목

- [System.Collections.Generic](#)
- [제네릭 소개](#)

- 제네릭

## ⌚ GitHub에서 Microsoft와 공동 작업

이 콘텐츠의 원본은 GitHub에서 찾을 수 있으며, 여기서 문제와 끌어오기 요청을 만들고 검토할 수도 있습니다. 자세한 내용은 [참여자 가이드](#)를 참조하세요.

.NET

## .NET 피드백

.NET은(는) 오픈 소스 프로젝트입니다. 다음 링크를 선택하여 피드백을 제공해 주세요.

🌟 설명서 문제 열기

↗️ 제품 사용자 의견 제공

# C# 언어 참조

언어 참조는 초보자와 숙련된 C# 및 .NET 개발자를 위한 C# 구문 및 관용구에 대한 비공식적 참조를 제공합니다.

## C# 언어 참조

### 개요

[C# 언어 전략](#)

### 참조

[C# 키워드](#)

[C# 연산자 및 식](#)

[언어 버전 구성](#)

[C# 언어 사양 - C# 8 초안 진행 중](#)

## 새로운 기능

### 새로운 기능

[C# 12의 새로운 기능](#)

[C# 11의 새로운 기능](#)

[C# 10의 새로운 기능](#)

### 참조

[C# 컴파일러의 호환성이 손상되는 변경](#)

[버전 호환성](#)

## 연결 유지하기

### 참조

[.NET 개발자 커뮤니티 ↗](#)

[YouTube](#)

[Twitter](#)

# 사양

C# 언어에 대한 자세한 사양과 최신 기능에 대한 자세한 사양을 읽어 보세요.

## ECMA 사양 및 최신 기능

### ▣ 개요

[사양 프로세스](#)

[자세한 ECMA 사양 내용](#)

### ▣ 참조

[최신 기능 사양](#)

## C# ECMA 사양 초안 작성 - 소개 자료

### ▣ 참조

[머리말](#)

[소개](#)

### ▣ 참조

[범위](#)

[표준 참조](#)

[용어 및 정의](#)

[일반 설명](#)

[규칙](#)

## C# ECMA 사양 초안 작성 - 언어 사양

### ▣ 참조

[어휘 구조](#)

기본 개념

유형

variables

변환

패턴

표현식

문

---

#### 참조

네임스페이스

클래스

구조체

배열

인터페이스

열거형

대리자

예외

특성

안전하지 않은 코드

---

## C# ECMA 사양 초안 작성- 부록

---

#### 참조

문법

이식성 문제

표준 라이브러리

설명서 주석

참고 문헌