

# Contents

C# 설명서

C# 둘러보기

소개

유형

프로그램 구성 요소

주 언어 영역

자습서

개요

C#을 사용한 프로그래밍 소개

첫 번째 단원 선택

Hello World

C#의 숫자

분기 및 루프

목록 컬렉션

로컬 환경에서 작업

환경 설정

C#의 숫자

분기 및 루프

목록 컬렉션

클래스 소개

개체 지향 프로그래밍

레코드 종류 탐색

최상위 문 탐색

개체 패턴 탐색

문자열 보간 탐색 - 대화형 자습서

사용자 환경에서 문자열 보간 탐색

문자열 보간용 고급 시나리오

기본 인터페이스 메서드로 인터페이스를 안전하게 업데이트

기본 인터페이스 메서드로 믹스인 기능 만들기

인덱스 및 범위 탐색  
nullable 참조 형식 작업  
앱을 nullable 참조 형식으로 업그레이드  
비동기 스트림 생성 및 사용  
패턴 일치를 통한 데이터 기반 알고리즘 빌드  
콘솔 애플리케이션  
REST 클라이언트  
C# 및 .NET의 상속  
LINQ 작업  
특성 사용  
C#의 새로운 기능  
C# 9.0  
C# 8.0  
C# 7.0~7.3  
컴파일러의 호환성이 손상되는 변경  
C# 버전 기록  
언어와 프레임워크 간 관계  
버전 및 업데이트 고려 사항  
C# 개념  
C# 형식 시스템  
nullable 참조 형식  
nullable 참조 형식을 사용하기 위한 전략 선택  
네임스페이스  
기본 형식  
클래스  
튜플 및 기타 형식 분해  
인터페이스  
메서드  
속성  
인덱서  
버림  
제네릭

반복기

대리자 및 이벤트

대리자 소개

System.Delegate 및 대리자 키워드

강력한 형식의 대리자

대리자에 대한 일반적인 패턴

이벤트 소개

표준 .NET 이벤트 패턴

업데이트된 .NET 이벤트 패턴

대리자 및 이벤트를 구별

LINQ(Language-Integrated Query)

LINQ 개요

쿼리 식 기본 사항

C#의 LINQ

C#에서 LINQ 쿼리 작성하기

개체의 컬렉션 쿼리

메서드에서 쿼리 반환

쿼리 결과를 메모리에 저장

쿼리 결과 그룹화

중첩 그룹 만들기

그룹화 작업에서 하위 쿼리 수행

연속 키를 기준으로 결과 그룹화

런타임에 동적으로 조건자 필터 지정

내부 조인 수행

그룹화 조인 수행

왼쪽 우선 외부 조인 수행

join 절 결과를 순서대로 정렬

복합 키를 사용하여 조인

사용자 지정 조인 작업 수행

쿼리 식의 Null 값 처리

쿼리 식의 예외 처리

비동기 프로그래밍

패턴 일치

안전하고 효율적인 코드 작성

식 트리

식 트리 소개

식 트리 설명

식 트리를 지원하는 프레임워크 형식

식 실행

식 해석

식 빌드

식 변환

요약

기본 상호 운용성

코드 문서화

버전 관리

C# 방법 문서

문서 인덱스

문자열을 부분 문자열로 분할

문자열 연결

검색 문자열

문자열 내용 수정

문자열 비교

패턴 일치 및 `is/as` 연산자를 사용하여 안전하게 캐스팅

.NET Compiler Platform SDK(Roslyn API)

.NET Compiler Platform SDK(Roslyn API) 개요

컴파일러 API 모델 이해

구문 작업

의미 체계 작업

작업 영역 작업

구문 시각화 도우미를 사용하여 코드 살펴보기

빠른 시작

구문 분석

의미 체계 분석

구문 변환

자습서

첫 번째 분석기 및 코드 수정 빌드

C# 프로그래밍 가이드

개요

C# 프로그램 내부

C# 프로그램 내용

Hello World -- 프로그램 처음 만들기

C# 프로그램의 일반적인 구조체

식별자 이름

C# 코딩 규칙

Main()과 명령줄 인수

개요

명령줄 인수

명령줄 인수를 표시하는 방법

Main() 반환 값

프로그래밍 개념

개요

비동기 프로그래밍

개요

작업 비동기 프로그래밍 모델

비동기 반환 형식

작업 취소

작업 목록 취소

일정 기간 이후 작업 취소

완료되면 비동기 작업 처리

비동기 파일 액세스

특성

개요

사용자 지정 특성 만들기

리플렉션을 사용하여 특성 액세스

특성을 사용하여 C/C++ 공용 구조체를 만드는 방법

컬렉션

공변성(Covariance) 및 반공변성(Contravariance)

개요

제네릭 인터페이스의 가변성

가변 제네릭 인터페이스 만들기

제네릭 컬렉션에 인터페이스의 가변성 사용

대리자의 가변성

대리자의 가변성 사용

Func 및 Action 제네릭 대리자에 가변성 사용

식 트리

개요

식 트리 실행 방법

식 트리 수정 방법

식 트리를 사용하여 동적 쿼리 빌드 방법

Visual Studio에서 식 트리 디버그

DebugView 구문

반복기

LINQ(Language-Integrated Query)

개요

C#에서 LINQ 시작

LINQ 쿼리 소개

LINQ 및 제네릭 형식

기본 LINQ 쿼리 작업

LINQ를 통한 데이터 변환

LINQ 쿼리 작업의 형식 관계

LINQ의 쿼리 구문 및 메서드 구문

LINQ를 지원하는 C# 기능

연습: C#에서 쿼리 작성(LINQ)

표준 쿼리 연산자 개요

개요

표준 쿼리 연산자의 쿼리 식 구문

실행 방식에 따른 표준 쿼리 연산자 분류

데이터 정렬

집합 작업

데이터 필터링

수량자 작업

프로젝션 작업

데이터 분할

조인 작업

데이터 그룹화

생성 작업

같음 연산

요소 작업

데이터 형식 변환

연결 작업

집계 작업

## LINQ to Objects

개요

LINQ 및 문자열

방법 문서

문자열에서 단어가 나오는 횟수를 세는 방법(LINQ)

지정된 단어 집합이 포함된 문장을 쿼리하는 방법(LINQ)

문자열의 문자를 쿼리하는 방법(LINQ)

LINQ 쿼리와 정규식을 결합하는 방법

두 목록 간의 차집합을 구하는 방법(LINQ)

단어 또는 필드를 기준으로 텍스트 데이터를 정렬하거나 필터링하는 방법(LINQ)

구분된 파일의 필드를 다시 정렬하는 방법(LINQ)

문자열 컬렉션을 결합하고 비교하는 방법(LINQ)

여러 소스로 개체 컬렉션을 채우는 방법(LINQ)

그룹을 사용하여 파일을 여러 파일로 분할하는 방법(LINQ)

서로 다른 파일의 콘텐츠를 조인하는 방법(LINQ)

CSV 텍스트 파일의 열 값을 계산하는 방법(LINQ)

## LINQ 및 리플렉션

리플렉션을 사용하여 어셈블리의 메타데이터를 쿼리하는 방법(LINQ)

## LINQ 및 파일 딕렉터리

### 개요

지정된 특성 또는 이름을 사용하여 파일을 쿼리하는 방법

확장명을 기준으로 파일을 그룹화하는 방법(LINQ)

폴더 집합의 총 바이트 수를 쿼리하는 방법(LINQ)

두 폴더의 내용을 비교하는 방법(LINQ)

디렉터리 트리에서 가장 큰 파일을 쿼리하는 방법(LINQ)

디렉터리 트리에서 중복 파일을 쿼리하는 방법(LINQ)

폴더의 파일 내용 쿼리 방법(LINQ)

LINQ를 사용하여 ArrayList를 쿼리하는 방법

LINQ 쿼리용 사용자 지정 메서드를 추가하는 방법

LINQ to ADO.NET(포털 페이지)

LINQ 쿼리에 대한 데이터 소스 활성화

LINQ를 위한 Visual Studio IDE 및 도구 지원

### 반사

## Serialization(C#)

### 개요

XML 파일에 개체 데이터를 쓰는 방법

XML 파일에서 개체 데이터를 읽는 방법

연습: Visual Studio에서 개체 유지

문, 식, 연산자

### 개요

문

식 본문 멤버

익명 함수

### 개요

쿼리에 람다식을 사용하는 방법

같음 및 같음 비교

같음 비교

형식의 값 일치를 정의하는 방법

참조 일치(ID)를 테스트하는 방법

유형

형식 사용 및 정의

캐스팅 및 형식 변환(C#)

boxing 및 unboxing

바이트 배열을 정수로 변환하는 방법

문자열을 숫자로 변환하는 방법

16진수 문자열과 숫자 형식 간에 변환하는 방법

유형 동적 사용

연습: 동적 개체 만들기 및 사용(C# 및 Visual Basic)

클래스 및 구조체

개요

클래스

개체

상속

다형성

개요

Override 및 New 키워드를 사용하여 버전 관리

Override 및 New 키워드를 사용해야 하는 경우

ToString 메서드 재정의 방법

멤버

멤버 개요

추상 및 봉인 클래스와 클래스 멤버

정적 클래스 및 정적 클래스 멤버

액세스 한정자

필드

상수

추상 속성 정의 방법

C#에서 상수 정의 방법

속성

속성 개요

속성 사용

인터페이스 속성

접근자 액세스 가능성 제한[C#]

읽기/쓰기 속성 선언 및 사용 방법

자동으로 구현된 속성

자동으로 구현된 속성을 사용하여 간단한 클래스를 구현하는 방법

## 메서드

메서드 개요

로컬 함수

참조 반환 및 참조 로컬

매개 변수

매개 변수 전달

값 형식 매개 변수 전달

참조 형식 매개 변수 전달

메서드에 대한 구조체 전달과 클래스 참조 전달 간의 차이점을 이해하는 방법

암시적으로 형식화한 지역 변수

쿼리 식에서 암시적으로 형식화된 지역 변수 및 배열 사용 방법

## 확장명 메서드

사용자 지정 확장 메서드 구현 및 호출 방법

새 열거형 메서드 만드는 방법

명명된 인수 및 선택적 인수

Office 프로그래밍에 명명된 인수와 선택적 인수 사용 방법

## 생성자

생성자 개요

생성자 사용

인스턴스 생성자

전용 생성자

정적 생성자

복사 생성자 작성 방법

## 종료자

개체 및 컬렉션 이니셜라이저

개체 이니셜라이저를 사용하여 개체를 초기화하는 방법

컬렉션 이니셜라이저를 사용하여 사전을 초기화하는 방법

## 중첩 형식

Partial 클래스 및 메서드

익명 형식

쿼리에서 요소 속성의 하위 집합을 반환하는 방법

인터페이스

개요

명시적 인터페이스 구현

인터페이스 멤버를 명시적으로 구현하는 방법

두 인터페이스의 멤버를 명시적으로 구현하는 방법

대리자

개요

대리자 사용

대리자 비교: 명명된 메서드 및 무명 메서드

대리자를 결합하는 방법(멀티캐스트 대리자)(C# 프로그래밍 가이드)

대리자를 선언, 인스턴스화, 사용하는 방법

배열

개요

1차원 배열

다차원 배열

가변 배열

배열에 foreach 사용

인수로 배열 전달

암시적으로 형식화된 배열

문자열

문자열을 사용한 프로그래밍

문자열이 숫자 값을 나타내는지 확인하는 방법

인덱서

개요

인덱서 사용

인터페이스의 인덱서

속성 및 인덱서 비교

이벤트

개요

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

제네릭

개요

제네릭 형식 매개 변수

형식 매개 변수에 대한 제약 조건

제네릭 클래스

제네릭 인터페이스

제네릭 메서드

제네릭 및 배열

제네릭 대리자

C++ 템플릿과 C# 제네릭의 차이점

런타임의 제네릭

제네릭 및 리플렉션

제네릭 및 특성

네임스페이스

개요

Using namespaces

My 네임스페이스를 사용하는 방법

안전하지 않은 코드 및 포인터

개요 및 제한 사항

고정 크기 버퍼

포인터 형식

개요

포인터 변환

포인터를 사용하여 바이트 배열을 복사하는 방법

XML 문서 주석

개요

문서 주석에 대한 권장 태그

XML 파일 처리

문서 태그에 대한 구분 기호

XML 문서 기능을 사용하는 방법

설명서 태그 참조

<c>

<code>

cref 특성

<example>

<exception>

<include>

<inheritdoc>

<list>

<para>

<param>

<paramref>

<permission>

<remarks>

<returns>

<see>

<seealso>

<summary>

<typeparam>

<typeparamref>

<value>

예외 및 예외 처리

개요

예외 사용

예외 처리

예외 만들기 및 Throw

컴파일러 생성 예외

try-catch를 사용하여 예외를 처리하는 방법

finally를 사용하여 정리 코드를 실행하는 방법

CLS 규격이 아닌 예외를 catch하는 방법

## 파일 시스템 및 레지스트리

### 개요

디렉터리 트리를 반복하는 방법

파일, 폴더 및 드라이브에 대한 정보를 가져오는 방법

파일 또는 폴더를 만드는 방법

파일 및 폴더를 복사, 삭제, 이동하는 방법

파일 작업에 진행률 대화 상자를 제공하는 방법

텍스트 파일에 쓰는 방법

텍스트 파일에서 읽는 방법

텍스트 파일을 한 번에 한 줄씩 읽는 방법

레지스트리에 키를 만드는 방법

## 상호 운용성

.NET 상호 운용성

상호 운용성 개요

C# 기능을 사용하여 Office interop 개체에 액세스하는 방법

COM interop 프로그래밍에서 인덱싱된 속성을 사용하는 방법

플랫폼 호출을 사용하여 WAV 파일을 재생하는 방법

연습: Office 프로그래밍(C# 및 Visual Basic)

COM 클래스 예제

## 언어 참조

### 개요

언어 버전 구성

유형

값 형식

개요

정수 숫자 형식

부동 소수점 숫자 형식

기본 제공 숫자 변환

bool

char

열거형 형식

구조체 형식

튜플 형식

Nullable 값 형식

참조 형식

참조 형식의 기능

기본 제공 참조 형식

class

interface(인터페이스)

nullable 참조 형식

void

var

기본 제공 형식

관리되지 않는 형식

기본값

키워드

개요

한정자

액세스 한정자

빠른 참조

액세스 가능성 수준

내게 필요한 옵션 도메인

내게 필요한 옵션 수준 사용에 대한 제한

internal

private

protected

public

protected internal

private protected

abstract

async

const

event

extern

in(제네릭 한정자)

new(멤버 한정자)

out(제네릭 한정자)

override

readonly

sealed

static

unsafe

virtual

volatile

## 문 키워드

명령문 범주

선택문

if-else

switch

반복 문

do

for

foreach, in

while

점프 문

break

continue

goto

return

예외 처리 문

throw

try-catch

try-finally

try-catch-finally

Checked 및 Unchecked

개요

checked  
unchecked  
fixed 문  
lock 문  
메서드 매개 변수  
매개 변수 전달  
params  
in(매개 변수 한정자)  
ref  
out(매개 변수 한정자)

네임스페이스 키워드  
namespace  
using  
    using 컨텍스트  
    using 지시문  
    using 정적 지시문  
    using 문  
extern alias

형식 테스트 키워드  
is

제네릭 형식 제약 조건 키워드  
new 제약 조건  
where

액세스 키워드  
base  
this

리터럴 키워드  
null  
true 및 false  
default

상황별 키워드  
빠른 참조

add

Get

partial(형식)

partial(메서드)

remove

set

when(필터 조건)

value

yield

쿼리 키워드

빠른 참조

from 절

where 절

select 절

group 절

into

orderby 절

join 절

let 절

ascending

descending

On

equals

by

in

연산자 및 식

개요

산술 연산자

부울 논리 연산자

비트 및 시프트 연산자

같음 연산자

비교 연산자

멤버 액세스 연산자 및 식

형식 테스트 연산자 및 캐스트 식

사용자 정의 전환 연산자

포인터 관련 연산자

대입 연산자

람다 식

+ 및 += 연산자

- 및 -= 연산자

? : 연산자

! (null-forgiving) 연산자

?? 및 ??= 연산자

=> 연산자

:: 연산자

await 연산자

기본값 식

delegate 연산자

nameof 식

new 연산자

sizeof 연산자

stackalloc 식

switch 식

true 및 false 연산자

with 식

연산자 오버로드

특수 문자

개요

\$ -- 문자열 보간

@ -- 약어 식별자

컴파일러에서 읽은 특성

전역 특성

일반

호출자 정보

null 허용 정적 분석

전처리기 지시문

개요

#if

#else

#elif

#endif

#define

#undef

#warning

#error

#line

#nullable

#region

#endregion

#pragma

#pragma warning

#pragma checksum

C# 컴파일러 옵션

개요

csc.exe를 사용한 명령줄 빌드

Visual Studio 명령줄에 필요한 환경 변수를 설정하는 방법

C# 컴파일러 옵션 범주별 목록

C# 컴파일러 옵션 사전순 목록

@

-addmodule

-appconfig

-baseaddress

-bugreport

-checked

-codepage

-debug

-define  
-delaysign  
-deterministic  
-doc  
-errorreport  
-filealign  
-fullpaths  
-help, -?  
-highentropyva  
-keycontainer  
-keyfile  
-langversion  
-lib  
-link  
-linkresource  
-main  
-moduleassemblyname  
-noconfig  
-nologo  
-nostdlib  
-nowarn  
-nowin32manifest  
-nullable  
-optimize  
-out  
-pathmap  
-pdb  
-platform  
-preferreduilang  
-publicsign  
-recurse  
-reference

- refout
- refonly
- resource
- subsystemversion
- target
  - target:appcontainerexe
  - target:exe
  - target:library
  - target:module
  - target:winexe
  - target:winmdobj
- unsafe
- utf8output
- warn
- warnaserror
- win32icon
- win32manifest
- win32res

컴파일러 오류

C# 6.0 초안 사양

C# 7.0 - 9.0 제안

연습



# C# 언어 둘러보기

2021-02-18 • 39 minutes to read • [Edit Online](#)

C#("씨샵"이라고 발음합니다.)은 형식이 안전한 최신 개체 지향 프로그래밍 언어입니다. 개발자는 C#을 사용하면 .NET 에코시스템에서 실행되는 다양한 형식의 안전하고 강력한 애플리케이션을 빌드할 수 있습니다. C#은 C 언어 제품군에서 시작되었으며 C, C++, Java 및 JavaScript 프로그래머에게 친숙할 것입니다. 이 둘러보기에서는 C# 8 이상 언어의 주요 구성 요소를 간략하게 설명합니다. 대화형 예제를 통해 언어를 살펴보려면 [C# 소개](#) 자습서를 사용해 보세요.

C#은 개체 지향, '구성 요소 지향' 프로그래밍 언어입니다.\* C#은 이러한 개념을 직접적으로 지원하는 언어 구문을 제공함으로써 소프트웨어 구성 요소를 만들고 사용할 수 있는 자연 언어로 자리매김하게 되었습니다. 원본 이후로 C#은 새로운 워크로드를 지원하기 위한 기능 및 새로운 소프트웨어 디자인 사례를 추가했습니다.

여러 가지 C# 기능은 강력한 지속형 애플리케이션을 만드는데 도움이 됩니다. '[가비지 수집](#)'은 연결할 수 없는 사용되지 않는 개체에서 사용하는 메모리를 자동으로 회수합니다. '[Nullable 형식](#)'은 할당된 개체를 참조하지 않는 변수로부터 보호합니다. '[예외 처리](#)'는 오류 검색 및 복구에 대한 구조적이고 확장 가능한 방법을 제공합니다. '[람다 식](#)'은 함수형 프로그래밍 기술을 지원합니다. '[LINQ\(언어 통합 쿼리\)](#)' 구문은 모든 소스의 데이터로 작업하기 위한 일반적인 패턴을 만듭니다. '[비동기 작업](#)'에 대한 언어 지원은 분산 시스템을 빌드하기 위한 구문을 제공합니다. C#에는 '[통합 형식 시스템](#)'이 있습니다. `int` 및 `double`과 같은 기본 형식을 포함하는 모든 C# 형식은 단일 루트 `object`에서 상속됩니다. 모든 형식은 일반 작업 집합을 공유합니다. 모든 형식의 값을 일관된 방식으로 저장 및 전송하고 작업을 수행할 수 있습니다. 또한 C#은 사용자 정의 [참조 형식](#) 및 [값 형식](#)을 모두 지원합니다. C#은 개체의 동적 할당 및 경량 구조체의 인라인 스토리지를 허용합니다. C#은 향상된 형식 안전성과 성능을 제공하는 제네릭 메서드 및 형식을 지원합니다. C#은 컬렉션 클래스의 구현자가 클라이언트 코드에 대한 사용자 지정 동작을 정의하는 데 사용할 수 있는 반복기를 제공합니다.

C#은 시간 경과에 따라 프로그램 및 라이브러리가 호환 가능한 방식으로 개선될 수 있도록 '버전 관리'를 강조합니다. 버전 관리 고려 사항의 직접적인 영향을 받은 C# 설계의 측면에는 별도의 `virtual` 및 `override` 한정자, 메서드 오버로드 확인 규칙 및 명시적 인터페이스 멤버 선언에 대한 지원이 포함됩니다.

## .NET 아키텍처

C# 프로그램은 CLR(공용 언어 런타임)이라는 가상 실행 시스템이며 클래스 라이브러리 세트인 .NET에서 실행됩니다. CLR은 국제 표준인 CLI(공용 언어 인프라)를 Microsoft에서 구현한 것입니다. CLI는 언어와 라이브러리가 원활하게 함께 작동하는 실행 및 개발 환경을 만들기 위한 기초입니다.

C#으로 작성된 소스 코드는 CLI 사양을 준수하는 [IL\(중간 언어\)](#)로 컴파일됩니다. IL 코드와 리소스(예: 비트맵 및 문자열)는 일반적으로 `.dll*` 확장명과 함께 어셈블리에 저장됩니다. 어셈블리는 어셈블리의 형식, 버전 및 문화에 대한 정보를 제공하는 매니페스트를 포함합니다.

C# 프로그램이 실행되면 어셈블리가 CLR에 로드됩니다. CLR은 JIT(Just-In-Time) 컴파일을 수행하여 IL 코드를 네이티브 기계어 명령으로 변환합니다. CLR은 자동 가비지 수집, 예외 처리 및 리소스 관리와 관련된 다른 서비스도 제공합니다. CLR에서 실행되는 코드는 "관리 코드"라고도 합니다. 즉, 특정 플랫폼을 대상으로 하는 네이티브 기계어로 컴파일되는 "비관리 코드"와는 반대됩니다.

언어 상호 운용성은 .NET의 주요 기능입니다. C# 컴파일러에서 생성된 IL 코드는 CTS(공용 형식 사양)를 따릅니다. C#에서 생성된 IL 코드는 F#의 .NET 버전, Visual Basic, C++ 또는 20개가 넘는 다른 CTS 규격 언어에서 생성된 코드와 상호 작용할 수 있습니다. 단일 어셈블리는 다른 .NET 언어로 작성된 여러 모듈을 포함할 수 있고 형식은 마치 같은 언어로 작성된 것처럼 서로를 참조할 수 있습니다.

런타임 서비스 외에도 .NET에는 광범위한 라이브러리가 포함되어 있습니다. 이러한 라이브러리는 다양한 워크로드를 지원합니다. 이들은 파일 입/출력부터 문자열 조작, XML 구문 분석, 웹 애플리케이션 프레임워크, Windows Forms 컨트롤에 이르기까지 모든 항목에 대해 다양하고 유용한 기능을 제공하는 네임스페이스로 구

성됩니다. 전형적인 C# 애플리케이션은 .NET 클래스 라이브러리를 광범위하게 사용하여 일반적인 “배관” 작업을 처리합니다.

.NET에 대한 자세한 내용은 [.NET의 개요](#)를 참조하세요.

## Hello World

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

```
using System;

class Hello
{
    static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

“Hello, World” 프로그램은 `System` 네임스페이스를 참조하는 `using` 지시문으로 시작합니다. 네임스페이스는 계층적으로 C# 프로그램 및 라이브러리를 구성하는 방법을 제공합니다. 네임스페이스에는 형식 및 다른 네임스페이스가 포함됩니다. 예를 들어 `System` 네임스페이스에는 많은 형식(예: 프로그램에 참조되는 `Console` 클래스) 및 많은 다른 네임스페이스(예: `IO` 및 `Collections`)가 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.

“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. 관례상 `Main`이라는 정적 메서드가 C# 프로그램의 진입점으로 사용됩니다.

프로그램의 출력은 `System` 네임스페이스에 있는 `Console` 클래스의 `WriteLine` 메서드에 의해 생성됩니다. 이 클래스는 기본적으로 컴파일러에서 자동으로 참조되는 표준 클래스 라이브러리를 통해 제공됩니다.

## 형식 및 변수

C#에는 두 가지 종류의 형식, 즉 `값` 형식과 `참조` 형식이 있습니다. `값` 형식의 변수에는 해당 데이터가 직접 포함됩니다. 참조 형식의 변수에는 개체라고도 하는 데이터에 대한 참조가 저장됩니다. 참조 형식에서는 두 개의 변수가 같은 개체를 참조할 수 있고 한 변수에 대한 작업이 다른 변수에서 참조하는 개체에 영향을 미칠 수 있습니다. `값` 형식에서는 변수가 자체적으로 데이터 사본을 갖고 있으며 한 변수에 대한 작업이 다른 변수에 영향을 미칠 수 없습니다(`ref` 및 `out` 매개 변수 변수 제외).

‘식별자’는 변수 이름입니다.\* \_식별자는 공백이 없는 유니코드 문자 시퀀스입니다. 식별자에 `@` 접두사가 있으면 C# 예약어일 수 있습니다. 예약어를 식별자로 사용하면 다른 언어와 상호 작용할 때 유용할 수 있습니다.

C#의 `값` 형식은 ‘단순 형식’, ‘열거형 형식’, ‘구조체 형식’, ‘null 허용 값 형식’, ‘튜플 값 형식’으로 세분화됩니다.\* . C#의 참조 형식은 클래스 형식, 인터페이스 형식, 배열 형식 및 대리자 형식으로 세분화됩니다.

다음 개요는 C#의 형식 시스템에 대한 개요를 제공합니다.

### ● `값` 형식

#### ○ 단순 형식

- `부호 있는 정수`: `sbyte`, `short`, `int`, `long`
- `부호 없는 정수`: `byte`, `ushort`, `uint`, `ulong`
- `유니코드 문자`: `char` (UTF-16 코드 단위)

- IEEE 이진 부동 소수점: `float`, `double`
  - 고정밀 10진수 부동 소수점: `decimal`
  - 부울: `bool` 은 부울 값, 즉 `true` 또는 `false` 를 나타내는 값입니다.
  - 열거형 형식
    - `enum E {...}` 의 사용자 정의 형식입니다. `enum` 형식은 명명된 상수가 있는 고유한 형식입니다. 모든 `enum` 형식은 8가지 정수 형식 중 하나인 내부 형식을 갖습니다. `enum` 형식의 값 집합은 내부 형식의 값 집합과 동일합니다.
  - 구조체 형식
    - `struct S {...}` 양식의 사용자 정의 형식
  - Nullable 값 형식
    - `null` 값을 갖는 다른 모든 값 형식의 확장
  - 튜플 값 형식
    - `(T1, T2, ...)` 양식의 사용자 정의 형식
- 참조 형식
    - 클래스 형식
      - 다른 모든 형식의 기본 클래스: `object`
      - 유니코드 문자열: `string` (UTF-16 코드 유닛 시퀀스)
      - `class C {...}` 양식의 사용자 정의 형식
    - 인터페이스 형식
      - `interface I {...}` 양식의 사용자 정의 형식
    - 배열 형식
      - 1차원 배열, 다차원 배열, 가변 배열. 예: `int[]`, `int[,]`, `int[][]`
    - 대리자 형식
      - `delegate int D(...)` 양식의 사용자 정의 형식

C# 프로그램에서는 형식 선언을 사용하여 새 형식을 만듭니다. 형식 선언은 새 형식의 이름과 멤버를 지정합니다. 사용자 정의가 가능한 C#의 6가지 형식 범주는 클래스 형식, 구조체 형식, 인터페이스 형식, 열거형 형식, 대리자 형식, 튜플 값 형식입니다.

- `class` 형식은 데이터 멤버(필드) 및 함수 멤버(메서드, 속성 및 기타)를 포함하는 데이터 구조를 정의합니다. 클래스 형식은 단일 상속 및 다형성과 파생된 클래스가 기본 클래스를 확장하고 특수화할 수 있는 메커니즘을 지원합니다.
- `struct` 형식은 데이터 멤버 및 함수 멤버로 구조체를 나타내는 클래스 형식과 유사합니다. 그러나 클래스와 달리 구조체는 값 형식이며 일반적으로 힙 할당이 필요하지 않습니다. 구조체 형식은 사용자 지정 상속을 지원하지 않으며 모든 구조체 형식은 `object` 형식으로부터 암시적으로 상속됩니다.
- `interface` 형식은 계약을 공용 멤버의 명명된 집합으로 정의합니다. `interface` 를 구현하는 `class` 또는 `struct` 는 인터페이스의 멤버 구현을 제공해야 합니다. `interface` 는 여러 기본 인터페이스에서 상속될 수 있으며 `class` 또는 `struct` 는 여러 인터페이스를 구현할 수 있습니다.
- `delegate` 형식은 특정 매개 변수 목록 및 반환 형식이 있는 메서드에 대한 참조를 나타내는 형식입니다. 대리자는 메서드를 변수에 할당되고 매개 변수로 전달될 수 있는 엔터티로 취급할 수 있도록 합니다. 대리자는 함수 언어에서 제공하는 함수 형식과 유사합니다. 대리자는 다른 언어의 함수 포인터와 개념이 비슷하지만 함수 포인터와 달리 대리자는 개체 지향적이며 형식이 안전한 방식입니다.

`class`, `struct`, `interface` 및 `delegate` 형식은 모두 제네릭을 지원하므로 다른 형식으로 매개 변수화할 수 있습니다.

C#은 모든 형식의 1차원 및 다차원 배열을 지원합니다. 위에 나열된 형식과 달리, 배열 형식은 사용하기 전에 먼저 선언할 필요가 없습니다. 대신, 배열 형식은 형식 이름을 대괄호로 둑어 생성합니다. 예를 들어 `int[]` 는 `int` 의 1차원 배열이고, `int[,]` 는 `int` 의 2차원 배열, `int[][]` 는 `int` 의 1차원 배열의 1차원 배열, 즉 "가변" 배열입니다.

nullable 형식에는 별도의 정의가 필요하지 않습니다. null을 허용하지 않는 형식 `T`의 경우, 대응되는 nullable 형식 `T?` 가 있으며 이는 추가 값 `null` 을 가질 수 있습니다. 예를 들어 `int?` 는 32비트 정수 또는 `null` 값을 보유할 수 있는 형식이고, `string?` 은 모든 `string` 또는 `null` 값을 보유할 수 있는 형식입니다.

C#의 형식 시스템은 모든 형식의 값이 `object` 로 취급될 수 있도록 통합됩니다. C#의 모든 형식은 `object` 클래스 형식에서 직접 또는 간접적으로 파생되고 `object` 는 모든 형식의 기본 클래스입니다. 참조 형식의 값은 `object` 로 인식함으로써 간단히 개체로 처리됩니다. 값 형식의 값은 *boxing* 및 *unboxing* 작업을 수행하여 개체로 처리됩니다. 다음 예제에서 `int` 값은 `object` 로 변환되었다가 다시 `int` 로 변환됩니다.

```
int i = 123;
object o = i;    // Boxing
int j = (int)o; // Unboxing
```

값 형식의 값이 `object` 참조에 할당되면 값을 보유하기 위해 "box"가 할당됩니다. 이 상자는 참조 형식의 인스턴스이며 해당 상자에 값이 복사됩니다. 반대로 `object` 참조가 값 형식으로 캐스트될 때 참조된 `object` 가 올바른 값 형식의 상자인지 확인합니다. 확인에 성공하면 상자의 값이 값 형식에 복사됩니다.

C# 통합 형식 시스템은 결과적으로 값 형식이 "요청 시" `object` 참조로 처리됨을 의미합니다. 통합 때문에 `object` 형식을 사용하는 범용 라이브러리는 참조 형식과 값 형식을 모두 포함하여 `object` 에서 파생되는 모든 유형에 사용할 수 있습니다.

C#에는 필드, 배열 요소, 지역 변수 및 매개 변수를 포함하는 여러 종류의 변수가 있습니다. 변수는 저장소 위치를 나타냅니다. 모든 변수에는 아래와 같이 해당 변수에 저장할 수 있는 값을 결정하는 유형이 있습니다.

- Null을 허용하지 않는 값 형식
  - 정확한 해당 형식의 값
- Null 허용 값 형식
  - `null` 값 또는 정확한 해당 형식의 값
- object
  - `null` 참조, 참조 형식의 개체에 대한 참조 또는 값 형식의 boxed 값에 대한 참조
- 클래스 형식
  - `null` 참조, 해당 클래스 형식의 인스턴스에 대한 참조 또는 해당 클래스 형식에서 파생된 클래스의 인스턴스에 대한 참조
- 인터페이스 유형
  - `null` 참조, 해당 인터페이스 형식을 구현하는 클래스 형식의 인스턴스에 대한 참조 또는 해당 인터페이스 형식을 구현하는 값 형식의 boxed 값에 대한 참조
- 배열 형식
  - `null` 참조, 해당 배열 형식의 인스턴스에 대한 참조 또는 호환되는 배열 형식의 인스턴스에 대한 참조
- 대리자 형식
  - `null` 참조 또는 호환되는 대리자 형식의 인스턴스에 대한 참조

## 프로그램 구조

C#의 핵심 조직 개념은 '[프로그램](#)', '[네임스페이스](#)', '[형식](#)', '[멤버](#)', '[어셈블리](#)'입니다.\*\_ 프로그램은 멤버를 포함하고 네임스페이스로 구성될 수 있는 형식을 선언합니다. 클래스, 구조체 및 인터페이스는 형식의 예입니다. 필드, 메서드, 속성 및 이벤트는 멤버의 예입니다. C# 프로그램을 컴파일하면 실제로 어셈블리로 패키지됩니다. 어셈블리는 일반적으로 애플리케이션을 구현하는지 또는 라이브러리를 구현하는지에 따라 각각 파일 확장명 `.exe` 또는 `.dll` 을 갖습니다.

간단한 예로, 다음 코드를 포함하는 어셈블리를 생각해 보세요.

```

using System;

namespace Acme.Collections
{
    public class Stack<T>
    {
        Entry _top;

        public void Push(T data)
        {
            _top = new Entry(_top, data);
        }

        public T Pop()
        {
            if (_top == null)
            {
                throw new InvalidOperationException();
            }
            T result = _top.Data;
            _top = _top.Next;

            return result;
        }

        class Entry
        {
            public Entry Next { get; set; }
            public T Data { get; set; }

            public Entry(Entry next, T data)
            {
                Next = next;
                Data = data;
            }
        }
    }
}

```

이 클래스의 정규화된 이름은 `Acme.Collections.Stack`입니다. 클래스에는 필드 `top`, 2개의 메서드 `Push` 및 `Pop`, 중첩된 클래스 `Entry` 등의 여러 멤버가 포함됩니다. `Entry` 클래스는 필드 `next` 및 필드 `data`, 생성자의 세 멤버가 포함됩니다. `Stack`은 '제네릭' 클래스입니다.\_\* 이는 사용 시 구체적인 형식으로 대체되는 `T` 형식 매개 변수 하나를 포함합니다.

#### NOTE

스택은 "FILO"(선입후출) 컬렉션입니다. 스택의 맨 위에 새 요소가 추가됩니다. 요소가 제거되면 스택의 맨 위에서 제거됩니다.

어셈블리에는 IL(중간 언어) 명령 형식의 실행 코드와 메타데이터 형식의 기호 정보가 포함됩니다. 어셈블리가 실행되기 전에, .NET 공용 언어 런타임의 JIT(Just-In-Time) 컴파일러가 어셈블리 안의 IL 코드를 해당 프로세서에 맞는 코드로 변환합니다.

어셈블리는 코드와 메타데이터를 모두 포함하는 기능의 자체 설명 단위이므로 C#에서는 `#include` 지시문과 해더 파일이 필요하지 않습니다. 특정 어셈블리에 포함된 공용 형식 및 멤버는 프로그램을 컴파일할 때 해당 어셈블리를 참조하는 것만으로 C# 프로그램에서 사용 가능해집니다. 예를 들어 이 프로그램에서는 `acme.dll` 어셈블리의 `Acme.Collections.Stack` 클래스를 사용합니다.

```
using System;
using Acme.Collections;

class Example
{
    public static void Main()
    {
        var s = new Stack<int>();
        s.Push(1); // stack contains 1
        s.Push(10); // stack contains 1, 10
        s.Push(100); // stack contains 1, 10, 100
        Console.WriteLine(s.Pop()); // stack contains 1, 10
        Console.WriteLine(s.Pop()); // stack contains 1
        Console.WriteLine(s.Pop()); // stack is empty
    }
}
```

이 프로그램을 컴파일하려면 이전 예제에 정의된 스택 클래스를 포함하는 어셈블리를 참조해야 합니다.

C# 프로그램은 여러 원본 파일에 저장될 수 있습니다. C# 프로그램을 컴파일하면 모든 원본 파일이 함께 처리되고 서로를 제약 없이 참조할 수 있습니다. 개념적으로 처리되기 전에 모든 원본 파일이 하나의 대량 파일에 연결된 것과 같습니다. 소수의 경우를 제외하고 선언 순서는 중요하지 않으므로 C#에서는 정방향 선언이 필요한 경우가 없습니다. C#은 소스 파일을 하나의 공용 형식만 선언하도록 제한하거나 소스 파일 이름이 소스 파일에 선언된 형식과 일치하도록 요구하지 않습니다.

이 둘러보기의 추가 문서에서는 이러한 조직 구성 요소에 대해 설명합니다.

다음

# 형식 및 멤버

2021-02-18 • 18 minutes to read • [Edit Online](#)

개체 지향 언어인 C#은 캡슐화, 상속 및 다형성의 개념을 지원합니다. 클래스는 단일 부모 클래스에서 직접 상속될 수 있으며 원하는 수의 인터페이스를 구현할 수 있습니다. 부모 클래스에서 가상 메서드를 재정의하는 메서드에는 우발적인 재정의를 방지하는 방법으로 `override` 키워드가 필요합니다. C#에서 구조체는 간단한 클래스와 같습니다. 즉, 인터페이스를 구현할 수 있지만 상속을 지원하지 않는 스택 할당 형식입니다. C#은 또한 주로 데이터 값을 저장하는 용도로 사용할 수 있는 클래스 형식인 레코드를 제공합니다.

## 클래스 및 개체

클래스는 C#의 가장 기본적인 형식입니다. 클래스는 상태(필드)와 작업(메서드 및 기타 함수 멤버)을 하나의 단위로 결합하는 데이터 구조입니다. 클래스는 해당 클래스의 '인스턴스'('개체'라고도 함)에 대한 정의를 제공합니다. 클래스는 상속 및 다형성과 파생된 클래스가 기본 클래스를 확장하고 특수화할 수 있는 메커니즘을 지원합니다.

새 클래스는 클래스 선언을 사용하여 만들어집니다. 클래스 선언은 헤더로 시작합니다. 헤더는 다음을 지정합니다.

- 클래스의 특성 및 한정자
- 클래스의 이름
- 기본 클래스([기본 클래스](#)에서 상속하는 경우)
- 이 클래스에서 구현한 인터페이스

헤더 다음에는 구분 기호 `{` 및 `}` 간에 작성되는 멤버 선언 목록으로 구성되는 클래스 본문이 나옵니다.

다음 코드는 `Point`라는 간단한 클래스의 선언입니다.

```
public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);
}
```

클래스의 인스턴스는 새 인스턴스에 대한 메모리를 할당하고, 인스턴스를 초기화하는 생성자를 호출하고, 인스턴스에 대한 참조를 반환하는 `new` 연산자를 사용하여 만들어집니다. 다음 문은 두 개의 `Point` 개체를 만들고 해당 개체에 대한 참조를 두 변수에 저장합니다.

```
var p1 = new Point(0, 0);
var p2 = new Point(10, 20);
```

개체가 차지하는 메모리는 개체에 더 이상 연결할 수 없을 때 자동으로 회수됩니다. C#에서는 개체를 명시적으로 할당 취소할 필요가 없으며 가능하지도 않습니다.

### 형식 매개 변수

제네릭 클래스는 '[형식 매개 변수](#)'를 정의합니다.\* 형식 매개 변수는 대괄호로 묶인 형식 매개 변수 이름 목록입니다. 형식 매개 변수는 클래스 이름을 따릅니다. 그런 후 형식 매개 변수를 클래스 선언 본문에 사용하여 클래스의 멤버를 정의할 수 있습니다. 다음 예제에서 `Pair`의 형식 매개 변수는 `TFirst` 및 `TSecond`입니다.

```

public class Pair<TFirst, TSecond>
{
    public TFirst First { get; }
    public TSecond Second { get; }

    public Pair(TFirst first, TSecond second) =>
        (First, Second) = (first, second);
}

```

형식 매개 변수를 사용하도록 선언된 클래스 형식을 '제네릭 클래스 형식'이라고 합니다. 구조체, 인터페이스 및 대리자 형식도 제네릭일 수 있습니다. 제네릭 클래스를 사용하는 경우 각 형식 매개 변수에 대해 다음과 같은 형식 인수가 제공되어야 합니다.

```

var pair = new Pair<int, string>(1, "two");
int i = pair.First;      // TFirst int
string s = pair.Second; // TSecond string

```

위의 `Pair<int, string>` 과 같이 형식 인수가 제공된 제네릭 형식을 **생성된 형식**이라고 합니다.

### 기본 클래스

클래스 선언은 기본 클래스를 지정할 수 있습니다. 클래스 이름 및 형식 매개 변수 뒤에 콜론과 기본 클래스의 이름을 사용하면 됩니다. 기본 클래스 지정을 생략하면 `object` 형식에서 파생되는 클래스와 같습니다. 다음 예제에서 `Point3D`의 기본 클래스는 `Point`입니다. 첫 번째 예제에서 `Point`의 기본 클래스는 `object`입니다.

```

public class Point3D : Point
{
    public int Z { get; set; }

    public Point3D(int x, int y, int z) : base(x, y)
    {
        Z = z;
    }
}

```

클래스는 기본 클래스의 멤버를 상속합니다. 상속은 클래스가 기본 클래스의 거의 모든 멤버를 암시적으로 포함함을 의미합니다. 클래스는 인스턴스 및 정적 생성자와 종결자는 상속하지 않습니다. 파생 클래스는 해당 파생된 클래스를 상속하는 멤버에 새 멤버를 추가할 수 있지만 상속된 멤버의 정의를 제거할 수 없습니다. 앞의 예제에서 `Point3D`는 `Point`에서 `x` 및 `y` 멤버를 상속하고 모든 `Point3D` 인스턴스는 세 개의 속성, 즉 `x`, `y` 및 `z`를 포함합니다.

클래스 형식에서 해당 기본 클래스 형식 간에 암시적 변환이 존재합니다. 클래스 형식의 변수는 해당 클래스의 인스턴스 또는 모든 파생 클래스의 인스턴스를 참조할 수 있습니다. 예를 들어 이전 클래스 선언에서 형식 `Point`의 변수는 `Point` 또는 `Point3D`를 참조할 수 있습니다.

```

Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);

```

## 구조체

클래스는 상속과 다형성을 지원하는 형식을 정의합니다. 이를 통해 파생 클래스의 계층 구조를 기반으로 정교한 동작을 만들 수 있습니다. 이와 대조적으로 '**구조체**' 형식은 보다 단순한 형식으로, 기본 용도는 데이터 값을 저장하는 것입니다.\* 구조체는 기본 형식을 선언할 수 없으며, [System.ValueType](#)에서 암시적으로 파생됩니다.

`struct` 형식에서 다른 `struct` 형식을 파생할 수 없으며 암시적으로 봉인됩니다.

```

public struct Point
{
    public double X { get; }
    public double Y { get; }

    public Point(double x, double y) => (X, Y) = (x, y);
}

```

## 인터페이스

'인터페이스'는 클래스 및 구조체로 구현할 수 있는 계약을 정의합니다. 인터페이스는 메서드, 속성, 이벤트 및 인덱서를 포함할 수 있습니다. 인터페이스는 일반적으로 해당 인터페이스가 정의하는 멤버의 구현을 제공하지 않으며 단지 해당 인터페이스를 구현하는 클래스 또는 구조체에서 제공해야 하는 멤버를 지정합니다.

인터페이스는 '다중 상속'을 사용할 수 있습니다. 다음 예제에서 인터페이스 `IComboBox` 는 `ITextBox` 및 `IListBox` 를 둘 다 상속합니다.

```

interface IControl
{
    void Paint();
}

interface ITextBox : IControl
{
    void SetText(string text);
}

interface IListBox : IControl
{
    void SetItems(string[] items);
}

interface IComboBox : ITextBox, IListBox { }

```

클래스 및 구조체는 여러 인터페이스를 구현할 수 있습니다. 다음 예제에서 클래스 `EditBox` 는 `IControl` 및 `IDataBound` 를 둘 다 구현합니다.

```

interface IDataBound
{
    void Bind(Binder b);
}

public class EditBox : IControl, IDataBound
{
    public void Paint() { }
    public void Bind(Binder b) { }
}

```

클래스 또는 구조체가 특정 인터페이스를 구현하는 경우 해당 클래스 또는 구조체의 인스턴스를 해당 인터페이스 형식으로 암시적으로 변환할 수 있습니다. 예를 들면 다음과 같습니다.

```

EditBox editBox = new EditBox();
IControl control = editBox;
IDataBound dataBound = editBox;

```

## 열거형

'열거형' 형식은 상수 값 세트를 정의합니다. 다음 `enum` 은 여러 뿌리채소를 정의하는 상수를 선언합니다.

```
public enum SomeRootVegetable
{
    HorseRadish,
    Radish,
    Turnip
}
```

플래그와 함께 사용되도록 `enum` 을 정의할 수도 있습니다. 다음 선언은 사계절에 대한 플래그 세트를 선언합니다. 모든 계절을 포함하는 `All` 값을 비롯해 계절의 모든 조합을 적용할 수 있습니다.

```
[Flags]
public enum Seasons
{
    None = 0,
    Summer = 1,
    Autumn = 2,
    Winter = 4,
    Spring = 8,
    All = Summer | Autumn | Winter | Spring
}
```

다음 예제에서는 위 열거형의 선언을 보여 줍니다.

```
var turnip = SomeRootVegetable.Turnip;

var spring = Seasons.Spring;
var startingOnEquinox = Seasons.Spring | Seasons.Autumn;
var theYear = Seasons.All;
```

## Nullable 유형

모든 형식의 변수는 'null을 허용하지 않는' 또는 'null 허용'으로 선언할 수 있습니다. null 허용 변수는 값이 없음을 나타내는 추가 `null` 값을 포함할 수 있습니다. null 허용 값 형식(구조체 또는 열거형)은

`System.Nullable<T>`로 표현됩니다. null을 허용하지 않는 형식과 null 참조 형식은 모두 기본 참조 형식으로 표현됩니다. 둘 사이의 구분은 컴파일러와 일부 라이브러리에서 읽어들이는 메타데이터로 표현됩니다. 컴파일러는 null 참조가 먼저 `null`에 대해 값을 검사하지 않고 역참조될 경우 경고를 발생시킵니다. 컴파일러는 null을 허용하지 않는 참조에 `null` 일 수 있는 값을 할당하는 경우에도 경고를 표시합니다. 다음 예제에서는 'null 허용 int'를 선언하고 `null`로 초기화합니다. 그런 다음 값을 5로 설정합니다. 이 예제는 'null 허용 문자열'과 동일한 개념을 보여 줍니다. 자세한 내용은 [null 허용 값 형식](#) 및 [null 허용 참조 형식](#)을 참조하세요.

```
int? optionalInt = default;
optionalInt = 5;
string? optionalText = default;
optionalText = "Hello World.;"
```

## 튜플

C#은 간단한 데이터 구조로 여러 데이터 요소를 그룹화하는 간결한 구문을 제공하는 '[튜플](#)'을 지원합니다.\* 다음 예제에서 볼 수 있듯이 튜플은 ( 와 ) 사이에서 멤버의 형식과 이름을 선언함으로써 인스턴스화합니다.

```
(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
// Output:
// Sum of 3 elements is 4.5.
```

튜플은 다음 문서에서 설명하는 기본 구성 요소를 사용하지 않고도 여러 멤버를 위한 또 다른 데이터 구조를 제공합니다.

이전

다음

# 프로그램 구성 요소

2021-02-18 • 61 minutes to read • [Edit Online](#)

이전 문서에 설명된 형식은 \*멤버\*, \*식\*, \*문\*이라는 구성 요소를 사용하여 빌드됩니다.

## 멤버

`class`의 멤버는 정적 멤버 또는 인스턴스 멤버입니다. 정적 멤버는 클래스에 속하며 인스턴스 멤버는 개체(클래스의 인스턴스)에 속합니다.

다음은 클래스가 포함할 수 있는 멤버의 종류입니다.

- 상수: 클래스와 연결된 상수 값
- 필드: 클래스와 연결된 변수
- 메서드: 클래스가 수행할 수 있는 작업
- Properties: 클래스의 명명된 속성에 대한 읽기 및 쓰기와 관련된 작업
- 인덱서: 클래스 인스턴스를 배열처럼 인덱싱하는 것과 관련된 작업
- 이벤트: 클래스에 의해 생성될 수 있는 알림
- 연산자: 클래스가 지원하는 변환 및 식 연산자
- 생성자: 클래스의 인스턴스 또는 클래스 자체를 초기화하는 데 필요한 작업
- 종료자: 클래스의 인스턴스가 영구적으로 삭제되기 전에 수행한 작업
- 형식: 클래스에 의해 선언된 중첩 형식

## 접근성

클래스의 각 멤버에는 해당 멤버에 액세스할 수 있는 프로그램 텍스트의 영역을 제어하는 액세스 수준이 있습니다. 액세스 가능성은 여섯 가지 형태로 제공됩니다. 다음은 액세스 한정자입니다.

- `public`: 액세스가 제한되지 않음
- `private`: 이 클래스로만 액세스가 제한됨
- `protected`: 이 클래스 또는 이 클래스에서 파생된 클래스로만 액세스가 제한됨
- `internal`: 액세스가 현재 어셈블리(`.exe` 또는 `.dll`)로 제한됩니다.
- `protected internal`: 액세스가 이 클래스, 이 클래스에서 파생된 클래스 또는 동일한 어셈블리 내의 클래스로만 제한됩니다.
- `private protected`: 액세스가 이 클래스 또는 동일한 어셈블리 내의 이 형식에서 파생된 클래스로만 제한됩니다.

## 필드

필드는 클래스 또는 클래스의 인스턴스와 연결된 변수입니다.

`static` 한정자를 사용하여 선언된 필드는 정적 필드를 정의합니다. 정적 필드는 정확히 하나의 스토리지 위치를 식별합니다. 클래스의 인스턴스가 몇 개나 만들어졌는지에 관계없이 정적 필드의 복사본은 하나뿐입니다.

`static` 한정자 없이 선언된 필드는 인스턴스 필드를 정의합니다. 클래스의 모든 인스턴스는 해당 클래스의 모든 인스턴스 필드의 별도 복사본을 포함합니다.

다음 예제에서 `Color` 클래스의 각 인스턴스는 `R`, `G` 및 `B` 인스턴스 필드의 별도 복사본을 갖지만 `Black`, `White`, `Red`, `Green` 및 `Blue` 정적 필드의 복사본은 하나뿐입니다.

```

public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);

    public byte R;
    public byte G;
    public byte B;

    public Color(byte r, byte g, byte b)
    {
        R = r;
        G = g;
        B = b;
    }
}

```

앞의 예제와 같이 읽기 전용 필드는 `readonly` 한정자를 사용하여 선언될 수 있습니다. 읽기 전용 필드에 대한 할당은 필드 선언의 일부로 또는 동일한 클래스의 생성자에서만 발생할 수 있습니다.

## 메서드

메서드는 개체 또는 클래스에서 수행할 수 있는 계산이나 작업을 구현하는 멤버입니다. 정적 메서드는 클래스를 통해 액세스됩니다. 인스턴스 메서드는 클래스의 인스턴스를 통해 액세스됩니다.

메서드에는 메서드로 전달되는 값 또는 변수 참조를 나타내는 '매개 변수' 목록이 있을 수 있습니다. 메서드에는 메서드에 의해 계산되고 반환되는 값의 형식을 지정하는 '반환 형식'이 있습니다. 메서드가 값을 반환하지 않을 경우 반환 형식은 `void`입니다.

형식과 마찬가지로 메서드에는 메서드가 호출될 때 형식 인수가 지정되어야 하는 형식 매개 변수 집합도 있을 수 있습니다. 형식과 달리 형식 인수는 종종 메서드 호출의 인수에서 유추될 수 있으므로 명시적으로 지정할 필요가 없습니다.

메서드의 시그니처는 메서드가 선언되는 클래스에서 고유해야 합니다. 메서드 시그니처는 메서드의 이름, 형식 매개 변수의 수, 해당 매개 변수의 수, 한정자 및 형식으로 구성됩니다. 메서드 시그니처는 반환 형식을 포함하지 않습니다.

메서드 본문이 단일 식인 경우 메서드는 다음 예제와 같이 간결한 식 형식을 사용하여 정의할 수 있습니다.

```
public override string ToString() => "This is an object";
```

### 매개 변수

매개 변수는 메서드에 값 또는 변수 참조를 전달하는 데 사용됩니다. 메서드의 매개 변수는 메서드가 호출될 때 지정된 인수에서 실제 값을 가져옵니다. 매개 변수에는 값 매개 변수, 참조 매개 변수, 출력 매개 변수 및 매개 변수 배열의 네 가지 종류가 있습니다.

**값 매개 변수**는 입력 매개 변수를 전달하는 데 사용됩니다. 값 매개 변수는 매개 변수에 전달된 인수에서 초기 값을 가져오는 지역 변수에 해당합니다. 값 매개 변수를 수정해도 해당 매개 변수에 전달된 인수에는 영향을 주지 않습니다.

해당 인수를 생략할 수 있도록 기본값을 지정하면 값 매개 변수는 선택적일 수 있습니다.

**참조 매개 변수**는 인수를 참조로 전달하는 데 사용됩니다. 참조 매개 변수에 전달되는 인수는 한정된 값을 가진 변수여야 합니다. 메서드를 실행하는 동안 참조 매개 변수는 인수 변수와 동일한 스토리지 위치를 나타냅니다. 참조 매개 변수는 `ref` 한정자를 사용하여 선언됩니다. 다음 예제에서는 `ref` 매개 변수를 사용하는 방법을 보

여 줍니다.

```
static void Swap(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}

public static void SwapExample()
{
    int i = 1, j = 2;
    Swap(ref i, ref j);
    Console.WriteLine($"{i} {j}"); // "2 1"
}
```

출력 매개 변수는 인수를 참조로 전달하는 데 사용됩니다. 호출자가 제공한 인수에 값을 명시적으로 할당할 필요가 없다는 점을 제외하고 참조 매개 변수와 비슷합니다. 출력 매개 변수는 `out` 한정자를 사용하여 선언됩니다. 다음 예제에서는 C# 7에서 도입된 구문으로 `out` 매개 변수를 사용하는 방법을 보여 줍니다.

```
static void Divide(int x, int y, out int result, out int remainder)
{
    result = x / y;
    remainder = x % y;
}

public static void OutUsage()
{
    Divide(10, 3, out int res, out int rem);
    Console.WriteLine($"{res} {rem}"); // "3 1"
}
```

매개 변수 배열은 다양한 개수의 인수가 메서드에 전달되도록 허용합니다. 매개 변수 배열은 `params` 한정자를 사용하여 선언됩니다. 메서드의 마지막 매개 변수만 매개 변수 배열일 수 있으며 매개 변수 배열의 형식은 1차원 배열 형식이어야 합니다. `System.Console` 클래스의 `Write` 및 `WriteLine` 메서드는 매개 변수 배열 사용의 좋은 예입니다. 이러한 메서드는 다음과 같이 선언됩니다.

```
public class Console
{
    public static void Write(string fmt, params object[] args) { }
    public static void WriteLine(string fmt, params object[] args) { }
    // ...
}
```

매개 변수 배열을 사용하는 메서드 내에서 매개 변수 배열은 배열 형식의 일반 매개 변수와 정확히 동일하게 동작합니다. 그러나 매개 변수 배열을 사용한 메서드 호출에서 매개 변수 배열 형식의 단일 인수 또는 매개 변수 배열에 있는 임의 개수의 요소 형식 인수를 전달할 수 있습니다. 후자의 경우 지정된 인수를 사용하여 배열 인스턴스가 자동으로 만들어지고 초기화됩니다. 다음 예제는

```
int x, y, z;
x = 3;
y = 4;
z = 5;
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

다음을 작성하는 것과 같습니다.

```

int x = 3, y = 4, z = 5;

string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);

```

## 메서드 본문 및 지역 변수

메서드의 본문은 메서드가 호출될 때 실행할 문을 지정합니다.

메서드 본문은 메서드 호출과 관련된 변수를 선언할 수 있습니다. 이러한 변수를 **지역 변수**라고 합니다. 지역 변수 선언은 형식 이름, 변수 이름을 지정하며 초기 값을 지정할 수도 있습니다. 다음 예제에서는 초기 값이 0인 지역 변수 **i**와 초기 값이 없는 지역 변수 **j**를 선언합니다.

```

class Squares
{
    public static void WriteSquares()
    {
        int i = 0;
        int j;
        while (i < 10)
        {
            j = i * i;
            Console.WriteLine($"{i} x {i} = {j}");
            i = i + 1;
        }
    }
}

```

C#에서는 해당 값을 얻기 위해 먼저 로컬 변수를 명확하게 할당해야 합니다. 예를 들어 이전 **i**의 선언이 초기 값을 포함하지 않으면 컴파일러는 **i**의 후속 사용에 대해 오류를 보고합니다. **i**는 프로그램에서 해당 시점에 명확하게 할당되지 않은 것이기 때문입니다.

메서드는 **return** 문을 사용하여 해당 호출자에게 컨트롤을 반환할 수 있습니다. **void**를 반환하는 메서드에서 **return** 문은 식을 지정할 수 없습니다. void 이외의 값을 반환하는 메서드에서 **return** 문은 반환 값을 계산하는 식을 포함해야 합니다.

## 정적 및 인스턴스 메서드

**static** 한정자를 사용하여 선언된 메서드는 '정적 메서드'입니다. 정적 메서드는 특정 인스턴스에 작동하지 않고 정적 멤버에 직접적으로만 액세스할 수 있습니다.

**static** 한정자를 사용하지 않고 선언된 메서드는 '인스턴스 메서드'입니다. 인스턴스 메서드는 특정 인스턴스에 작동하며 정적 및 인스턴스 멤버 둘 다에 액세스할 수 있습니다. 인스턴스 메서드가 호출된 인스턴스는 **this**로 명시적으로 액세스할 수 있습니다. 정적 메서드에서 **this**를 참조하면 오류가 발생합니다.

다음 **Entity** 클래스에는 정적 멤버와 인스턴스 멤버가 모두 있습니다.

```

class Entity
{
    static int s_nextSerialNo;
    int _serialNo;

    public Entity()
    {
        _serialNo = s_nextSerialNo++;
    }

    public int GetSerialNo()
    {
        return _serialNo;
    }

    public static int GetNextSerialNo()
    {
        return s_nextSerialNo;
    }

    public static void SetNextSerialNo(int value)
    {
        s_nextSerialNo = value;
    }
}

```

각 `Entity` 인스턴스에는 일련 번호(및 여기에 표시되지 않는 몇몇 정보)가 포함되어 있습니다. `Entity` 생성자(인스턴스 메서드와 유사함)는 사용 가능한 다음 일련 번호를 사용하여 새 인스턴스를 초기화합니다. 생성자가 인스턴스 멤버이기 때문에 `_serialNo` 인스턴스 필드 및 `s_nextSerialNo` 정적 필드 둘 다에 액세스하도록 허용됩니다.

`GetNextSerialNo` 및 `SetNextSerialNo` 정적 메서드는 `s_nextSerialNo` 정적 필드에 액세스할 수 있지만 `_serialNo` 인스턴스 필드에 직접 액세스하면 오류가 발생합니다.

다음 예제에서는 사용 된 `Entity` 클래스입니다.

```

Entity.SetNextSerialNo(1000);
Entity e1 = new Entity();
Entity e2 = new Entity();
Console.WriteLine(e1.GetSerialNo());           // Outputs "1000"
Console.WriteLine(e2.GetSerialNo());           // Outputs "1001"
Console.WriteLine(Entity.GetNextSerialNo());   // Outputs "1002"

```

`SetNextSerialNo` 및 `GetNextSerialNo` 정적 메서드는 클래스에 대해 호출되지만 `GetSerialNo` 인스턴스 메서드는 클래스의 인스턴스에 대해 호출됩니다.

### 가상, 재정의 및 추상 메서드

인스턴스 메서드 선언에 `virtual` 한정자가 포함되면 해당 메서드를 **가상 메서드**라고 합니다. 가상 한정자가 없으면 해당 메서드를 **비가상 메서드**라고 합니다.

가상 메서드가 호출되면 호출이 발생하는 인스턴스의 **런타임 형식**에 따라 호출할 실제 메서드 구현이 결정됩니다. 비가상 메서드 호출에서는 인스턴스의 **컴파일 타입 형식**이 결정 요인입니다.

가상 메서드는 파생된 클래스에서 **재정의**될 수 있습니다. 인스턴스 메서드 선언에 재정의 한정자가 포함될 경우 메서드는 동일한 시그니처로 상속된 가상 메서드를 재정의합니다. 가상 메서드 선언은 새 메서드를 도입합니다. 재정의 메서드 선언은 해당 메서드의 새 구현을 제공하여 기존의 상속된 가상 메서드를 특수화합니다.

**추상 메서드**는 구현이 없는 가상 메서드입니다. 추상 메서드는 `abstract` 한정자를 사용하여 선언되며 추상 클래스에서만 허용됩니다. 추상 메서드는 모든 비추상 파생 클래스에서 재정의해야 합니다.

다음 예제에서는 식 트리 노드를 나타내는 추상 클래스 `Expression` 와 상수, 변수 참조 및 산술 연산에 대한 식 트리 노드를 구현하는 세 개의 파생 클래스 `Constant` , `VariableReference` 및 `Operation` 을 선언합니다. (이 예제는 식 트리 형식과 비슷하지만 관련은 없습니다.)

```

public abstract class Expression
{
    public abstract double Evaluate(Dictionary<string, object> vars);
}

public class Constant : Expression
{
    double _value;

    public Constant(double value)
    {
        _value = value;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        return _value;
    }
}

public class VariableReference : Expression
{
    string _name;

    public VariableReference(string name)
    {
        _name = name;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        object value = vars[_name] ?? throw new Exception($"Unknown variable: {_name}");
        return Convert.ToDouble(value);
    }
}

public class Operation : Expression
{
    Expression _left;
    char _op;
    Expression _right;

    public Operation(Expression left, char op, Expression right)
    {
        _left = left;
        _op = op;
        _right = right;
    }

    public override double Evaluate(Dictionary<string, object> vars)
    {
        double x = _left.Evaluate(vars);
        double y = _right.Evaluate(vars);
        switch (_op)
        {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;

            default: throw new Exception("Unknown operator");
        }
    }
}

```

이전의 4개 클래스는 산술 연산자를 모델링하는 데 사용할 수 있습니다. 예를 들어 이러한 클래스의 인스턴스를

사용할 경우 식 `x + 3` 을 다음과 같이 나타낼 수 있습니다.

```
Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));
```

`Expression` 인스턴스의 `Evaluate` 메서드는 지정된 식을 계산하고 `double` 값을 생성하기 위해 호출됩니다. 이 메서드는 변수 이름(항목의 키)과 값(항목의 값)을 포함하는 `Dictionary` 인수를 사용합니다. `Evaluate` 가 추상 메서드이기 때문에 `Expression`에서 파생된 비추상 클래스는 `Evaluate` 를 재정의해야 합니다.

`Evaluate` 의 `Constant` 구현은 단순히 저장된 상수를 반환합니다. `VariableReference` 의 구현은 사전에서 변수 이름을 조회하고 결과 값을 반환합니다. `Operation` 의 구현은 먼저 왼쪽 및 오른쪽 피연산자를 계산하고(재귀적 으로 해당 `Evaluate` 메서드 호출) 지정된 산술 연산을 수행합니다.

다음 프로그램에서는 `Expression` 클래스를 사용하여 `x` 및 `y`의 다른 값에 대해 식 `x * (y + 2)` 를 계산합니다.

```
Expression e = new Operation(
    new VariableReference("x"),
    '*',
    new Operation(
        new VariableReference("y"),
        '+',
        new Constant(2)
    )
);
Dictionary<string, object> vars = new Dictionary<string, object>();
vars["x"] = 3;
vars["y"] = 5;
Console.WriteLine(e.Evaluate(vars)); // "21"
vars["x"] = 1.5;
vars["y"] = 9;
Console.WriteLine(e.Evaluate(vars)); // "16.5"
```

## 메서드 오버로드

메서드 오버로드는 동일한 클래스가 고유한 시그니처를 갖는 한, 동일한 이름을 갖도록 허용합니다. 오버로드 된 메서드의 호출을 컴파일할 때 컴파일러는 오버로드 확인을 사용하여 호출할 특정 메서드를 결정합니다. 오버로드 확인은 인수와 가장 적합하게 일치하는 단일 메서드를 찾습니다. 최상의 일치 메서드를 찾을 수 있는 경우 오류를 보고합니다. 다음 예제에서는 실제로 진행되는 오버로드 확인을 보여 줍니다. `UsageExample` 메서드의 각 호출에 대한 주석은 어느 메서드가 호출되었는지 보여 줍니다.

```

class OverloadingExample
{
    static void F() => Console.WriteLine("F()");
    static void F(object x) => Console.WriteLine("F(object)");
    static void F(int x) => Console.WriteLine("F(int)");
    static void F(double x) => Console.WriteLine("F(double)");
    static void F<T>(T x) => Console.WriteLine("F<T>(T)");
    static void F(double x, double y) => Console.WriteLine("F(double, double)");

    public static void UsageExample()
    {
        F();                // Invokes F()
        F(1);              // Invokes F(int)
        F(1.0);            // Invokes F(double)
        F("abc");          // Invokes F<string>(string)
        F((double)1);      // Invokes F(double)
        F((object)1);      // Invokes F(object)
        F<int>(1);         // Invokes F<int>(int)
        F(1, 1);           // Invokes F(double, double)
    }
}

```

예제와 같이, 인수를 정확한 매개 변수 형식과 형식 인수로 명시적으로 캐스팅하여 항상 특정 메서드를 선택할 수 있습니다.

## 기타 함수 멤버

실행 코드를 포함하는 멤버를 통칭하여 클래스의 **함수 멤버**라고 합니다. 이전 섹션에서는 함수 멤버의 기본 종류인 메서드에 대해 설명했습니다. 이 섹션에서는 C#에서 지원하는 다른 종류의 함수 멤버인 생성자, 속성, 인덱서, 이벤트, 연산자 및 종료자에 대해 설명합니다.

다음 예제에서는 늘어날 수 있는 개체 목록을 구현하는 `MyList<T>`라는 제네릭 클래스를 보여 줍니다. 이 클래스는 함수 멤버의 가장 일반적인 몇 가지 예제를 포함합니다.

```

public class MyList<T>
{
    const int DefaultCapacity = 4;

    T[] _items;
    int _count;

    public MyList(int capacity = DefaultCapacity)
    {
        _items = new T[capacity];
    }

    public int Count => _count;

    public int Capacity
    {
        get => _items.Length;
        set
        {
            if (value < _count) value = _count;
            if (value != _items.Length)
            {
                T[] newItems = new T[value];
                Array.Copy(_items, 0, newItems, 0, _count);
                _items = newItems;
            }
        }
    }

    public T this[int index]
    {
        get => _items[index];
        set
        {
            if (index < 0 || index >= _count)
                throw new ArgumentOutOfRangeException("index");
            _items[index] = value;
        }
    }
}

```

```

public T this[int index]
{
    get => _items[index];
    set
    {
        _items[index] = value;
        OnChanged();
    }
}

public void Add(T item)
{
    if (_count == Capacity) Capacity = _count * 2;
    _items[_count] = item;
    _count++;
    OnChanged();
}
protected virtual void OnChanged() =>
    Changed?.Invoke(this, EventArgs.Empty);

public override bool Equals(object other) =>
    Equals(this, other as MyList<T>);

static bool Equals(MyList<T> a, MyList<T> b)
{
    if (Object.ReferenceEquals(a, null)) return Object.ReferenceEquals(b, null);
    if (Object.ReferenceEquals(b, null) || a._count != b._count)
        return false;
    for (int i = 0; i < a._count; i++)
    {
        if (!object.Equals(a._items[i], b._items[i]))
        {
            return false;
        }
    }
    return true;
}

public event EventHandler Changed;

public static bool operator ==(MyList<T> a, MyList<T> b) =>
    Equals(a, b);

public static bool operator !=(MyList<T> a, MyList<T> b) =>
    !Equals(a, b);
}

```

## 생성자

C#은 인스턴스 및 정적 생성자를 모두 지원합니다. **인스턴스 생성자**는 클래스의 인스턴스를 초기화하는 데 필요한 작업을 구현하는 멤버입니다. '정적 생성자'는 처음 로드될 때 클래스 자체를 초기화하는 데 필요한 동작을 구현하는 멤버입니다.

생성자는 반환 형식이 없고 포함하는 클래스와 동일한 이름을 갖는 메서드처럼 선언됩니다. 생성자 선언에 **static** 한정자가 포함될 경우 정적 생성자를 선언합니다. 그렇지 않으면 인스턴스 생성자를 선언합니다.

인스턴스 생성자는 오버로드될 수 있으며 선택적 매개 변수를 가질 수 있습니다. 예를 들어 **MyList<T>** 클래스는 단일 선택적 **int** 매개 변수를 사용하여 하나의 인스턴스 생성자를 선언합니다. 인스턴스 생성자는 **new** 연산자를 사용하여 호출됩니다. 다음 문은 각각 선택적 인수를 사용하거나 사용하지 않고 **MyList<string>** 클래스의 생성자를 사용하여 2가지 **MyList** 인스턴스를 할당합니다.

```

MyList<string> list1 = new MyList<string>();
MyList<string> list2 = new MyList<string>(10);

```

다른 멤버와 달리 인스턴스 생성자는 상속되지 않습니다. 클래스에는 클래스에서 실제로 선언된 인스턴스 생성자 외에는 인스턴스 생성자가 없습니다. 클래스에 대해 인스턴스 생성자가 제공되지 않으면 매개 변수가 없는 빈 인스턴스 생성자가 자동으로 제공됩니다.

## 속성

속성은 필드의 기본 확장입니다. 둘 다 연결된 형식으로 명명되는 멤버이며, 필드 및 속성에 액세스하는 구문은 동일합니다. 그러나 필드와 달리 속성은 스토리지 위치를 명시하지 않습니다. 대신, 속성에는 해당 값을 읽거나 쓸 때 실행될 문을 지정하는 ‘접근자’가 있습니다.

속성은 필드처럼 선언되지만, 선언이 세미콜론으로 끝나지 않고, 구분 기호 `{` 및 `}` 사이에 쓴 `get` 접근자 또는 `set` 접근자로 끝난다는 점이 다릅니다. `get` 접근자와 `set` 접근자가 모두 있는 속성은 ‘읽기/쓰기 속성’입니다. `get` 접근자만 있는 속성은 ‘읽기 전용 속성’입니다. `set` 접근자만 있는 속성은 ‘쓰기 전용 속성’입니다.

`get` 접근자는 속성 형식의 반환 값을 갖는 매개 변수 없는 메서드에 해당합니다. `set` 접근자는 `value`라는 단일 매개 변수를 가지며 반환 형식이 없는 메서드에 해당합니다. `get` 접근자는 속성의 값을 계산합니다. `set` 접근자는 속성에 새 값을 제공합니다. 속성이 할당의 대상이거나 `++` 또는 `--`의 피연산자인 경우 `set` 접근자가 호출됩니다. 속성이 참조되는 그 밖의 경우에는 `get` 접근자가 호출됩니다.

`MyList<T>` 클래스는 각각 읽기 전용 및 읽기/쓰기 특성을 갖는 두 개의 속성 `Count` 및 `Capacity`를 선언합니다. 다음 코드는 이러한 속성을 사용하는 예입니다.

```
MyList<string> names = new MyList<string>();
names.Capacity = 100; // Invokes set accessor
int i = names.Count; // Invokes get accessor
int j = names.Capacity; // Invokes get accessor
```

필드 및 메서드와 마찬가지로, C#은 인스턴스 속성 및 정적 속성을 모두 지원합니다. 정적 속성은 `static` 한정자를 사용하여 선언되고 인스턴스 속성은 이 한정자를 사용하지 않고 선언됩니다.

속성의 접근자는 가상일 수 있습니다. 속성 선언에 `virtual`, `abstract`, 또는 `override` 한정자가 포함되면 속성의 접근자에 적용됩니다.

## 인덱서

인덱서는 개체가 배열과 같은 방식으로 인덱싱될 수 있도록 하는 멤버입니다. 인덱서는 `this` 과(와) 구분 기호 `[` 및 `]` 사이에 작성된 매개 변수 목록을 합쳐서 구성원 이름으로 사용한다는 점을 제외하고 속성처럼 선언됩니다. 매개 변수는 인덱서의 접근자에서 사용할 수 있습니다. 속성과 마찬가지로 인덱서는 읽기/쓰기, 읽기 전용 및 쓰기 전용일 수 있으며 인덱서의 접근자는 가상일 수 있습니다.

`MyList<T>` 클래스는 `int` 매개 변수를 사용하는 단일 읽기/쓰기 인덱서를 선언합니다. 인덱서는 `MyList<T>` 인스턴스를 `int` 값으로 인덱싱할 수 있도록 합니다. 예를 들어 다음과 같은 가치를 제공해야 합니다.

```
MyList<string> names = new MyList<string>();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++)
{
    string s = names[i];
    names[i] = s.ToUpper();
}
```

인덱서는 오버로드될 수 있습니다. 하나의 클래스는 매개 변수의 수와 형식이 다른 여러 인덱서를 선언할 수 있습니다.

## 이벤트

이벤트는 클래스 또는 개체가 알림을 제공할 수 있도록 하는 멤버입니다. 이벤트는 선언에 `event` 키워드가 포함됩니다.

함되고 형식이 대리자 형식이어야 한다는 점을 제외하고 필드처럼 선언됩니다.

이벤트 멤버를 선언하는 클래스 내에서 이벤트는 대리자 형식의 필드처럼 동작합니다(이벤트가 추상이 아니고 접근자를 선언하지 않을 경우). 필드는 이벤트에 추가된 이벤트 처리기를 나타내는 대리자에 대한 참조를 저장합니다. 이벤트 처리기가 없는 경우 필드는 `null`입니다.

`MyList<T>` 클래스는 `Changed`라는 단일 이벤트 멤버를 선언합니다. 이것은 새 항목이 목록에 추가되었음을 나타냅니다. `Changed` 이벤트는 `OnChanged` 가상 메서드에 의해 발생합니다. 이 메서드는 먼저 이벤트가 `null`인지 확인합니다(처리기가 없음을 의미함). 이벤트 발생 개념은 이벤트가 나타내는 대리자를 호출하는 것과 정확히 동일하므로 이벤트 발생을 위한 특수한 언어 구문은 없습니다.

클라이언트는 `이벤트 처리기`를 통해 이벤트에 반응합니다. 이벤트 처리기는 `+=` 연산자를 사용하여 추가되고, `-=` 연산자를 사용하여 제거됩니다. 다음 예제에서는 이벤트 처리기를 `MyList<string>`의 `Changed` 이벤트에 추가합니다.

```
class EventExample
{
    static int s_changeCount;

    static void ListChanged(object sender, EventArgs e)
    {
        s_changeCount++;
    }

    public static void Usage()
    {
        var names = new MyList<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(s_changeCount); // "3"
    }
}
```

이벤트의 기본 스토리지를 제어하려고 하는 고급 시나리오의 경우 이벤트 선언에서 속성의 `set` 접근자와 비슷한 `add` 및 `remove` 접근자를 명시적으로 제공할 수 있습니다.

### 연산자

연산자는 클래스 인스턴스에 특정 식 연산자를 적용하는 것의 의미를 정의하는 멤버입니다. 세 가지 종류의 연산자, 즉, 단항 연산자, 이항 연산자 및 변환 연산자를 정의할 수 있습니다. 모든 연산자는 `public` 및 `static`으로 선언해야 합니다.

`MyList<T>` 클래스는 두 개의 연산자 `operator ==` 및 `operator !=`를 선언합니다. 이들 재정의된 연산자는 해당 연산자를 `MyList` 인스턴스에 적용하는 식에 새로운 의미를 부여합니다. 특히, 이러한 연산자는 해당 `Equals` 메서드를 사용하여 포함된 각 개체를 비교할 때 두 `MyList<T>` 인스턴스의 같음을 정의합니다. 다음 예제에서는 `==` 연산자를 사용하여 두 `MyList<int>` 인스턴스를 비교합니다.

```
MyList<int> a = new MyList<int>();
a.Add(1);
a.Add(2);
MyList<int> b = new MyList<int>();
b.Add(1);
b.Add(2);
Console.WriteLine(a == b); // Outputs "True"
b.Add(3);
Console.WriteLine(a == b); // Outputs "False"
```

두 목록은 같은 순서로 같은 값을 갖는 동일한 수의 개체를 포함하므로 첫 번째 `Console.WriteLine`은 `True`를 출

력합니다. `MyList<T>`에서 `operator ==`이 정의되지 않았으면 `a` 및 `b`은 다른 `MyList<int>` 인스턴스를 참조하므로 첫 번째 `Console.WriteLine`은 `False`를 출력합니다.

## 종료자

종료자는 클래스의 인스턴스를 종결하는 데 필요한 작업을 구현하는 멤버입니다. 종료자는 일반적으로 관리되지 않는 리소스를 해제하는 데 필요합니다. 종료자는 매개 변수를 가질 수 없고, 액세스 수준 한정자를 가질 수 없으며, 명시적으로 호출할 수 없습니다. 인스턴스에 대한 종료자는 가비지 수집 중에 자동으로 호출됩니다. 자세한 내용은 [종료자](#) 문서를 참조하세요.

가비지 수집기는 개체를 수집하고 종료자를 실행할 시기를 유연하게 결정할 수 있도록 합니다. 특히, 종료자 호출 타이밍은 결정적이 아니며, 모든 스레드에서 종료자를 실행할 수 있습니다. 이러한 이유 및 기타 이유로 클래스는 가능한 다른 솔루션이 없을 때만 종료자를 구현해야 합니다.

`using` 문은 개체 소멸을 위한 더 나은 방법을 제공합니다.

## 표현식

식은 피연산자 및 연산자로 생성됩니다. 식의 연산자는 피연산자에 적용할 연산을 나타냅니다. 연산자의 예로 `+`, `-`, `*`, `/` 및 `new`가 있습니다. 피연산자의 예로는 리터럴, 필드, 지역 변수 및 식이 있습니다.

식에 여러 연산자가 포함된 경우 연산자의 '우선 순위'는 개별 연산자가 평가되는 순서를 제어합니다. 예를 들어 `*` 연산자는 `+` 연산자보다 우선 순위가 더 높기 때문에 식 `x + y * z`는 `x + (y * z)`로 계산됩니다.

피연산자는 동일한 우선 순위를 가진 두 연산자 사이에 나올 경우 연산자의 결합성에 따라 연산이 수행되는 순서가 제어됩니다.

- 대입 및 Null 병합 연산자를 제외하고 모든 이항 연산자는 왼쪽 결합성을 갖습니다. 즉, 연산이 왼쪽에서 오른쪽으로 수행됩니다. 예를 들어, `x + y + z`는 `(x + y) + z`로 계산됩니다.
- 대입 연산자, Null 병합 `??` 및 `??=` 연산자, 조건부 연산자(`? :`)는 오른쪽 결합성을 갖습니다. 즉, 연산이 오른쪽에서 왼쪽으로 수행됩니다. 예를 들어, `x = y = z`는 `x = (y = z)`로 계산됩니다.

우선 순위 및 결합성은 괄호를 사용하여 제어할 수 있습니다. 예를 들어 `x + y * z`는 먼저 `y`와 `z`를 곱한 다음 그 결과를 `x`와 더하지만 `(x + y) * z`는 먼저 `x`와 `y`를 더한 다음 그 결과에 `z`를 곱합니다.

대부분의 연산자는 [오버로드](#)할 수 있습니다. 연산자 오버로드는 피연산자 중 하나 또는 둘 다가 사용자 정의 클래스 또는 구조체 형식인 연산에 대해 사용자 정의 연산자 구현을 지정할 수 있도록 허용합니다.

C#은 [산술](#), [논리](#), [비트](#) 및 [시프트](#) 연산과 [같음](#) 및 [순서](#) 비교를 수행하는 여러 연산자를 제공합니다.

우선 순위 수준에 따라 정렬된 전체 연산자 목록은 [C# 연산자](#)를 참조하세요.

## 문

프로그램의 동작은 문을 사용하여 표현됩니다. C#은 여러 다른 종류의 문을 지원하며 이중 많은 문이 포함 문에 대해 정의됩니다.

- 블록은 단일 문이 허용되는 컨텍스트에서 여러 문을 쓸 수 있도록 허용합니다. 블록은 구분 기호 `{`와 `}` 사이에 쓴 문 목록으로 구성됩니다.
- 선언 문은 지역 변수 및 상수를 선언하는 데 사용됩니다.
- 식 문은 식을 평가하는 데 사용됩니다. 문으로 사용할 수 있는 식에는 메서드 호출, `new` 연산자를 사용하는 개체 할당, `=` 및 복합 할당 연산자를 사용하는 대입, `++` 및 `--` 연산자를 사용하는 증가 및 감소 연산, `await` 식이 포함됩니다.
- 선택 문은 일부 식 값에 따라 실행할 수 있는 다양한 문 중에서 하나를 선택하는 데 사용됩니다. 이 그룹에는 `if` 문과 `switch` 문이 포함됩니다.
- 반복 문은 포함 문을 반복해서 실행하는 데 사용됩니다. 이 그룹에는 `while` 문, `do` 문, `for` 문과 `foreach` 문이 포함됩니다.

- 점프 문은 제어를 전달하는 데 사용됩니다. 이 그룹에는 `break` 문, `continue` 문, `goto` 문, `throw` 문, `return` 문과 `yield` 문이 포함됩니다.
- `try ... catch` 문은 블록 실행 중에 발생하는 예외를 `catch`하는 데 사용되고 `try ... finally` 문은 예외 발생 여부에 관계 없이 항상 실행되는 종료 코드를 지정하는 데 사용됩니다.
- `checked` 및 `unchecked` 문은 정수 계열 형식 산술 연산 및 변환에 대한 오버플로 검사 컨텍스트를 제어하는 데 사용됩니다.
- `lock` 문은 지정된 개체에 대한 상호 배타적 잠금을 획득하고, 문을 실행한 후 잠금을 해제하는 데 사용됩니다.
- `using` 문은 리소스를 획득하고, 문을 실행한 후 해당 리소스를 삭제하는 데 사용됩니다.

다음은 사용할 수 있는 문 종류입니다.

- 지역 변수 선언
- 지역 상수 선언
- 식 문
- `if` 문
- `switch` 문
- `while` 문
- `do` 문
- `for` 문
- `foreach` 문
- `break` 문
- `continue` 문
- `goto` 문
- `return` 문
- `yield` 문
- `throw` 문과 `try` 문
- `checked` 및 `unchecked` 문
- `lock` 문
- `using` 문

[이전](#) [다음](#)

# 주 언어 영역

2021-02-18 • 23 minutes to read • [Edit Online](#)

## 배열, 컬렉션 및 LINQ

C#과 .NET은 여러 컬렉션 형식을 제공합니다. 배열에는 언어에 의해 정의된 구문이 있습니다. 제네릭 컬렉션 형식은 [System.Collections.Generic](#) 네임스페이스에 나열되어 있습니다. 특수 컬렉션에는 스택 프레임에서 연속 메모리에 액세스하기 위한 [System.Span<T>](#)와 관리되는 힙에서 연속 메모리에 액세스하기 위한 [System.Memory<T>](#)가 있습니다. 배열, [Span<T>](#), [Memory<T>](#)를 포함하는 모든 컬렉션은 반복을 위한 통일된 원칙을 공유합니다. 사용자는 [System.Collections.Generic.IEnumerable<T>](#) 인터페이스를 사용합니다. 이 통일된 원칙은 모든 컬렉션 형식을 LINQ 쿼리 또는 기타 알고리즘과 함께 사용할 수 있음을 의미합니다. 메서드는 [IEnumerable<T>](#)를 사용하여 작성하며, 이러한 알고리즘은 모든 컬렉션에서 작동합니다.

### 배열

**배열**은 계산된 인덱스를 통해 액세스되는 여러 변수를 포함하는 데이터 구조입니다.\* 배열에 포함된 변수, 즉 배열의 요소라고도 하는 배열은 모두 같은 형식이며, 이 형식을 배열의 요소 형식이라고 합니다.

배열 형식은 참조 형식이고 배열 변수의 선언은 배열 인스턴스에 대한 참조를 위한 공간을 설정합니다. 실제 배열 인스턴스는 `new` 연산자를 사용하여 런타임에 동적으로 만들어집니다. `new` 연산은 새 배열 인스턴스의 길이(인스턴스 수명 동안 고정됨)를 지정합니다. 배열 요소의 인덱스 범위는 `0`에서 `Length - 1` 사이입니다. `new` 연산자는 배열의 요소를 모든 숫자 형식에 대해 0이고, 모든 참조 형식에 대해 `null`인 기본값으로 자동으로 초기화합니다.

다음 예제에서는 `int` 요소의 배열을 만들고 배열을 초기화하고 배열의 콘텐츠를 출력합니다.

```
int[] a = new int[10];
for (int i = 0; i < a.Length; i++)
{
    a[i] = i * i;
}
for (int i = 0; i < a.Length; i++)
{
    Console.WriteLine($"a[{i}] = {a[i]}");
}
```

이 예제에서는 1차원 배열을 만들고 작업을 수행합니다. C#에서는 다차원 배열도 지원합니다. 배열 형식의 '순위'라고도 하는 배열 형식의 차원 수는 배열 형식의 대괄호 사이에 사용된 쉼표 수에 1을 더한 값입니다. 다음 예제에서는 1차원, 2차원 및 3차원 배열을 각각 할당합니다.

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

`a1` 배열에는 10개의 요소가 들어 있고 `a2` 배열에는 50( $10 \times 5$ )개의 요소가 들어 있고 `a3` 배열에는 100( $10 \times 5 \times 2$ )개 요소가 들어 있습니다. 배열의 요소 형식은 배열 형식을 비롯한 어떤 형식도 될 수 있습니다. 배열 형식의 요소가 있는 배열을 가변 배열이라고도 합니다. 요소 배열의 길이가 항상 동일할 필요는 없기 때문입니다. 다음 예제에서는 `int` 배열의 배열을 할당합니다.

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

첫 번째 줄은 형식이 `int[]`이고 초기 값이 `null`인 3개 요소가 있는 배열을 만듭니다. 다음 줄은 가변 길이의 개별 배열 인스턴스에 대한 참조로 3개 요소를 초기화합니다.

`new` 연산자는 구분 기호 `{` 와 `}` 사이에 쓰인 식 목록인 배열 이니셜라이저를 사용하여 배열 요소의 초기 값이 지정되도록 허용합니다. 다음 예제에서는 3개 요소로 `int[]`를 할당하고 초기화합니다.

```
int[] a = new int[] { 1, 2, 3 };
```

배열의 길이는 `{`와 `}` 사이에 있는 식의 개수에서 유추됩니다. 배열 형식을 다시 시작할 필요가 없도록 배열 초기화를 더 줄일 수 있습니다.

```
int[] a = { 1, 2, 3 };
```

앞의 두 예제는 다음 코드와 동일합니다.

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

`foreach` 문은 컬렉션의 요소를 열거하는 데 사용할 수 있습니다. 다음 코드는 앞에 나온 예제의 배열을 열거합니다.

```
foreach (int item in a)
{
    Console.WriteLine(item);
}
```

`foreach` 문은 `IEnumerable<T>` 인터페이스를 사용하므로 모든 컬렉션에서 사용할 수 있습니다.

## 문자열 보간

C# [문자열 보간](#)을 사용하면 결과가 서식 문자열에 배치되는 식을 정의하여 문자열에 서식을 지정할 수 있습니다. 예를 들어 다음 예제에서는 날씨 데이터 세트에서 지정된 날짜의 온도를 출력합니다.

```
Console.WriteLine($"The low and high temperature on {weatherData.Date:MM-DD-YYYY}");
Console.WriteLine($"    was {weatherData.LowTemp} and {weatherData.HighTemp}.");
// Output (similar to):
// The low and high temperature on 08-11-2020
//     was 5 and 30.
```

보간된 문자열은 `$` 토큰을 사용하여 선언됩니다. 문자열 보간은 `{` 와 `}` 사이의 식을 계산한 다음 결과를 `string`으로 변환하고 대괄호 사이에 있는 텍스트를 식의 문자열 결과로 바꿉니다. 첫 번째 식 `{weatherData.Date:MM-DD-YYYY}`의 `:`은 서식 문자열을 지정합니다. 앞의 예제에서는 "MM-DD\_YYYY" 형식으로 출력해야 할 날짜를 지정합니다.

## 패턴 일치

C# 언어는 개체 상태를 쿼리하고 이 상태에 따라 코드를 실행하는 **패턴 일치** 식을 제공합니다.\* 형식과 속성 및 필드의 값을 검사하여 수행할 동작을 결정할 수 있습니다. `switch` 식은 패턴 일치를 위한 기본 식입니다.

## 대리자와 람다 식

**대리자 형식**은 특정 매개 변수 목록 및 반환 형식이 있는 메서드에 대한 참조를 나타냅니다. 대리자는 메서드를 변수에 할당되고 매개 변수로 전달될 수 있는 엔터티로 취급할 수 있도록 합니다. 대리자는 다른 언어의 함수 포인터와 개념이 비슷하지만 함수 포인터와 달리 대리자는 개체 지향적이며 형식이 안전한 방식입니다.

다음 예제에서는 `Function`라는 대리자 형식을 선언하고 사용합니다.

```
delegate double Function(double x);

class Multiplier
{
    double _factor;

    public Multiplier(double factor) => _factor = factor;

    public double Multiply(double x) => x * _factor;
}

class DelegateExample
{
    static double[] Apply(double[] a, Function f)
    {
        var result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    public static void Main()
    {
        double[] a = { 0.0, 0.5, 1.0 };
        double[] squares = Apply(a, (x) => x * x);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new Multiplier(2.0);
        double[] doubles = Apply(a, m.Multiply);
    }
}
```

`Function` 대리자 형식의 인스턴스는 `double` 인수를 사용하고 `double` 값을 반환하는 메서드를 참조할 수 있습니다. `Apply` 메서드는 `double[]`의 요소에 지정된 `Function`을 적용하여 결과가 있는 `double[]`을 반환합니다. `Main` 메서드에서 `Apply`는 세 가지 다른 함수를 `double[]`에 적용하는 데 사용됩니다.

대리자는 정적 메서드(예: 이전 예제의 `Square` 또는 `Math.Sin`) 또는 인스턴스 메서드(예: 이전 예제의 `m.Multiply`)를 참조할 수 있습니다. 인스턴스 메서드를 참조하는 대리자는 특정 개체도 참조하며, 인스턴스 메서드가 대리자를 통해 호출되는 경우 해당 개체도 이 호출에서 `this`가 됩니다.

선언 즉시 만들어지는 “인라인 메서드”인 익명 함수를 사용하여 대리자를 만들 수도 있습니다. 익명 함수는 주변 메서드의 지역 변수를 볼 수 있습니다. 다음 예제에서는 클래스를 만들지 않습니다.

```
double[] doubles = Apply(a, (double x) => x * 2.0);
```

대리자는 해당 대리자가 참조하는 메서드의 클래스를 알지 못하고 클래스가 무엇인지와 관계없이 작동합니다. 중요한 것은 참조되는 메서드가 대리자와 동일한 매개 변수와 반환 형식을 갖는다는 사실입니다.

## async / await

C#은 두 개의 키워드 `async` 와 `await` 로 비동기 프로그램을 지원합니다. 메서드를 비동기로 선언하려면 메서드 선언에 `async` 한정자를 추가합니다. `await` 연산자는 컴파일러에 결과가 완료될 때까지 비동기적으로 대기하도록 지시합니다. 제어는 호출자에게 반환되고, 메서드는 비동기 작업의 상태를 관리하는 구조체를 반환합니다. 구조체는 일반적으로 `System.Threading.Tasks.Task<TResult>` 이지만, awaite 패턴을 지원하는 어떤 형식도 될 수 있습니다. 이러한 기능을 사용하면 동기식으로 읽히지만 비동기적으로 실행되는 코드를 작성할 수 있습니다. 예를 들어 다음 코드는 Microsoft docs의 홈페이지를 다운로드합니다.

```
public async Task<int> RetrieveDocsHomePage()
{
    var client = new HttpClient();
    byte[] content = await client.GetByteArrayAsync("https://docs.microsoft.com/");

    Console.WriteLine($"{nameof(RetrieveDocsHomePage)}: Finished downloading.");
    return content.Length;
}
```

이 작은 샘플은 비동기 프로그래밍의 주요 특징을 보여 줍니다.

- 메서드 선언에는 `async` 한정자가 포함됩니다.
- 메서드의 본문은 `GetByteArrayAsync` 메서드의 반환을 `await` 합니다.
- `return` 문에 지정된 형식은 메서드의 `Task<T>` 선언에 있는 형식 인수와 일치합니다. (`Task`를 반환하는 메서드는 `return` 문을 인수 없이 사용합니다.)

## 특성

C# 프로그램의 형식, 멤버 및 기타 엔터티는 동작의 특정 측면을 제어하는 한정자를 지원합니다. 예를 들어 메서드의 액세스 가능성은 `public`, `protected`, `internal` 및 `private` 한정자를 사용하여 제어됩니다. C#은 선언적 정보의 사용자 정의 형식을 프로그램 엔터티에 연결하고 런타임에 검색할 수 있도록 이러한 기능을 일반화합니다. 프로그램은 **특성\***을 정의하고 사용하여 이러한 추가적인 선언적 정보를 지정합니다.

다음 예제에서는 관련 설명서에 대한 링크를 제공하기 위해 프로그램 엔터티에 배치될 수 있는 `HelpAttribute` 특성을 선언합니다.

```
public class HelpAttribute : Attribute
{
    string _url;
    string _topic;

    public HelpAttribute(string url) => _url = url;

    public string Url => _url;

    public string Topic
    {
        get => _topic;
        set => _topic = value;
    }
}
```

모든 특성 클래스는 .NET 라이브러리에서 제공하는 `Attribute` 기본 클래스에서 파생됩니다. 연결된 선언 바로 앞에 대괄호로 묶은 특성 이름을 인수와 함께 적용할 수 있습니다. 특성 이름이 `Attribute`로 끝나는 경우 특성이 참조될 때 이름의 해당 부분을 생략해도 됩니다. 예를 들어 `HelpAttribute` 는 다음과 같이 사용할 수 있습니다.

```
[Help("https://docs.microsoft.com/dotnet/csharp/tour-of-csharp/features")]
public class Widget
{
    [Help("https://docs.microsoft.com/dotnet/csharp/tour-of-csharp/features",
        Topic = "Display")]
    public void Display(string text) { }
}
```

이 예제에서는 `HelpAttribute` 를 `Widget` 클래스에 연결합니다. 그런 후 다른 `HelpAttribute` 를 클래스의 `Display` 메서드에 추가합니다. 특성 클래스의 공용 생성자는 프로그램 엔터티에 특성을 추가할 때 제공해야 하는 정보를 제어합니다. 특성 클래스의 공용 읽기/쓰기 속성을 참조하여 추가 정보를 제공할 수 있습니다(예: 앞에 나온 `Topic` 속성 참조).

특성에 의해 정의된 메타데이터는 리플렉션을 사용하여 런타임 시 읽고 조작할 수 있습니다. 이 기술을 사용하여 특정 특성이 요청되면 특성 클래스에 대한 생성자가 프로그램 소스에 제공된 정보와 함께 호출됩니다. 결과 특성 인스턴스가 반환됩니다. 속성을 통해 추가 정보를 제공한 경우 해당 속성은 특성 인스턴스가 반환되기 전에 지정된 값으로 설정됩니다.

다음 코드 샘플에서는 `Widget` 클래스와 해당 `Display` 메서드에 연결된 `HelpAttribute` 인스턴스를 가져오는 방법을 보여줍니다.

```
Type widgetType = typeof(Widget);

object[] widgetClassAttributes = widgetType.GetCustomAttributes(typeof(HelpAttribute), false);

if (widgetClassAttributes.Length > 0)
{
    HelpAttribute attr = (HelpAttribute)widgetClassAttributes[0];
    Console.WriteLine($"Widget class help URL : {attr.Url} - Related topic : {attr.Topic}");
}

System.Reflection.MethodInfo displayMethod = widgetType.GetMethod(nameof(Widget.Display));

object[] displayMethodAttributes = displayMethod.GetCustomAttributes(typeof(HelpAttribute), false);

if (displayMethodAttributes.Length > 0)
{
    HelpAttribute attr = (HelpAttribute)displayMethodAttributes[0];
    Console.WriteLine($"Display method help URL : {attr.Url} - Related topic : {attr.Topic}");
}
```

## 자세한 정보

여러 [자습서](#)를 통해 C#에 대해 자세히 알아볼 수 있습니다.

이전

# C# 자습서

2021-02-18 • 4 minutes to read • [Edit Online](#)

C# 자습서를 시작합니다. 브라우저에서 실행할 수 있는 대화형 단원을 시작합니다. 최신 자습서 및 고급 자습서는 머신에서 .NET 개발 도구를 사용해 C# 프로그램을 만드는 데 도움이 됩니다.

## C#의 새 기능 살펴보기

- [문자열 보간](#): C#에서 형식이 지정된 문자열을 만들기 위해 문자열 보간을 사용하는 방법을 설명합니다.
- [Nullable 참조 형식](#): nullable 참조 형식을 사용하여 null 참조에 대한 의도를 나타내는 방법을 설명합니다.
- [Nullable 참조 형식을 사용하도록 프로젝트 업데이트](#): Nullable 참조 형식을 사용하도록 기존 프로젝트를 업그레이드하는 기술을 보여 줍니다.
- [패턴 일치를 사용하여 데이터 기능 확장](#): 패턴 일치를 사용하여 핵심 기능 이상으로 형식을 확장하는 방법을 보여 줍니다.
- [인덱스 및 범위를 사용하여 데이터 시퀀스 작업](#): 순차 데이터 컨테이너의 단일 요소 또는 범위에 액세스하기 위한 새로운 편리한 구문을 보여 줍니다.

## 일반 자습서

다음 자습서를 사용하면 [.NET Core](#)를 사용하여 C# 프로그램을 빌드할 수 있습니다.

- [콘솔 애플리케이션](#): 콘솔 I/O, 콘솔 애플리케이션의 구조 및 태스크 기반 비동기 프로그래밍 모델의 기본 사항에 대해 설명합니다.
- [REST 클라이언트](#): C# 언어의 웹 통신, JSON serialization 및 개체 지향 기능에 대해 설명합니다.
- [C# 및 .NET의 상속](#): 상속을 사용하여 기본 클래스, 추상 기본 클래스 및 파생 클래스를 정의하는 방법을 비롯하여 C#의 상속에 대해 설명합니다.
- [LINQ 작업](#): LINQ의 다양한 기능과 LINQ를 지원하는 언어 요소를 보여 줍니다.
- [특성 사용](#): C#에서 특성을 만들고 사용하는 방법을 설명합니다.
- [문자열 보간](#): 문자열에 값을 삽입하는 방법을 보여줍니다. 포함된 C# 식을 사용하여 보간된 문자열을 만드는 방법과 결과 문자열에서 식 결과의 텍스트 모양을 제어하는 방법을 설명합니다. 이 자습서는 머신에서 [로컬로 실행](#) 할 수도 있습니다.

# 클래스 및 객체를 사용한 객체 지향 프로그래밍 살펴보기

2021-02-18 • 24 minutes to read • [Edit Online](#)

이 자습서에서는 콘솔 애플리케이션을 빌드하고 C# 언어의 일부인 기본 객체 지향 기능을 확인합니다.

## 사전 준비 사항

이 자습서에서는 컴퓨터가 로컬 개발용으로 설정되어 있다고 가정합니다. Windows, Linux 또는 macOS에서 .NET CLI를 사용하여 애플리케이션을 만들고, 빌드하고, 실행할 수 있습니다. Windows에서 Visual Studio 2019를 사용할 수 있습니다. 설정 지침은 [로컬 환경 설정](#)을 참조하세요.

## 애플리케이션 만들기

터미널 창을 사용하여 `클래스`라는 디렉터리를 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.

```
using System;

namespace classes
{
    class Program
    {
        static void Main(string[] args)
        {
            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 디렉터리`에 `BankAccount.cs`라는 새 파일을 만듭니다.

이 파일에는 \*은행 계좌\*의 정의가 포함됩니다. 객체 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.

1. 은행 계좌를 고유하게 식별하는 10자리 숫자가 있습니다.
2. 소유자의 이름을 저장하는 문자열이 있습니다.
3. 잔액을 검색할 수 있습니다.
4. 예금을 허용합니다.
5. 인출을 허용합니다.
6. 초기 잔액은 양수여야 합니다.
7. 인출로 인해 음의 잔액이 발생할 수 없습니다.

## 은행 계좌 형식 정의

동작을 정의하는 클래스의 기본 사항을 만들어 시작할 수 있습니다. `File>New` 명령을 사용하여 새 파일을 만듭니다.

니다. 파일의 이름을 *BankAccount.cs*로 지정합니다. *BankAccount.cs* 파일에 다음 코드를 추가합니다.

```
using System;

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`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. 처음 세 개는 '속성'입니다. 속성은 데이터 요소이며 유효성 검사 또는 기타 규칙을 적용하는 코드가 있을 수 있습니다. 마지막 두 개는 메서드입니다. 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 할당하는 **생성자**를 정의하는 것입니다. 생성자는 클래스와 이름이 같은 멤버입니다. 생성자는 해당 클래스 형식의 개체를 초기화하는 데 사용됩니다. `BankAccount` 형식에 다음 생성자를 추가합니다. `MakeDeposit` 선언 위에 다음 코드를 배치합니다.

```
public BankAccount(string name, decimal initialBalance)
{
    this.Owner = name;
    this.Balance = initialBalance;
}
```

생성자는 `new`를 사용하여 개체를 만들 때 호출됩니다. *Program.cs*의 `Console.WriteLine("Hello World!");` 줄을 다음 코드로 바꿉니다(`<name>`을 사용자의 이름으로 바꿈).

```
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 실행 메뉴에서 디버깅하지 않고 시작을 선택합니다. 명령줄을 사용하는 경우 프로젝트를 만든 디렉터리에 `dotnet run`을 입력합니다.

계좌 번호가 공백인가요? 이를 수정해 보겠습니다. 개체가 생성될 때 계좌 번호를 지정해야 합니다. 그러나 계

좌 번호 생성은 호출자의 책임이 아닙니다. `BankAccount` 클래스 코드는 새 계좌 번호를 지정하는 방법을 알아야 합니다. 이를 수행하는 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.

```
private static int accountNumberSeed = 1234567890;
```

이는 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. 모든 `BankAccount` 개체에서 공유됨을 의미하는 `static` 이기도 합니다. 비정적 변수의 값은 `BankAccount` 개체의 각 인스턴스에 고유합니다. 생성자에 다음 두 줄을 추가하여 계좌 번호를 지정합니다.

`this.Balance = initialBalance` 줄 뒤에 배치합니다.

```
this.Number = accountNumberSeed.ToString();
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. 단순히 각 트랜잭션의 잔액을 업데이트하는 것보다 많은 장점이 있습니다. 기록을 사용하여 모든 트랜잭션을 감사하고 일별 잔액을 관리할 수 있습니다. 필요한 경우 모든 트랜잭션의 기록에서 잔액을 계산함으로써, 수정된 단일 트랜잭션의 오류가 다음 계산의 잔액에 올바르게 반영됩니다.

트랜잭션을 나타내는 새 형식을 생성해 보겠습니다. 이는 책임이 없는 단순 형식입니다. 몇 가지 속성이 필요합니다. `Transaction.cs`라는 새 파일을 만듭니다. 파일에 다음 코드를 추가합니다.

```
using System;

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)
        {
            this.Amount = amount;
            this.Date = date;
            this.Notes = note;
        }
    }
}
```

이제 `Transaction` 개체의 `List<T>`를 `BankAccount` 클래스에 추가하겠습니다. `BankAccount.cs` 파일의 생성자 뒤에 다음 선언을 추가합니다.

```
private List<Transaction> allTransactions = new List<Transaction>();
```

`List<T>` 클래스를 사용하려면 다른 네임스페이스를 가져와야 합니다. `BankAccount.cs`의 시작 부분에 다음을 추가합니다.

```
using System.Collections.Generic;
```

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.

```
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 최종 두 규칙을 적용합니다. 초기 잔액은 양수여야 하고 인출로 인해 음수의 잔액이 발생되어서는 안 됩니다.

이는 예외의 개념을 소개합니다. 메서드가 작업을 성공적으로 완료할 수 없음을 나타내는 일반적인 방법은 예외를 `throw`하는 것입니다. 예외 형식 및 관련 메시지는 오류를 설명합니다. 여기에서 `MakeDeposit` 메서드는 인출 금액이 음수인 경우 예외를 `throw`합니다. 인출 금액이 음수이거나 인출 적용 후 잔액이 음수인 경우 `MakeWithdrawal` 메서드는 예외를 `throw`합니다. `allTransactions` 목록의 선언 뒤에 다음 코드를 추가합니다.

```
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 `throw` 합니다. 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` 메서드

를 이미 작성했으므로 생성자에서 호출합니다. 완성된 생성자는 다음과 같아야 합니다.

```
public BankAccount(string name, decimal initialBalance)
{
    this.Number = accountNumberSeed.ToString();
    accountNumberSeed++;

    this.Owner = name;
    MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

`DateTime.Now`은 현재 날짜 및 시간을 반환하는 속성입니다. 새 `BankAccount`를 만드는 코드를 따라 `Main` 메서드에서 몇 가지 예금 및 인출을 추가하여 테스트합니다.

```
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 음수 잔액을 사용하여 계정을 생성하여 오류 조건을 알아보는 테스트를 수행합니다. 방금 추가한 이전 코드 뒤에 다음 코드를 추가합니다.

```
// Test that the initial balances must be positive.
try
{
    var invalidAccount = new BankAccount("invalid", -55);
}
catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine("Exception caught creating account with negative balance");
    Console.WriteLine(e.ToString());
}
```

`try` 및 `catch` 문을 사용하여 예외를 `throw`할 수 있는 코드 블록을 표시하고 예상한 오류를 `catch`합니다. 동일한 기술을 사용하여 음수 잔액에 대한 예외를 `throw`하는 코드를 테스트할 수 있습니다. `Main` 메서드의 끝에 다음 코드를 추가합니다.

```
// 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.

```
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();
}
```

`StringBuilder` 클래스를 사용하여 각 트랜잭션에 대해 한 줄을 포함하는 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.

```
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.

- [If 및 else 문](#)
- [While 문](#)
- [Do 문](#)
- [For 문](#)

# 개체 지향 프로그래밍(C#)

2021-02-18 • 30 minutes to read • [Edit Online](#)

C#는 개체 지향 언어입니다. 개체 지향 프로그래밍에 사용되는 네 가지 주요 방법은 다음과 같습니다.

- '추상화'는 형식 소비자의 불필요한 세부 정보를 숨기는 것입니다.
- **캡슐화**는 서로 관련된 속성, 메서드 및 기타 멤버의 그룹을 하나의 단위나 개체로 취급하는 것을 말합니다.
- **상속**은 기존 클래스를 기반으로 새로운 클래스를 만들 수 있는 능력을 나타냅니다.
- **다형성**은 동일한 속성 또는 메서드를 각각 다른 방식으로 구현하는 여러 클래스를 서로 교체하여 사용할 수 있음을 의미합니다.

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 설명하는데 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 한번 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.

```
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.

```
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.

```
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.
- 선불 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.

```
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.

```

public override void PerformMonthEndTransactions()
{
    if (Balance > 500m)
    {
        var 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.

```

public override void PerformMonthEndTransactions()
{
    if (Balance < 0)
    {
        // Negate the balance to get a positive interest charge:
        var 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.

```

private 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 `0`을 생략할 수 있습니다. 다음으로 생성자에서 `0`이 아닌 값으로 설정된 경우 월별 예치금을 추가하도록 `PerformMonthEndTransactions` 메서드를 재정의합니다.

```

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.

```

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.

```

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.

```

Unhandled exception. System.ArgumentOutOfRangeException: Amount of deposit must be positive (Parameter 'amount')
at OOPProgramming.BankAccount.MakeDeposit(Decimal amount, DateTime date, String note) in
BankAccount.cs:line 42
at OOPProgramming.BankAccount..ctor(String name, Decimal initialBalance) in BankAccount.cs:line 31
at OOPProgramming.LineOfCreditAccount..ctor(String name, Decimal initialBalance) in
LineOfCreditAccount.cs:line 9
at OOPProgramming.Program.Main(String[] args) in Program.cs:line 29

```

#### NOTE

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. 기본 값은 `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 수도 있지만 그러면 나중에 두 위치를 변경해야 합니다. 대신 생성자 연결을 사용하여 한 생성자가 다른 생성자를 호출하도록 할 수 있습니다. 다음 코드는 두 개의 생성자와 새 추가 필드를 보여 줍니다.

```

private readonly decimal minimumBalance;

public BankAccount(string name, decimal initialBalance) : this(name, initialBalance, 0) { }

public BankAccount(string name, decimal initialBalance, decimal minimumBalance)
{
    this.Number = accountNumberSeed.ToString();
    accountNumberSeed++;

    this.Owner = name;
    this.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 만들어지면 `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.

```
if (Balance - amount < minimumBalance)
```

`BankAccount` 클래스를 확장한 후 다음 코드에 나온 것처럼 새 기본 생성자를 호출하도록 `LineOfCreditAccount` 생성자를 수정할 수 있습니다.

```

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.

```

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.

```

public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of withdrawal must be positive");
    }
    var overdraftTransaction = CheckWithdrawalLimit(Balance - amount < minimumBalance);
    var withdrawal = new Transaction(-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.

```

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.

```
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 클래스에 많은 세부 정보를 `private`으로 유지하는 경우 추상화를 사용했습니다.
- 서로 다른 각 계좌 유형의 클래스를 정의할 때 캡슐화를 사용했습니다. 이러한 클래스는 해당 계좌 유형의 동작을 설명합니다.
- `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.

축하합니다. 모든 C# 소개 자습서를 완료했습니다. 더 자세한 내용은 추가 [자습서](#)를 확인하세요.

# 레코드 종류 만들기

2021-02-18 • 30 minutes to read • [Edit Online](#)

C# 9에서는 클래스나 구조체 대신 만들 수 있는 새 참조 형식인 '레코드'를 소개합니다. 레코드는 레코드 종류가 '값 기반 같음'을 사용한다는 점에서 클래스와 구분됩니다. 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.

이 자습서에서 학습할 방법은 다음과 같습니다.

- `class` 를 선언할지 `record` 를 선언할지를 결정합니다.
- 레코드 종류 및 위치 레코드 종류를 선언합니다.
- 레코드에서 컴파일러 생성 메서드를 자신의 메서드로 대체합니다.

## 사전 요구 사항

C# 9.0 이상 컴파일러를 포함하여 .NET 5 이상을 실행하도록 머신을 설정해야 합니다. C# 9.0 컴파일러는 [Visual Studio 2019 버전 16.8](#) 또는 [.NET 5.0 SDK](#)부터 사용할 수 있습니다.

## 레코드의 특징

`class` 또는 `struct` 키워드 대신 `record` 키워드로 형식을 선언하여 '레코드'를 정의합니다. 레코드는 참조 형식이며 값 기반 같음 의미 체계를 따릅니다. 값 의미 체계를 적용하기 위해 컴파일러는 레코드 종류에 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 `init` 전용 속성.
- 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 개체군 관리에 도움이 됩니다. 도일은 계절

변화에 따른 식물 성장과 상관 관계가 있습니다. 도일은 기후에 따라 이동하는 동물 종의 이동을 추적하는데 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.

```
public record DailyTemperature(double HighTemp, double LowTemp);
```

위의 코드는 '위치 레코드'를 정의합니다. `HighTemp` 와 `LowTemp` 의 두 속성을 포함하는 참조 형식을 만들었습니다. 이러한 속성은 'init 전용 속성'이므로 생성자에서 설정하거나 속성 이니셜라이저를 사용하여 설정할 수 있습니다. `DailyTemperature` 형식에는 두 개의 속성과 일치하는 두 개의 매개 변수가 있는 '기본 생성자'도 있습니다. 기본 생성자를 사용하여 `DailyTemperature` 레코드를 초기화합니다.

```
private static DailyTemperature[] data = new DailyTemperature[]
{
    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.

```
public record DailyTemperature(double HighTemp, double LowTemp)
{
    public double Mean => (HighTemp + LowTemp) / 2.0;
}
```

이 데이터를 사용할 수 있는지 확인해 보겠습니다. `Main` 메서드에 다음 코드를 추가합니다.

```
foreach (var item in data)
    Console.WriteLine(item);
```

애플리케이션을 실행하면 다음과 같은 출력이 표시됩니다(공간상의 이유로 여러 행을 제거함).

```
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 원하는 경우 `ToString`의 고유한 버전을 작성할 수 있습니다. 그러면 컴파일러에서 자동으로 버전을 합성할 수 없습니다.

## 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.

```
public abstract record DegreeDays(double BaseTemperature, IEnumerable<DailyTemperature> TempRecords);  
  
public 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.

```
var heatingDegreeDays = new HeatingDegreeDays(65, data);  
Console.WriteLine(heatingDegreeDays);  
  
var coolingDegreeDays = new CoolingDegreeDays(65, data);  
Console.WriteLine(coolingDegreeDays);
```

다음과 같은 출력이 표시됩니다.

```
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 종류가 `sealed`이면 서명은 `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.

```
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.

```
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.

## 비파괴적 변경

위치 레코드의 합성 멤버는 레코드의 상태를 수정하지 않습니다. 목표는 변경이 불가능한 레코드를 더욱 쉽게 만들 수 있도록 하는 것입니다. `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.

```
// 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.

```
// 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.

```
var growingDegreeDaysCopy = growingDegreeDays with { };
```

완성된 애플리케이션을 실행하여 결과를 확인합니다.

## 요약

이 자습서에서는 레코드의 여러 측면을 보여 주었습니다. 레코드는 기본 용도가 데이터 저장인 참조 형식에 대해 간결한 구문을 제공합니다. 개체 지향 클래스의 기본 용도는 책임을 정의하는 것입니다. 이 자습서에서는 간결한 구문을 사용하여 레코드에 대한 init 전용 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. 위치 레코드의 경우 `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 같음 및 비교를 원하지만 값을 복사하지 않고 참조 변수를 사용하려는 경우 레코드를 만듭니다.

레코드에 대한 전체 설명은 [제안된 레코드 종류 사양](#)을 참조하세요.

# 자습서: 배우는 동안 최상위 문을 사용하여 코드를 빌드하는 아이디어 탐색

2021-02-18 • 18 minutes to read • [Edit Online](#)

이 자습서에서 학습할 방법은 다음과 같습니다.

- 최상위 문 사용을 제어하는 규칙을 알아봅니다.
- 최상위 문을 사용하여 알고리즘을 탐색합니다.
- 탐색을 재사용 가능한 구성 요소로 리팩터링합니다.

## 사전 요구 사항

C# 9 컴파일러를 포함하는 .NET 5를 실행하도록 머신을 설정해야 합니다. C# 9 컴파일러는 [Visual Studio 2019 버전 16.9 미리 보기 1](#) 또는 [.NET 5.0 SDK](#)부터 사용할 수 있습니다.

이 자습서에서는 Visual Studio 또는 .NET Core CLI를 포함하여 C# 및 .NET에 익숙하다고 가정합니다.

## 다양한 콘텐츠 살펴보기

최상위 문을 사용하면 프로그램의 진입점을 클래스의 정적 메서드에 배치하여 필요한 추가 공식 절차를 방지할 수 있습니다. 새 콘솔 애플리케이션의 일반적인 시작점은 다음 코드와 같습니다.

```
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 해당 프로그램을 단순화할 수 있습니다. 이렇게 하면 해당 프로그램에서 두 개 줄을 제외한 모든 줄을 제거할 수 있습니다.

```
using System;

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

이 작업은 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 가능한 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.

```
using System;

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.

```
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.

```
System.String[]
```

콘솔에 질문을 쓰려면 인수를 열거하고 공백으로 구분해야 합니다. `WriteLine` 호출을 다음 코드로 바꿉니다.

```
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.

```
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 12개, 커밋이 아닌 응답 6개, 부정 응답 6개가 있습니다. 다음으로, 다음 코드를 추가하여

배열에서 임의 응답을 생성하고 표시합니다.

```
var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

애플리케이션을 다시 실행하여 결과를 볼 수 있습니다. 다음 출력과 같은 정보가 표시됩니다.

```
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.

```
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();
```

또한 소스 파일의 맨 위에 `using` 문을 추가해야 합니다.

```
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 지연 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.

```

using System;
using System.Threading.Tasks;

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.

```

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.

```

using System;
using System.Threading.Tasks;

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 문은 한 파일에만 있을 수 있으며, 해당 파일은 네임스페이스나 형식을 포함할 수 없습니다.

마지막으로, 애니메이션 코드를 정리하여 일부 중복을 제거할 수 있습니다.

```

foreach (string s in new[] { "| -", "/ \\", "- |", "\\ /", })
{
    Console.WriteLine(s);
    await Task.Delay(50);
    Console.WriteLine("\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 유ти리티 메서드를 호출할 수 있습니다.

```

using System;
using MyNamespace;

Console.WriteLine();
foreach(var s in args)
{
    Console.WriteLine(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]);

```

그러면 `Utilities.ShowConsoleAnimation` 호출이 추가되고 `using` 문이 추가됩니다.

## 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.

# 패턴 일치를 사용하여 코드를 개선하는 클래스 동작 빌드

2021-02-18 • 24 minutes to read • [Edit Online](#)

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 자습서에서 학습할 방법은 다음과 같습니다.

- 데이터 패턴을 사용하여 개체 지향 클래스를 표현합니다.
- C#의 패턴 일치 기능을 사용하여 이러한 패턴을 구현합니다.
- 컴파일러 진단을 활용하여 구현의 유효성을 검사합니다.

## 필수 구성 요소

C# 9.0 컴파일러를 포함하여 .NET 5를 실행하도록 컴퓨터를 설정해야 합니다. C# 9.0 컴파일러는 [Visual Studio 2019 버전 16.8 미리 보기](#) 또는 [.NET 5.0 SDK 미리 보기](#)부터 사용할 수 있습니다.

## 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.

```
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.

```

// 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}");
Console.WriteLine(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.

```

// 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.

```

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.

```

// 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 개체가 명령을 실행하기에 유효한 상태에 있는지 확인하는 것입니다. 명령이 세 변수(수문 상태, 수위, 새 설정)의 함수로 허용되는지 여부를 표현할 수 있습니다.

새 설정	수문 상태	수위	결과
종결	종결	높은	종결
종결	종결	낮음	종결
종결	열기	높은	열기
해결됨	열기	낮음	해결됨
열기	해결됨	높은	열기
열기	해결됨	낮음	닫힘(오류)
열기	열기	높은	열기
열기	열기	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 나타냄을 명심하세요).

```

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 통과되며 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 컴파일러가 다음과 같은 경고를 생성하는 것을 볼 수 있습니다. *CS8524*는 `switch` 식에 가능한 모든 입력이 포함되지 않음을 나타냅니다. 이 경고가 발생하는 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.

```
_ => throw new InvalidOperationException("Invalid internal state"),
```

위의 `switch arm`은 모든 입력과 일치하므로 `switch` 식에서 마지막에 와야 합니다. 앞 순서로 옮겨 실행해 보세요. 그러면 패턴의 연결할 수 없는 코드를 나타내는 컴파일러 오류 *CS8510*이 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 `arm`이 발생하는 경우 오류를 생성하고, 필요하지 않은 `arm`을 제거하는 경우 경고를 생성합니다.

첫 번째 변경은 명령이 수문 닫기인 모든 `arm`을 결합하는 것입니다. 이는 항상 허용됩니다. `switch` 식의 첫 번째 `arm`으로 다음 코드를 추가합니다.

```
(false, _, _) => false,
```

이전 `switch arm`을 추가한 후 명령이 `false`인 각 `arm`에 하나씩 4개의 컴파일러 오류가 발생합니다. 이러한 `arm`은 새로 추가된 `arm`에 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 `arm`을 단순화할 수 있습니다. 수위가 높은 두 경우 모두 수문을 열 수 있습니다. (한 경우에는 수문이 이미 열려 있습니다.) 수위가 낮은 한 경우는 예외를 `throw`하고 다른 경우는 발생하면 안 됩니다. 갑문이 이미 잘못된 상태인 경우 동일한 예외를 안전하게 `throw`해야 합니다. 이러한 `arm`에 대해 다음과 같은 단순화를 만들 수 있습니다.

```

(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.

```

// 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.

```

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 설정

하는 메서드에 동일한 프로세스를 사용하는 것이 유용할 수 있습니다. 다음 네 개의 입력 모두로 시작합니다. 양 쪽 수문의 상태, 수위의 현재 상태, 요청된 새 수위. switch 식은 다음으로 시작해야 합니다.

```
CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen, HighWaterGateOpen) switch
{
    // elided
};
```

입력할 switch arm은 총 16개입니다. 그런 다음 테스트하고 단순화합니다.

만든 메서드가 다음과 비슷합니까?

```
// 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, 문하 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.



# 문자열 보간을 사용하여 형식이 지정된 문자열 생성

2020-05-11 • 20 minutes to read • [Edit Online](#)

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.

```
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.

```
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.

- `$` 문자로 시작한 후 여는 따옴표 문자가 다음에 나오는 문자열 리터럴. `$` 기호와 따옴표 문자 사이에는 공백이 없어야 합니다.(공백을 포함하면 어떻게 되는지 확인하려면 `$` 문자 뒤에 공백을 삽입하고 파일을 저장한 후 콘솔 창에 `dotnet run`을 입력하여 프로그램을 다시 실행합니다. C# 컴파일러가 "오류 CS1056: 예기치 않은 문자 '\$'"라는 오류 메시지를 표시합니다.)
- 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 삽입하는데 문자열 보간을 사용했습니다. 보간 식의 결과는 모든 데이터 형식일 수 있습니다. 보간된 문자열에 다양한 데이터 형식의 값을 포함시켜 보겠습니다.

다음 예제에서는 먼저 `Name` 속성과 `Object.ToString()` 메서드의 동작을 재정의하는 `ToString` 메서드가 있는 [클래스](#) 데이터 형식 `Vegetable`을 정의합니다. `public` 액세스 한정자를 지정하면 해당 메서드를 모든 클라이언트

코드에 사용하여 `Vegetable` 인스턴스의 문자열 표현을 가져올 수 있습니다. 예제에서 `Vegetable.ToString` 메서드는 `Vegetable` 생성자에서 초기화된 `Name` 속성의 값을 반환합니다.

```
public Vegetable(string name) => Name = name;
```

그런 다음, `new` 연산자를 사용하고 생성자 `Vegetable`의 이름을 제공하여 `item`이라는 `Vegetable` 클래스의 인스턴스를 만듭니다.

```
var item = new Vegetable("eggplant");
```

마지막으로 `DateTime` 값, `Decimal` 값 및 `Unit` 열거형 값을 포함하는 보간된 문자열에 `item` 변수를 포함시킵니다. 편집기에서 모든 C# 코드를 다음 코드로 바꾼 후 `dotnet run` 명령을 사용하여 실행합니다.

```
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. 하나는 날짜만 적절한 날짜 및 시간 값이었습니다. 두 번째는 통화 단위를 나타내지 않는 가격이었습니다. 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 포함시킵니다.

```
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 간단한 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 생성하는데 도움이 됩니다. 이것을 확인하려면 텍스트 편집기에서 모든 코드를 다음 코드로 바꾼 후 `dotnet run`을 입력하여 프로그램을 실행합니다.

```
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 보세

요.

```
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.

```
Console.WriteLine($"{DateTime.Now,-20:d} {Hour} [{DateTime.Now,-10:HH}] [{1063.342,15:N2}] feet");
```

다음과 같은 출력을 얻을 수 있습니다.

```
[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

2020-03-18 • 14 minutes to read • [Edit Online](#)

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. .NET에서 형식 서식 지정하는 방법에 대한 자세한 내용은 [.NET의 형식 지정](#) 항목을 참조하세요.

## NOTE

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.

## 소개

문자열 보간 기능은 [복합 서식 지정](#) 기능을 기반으로 빌드되고 결과 문자열에 서식이 지정된 식 결과를 포함하는 읽기 쉽고 편리한 구문을 제공합니다.

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.

```
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);

// Expected 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.

```
{<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

콜론(:)과 형식 문자열을 사용하여 보간 식에 따라 식 결과의 형식에서 지원하는 형식 문자열을 지정합니다.

```
{<interpolationExpression>:<formatString>}
```

다음 예제에서는 날짜 및 시간 또는 숫자 결과를 생성하는 식의 표준 및 사용자 지정 서식 지정 문자열을 지정하는 방법을 보여줍니다.

```

var date = new DateTime(1731, 11, 25);
Console.WriteLine($"On {date:dddd, MMMM dd, yyyy} Leonhard Euler introduced the letter e to denote
{Math.E:F5} in a letter to Christian Goldbach.");

// Expected output:
// On Sunday, November 25, 1731 Leonhard Euler introduced the letter e to denote 2.71828 in a letter to
Christian Goldbach.

```

자세한 내용은 [복합 서식 지정](#) 항목의 [문자열 구성 요소 서식 지정](#) 섹션을 참조하세요. 해당 섹션에서는 .NET 기본 형식에서 지원하는 표준 및 사용자 지정 서식 지정 문자열을 설명하는 항목에 대한 링크를 제공합니다.

## 필드 너비와 서식이 지정된 보간 식의 맞춤을 제어하는 방법

쉼표(",") 및 상수 식을 포함한 보간 식에 따라 최소 필드 너비 및 서식이 지정된 식 결과의 맞춤을 지정합니다.

```
{<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.

```
{<interpolationExpression>,<alignment>:<formatString>}
```

다음 예제에서는 맞춤을 지정하고 파이프 문자("|")를 사용하여 텍스트 필드를 구분하는 방법을 보여줍니다.

```

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}|");

// Expected output:
// Three classical Pythagorean means of 3 and 4:
// |Arithmetic| 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 지정](#) 항목의 [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 약어 문자열은 \$ 문자가 뒤에 오는 @ 문자로 시작합니다. C# 8.0부터는 \$ 및 @ 토큰을 순서에 관계없이 사용할 수 있습니다. \$@"..." 및 @\$..."는 모두 유효한 보간된 약어 문자열입니다.

중괄호("{} 또는 "}")를 포함하려면 결과 문자열에서 2개의 중괄호("{{" 또는 "}}")를 사용합니다. 자세한 내용은 [복합 서식 지정](#) 항목의 [중괄호 이스케이프 처리](#) 섹션을 참조하세요.

다음 예제에서는 결과 문자열에 중괄호를 포함하고 약어 보간된 문자열을 만드는 방법을 보여줍니다.

```
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.");
Console.WriteLine("Jane");
var stringWithEscapes = $"C:\\Users\\{userName}\\Documents";
var verbatimInterpolated = @"C:\Users\{userName}\Documents";
Console.WriteLine(stringWithEscapes);
Console.WriteLine(verbatimInterpolated);

// Expected output:
// Find the intersection of the {1, 2, 7, 9} and {7, 9, 12} sets.
// C:\Users\Jane\Documents
// C:\Users\Jane\Documents
```

## 보간 식에서 3개로 구성된 ?: 조건부 연산자를 사용하는 방법

보간 식에서 콜론(":")에 특별한 의미가 있으므로 식에서 [조건부 연산자](#)를 사용하기 위해 다음 예제에서 볼 수 있듯이 해당 식을 괄호로 묶습니다.

```
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. [System.FormattableString](#) 인스턴스에 대한 보간된 문자열의 암시적 변환을 사용하고 해당 [ToString\(IFormatProvider\)](#) 메서드를 호출하여 문화권별 결과 문자열을 만듭니다. 다음 예제에서는 해당 작업을 수행하는 방법을 보여줍니다.

```
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 = $"{date,20}{number,20:N3}";
foreach (var culture in cultures)
{
    var cultureSpecificMessage = message.ToString(culture);
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
}

// Expected output is like:
// en-US      5/17/18 3:44:55 PM      31,415,926.536
// en-GB      17/05/2018 15:44:55      31,415,926.536
// nl-NL      17-05-18 15:44:55      31.415.926,536
//          05/17/2018 15:44:55      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

[FormattableString.ToString\(IFormatProvider\)](#) 메서드와 함께 고정 [FormattableString.Invariant](#) 메서드를 사용하여 [InvariantCulture](#)에서 보간된 문자열을 결과 문자열로 해결할 수 있습니다. 다음 예제에서는 해당 작업을 수행하는 방법을 보여줍니다.

```
string messageInInvariantCulture = FormattableString.Invariant($"Date and time in invariant culture:  
{DateTime.Now}");  
Console.WriteLine(messageInInvariantCulture);  
  
// Expected output is like:  
// 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 보간 항목](#)을 참조하세요. .NET에서 형식 서식 지정하는 방법에 대한 자세한 내용은 [.NET의 형식 지정](#) 및 [복합 서식 지정](#) 항목을 참조하세요.

## 참고 항목

- [String.Format](#)
- [System.FormattableString](#)
- [System.IFormattable](#)
- [문자열](#)

# 자습서: C# 8.0에서 기본 인터페이스 메서드로 인터페이스를 업데이트

2020-11-02 • 17 minutes to read • [Edit Online](#)

.NET Core 3.0의 C# 8.0에서부터, 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# 8.0 컴파일러를 포함하여 .NET Core를 실행하도록 머신을 설정해야 합니다. C# 8.0 컴파일러는 [Visual Studio 2019 버전 16.3](#) 또는 [.NET CORE 3.0 SDK](#)부터 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.

```
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.

```
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! 주요 변경 사항입니다!" C#8.0은 인터페이스 업그레이드를 위해 기본 인터페이스 구현을 추가했습니다. 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. 이로써 기본 인터페이스 메서드에 완벽한 시나리오가 됩니다. `ICustomer` 인터페이스에 메서드를 추가하고 가장 가능성이 높은 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.

```
// 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.

```
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(2103, 7, 4), 25m);  
c.AddOrder(o);  
  
// Check the discount:  
ICustomer theCustomer = c;  
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

테스트의 다음 부분을 확인합니다.

```
// Check the discount:  
ICustomer theCustomer = c;  
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

`SampleCustomer`에서 `ICustomer`로의 캐스트가 필요합니다. `SampleCustomer` 클래스는 `ComputeLoyaltyDiscount`에 대한 구현을 제공할 필요가 없으며, `ICustomer` 인터페이스에서 제공합니다. 그러나 `SampleCustomer` 클래스

는 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

시작이 좋습니다. 그러나 기본 구현은 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.

```
// 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. 추가 필드는 비공개이고 새 메서드는 공개입니다. 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.

```
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.

```
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.

```
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. 여러 클래스를 통해 구현할 수 있는 단일 기능 아이디어를 표현하려면 인터페이스를 신중하게 설계하세요. 이를 통해 동일한 기능 아이디어에 새로운 요구 사항이 발견될 경우 해당 인터페이스 정의를 훨씬 쉽게 업그레이드할 수 있습니다.

# 자습서: 기본 인터페이스 메서드를 사용하는 인터페이스를 통해 클래스를 만드는 경우의 기능 혼합

2020-11-02 • 23 minutes to read • [Edit Online](#)

.NET Core 3.0의 C# 8.0에서부터, 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# 8.0 컴파일러를 포함하여 .NET Core를 실행하도록 머신을 설정해야 합니다. C# 8.0 컴파일러는 [Visual Studio 2019 버전 16.3](#) 또는 [.NET CORE 3.0 SDK](#) 이상부터 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 하는지 제공하지 않아야 하는지 여부에 관계없이 선언할 수 없습니다.

C# 8.0부터 기본 구현을 인터페이스 메서드로 선언할 수 있습니다. 그러면 모든 클래스에서 자동으로 기본 구현을 사용합니다. 더 나은 구현을 제공할 수 있는 모든 클래스는 더 나은 알고리즘으로 인터페이스 메서드를 재정의할 수 있습니다. 어떤 의미에서 이 기술은 [확장 메서드](#)를 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 이 시나리오에서 확장 메서드보다 더 나은 솔루션입니다. 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.

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
}
```

기본 오버헤드 광원 퍽스쳐는 다음 코드와 같이 이 인터페이스를 구현할 수 있습니다.

```
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.

```
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.

```
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.");
    }
}
```

이러한 변경 사항 추가로 `OverheadLight` 클래스는 인터페이스에 대한 지원을 선언하여 타이머 함수를 구현할 수 있습니다.

```
public class OverheadLight : ITimerLight { }
```

다른 광원 형식은 보다 정교한 프로토콜을 지원할 수 있습니다. 다음 코드와 같이 `TurnOnFor`에 대한 자체 구현을 제공할 수 있습니다.

```
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}";
}
```

가상 클래스 메서드를 재정의하는 것과 달리 `HalogenLight` 클래스의 `TurnOnFor` 선언에는 `override` 키워드가 사용되지 않습니다.

## 조합 및 일치 기능

고급 기능을 도입할 수록 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.

```
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.

```

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.

```

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.

```

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.

```

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.

```

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 확인해 보겠습니다. 모든 표시등 광원은 해당 전원 상태를 열거형 값으로 보고할 수 있습니다.

```
public enum PowerStatus
{
    NoPower,
    ACPower,
    FullBattery,
    MidBattery,
    LowBattery
}
```

기본 구현에서는 전원이 없는 것으로 가정합니다.

```
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 광원의 각 기능은 고유한 인터페이스입니다. 여러 인터페이스는 클래스에 의해서만 상속됩니다.

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 종인 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.

# 인덱스 및 범위

2021-02-18 • 15 minutes to read • [Edit Online](#)

범위와 인덱스는 시퀀스의 단일 요소 또는 범위에 액세스하기 위한 간결한 구문을 제공합니다.

이 자습서에서는 다음과 같은 작업을 수행하는 방법을 알아봅니다.

- 시퀀스에서 범위에 구문을 사용합니다.
- 각 시퀀스의 시작 및 끝에 대한 설계 의사 결정을 이해합니다.
- [Index](#) 및 [Range](#) 형식에 대한 시나리오를 살펴봅니다.

## 인덱스 및 범위에 대한 언어 지원

이 언어 지원은 다음과 같은 두 가지 새 형식 및 두 가지 새 연산자를 사용합니다.

- [System.Index](#)는 인덱스를 시퀀스로 표현합니다.
- 인덱스가 시퀀스의 끝을 기준으로 하도록 지정하는 끝부터 인덱스 연산자 `^`입니다.
- [System.Range](#)는 시퀀스의 하위 범위를 나타냅니다.
- 범위의 시작과 끝을 피연산자로 지정하는 범위 연산자 `..`입니다.

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. `n`이 어떤 숫자이든, 인덱스 `^n`은 `sequence[sequence.Length - n]`과 동일합니다.

```
string[] words = new string[]
{
    // index from start      index from end
    "The",        // 0           ^9
    "quick",      // 1           ^8
    "brown",      // 2           ^7
    "fox",        // 3           ^6
    "jumped",     // 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.

```
Console.WriteLine($"The last word is {words[^1]}");
```

한 범위는 어떤 범위의 시작 및 끝을 지정합니다. 여러 범위는 배타적입니다. 즉, '끝'이 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. 다음 코드를 같은 메서드에 추가합니다. 대화형 창의 맨 아래에 다음 코드를 복사하여 붙여넣습니다.

```
string[] quickBrownFox = words[1..4];
foreach (var word in quickBrownFox)
    Console.WriteLine($"< {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.

```
string[] lazyDog = words[^2..^0];
foreach (var word in lazyDog)
    Console.WriteLine($"< {word} >");
Console.WriteLine();
```

다음 예제는 시작만, 끝만, 그리고 시작과 끝이 모두 열린 범위를 만듭니다.

```
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.WriteLine($"< {word} >");
Console.WriteLine();
foreach (var word in firstPhrase)
    Console.WriteLine($"< {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.

```
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 `z`를 수정하여 다양한 조합을 시도해 봅니다. 실험할 때는 올바른 조합을 위해 `x`가 `y`보다 작고 `y`가 `z`보다 작은 값을 사용합니다. 새 메서드에 다음 코드를 추가합니다. 다양한 조합을 시도해 봅니다.

```

int[] numbers = Enumerable.Range(0, 100).ToArray();
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]}, numbers[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 {x_x[^1]}");

Console.WriteLine("numbers[..x] means numbers[0..x] and numbers[x..] means numbers[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]}");

```

## 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.

### IMPORTANT

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 N은 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 섹션](#)을 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. 다차원 배열은 인덱서 또는 범위를 지원하지 않습니다. 다차원 배열에 대한 인덱서에는 단일 매개 변수가 아닌 여러 개의 매개 변수가 포함됩니다. 예를 들어, [int\[,\]](#)는 두 개의 인덱스를 필요로 하는 반면, [int\[,,\]](#)는 세 개의 인덱스를 필요로 합니다.

수가 있습니다. 배열의 배열이라고도 하는 가변 배열은 범위와 인덱스를 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.

```
var jagged = new int[10][]{  
    new int[10] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},  
    new int[10] { 10,11,12,13,14,15,16,17,18,19},  
    new int[10] { 20,21,22,23,24,25,26,27,28,29},  
    new int[10] { 30,31,32,33,34,35,36,37,38,39},  
    new int[10] { 40,41,42,43,44,45,46,47,48,49},  
    new int[10] { 50,51,52,53,54,55,56,57,58,59},  
    new int[10] { 60,61,62,63,64,65,66,67,68,69},  
    new int[10] { 70,71,72,73,74,75,76,77,78,79},  
    new int[10] { 80,81,82,83,84,85,86,87,88,89},  
    new int[10] { 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.

```
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)).ToArray();
```

# 자습서: nullable 참조 형식 및 nullable이 아닌 참조 형식을 사용하여 디자인 의도를 보다 명확하게 표현

2020-11-02 • 27 minutes to read • [Edit Online](#)

C# 8.0에서는 nullable 값 형식이 값 형식을 보완하는 것과 동일한 방식으로 참조 형식을 보완하는 nullable 참조 형식이 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# 8.0 컴파일러를 포함하여 .NET Core를 실행하도록 컴퓨터를 설정해야 합니다. C# 8.0 컴파일러는 [Visual Studio 2019](#) 또는 [.NET Core 3.0](#)에서 사용할 수 있습니다.

이 자습서에서는 Visual Studio 또는 .NET Core CLI를 포함하여 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# 8.0 프로젝트에서도 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.

```
<Nullable>enable</Nullable>
```

### 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.

```
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.

```

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 초기화되지 않았다는 경고를 실행합니다. 디자인에 따라 질문 텍스트는 null이 아니어야 하므로 초기화할 생성자와 `QuestionType` 값도 추가합니다. 완성된 클래스 정의는 다음 코드와 같습니다.

```

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.

```

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 `null` 일 수 있습니다.

편집기에서 `Program.cs`로 전환하고 `Main`의 내용을 다음 코드 줄로 바꿉니다.

```
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 `null`을 전달하면 경고가 발생합니다. 다음 줄을 `Main`에 추가하여 시도해 보세요.

```
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.

```
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.

```
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.

```

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 추

가합니다.

```
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 수행 할 호출을 추가하는 것입니다.

```
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.

```
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.

```
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.

```
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 역참조 하지 않도록 어떻게 다양한 경고를 생성하는지 확인합니다.

## 다음 단계

nullable 참조 형식을 사용할 수 있도록 기존 애플리케이션을 마이그레이션하여 자세히 알아보세요.

[자습서: nullable 참조 형식이 있는 기존 코드 마이그레이션](#)

Entity Framework를 사용할 때 nullable 참조 형식을 사용하는 방법을 알아봅니다.

[Entity Framework Core 기본 사항: nullable 참조 형식 사용](#)

# 자습서: nullable 참조 형식이 있는 기존 코드 마이그레이션

2020-03-18 • 37 minutes to read • [Edit Online](#)

C# 8에서는 nullable 값 형식이 값 형식을 보완하는 것과 동일한 방식으로 참조 형식을 보완하는 nullable 참조 형식이 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. 참조 형식을 갖는 기존 변수는 모두 nullable이 아닌 참조 형식으로 해석됩니다.

이 자습서에서는 다음과 같은 작업을 수행하는 방법을 알아봅니다.

- 코드를 작업할 때 null 참조 검사를 활성화합니다.
- null 값과 각종 경고를 진단하고 수정합니다.
- nullable이 활성화된 컨텍스트와 nullable이 비활성화된 컨텍스트 간의 인터페이스를 관리합니다.
- nullable 주석 컨텍스트를 제어합니다.

## 사전 요구 사항

C# 8.0 컴파일러를 포함하여 .NET Core를 실행하도록 머신을 설정해야 합니다. C# 8 컴파일러는 [Visual Studio 2019 버전 16.3](#) 또는 [.NET CORE 3.0 SDK](#)부터 사용할 수 있습니다.

이 자습서에서는 Visual Studio 또는 .NET Core CLI를 포함하여 C# 및 .NET에 익숙하다고 가정합니다.

## 샘플 애플리케이션 살펴보기

여기서 마이그레이션 할 샘플 애플리케이션은 RSS 피드 리더기 웹앱입니다. 이 애플리케이션은 하나의 RSS 피드를 읽고 가장 최근 기사의 요약을 표시합니다. 표시되는 기사를 선택하여 사이트를 방문할 수 있습니다. 이 애플리케이션은 비교적 최근에 작성되었지만, nullable 참조 형식을 사용할 수 있기 전에 작성되었습니다. 이 애플리케이션에는 바람직한 설계 원칙이 적용되었지만, 이 중요한 언어 기능을 남용하지는 마시기 바랍니다.

샘플 애플리케이션에는 앱의 주요 기능의 유효성을 검사하는 단위 테스트 라이브러리가 포함되어 있습니다. 생성되는 경고에 따라 구현을 조금이라도 변경하는 경우, 이 프로젝트를 사용하여 안전하게 업그레이드할 수 있습니다. 시작 코드는 [dotnet/samples](#) GitHub 리포지토리에서 다운로드할 수 있습니다.

프로젝트를 마이그레이션 할 때 목표는 변수의 nullable 여부를 어떻게 설정할지 명확히 표현할 수 있도록 새로운 언어 기능을 활용하고, nullable 주석 컨텍스트와 nullable 경고 컨텍스트를 `enabled` 로 설정할 때 컴파일러에서 경고가 생성되지 않도록 하는 것이 되어야 합니다.

## 프로젝트를 C# 8로 업그레이드

첫 번째 단계로 마이그레이션 작업의 범위를 확인하는 것이 좋습니다. 시작하려면 먼저 프로젝트를 C# 8.0(이상)으로 업그레이드합니다. 웹 프로젝트의 csproj 파일과 단위 테스트 프로젝트의 csproj 파일에 있는 PropertyGroup에 각각 `LangVersion`  요소를 추가합니다.

```
<LangVersion>8.0</LangVersion>
```

언어 버전을 업그레이드하면 C# 8.0이 선택되지만, nullable 주석 컨텍스트와 nullable 경고 컨텍스트가 활성화 되지는 않습니다. 프로젝트를 다시 빌드하여 경고 없이 빌드되는지 확인합니다.

다음 단계로 nullable 주석 컨텍스트를 켜고 경고가 몇 개나 생성되는지 살펴보는 것이 좋습니다. 솔루션의 두 csproj 파일에서 `LangVersion` 요소 아래에 각각 다음 요소를 추가합니다.

```
<Nullable>enable</Nullable>
```

테스트 빌드를 수행하고 경고 목록을 살펴봅니다. 이 작은 애플리케이션에서 컴파일러가 경고를 5개 생성하는 것을 볼 수 있습니다. 여기서는 nullable 주석 컨텍스트를 활성화 상태로 두고 프로젝트 전체의 경고를 수정할 수 있습니다.

이 방법은 규모가 작은 프로젝트에서만 사용할 수 있습니다. 규모가 큰 프로젝트에서는 전체 코드베이스에서 nullable 주석 컨텍스트를 활성화하여 생성되는 경고의 개수가 많기 때문에 경고를 체계적으로 수정하기가 어렵습니다. 규모가 큰 엔터프라이즈 프로젝트에서는 한 번에 프로젝트 하나씩 마이그레이션하는 것이 좋습니다. 각 프로젝트에서 한 번에 하나의 클래스 또는 파일을 마이그레이션합니다.

## 원래의 설계 의도를 파악하는 데 도움이 되는 경고

2개의 클래스에서 여러 개의 경고가 생성되었습니다. `NewsStoryViewModel` 클래스부터 시작합니다. 경고의 범위를 현재 작업 중인 코드 섹션으로 한정할 수 있도록 두 csproj 파일에서 각각 `Nullable` 요소를 제거합니다.

`NewsStoryViewModel.cs` 파일을 열고 다음 지시문을 추가하여 `NewsStoryViewModel`에서 nullable 주석 컨텍스트를 활성화하고 클래스 정의 후에 이를 복원합니다.

```
#nullable enable
public class NewsStoryViewModel
{
    public DateTimeOffset Published { get; set; }
    public string Title { get; set; }
    public string Uri { get; set; }
}
#nullable restore
```

위의 2개의 지시문은 마이그레이션 노력을 집중하는 데 도움이 됩니다. 현재 작업 중인 코드 영역에 해당하는 nullable 경고가 생성되었습니다. 프로젝트 전체에서 경고를 걸 준비가 될 때까지 일단 그대로 둡니다. 나중에 프로젝트 전체에서 nullable 주석을 켰을 때 컨텍스트가 실수로 비활성화되지 않도록 `disable` 값 대신 `restore` 값을 사용해야 합니다. 프로젝트 전체에서 nullable 주석 컨텍스트를 켠 후에 해당 프로젝트에서 모든 `#nullable` pragma를 제거할 수 있습니다.

`NewsStoryViewModel` 클래스는 DTO(데이터 전송 개체)로, 2개의 속성이 읽기/쓰기 문자열입니다.

```
public class NewsStoryViewModel
{
    public DateTimeOffset Published { get; set; }
    public string Title { get; set; }
    public string Uri { get; set; }
}
```

이 2개의 속성 때문에 `cs8618`, “Non-nullable property is uninitialized”(nullable이 아닌 속성이 초기화되지 않았습니다.)가 발생합니다. 2개의 `string` 속성 모두 `NewsStoryViewModel`이 생성될 때 기본값이 `null`이므로 원인을 쉽게 파악할 수 있습니다. 이때 중요한 것은 `NewsStoryViewModel` 개체가 어떻게 생성되는지 알아내는 것입니다. 클래스를 살펴봐도 `null` 값이 설계의 일부인지 아니면 이러한 개체가 생성될 때마다 해당 개체가 `null`이 아닌 값으로 설정되는 것인지 파악하기가 어렵습니다. 뉴스 기사는 `NewsService` 클래스의 `GetNews` 메서드에서 생성됩니다.

```
ISyndicationItem item = await feedReader.ReadItem();
var newsStory = _mapper.Map<NewsStoryViewModel>(item);
news.Add(newsStory);
```

위 코드 블록에서는 몇 가지 작업이 진행되고 있습니다. 이 애플리케이션은 [AutoMapper](#) NuGet 패키지를 사용하여 `ISyndicationItem` 으로부터 뉴스 항목을 생성합니다. 이 하나의 문에서 뉴스 기사 항목이 생성되고 속성이 설정된다는 사실을 파악했습니다. 따라서 `NewsStoryViewModel` 의 설계 의도는 이러한 속성이 `null` 값을 갖지 않도록 하는 것임을 알 수 있습니다. 이러한 속성은 `nullable`이 아닌 참조 형식이 되어야 합니다. 이렇게 해야 원래 설계 의도가 가장 잘 표현됩니다. 실제로 모든 `NewsStoryViewModel` 이 `null`이 아닌 값으로 올바르게 인스턴스화되었습니다. 그렇다면 다음과 같은 초기화 코드가 유효한 수정이 될 수 있습니다.

```
public class NewsStoryViewModel
{
    public DateTimeOffset Published { get; set; }
    public string Title { get; set; } = default!;
    public string Uri { get; set; } = default!;
}
```

`Title` 과 `Uri`에 `default` (`string` 형식의 경우 `null`)를 할당해도 프로그램의 런타임 동작이 변경되지 않습니다. `NewsStoryViewModel` 은 전과 동일하게 `null` 값으로 생성되지만, 이제 컴파일러가 경고를 보고하지 않습니다. `default` 식 뒤에 `null` 허용 연산자인 `!` 문자가 오기 때문에 컴파일러는 선행 식이 `null`이 아님을 알 수 있습니다. 이 방법은 다른 변경 사항을 수행하는 경우 코드 베이스에 훨씬 더 많은 변경이 적용될 때는 유용하게 사용할 수 있지만, 이 애플리케이션에서는 보다 빠르고 효과적인 솔루션이 있습니다. 바로 `NewsStoryViewModel` 을 모든 속성이 생성자에서 설정되는, 변경이 불가능한 형식으로 만드는 것입니다. `NewsStoryViewModel` 에 다음 변경 내용을 적용합니다.

```
#nullable enable
public class NewsStoryViewModel
{
    public NewsStoryViewModel(DateTimeOffset published, string title, string uri) =>
        (Published, Title, Uri) = (published, title, uri);

    public DateTimeOffset Published { get; }
    public string Title { get; }
    public string Uri { get; }
}
(nullable restore)
```

그런 다음, [AutoMapper](#)를 구성하는 코드가 속성을 설정하는 대신 생성자를 사용하도록 수정해야 합니다.

`NewsService.cs` 를 열고 파일 하단에서 다음 코드를 찾습니다.

```
public class NewsStoryProfile : Profile
{
    public NewsStoryProfile()
    {
        // Create the AutoMapper mapping profile between the 2 objects.
        // ISyndicationItem.Id maps to NewsStoryViewModel.Uri.
        CreateMap<ISyndicationItem, NewsStoryViewModel>()
            .ForMember(dest => dest.Uri, opts => opts.MapFrom(src => src.Id));
    }
}
```

이 코드는 `ISyndicationItem` 개체의 속성을 `NewsStoryViewModel` 속성에 매핑합니다. [AutoMapper](#)가 대신 생성자를 사용하여 매핑을 수행하도록 수정해야 합니다. 위 코드를 다음 automapper 구성으로 바꿉니다.

```

#ifndef nullable enable
public class NewsStoryProfile : Profile
{
    public NewsStoryProfile()
    {
        // Create the AutoMapper mapping profile between the 2 objects.
        // ISyndicationItem.Id maps to NewsStoryViewModel.Uri.
        CreateMap<ISyndicationItem, NewsStoryViewModel>()
            .ForCtorParam("uri", opt => opt.MapFrom(src => src.Id));
    }
}

```

이 클래스는 규모가 작고 신중히 검사했기 때문에 이 클래스 선언 위에서 `#nullable enable` 지시문을 켜야 합니다. 생성자를 변경한 결과 무언가 잘못되었을 수 있으므로 계속 진행하기 전에 먼저 모든 테스트를 실행하여 애플리케이션을 테스트하는 것이 좋습니다.

지금까지의 작업을 통해 원래 설계 의도가 변수를 `null`로 설정하지 않는 것인지 확인하는 방법을 알아보았습니다. 이 방법을 **correct by construction**(생성에 의한 올바름)이라고 부릅니다. 개체와 개체의 속성을 생성할 때 이것이 `null`이 될 수 없다고 선언하는 것입니다. 컴파일러는 해당 속성이 생성 후에 `null`로 설정되지 않도록 허름 분석을 수행합니다. 이 생성자는 외부 코드에 의해 호출되는 것을 볼 수 있는데, 이 코드는 `nullable` 까지 불가입니다. 새로운 구문에서는 런타임 검사를 제공하지 않습니다. 따라서 외부 코드가 컴파일러의 허름 분석을 피해갈 수 있습니다.

클래스의 구조가 설계 의도에 대한 또 다른 힌트를 주는 경우도 있습니다. *Pages* 폴더에서 *Error.cshtml.cs* 파일을 엽니다. `ErrorViewModel`에 다음 코드가 포함되어 있습니다.

```

public class ErrorModel : PageModel
{
    public string RequestId { get; set; }

    public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);

    public void OnGet()
    {
        RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;
    }
}

```

클래스 선언 앞에 `#nullable enable` 지시문을 추가하고, 클래스 선언 뒤에 `#nullable restore` 지시문을 추가합니다. `RequestId` 가 초기화되지 않았다는 경고가 표시됩니다. 클래스를 살펴보니 `RequestId` 속성은 경우에 따라 `null`이 되어야 한다는 사실을 알 수 있습니다. `ShowRequestId` 속성이 존재한다는 사실이 누락된 값이 있을 수 있음을 나타냅니다. `null`이 유효하므로 `string` 형식에 `?` 을 추가하여 `RequestId` 속성이 'nullable 참조 형식'임을 나타냅니다. 다음은 완성된 클래스입니다.

```

#ifndef nullable enable
public class ErrorModel : PageModel
{
    public string? RequestId { get; set; }

    public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);

    public void OnGet()
    {
        RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier;
    }
}
#endif nullable restore

```

속성이 사용되는 방식을 살펴보면 표시에서 속성이 렌더링되기 전에 연결된 페이지에서 속성이 `null`인지 검사

되는 것을 알 수 있습니다. 이처럼 nullable 참조 형식이 안전하게 사용되고 있으므로 이 클래스는 완성되었습니다.

## null을 수정하면 변경이 발생하는 경우

하나의 경고 세트를 수정하면 관련 코드에서 새로운 경고가 생성되는 경우가 있습니다. `index.cshtml.cs` 클래스를 수정하여 실제로 경고를 살펴보겠습니다. `index.cshtml.cs` 파일을 열고 코드를 살펴봅니다. 이 파일에는 인덱스 페이지를 구동하는 코드가 포함되어 있습니다.

```
public class IndexModel : PageModel
{
    private readonly NewsService _newsService;

    public IndexModel(NewsService newsService)
    {
        _newsService = newsService;
    }

    public string ErrorText { get; private set; }

    public List<NewsStoryViewModel> NewsItems { get; private set; }

    public async Task OnGet()
    {
        string feedUrl = Request.Query["feedurl"];

        if (!string.IsNullOrEmpty(feedUrl))
        {
            try
            {
                NewsItems = await _newsService.GetNews(feedUrl);
            }
            catch (UriFormatException)
            {
                ErrorText = "There was a problem parsing the URL.";
                return;
            }
            catch (WebException ex) when (ex.Status == WebExceptionStatus.NameResolutionFailure)
            {
                ErrorText = "Unknown host name.";
                return;
            }
            catch (WebException ex) when (ex.Status == WebExceptionStatus.ProtocolError)
            {
                ErrorText = "Syndication feed not found.";
                return;
            }
            catch (AggregateException ae)
            {
                ae.Handle((x) =>
                {
                    if (x is XmlException)
                    {
                        ErrorText = "There was a problem parsing the feed. Are you sure that URL is a syndication feed?";
                        return true;
                    }
                    return false;
                });
            }
        }
    }
}
```

`#nullable enable` 지시문을 추가하면 `ErrorText` 속성과 `NewsItems` 속성이 초기화되지 않았다는 2개의 경고를 볼 수 있습니다. 이 클래스를 살펴보면 두 속성 모두 nullable 참조 형식이 되어야 한다는 사실을 알 수 있습니다. 두 속성 모두 `private setter`를 갖고 있기 때문입니다. `OnGet` 메서드에서 정확히 1개가 할당되어 있습니다. 변경을 수행하기 전에 두 속성을 사용하는 주체를 찾아보겠습니다. 페이지에서 오류 표시가 생성되기 전에 `ErrorText` 가 `null`에 대해 검사됩니다. `NewsItems` 컬렉션이 `null`에 대해 검사되고, 컬렉션에 항목이 있는지도 검사됩니다. 두 속성을 모두 nullable 참조 형식으로 만들어서 빠르게 수정할 수도 있지만, 이보다 나은 방법은 컬렉션을 nullable이 아닌 참조 형식으로 만들고, 뉴스를 가져올 때 기존 컬렉션에 항목을 추가하는 것입니다. 먼저 `ErrorText` 의 `string` 형식에 `?`를 추가합니다.

```
public string? ErrorText { get; private set; }
```

이 `ErrorText` 속성에 대한 모든 액세스가 이미 `null` 검사에 의해 보호되었으므로 이 변경 사항은 다른 코드로 전파되지 않습니다. 다음으로 `NewsItems` 목록을 초기화하고 속성 `setter`를 제거하여 읽기 전용 속성으로 만듭니다.

```
public List<NewsStoryViewModel> NewsItems { get; } = new List<NewsStoryViewModel>();
```

이렇게 하면 경고는 해결되지만 오류가 발생합니다. `NewsItems` 목록은 이제 **correct by construction**(생성에 의한 올바름)이지만, `OnGet`에서 목록을 설정하는 코드가 새 API에 대응되도록 변경해야 합니다. 할당을 사용하는 대신 `AddRange`를 호출하여 기존 목록에 뉴스 항목을 추가합니다.

```
NewsItems.AddRange(await _newsService.GetNews(feedUrl));
```

할당 대신 `AddRange`를 사용한다는 것은 `GetNews` 메서드가 `List` 대신 `IEnumerable`을 반환할 수 있음을 의미합니다. 이렇게 하면 할당이 하나 절약됩니다. 다음 코드 샘플과 같이 메서드의 시그니처를 변경하고 `ToList` 호출을 제거합니다.

```

public async Task<IEnumerable<NewsStoryViewModel>> GetNews(string feedUrl)
{
    var news = new List<NewsStoryViewModel>();
    var feedUri = new Uri(feedUrl);

    using (var xmlReader = XmlReader.Create(feedUri.ToString(),
        new XmlReaderSettings { Async = true }))
    {
        try
        {
            var feedReader = new RssFeedReader(xmlReader);

            while (await feedReader.Read())
            {
                switch (feedReader.ElementType)
                {
                    // RSS Item
                    case SyndicationElementType.Item:
                        ISyndicationItem item = await feedReader.ReadItem();
                        var newsStory = _mapper.Map<NewsStoryViewModel>(item);
                        news.Add(newsStory);
                        break;

                    // Something else
                    default:
                        break;
                }
            }
        }
        catch (AggregateException ae)
        {
            throw ae.Flatten();
        }
    }

    return news.OrderByDescending(story => story.Published);
}

```

시그니처를 변경하면 테스트 중 하나도 중단됩니다. `SimpleFeedReader.Tests` 프로젝트의 `Services` 폴더에서 `NewsServiceTests.cs` 파일을 엽니다. `Returns_News_Stories_Given_Valid_Uri` 테스트로 이동하고 `result` 변수의 형식을 `IEnumerable<NewsItem>` 으로 변경합니다. 형식을 변경한다는 것은 `Count` 속성을 더 이상 사용할 수 없음을 의미하므로, `Assert` 의 `Count` 속성을 `Any()` 호출로 변경합니다.

```

// Act
IEnumerable<NewsStoryViewModel> result =
    await _newsService.GetNews(feedUrl);

// Assert
Assert.True(result.Any());

```

파일의 시작 부분에 `using System.Linq` 문도 추가해야 합니다.

지금까지 제네릭 인스턴스화를 포함하는 코드를 업데이트할 때 특히 유의해야 하는 고려 사항을 살펴보았습니다. 목록과 목록의 요소는 모두 nullable이 아닌 형식입니다. 둘 중 하나 또는 둘 다 nullable 형식이 될 수 있습니다. 다음 지시문이 모두 허용됩니다.

- `List<NewsStoryViewModel>` : nullable이 아닌 보기 모델의 nullable이 아닌 목록
- `List<NewsStoryViewModel?>` : nullable 보기 모델의 nullable이 아닌 목록
- `List<NewsStoryViewModel?>` : nullable이 아닌 보기 모델의 nullable 목록
- `List<NewsStoryViewModel?>?` : nullable 보기 모델의 nullable 목록

## 외부 코드와의 인터페이스

지금까지 `NewsService` 클래스를 변경했습니다. 이번에는 이 클래스에서 `#nullable enable` 주석을 캡니다. 이렇게 해도 새로운 경고가 생성되지 않습니다. 그렇지만 클래스를 주의 깊게 살펴보면 컴파일러의 흐름 분석에 어떤 제한이 있는지 알 수 있습니다. 생성자를 살펴봅니다.

```
public NewsService(IMapper mapper)
{
    _mapper = mapper;
}
```

`IMapper` 매개 변수는 nullable이 아닌 참조 형식입니다. ASP.NET Core 인프라 코드에 의해 호출되므로, 컴파일러는 `IMapper` 가 null이 될 수 없다는 것을 알지 못합니다. 기본적인 ASP.NET Core DI(종속성 주입) 컨테이너는 필수 서비스를 확인할 수 없는 경우 예외를 throw하므로 이 코드는 올바릅니다. nullable 주석 컨텍스트를 활성화한 상태로 코드를 컴파일하더라도 컴파일러는 모든 공용 API 호출의 유효성을 검사할 수 없습니다. 또한, nullable 참조 형식을 사용하도록 설정되지 않은 프로젝트에 의해 라이브러리가 과도하게 사용될 수 있습니다. nullable이 아닌 형식으로 선언한 경우에도 공용 API의 입력은 유효성을 검사해야 합니다.

## 코드 가져오기

초기 테스트 컴파일에서 확인한 경고를 모두 수정했으므로 이제 두 프로젝트에서 모두 nullable 주석 컨텍스트를 결 수 있습니다. 프로젝트를 다시 빌드하면 컴파일러에서 생성하는 경고가 없는 것을 볼 수 있습니다. 완성된 프로젝트의 코드는 [dotnet/samples GitHub 리포지토리](#)에서 가져올 수 있습니다.

nullable 참조 형식을 지원하는 새로운 기능을 사용하면 코드에서 `null` 값을 처리하는 방식의 잠재적인 오류를 찾아서 수정할 수 있습니다. nullable 주석 컨텍스트를 활성화하면 어떤 변수는 null이 되면 안 되고, 어떤 변수는 null 값을 포함해도 된다는 설계 의도를 원하는 대로 표현할 수 있습니다. 이러한 기능을 사용하여 설계 의도를 보다 쉽게 드러낼 수 있습니다. 마찬가지로, nullable 경고 컨텍스트는 의도가 위반된 경우 경고를 발생하라고 컴파일러에 알립니다. 이러한 경고를 살펴보고 업데이트하여 코드의 복원력을 높이고 실행 중에

`NullReferenceException` 을 throw할 확률을 줄일 수 있습니다. 이러한 컨텍스트의 범위를 제어하여 나머지 코드 베이스는 그대로 두고 마이그레이션 할 로컬 코드 영역에만 집중할 수 있습니다. 실전에서는 클래스의 정기적인 유지 관리의 일환으로 이러한 마이그레이션 작업을 수행할 수 있습니다. 이 자습서에서는 nullable 참조 형식을 사용하도록 애플리케이션을 마이그레이션하는 과정을 살펴봤습니다. [NodaTime](#)에 nullable 참조 형식을 통합하도록 구현된 PR [Jon Skeet](#)에서 더 많은 실제 사례를 참조할 수 있습니다. 또는 [Entity Framework Core - nullable 참조 형식 사용](#)에서 Entity Framework Core에 nullable 참조 형식을 사용하는 방법을 배울 수도 있습니다.

# 자습서: C# 8.0 및 .NET Core 3.0을 사용하여 비동기 스트림 생성 및 사용

2020-11-02 • 24 minutes to read • [Edit Online](#)

C# 8.0에서는 데이터의 스트리밍 소스를 모델링하는 비동기 스트림을 도입합니다. 데이터 스트림은 종종 요소를 비동기적으로 검색하거나 생성합니다. 비동기 스트림은 .NET Standard 2.1에 도입된 새 인터페이스를 사용합니다. 이 인터페이스는 .NET Core 3.0 이상에서 지원됩니다. 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# 8.0 컴파일러를 포함하여 .NET Core를 실행하도록 컴퓨터를 설정해야 합니다. C# 8 컴파일러는 [Visual Studio 2019 버전 16.3](#) 또는 [.NET CORE 3.0 SDK](#)부터 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.

### WARNING

개인용 액세스 토큰을 안전하게 보관합니다. 개인용 액세스 토큰이 있는 소프트웨어는 액세스 권한을 사용하여 GitHub API 호출을 수행할 수 있습니다.

이 자습서에서는 Visual Studio 또는 .NET Core CLI를 포함하여 C# 및 .NET에 익숙하다고 가정합니다.

## 시작 애플리케이션 실행

[csharp/tutorials/AsyncStreams](#) 폴더의 [dotnet/docs](#) 리포지토리에서 이 자습서에 사용된 시작 애플리케이션의 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.

```

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.

```

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"];
}

```

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 사용합니

다.

- [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 `IAsyncEnumerable<JToken>`을 반환하고 다음 코드에 표시된 대로 매개 변수 목록에서 취소 토큰 및 진행 개체를 제거합니다.

```
private static async IAsyncEnumerable<JToken> runPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
```

시작 코드는 다음 코드에 표시된 대로 페이지가 검색될 때 각 페이지를 처리합니다.

```
finalResults.Merge(issues(results)["nodes"]);
progress?.Report(issuesReturned);
cancel.ThrowIfCancellationRequested();
```

해당 세 줄을 다음 코드로 바꿉니다.

```
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.

```

private static async IAsyncEnumerable<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.

```

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.

```
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.

```
int num = 0;
var enumerator = runPagedQueryAsync(client, PagedIssueQuery, "docs").GetEnumeratorAsync();
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.

```

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"];
}

```

[IEnumeratorCancellationAttribute](#) 특성을 사용하면 컴파일러는 비동기 반복기 본문에 표시되는

`GetAsyncEnumerator`에 전달된 토큰을 해당 인수로 만드는 `IAsyncEnumerator<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.

```

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");
    }
}

```

[csharp/tutorials/AsyncStreams](#) 풀더의 [dotnet/docs](#) 리포지토리에서 완료된 자습서의 코드를 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 추적하는데 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.

# 자습서: 패턴 일치를 사용하여 형식 기반 및 데이터 기반 알고리즘 빌드

2021-02-18 • 38 minutes to read • [Edit Online](#)

C# 7에서는 기본 패턴 일치 기능을 도입했습니다. 새로운 식과 패턴을 포함하여 C# 8 및 C# 9에서 기능을 확장했습니다. 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

C# 9 컴파일러를 포함하는 .NET 5를 실행하도록 머신을 설정해야 합니다. C# 9 컴파일러는 [Visual Studio 2019 버전 16.9 미리 보기 1](#) 또는 [.NET 5.0 SDK](#)부터 사용할 수 있습니다.

이 자습서에서는 Visual Studio 또는 .NET Core CLI를 포함하여 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 민들어야 합니다. 그런 다음, 애플리케이션은 이 새로운 형식을 사용하고, 상속 계층 구조를 빌드하고, 가상 메서드를 만들고, 추상화를 구현합니다. 해당 기술이 적용되고 때때로 가장 적합한 도구가 됩니다. 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 일치가 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.

```

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 있습니다. `System.Object` 이외의 공용 기본 클래스는 활용할 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 작업을 분리해서 설명하면 C#의 패턴 일치 기능을 더 쉽게 사용할 수 있습니다.

## 기본 통행료 계산 구현

가장 기본적인 통행료 계산에는 차량 형식만 사용됩니다.

- `Car`은 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.

```

using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace toll_calculator
{
    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", paramName:
nameof(vehicle)),
            null            => throw new ArgumentNullException(nameof(vehicle))
        };
    }
}

```

앞의 코드는 형식 패턴을 테스트하는 `switch` 식(`switch` 문과 같지 않음)을 사용합니다. `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 `null`이 이 메서드에 전달될 때 `null` 패턴이 검색됩니다. 다른 형식 패턴은 올바른 형식의 `null`이 아닌 개체와만 일치하므로 `null` 패턴이 마지막일 수 있습니다.

`Program.cs`에서 다음 코드를 사용하여 이 코드를 테스트할 수 있습니다.

```

using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace toll_calculator
{
    class Program
    {
        static void Main(string[] args)
        {
            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 패턴을 사용하여 구현할 수 있습니다. 속성 패턴은 속성 값을 상수 값과 비교하는 `when` 절입니다. 속성 패턴은 형식이 결정된 후 개체의 속성을 검사합니다. `Car` 사례 1개는 다른 사

례 4개로 확장됩니다.

```
vehicle switch
{
    Car {Passengers: 0}      => 2.00m + 0.50m,
    Car {Passengers: 1}      => 2.0m,
    Car {Passengers: 2}      => 2.0m - 0.50m,
    Car c                   => 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.

```
vehicle switch
{
    // ...

    Taxi {Fares: 0}  => 3.50m + 1.00m,
    Taxi {Fares: 1}  => 3.50m,
    Taxi {Fares: 2}  => 3.50m - 0.50m,
    Taxi t           => 3.50m - 1.00m,

    // ...
};
```

앞의 예제에서는 `when` 절이 마지막 사례에서 생략되었습니다.

다음으로, 다음 예제와 같이 버스 사례를 확장하여 점유 규칙을 구현합니다.

```
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 b => 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.

```

vehicle switch
{
    // ...

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck t => 10.00m,
};

}

```

앞의 코드는 스위치 암(arm)의 `when` 절을 표시합니다. `when` 절을 사용하여 속성에서 다음 이외의 조건을 테스트합니다. 작업을 마치면 다음 코드와 같은 메서드가 생성됩니다.

```

vehicle switch
{
    Car {Passengers: 0}      => 2.00m + 0.50m,
    Car {Passengers: 1}      => 2.0m,
    Car {Passengers: 2}      => 2.0m - 0.50m,
    Car c                   => 2.00m - 1.0m,

    Taxi {Fares: 0}    => 3.50m + 1.00m,
    Taxi {Fares: 1}    => 3.50m,
    Taxi {Fares: 2}    => 3.50m - 0.50m,
    Taxi t             => 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))
};

}

```

대부분의 이 스위치 암(arm)은 재귀 패턴의 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 둘 다 앞의 예제에 있는 서로 다른 암(arm) 4개가 포함됩니다. 두 경우에 모두 속성 패턴에 제공되는 형식 패턴을 만들 수 있습니다. 이 기술이 다음 코드에 나옵니다.

```

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. 다음 코드를 사용하여 `if` 및 `else` 문으로 표현하는 경우 계열을 사용할 수 있습니다.

```

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 일치식을 빌드할 수 있습니다. 복잡한 식이 생성됩니다. 읽기 힘들고 이해하기 어려운 식입니다. 따라서 식의 정확성을 보장하기 어렵습니다. 대신, 해당 메서드를 결합하여 모든 상태를 간결하게 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 단순화합니다.

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.

```
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.

```

private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Saturday => false,
        DayOfWeek.Sunday => false,
        _ => true
    };

```

다음으로, 시간을 블록으로 분류하는 비슷한 함수를 추가합니다.

```

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 'or 결합 패턴'을 사용합니다. 두 패턴은 모두 C# 9.0에서 추가된 것입니다. 관계형 패턴을 사용하면 `<`, `>`, `<=` 또는 `>=`로 숫자 값을 테스트할 수 있습니다. `or` 패턴은 식이 하나 이상의 패턴과 일치하는지 테스트 합니다. `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 16 개 암(arm)을 사용하여 `switch` 식을 작성할 수 있습니다.

```

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.

```
(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.

```
(true, TimeBand.OVERNIGHT, _) => 0.75m,  
(true, TimeBand.DAYTIME, _) => 1.5m,
```

해당하는 두 줄이 변경된 후 코드는 다음과 같이 표시됩니다.

```
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.

```
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.

# 콘솔 앱

2020-11-02 • 32 minutes to read • [Edit Online](#)

이 자습서에서는 .NET Core 및 C# 언어의 다양한 기능에 대해 설명합니다. 다음에 대해 알아봅니다.

- .NET Core 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 작은) 또는 '>'(보다 큼) 키를 눌러 속도를 높이거나 낮출 수 있습니다.

이 자습서에는 많은 기능이 있습니다. 하나씩 빌드해 보겠습니다.

## 사전 요구 사항

- .NET Core를 실행하도록 컴퓨터를 설정합니다. [.NET Core 다운로드](#) 페이지에서 설치 지침을 찾을 수 있습니다. Windows, Linux, macOS 또는 Docker 컨테이너에서 이 애플리케이션을 실행할 수 있습니다.
- 선호하는 코드 편집기를 설치합니다.

## 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.

파일 수정을 시작하기 전에 간단한 Hello World 애플리케이션을 실행하는 단계를 진행해 보겠습니다. 애플리케이션을 만든 후에 명령 프롬프트에서 `dotnet restore` 를 입력합니다. 이 명령은 NuGet 패키지 복원 프로세스를 실행합니다. NuGet은 .NET 패키지 관리자입니다. 이 명령은 프로젝트에 대한 누락된 종속성 중 하나를 다운로드합니다. 이 프로젝트는 새 프로젝트이므로 어떤 종속성도 없습니다. 따라서 처음 실행하면 .NET Core 프레임워크가 다운로드됩니다. 이 초기 단계 후에 새 종속 패키지를 추가하거나 종속성 버전을 업데이트할 때 `dotnet restore` 를 실행하기만 하면 됩니다.

`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](#) 설명서를 참조하세요.

패키지를 복원한 후 `dotnet build` 를 실행합니다. 이렇게 하면 빌드 엔진이 실행되고 애플리케이션 실행 파일이 만들어집니다. 마지막으로 `dotnet run` 을 실행하여 애플리케이션을 실행합니다.

이 간단한 Hello World 애플리케이션 코드은 모두 `Program.cs`에 있습니다. 원하는 텍스트 편집기를 사용하여 해당 파일을 엽니다. 이제 첫 번째 변경을 수행해 보겠습니다. 파일 맨 위에서 `using` 문을 보세요.

```
using System;
```

이 문은 `System` 네임스페이스의 모든 형식이 범위 내에 있음을 컴파일러에 알려줍니다. 사용해본 적이 있을 수 다른 개체 지향 언어와 마찬가지로 C#에서도 네임스페이스를 사용하여 형식을 구성합니다. 이 Hello World 프로그램도 다르지 않습니다. 프로그램이 현재 디렉터리의 이름에 기반한 이름으로 네임스페이스에 묶인 것을 볼 수 있습니다. 이 자습서에서는 네임스페이스 이름을 `TeleprompterConsole`로 변경해 보겠습니다.

```
namespace TeleprompterConsole
```

## 파일 읽기 및 쓰기

추가할 첫 번째 기능은 텍스트 파일을 읽고 콘솔에 해당 텍스트를 모두 표시하는 기능입니다. 먼저 텍스트 파일을 추가해 보겠습니다. 이 샘플에 대한 GitHub 리포지토리의 `sampleQuotes.txt` 파일을 프로젝트 디렉터리로 복사합니다. 이 파일은 애플리케이션에 대한 스크립트로 작동합니다. 이 항목에 대한 샘플 앱을 다운로드하는 방법에 대한 정보를 원하는 경우 [샘플 및 자습서](#) 항목의 지침을 참조하세요.

다음에는 다음 메서드를 `Program` 클래스에 추가합니다(`Main` 메서드 바로 아래).

```
static IEnumerable<string> ReadFrom(string file)
{
    string line;
    using (var reader = File.OpenText(file))
    {
        while ((line = reader.ReadLine()) != null)
        {
            yield return line;
        }
    }
}
```

이 메서드는 두 개의 새 네임스페이스에 있는 형식을 사용합니다. 이를 컴파일하려면 파일 맨 위에 다음 두 줄을 추가해야 합니다.

```
using System.Collections.Generic;
using System.IO;
```

`IEnumerable<T>` 인터페이스는 `System.Collections.Generic` 네임스페이스에 정의되어 있습니다. `File` 클래스는 `System.IO` 네임스페이스에 있습니다.

이 메서드는 [반복기 메서드](#)라는 특수한 형식의 C# 메서드입니다. 열거자 메서드는 자연 계산되는 시퀀스를 반환합니다. 즉, 시퀀스의 각 항목은 시퀀스를 사용하는 코드에서 요청된 것처럼 생성됩니다. 열거자 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 다른 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 해

당하는 `OpenText(String)` 메서드의 반환 값입니다.

이제 `Main` 메서드에서 파일을 읽는 코드를 채웁니다.

```
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 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.

```
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.

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

`Task` 클래스는 `System.Threading.Tasks` 네임스페이스에 있으므로 해당 `using` 문을 파일 맨 위에 추가해야 합니다.

```
using System.Threading.Tasks;
```

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.

```
var lineLength = 0;
```

그린 후 `yield return word + " ";` 문 뒤에(닫는 중괄호 이전) 다음 코드를 추가합니다.

```
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 마지막에 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.

```
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.

**Main** 메서드에서 다음 새 메서드를 호출할 수 있습니다.

```
ShowTeleprompter().Wait();
```

여기 **Main**에서 코드는 동기적으로 대기합니다. 가능한 경우 동기적으로 대기하는 대신 **await** 연산자를 사용하는 것이 좋습니다. 그러나 콘솔 애플리케이션의 **Main** 메서드에서는 **await** 연산자를 사용할 수 없습니다. 결과적으로 모든 작업을 완료하기 전에 애플리케이션이 종료됩니다.

### NOTE

C# 7.1 이상을 사용하는 경우 **async** **Main** 메서드로 콘솔 애플리케이션을 만들 수 있습니다.

다음에는 콘솔에서 읽는 두 번째 비동기 메서드를 작성하고 '<'(보다 작음), '>'(보다 큼) 및 'X' 또는 'x' 키를 확인해야 합니다. 해당 작업에 대해 추가하는 메서드는 다음과 같습니다.

```

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);
}

```

이 메서드는 콘솔에서 키를 읽고 사용자가 '<'(보다 작음) 또는 '>'(보다 큼) 키를 누를 때 지연을 나타내는 지역 변수를 설정하는 `Action` 대리자를 나타내는 람다 식을 만듭니다. 사용자가 'X' 또는 'x' 키를 누를 때 대리자 메서드가 완료되면 사용자가 언제든지 텍스트 표시를 중지할 수 있습니다. 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.

```

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;
        }
    }
}

```

해당 클래스를 새 파일에 추가하고 위와 같이 `TeleprompterConsole` 네임스페이스에 클래스를 포함합니다. 또한 바깥쪽 클래스 또는 네임스페이스 이름 없이 `Min` 및 `Max` 메서드를 참조할 수 있도록 `using static` 문을 추가해야 합니다. `using static` 문은 하나의 클래스에서 메서드를 가져옵니다. 이것은 네임스페이스에서 모든 클래스를 가져오는 지금까지 사용했던 `using` 문과는 반대됩니다.

```
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.

```
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 자연을 위해 `config` 개체를 사용하도록 `ShowTeleprompter` 및 `GetInput` 메서드를 모두 업데이트해야 합니다.

```
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.

```
RunTeleprompter().Wait();
```

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 Core CLI에 대해 살펴보았습니다.

파일 I/O에 대한 자세한 내용은 [파일 및 스트림 I/O](#) 항목을 참조하세요. 이 자습서에서 사용된 비동기 프로그래밍 모델에 대한 자세한 내용은 [작업 기반 비동기 프로그래밍](#) 항목 및 [비동기 프로그래밍](#) 항목을 참조하세요.

# REST 클라이언트

2021-02-18 • 32 minutes to read • [Edit Online](#)

이 자습서에서는 .NET Core 및 C# 언어의 다양한 기능에 대해 설명합니다. 다음에 대해 알아봅니다.

- .NET Core CLI의 기본 사항
- C# 언어 기능의 개요
- NuGet으로 종속성 관리
- HTTP 통신
- JSON 정보 처리
- 특성을 사용하여 구성 관리

GitHub에서 REST 서비스에 HTTP 요청을 실행하는 애플리케이션을 빌드합니다. JSON 형식의 정보를 읽은 후 해당 JSON 패킷을 C# 개체로 변환합니다. 마지막으로 C# 개체를 사용하는 방법을 배웁니다.

이 자습서에는 많은 기능이 있습니다. 하나씩 빌드해 보겠습니다.

이 문서의 [최종 샘플](#)을 따르려는 경우 해당 샘플을 다운로드할 수 있습니다. 다운로드 지침은 [샘플 및 자습서](#)를 참조하세요.

## 사전 요구 사항

.NET Core를 실행하려면 머신을 설정해야 합니다. [.NET Core 다운로드](#) 페이지에서 설치 지침을 찾을 수 있습니다. Windows, Linux, macOS에서나 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. 콘솔 창에 다음 명령을 입력합니다.

```
dotnet new console --name WebAPIClient
```

이렇게 하면 기본 "Hello World" 애플리케이션에 대한 시작 파일이 만들어집니다. 프로젝트 이름은 "WebAPIClient"입니다. 새 프로젝트이므로 어떤 종속성도 없습니다. 첫 번째 실행에서는 .NET Core 프레임워크를 다운로드하고, 개발 인증서를 설치하며, NuGet 패키지 관리자를 실행하여 누락된 종속성을 복원합니다.

수정하기 전에 "WebAPIClient" 디렉터리에 `cd`하고 명령 프롬프트에서 `dotnet run` ([참고 참조](#))를 입력하여 애플리케이션을 실행합니다. 환경에 누락된 종속성이 있으면 `dotnet run`이 자동으로 `dotnet restore`를 수행합니다. 애플리케이션을 다시 빌드해야 하면 `dotnet build`도 수행합니다. 최초 설치 후에는 프로젝트에 해당할 때만 `dotnet restore` 또는 `dotnet build`를 실행하면 됩니다.

## 새 종속성 추가

.NET Core의 주요 디자인 목표 중 하나는 .NET 설치의 크기를 최소화하는 것입니다. 애플리케이션에 일부 기능을 위해 추가 라이브러리가 필요한 경우 C# 프로젝트(\*.csproj) 파일에 해당 종속성을 추가합니다. 현재 예제에서는 애플리케이션이 JSON 응답을 처리할 수 있도록 `System.Runtime.Serialization.Json` 패키지를 추가해야 합니다.

이 애플리케이션에는 `System.Runtime.Serialization.Json` 패키지가 필요합니다. 다음 .NET CLI 명령을 실행하여 패키지를 프로젝트에 추가합니다.

```
dotnet add package System.Text.Json
```

## 웹 요청 수행

이제 웹에서 데이터 검색을 시작할 준비가 되었습니다. 이 애플리케이션에서는 GitHub API에서 정보를 읽게 됩니다. .NET Foundation 상위 항목 아래에서 프로젝트에 대한 정보를 읽어 보겠습니다. 먼저 GitHub API에 대해 요청을 수행하여 프로젝트에 대한 정보를 검색합니다. 사용할 엔드포인트는 <https://api.github.com/orgs/dotnet/repos>입니다. 이러한 프로젝트에 대해 모든 정보를 검색하려고 하므로 HTTP GET 요청을 사용합니다. 브라우저도 HTTP GET 요청을 사용하므로 해당 URL을 브라우저에 붙여 넣어 수신되고 처리 중인 정보를 볼 수 있습니다.

`HttpClient` 클래스를 사용하여 웹 요청을 수행합니다. 모든 최신 .NET API와 마찬가지로 `HttpClient`는 장기 실행되는 API에 대해 비동기 메서드만 지원합니다. 먼저 비동기 메서드를 만들어 보겠습니다. 애플리케이션의 기능을 빌드할 때 구현을 채웁니다. 먼저 프로젝트 디렉터리에서 `program.cs` 파일을 열고 `Program` 클래스에 다음 메서드를 추가합니다.

```
private static async Task ProcessRepositories()
{
}
```

C# 컴파일러에서 `Task` 형식을 인식하도록 `using` 지시문을 `Main` 메서드 맨 위에 추가해야 합니다.

```
using System.Threading.Tasks;
```

이때 프로젝트를 빌드하면 이 메서드에는 `await` 연산자가 없어서 동기적으로 실행되므로 경고가 생성됩니다. 지금은 이 경고를 무시하세요. 메서드를 입력할 때 `await` 연산자를 추가할 것입니다.

다음으로 `Main` 메서드를 업데이트하여 `ProcessRepositories` 메서드를 호출합니다. `ProcessRepositories` 메서드는 작업을 반환합니다. 이 작업이 완료되기 전에 프로그램을 종료하지 않아야 합니다. 따라서 `Main`의 서명을 변경해야 합니다. `async` 한정자를 추가하고 반환 형식을 `Task`로 변경합니다. 그런 다음 메서드의 본문에서 `ProcessRepositories`에 대한 호출을 추가합니다. 해당 메서드 호출에 `await` 키워드를 추가합니다.

```
static async Task Main(string[] args)
{
    await ProcessRepositories();
}
```

이제 아무 작업도 수행하지 않지만 비동기적으로는 수행하는 프로그램이 되었습니다. 이것을 개선해보겠습니다.

먼저 웹에서 데이터를 검색할 수 있는 개체가 필요합니다. 이를 위해 `HttpClient`를 사용할 수 있습니다. 이 개체는 요청 및 응답을 처리합니다. `Program.cs` 파일 내의 `Program` 클래스에서 해당 형식의 단일 인스턴스를 인스턴스화합니다.

```

namespace WebAPIClient
{
    class Program
    {
        private static readonly HttpClient client = new HttpClient();

        static async Task Main(string[] args)
        {
            //...
        }
    }
}

```

다시 `ProcessRepositories` 메서드로 돌아가 첫 번째 버전을 채워 보겠습니다.

```

private static async Task ProcessRepositories()
{
    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 stringTask = client.GetStringAsync("https://api.github.com/orgs/dotnet/repos");

    var msg = await stringTask;
    Console.WriteLine(msg);
}

```

컴파일을 위해 파일 맨 위에 2개의 새 `using` 지시문도 추가해야 합니다.

```

using System.Net.Http;
using System.Net.Http.Headers;

```

이 첫 번째 버전은 dotnet foundation 조직의 모든 리포지토리 목록을 읽으라는 웹 요청을 수행합니다. (.NET Foundation의 GitHub ID는 `dotnet`임) 첫 몇 줄은 이 요청에 대해 `HttpClient`를 설정합니다. 먼저 GitHub JSON 응답을 수락하도록 구성됩니다. 이 형식은 단순히 JSON입니다. 다음 줄은 이 개체의 모든 요청에 사용자 에이전트 헤더를 추가합니다. 이러한 두 가지 헤더는 GitHub 서버 코드에서 확인되며 GitHub에서 정보를 검색하는데 필요합니다.

`HttpClient`를 구성한 후에는 웹 요청을 수행하고 응답을 검색합니다. 이 첫 번째 버전에서는 `HttpClient.GetStringAsync(String)` 편의 메서드를 사용합니다. 이 편의 메서드는 웹 요청을 수행하는 작업을 시작한 다음, 요청이 반환될 때 응답 스트림을 읽고 해당 스트림에서 콘텐츠를 추출합니다. 응답의 본문은 `String`으로 반환됩니다. 작업이 완료되면 이 문자열을 사용할 수 있습니다.

이 메서드의 마지막 두 줄은 해당 작업을 대기하고 콘솔에 응답을 출력합니다. 앱을 빌드한 다음 실행합니다. `ProcessRepositories`에는 `await` 연산자가 포함되므로 이제 빌드 경고는 사라집니다. 길게 표시되는 JSON 형식 텍스트를 볼 수 있습니다.

## JSON 결과 처리

지금까지 웹 서버에서 응답을 검색하고 해당 응답에 포함된 텍스트를 표시하는 코드를 작성했습니다. 다음에는 JSON 응답을 C# 개체로 변환해 보겠습니다.

`System.Text.Json.JsonSerializer` 클래스는 개체를 JSON으로 직렬화하고 JSON을 개체로 역직렬화합니다. 먼저 GitHub API에서 반환된 `repo` JSON 개체를 나타내도록 클래스를 정의합니다.

```

using System;

namespace WebAPIClient
{
    public class Repository
    {
        public string name { get; set; }
    }
}

```

'repo.cs'라는 새 파일에 위 코드를 넣습니다. 이 버전의 클래스는 JSON 데이터를 처리하는 가장 간단한 경로를 나타냅니다. 클래스 이름 및 멤버 이름은 다음 C# 규칙 대신 JSON 패킷에 사용된 이름과 일치합니다. 나중에 몇 가지 구성 특성을 제공하여 수정할 것입니다. 이 클래스에서는 JSON serialization 및 deserialization의 또 다른 중요한 특징을 보여 줍니다. 즉, JSON 패킷의 모든 필드가 이 클래스에 속하는 것은 아닙니다. JSON serializer는 사용되는 클래스 형식에 포함되지 않은 정보를 무시합니다. 이 특징 때문에 JSON 패킷의 필드 일부에만 작용하는 형식을 보다 쉽게 만들 수 있습니다.

이제 형식을 만들었으므로 역직렬화를 수행해 보겠습니다.

다음으로, serializer를 사용하여 JSON을 C# 개체로 변환합니다. `ProcessRepositories` 메서드의 `GetStringAsync(String)` 호출을 다음 줄로 바꿉니다.

```

var streamTask = client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories = await JsonSerializer.DeserializeAsync<List<Repository>>(await streamTask);

```

새 네임 스페이스를 사용하고 있으므로 파일의 맨 위에도 추가해야 합니다.

```

using System.Collections.Generic;
using System.Text.Json;

```

현재 `GetStringAsync(String)` 대신 `GetStreamAsync(String)`을 사용하고 있습니다. serializer는 해당 소스로 문자열 대신 스트림을 사용합니다. 앞의 코드 조각 두 번째 줄에서 사용되는 C# 언어의 몇 가지 기능에 대해 설명해 보겠습니다. `JsonSerializer.DeserializeAsync< TValue >(Stream, JsonSerializerOptions, CancellationToken)`에 대한 첫 번째 인수는 `await` 식입니다. (다른 두 매개 변수는 선택 사항이며 코드 조각에서 생략됩니다.) 지금까지는 대입문의 일부로만 볼 수 있었지만 Await 식은 코드의 거의 모든 위치에 나올 수 있습니다. `Deserialize` 메서드는 제네릭입니다. 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<>` 클래스는 개체의 컬렉션을 저장합니다. 형식 인수는 `List<>`에 저장된 개체의 형식을 선언합니다. JSON 텍스트는 리포지토리 개체의 컬렉션을 나타내므로 형식 인수는 `Repository`입니다.

이 섹션이 거의 완료되었습니다. JSON을 C# 개체로 변환했으므로 각 리포지토리의 이름을 표시해 보겠습니다. 다음 줄을

```

var msg = await stringTask;    /**Deleted this
Console.WriteLine(msg);

```

다음으로 바꿉니다.

```

foreach (var repo in repositories)
    Console.WriteLine(repo.name);

```

애플리케이션을 컴파일하고 실행합니다. .NET Foundation에 포함된 리포지토리의 이름이 출력됩니다.

## serialization 제어

더 많은 기능을 추가하기 전에 `[JsonPropertyName]` 특성을 사용하여 `name` 속성을 다뤄 보겠습니다. `repo.cs`에서 `name` 필드의 선언을 다음과 같이 변경합니다.

```
[JsonPropertyName("name")]
public string Name { get; set; }
```

`[JsonPropertyName]` 특성을 사용하려면 `using` 지시문에 `System.Text.Json.Serialization` 네임스페이스를 추가해야 합니다.

```
using System.Text.Json.Serialization;
```

이렇게 변경할 경우 `program.cs`에서 각 리포지토리의 이름을 쓰는 코드를 변경해야 합니다.

```
Console.WriteLine(repo.Name);
```

`dotnet run` 을 실행하여 매핑이 올바르게 수행되었는지 확인합니다. 이전과 동일한 출력이 표시되어야 합니다.

새로운 기능을 추가하기 전에 한 가지 더 변경해 보겠습니다. `ProcessRepositories` 메서드는 비동기 작업을 수행하고 리포지토리 컬렉션을 반환할 수 있습니다. 이 메서드에서 `List<Repository>`로 돌아가 정보를 쓰는 코드를 `Main` 메서드로 이동해 보겠습니다.

`ProcessRepositories`의 시그니처를 변경하여 `Repository` 개체의 목록을 해당 결과로 표시하는 작업을 반환합니다.

```
private static async Task<List<Repository>> ProcessRepositories()
```

다음에는 JSON 응답을 처리한 후 리포지토리를 반환합니다.

```
var streamTask = client.GetStreamAsync("https://api.github.com/orgs/dotnet/repos");
var repositories = await JsonSerializer.DeserializeAsync<List<Repository>>(await streamTask);
return repositories;
```

이 개체를 `async`로 표시했으므로 컴파일러는 해당 반환에 대한 `Task<T>` 개체를 생성합니다. 그런 다음 해당 결과를 캡처하고 각 리포지토리 이름을 콘솔에 쓰도록 `Main` 메서드를 수정해 보겠습니다. `Main` 메서드는 이제 다음과 같이 표시됩니다.

```
public static async Task Main(string[] args)
{
    var repositories = await ProcessRepositories();

    foreach (var repo in repositories)
        Console.WriteLine(repo.Name);
}
```

## 추가 정보 읽기

GitHub API에서 전송되는 JSON 패킷에 있는 속성을 몇 가지 더 처리하는 것으로 마무리해 보겠습니다. 모든 기능을 알 수는 없겠지만 일부 속성을 추가하면 C# 언어의 몇 가지 기능이 추가로 확인됩니다.

먼저 `Repository` 클래스 정의에 몇 가지 간단한 형식을 더 추가해 보겠습니다. 해당 클래스에 다음 속성을 추가

합니다.

```
[JsonPropertyName("description")]
public string Description { get; set; }

[JsonPropertyName("html_url")]
public Uri GitHubHomeUrl { get; set; }

[JsonPropertyName("homepage")]
public Uri Homepage { get; set; }

[JsonPropertyName("watchers")]
public int Watchers { get; set; }
```

이러한 속성은 기본적으로 문자열 형식(JSON 패킷에 포함된 형식)을 대상 형식으로 변환합니다. `Uri` 형식은 생소할 수 있습니다. 이 형식은 URI 또는 이 경우에는 URL을 나타냅니다. `Uri` 및 `int` 형식의 경우 JSON 패킷에 대상 형식으로 변환되지 않는 데이터가 포함되어 있으면 serialization 작업은 예외를 throw합니다.

이러한 데이터를 추가한 경우에는 해당 요소를 표시하도록 `Main` 메서드를 업데이트합니다.

```
foreach (var repo in repositories)
{
    Console.WriteLine(repo.Name);
    Console.WriteLine(repo.Description);
    Console.WriteLine(repo.GitHubHomeUrl);
    Console.WriteLine(repo.Homepage);
    Console.WriteLine(repo.Watchers);
    Console.WriteLine();
}
```

마지막 단계로, 최종 밀어넣기 작업을 위한 정보를 추가해 보겠습니다. 이 정보는 JSON 응답에서 다음 방식으로 형식이 지정됩니다.

```
2016-02-08T21:27:00Z
```

해당 형식은 UTC(협정 세계시)이므로 `Kind` 속성이 `Utc`인 `DateTime` 값을 얻습니다. 표준 시간대로 표시되는 날짜를 선호하는 경우 사용자 지정 변환 메서드를 작성해야 합니다. 먼저 `Repository` 클래스에서 날짜 및 시간을 UTC로 표현하는 `public` 속성을 정의하고 현지 시간으로 변환된 날짜를 반환하는 `LastPush` `readonly` 속성을 정의합니다.

```
[JsonPropertyName("pushed_at")]
public DateTime LastPushUtc { get; set; }

public DateTime LastPush => LastPushUtc.ToLocalTime();
```

방금 정의한 새 구문을 살펴보겠습니다. `LastPush` 속성은 `get` 접근자에 대한 식 `본문 멤버`를 사용하여 정의됩니다. `set` 접근자가 없습니다. `set` 접근자를 생략하는 것이 바로 C#에서 읽기 전용 속성을 정의하는 방식입니다. (C#에서 쓰기 전용 속성을 만들 수 있지만 해당 값은 제한됩니다.)

마지막으로 콘솔에 `output` 문을 하나 더 추가하면 이 앱을 빌드하고 다시 실행할 준비가 됩니다.

```
Console.WriteLine(repo.LastPush);
```

이제 해당 버전이 완성된 샘플과 일치해야 합니다.

## 결론

이 자습서에서는 웹 요청을 수행하고, 결과를 구문 분석하고, 해당 결과의 속성을 표시하는 방법을 알아보았습니다. 또한 프로젝트에 종속성으로 새 패키지를 추가했습니다. 개체 지향 기술을 지원하는 C# 언어의 일부 기능을 확인했습니다.

`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 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.

# C# 및 .NET의 상속

2020-11-02 • 63 minutes to read • [Edit Online](#)

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

이 자습서에서는 .NET Core SDK를 설치했다고 가정합니다. 다운로드하려면 [.NET Core 다운로드](#) 페이지를 방문하세요. 코드 편집기도 필요합니다. 원하는 어떤 코드 편집기도 사용 가능하지만 이 자습서에서는 [Visual Studio Code](#)를 사용합니다.

## 예제 실행

이 자습서의 예제를 만들고 실행하기 위해 명령줄에서 `dotnet` 유ти리티를 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 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 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. 개인 `A.value` 필드는 `A.B`에 표시됩니다. 그러나 `C.GetValue` 메서드에서 주석을 제거하고 예제를 컴파일하고 하면 컴파일러 오류 CS0122: "보호 수준 때문에 'A.value'에 액세스할 수 없습니다."가 표시됩니다.

```
using System;

public class A
{
    private int value = 10;

    public class B : A
    {
        public int GetValue()
        {
            return this.value;
        }
    }
}

public class C : A
{
    //    public int GetValue()
    //    {
    //        return this.value;
    //    }
}

public class Example
{
    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.

```

public class A
{
    public void Method1()
    {
        // Method implementation.
    }
}

public class B : A
{ }

public class Example
{
    public static void Main()
    {
        B b = new B();
        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 비가상 멤버를 재정의하려고 하면 컴파일러 오류 CS0506: "<member>: 상속된 '<member>' 멤버는 virtual, abstract 또는 override로 표시되지 않았으므로 재정의할 수 없습니다."가 표시됩니다.

```

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.

```

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 하면 컴파일러 오류 CS0527이 발생합니다. "인터페이스 목록에 있는 'ValueType' 형식이 인터페이스가 아닙니다." 이 오류 메시지는 구조체가 구현하는 인터페이스를 정의할 수 있지만 상속은 지원되지 않음을 나타냅니다.

```
using System;

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.

```
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.

```

using System;
using System.Reflection;

public class Example
{
    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 (var 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";
            }
            var 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 객체의 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.

```
using System;

public class SimpleClass
{}

public class Example
{
    public static void Main()
    {
        SimpleClass sc = new SimpleClass();
        Console.WriteLine(sc.ToString());
    }
}

// The example displays the following output:
//      SimpleClass
```

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	<code>Object</code>
struct	<code>ValueType, Object</code>
enum	<code>Enum, ValueType, Object</code>
대리자(delegate)	<code>MulticastDelegate, Delegate, Object</code>

## 상속 및 "~이다(is a)" 관계

일반적으로 상속은 기본 클래스와 하나 이상의 파생 클래스 간 "~이다(is a)" 관계를 나타내는데 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.

### NOTE

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.

```

using System;

public class Automobile
{
    public Automobile(string make, string model, int year)
    {
        if (make == null)
            throw new ArgumentNullException("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("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.

```

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 재정의 할 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 호출에 의해 `abstract` 키워드로 표시된 클래스를 인스턴스화하려고 하면 C# 컴파일러는 오류 CS0144, "추상 클래스 또는 인터페이스의 인스턴스를 만들 수 없습니다."를 생성합니다. 리플렉션을 사용하여 클래스를 인스턴스화하려고 하면 리플렉션 메서드가 `MemberAccessException`을 throw합니다.

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 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. `sealed` 클래스에서 파생하려고 하면 컴파일러 오류 CS0509 "sealed 형식 '<typeName>'에서 파생될 수

없습니다."를 생성합니다.

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.

```
using System;

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("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;
        this.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.

```

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 `throw` 됩니다.

- 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. `Publish` 메서드는 호출될 때 `private published` 플래그를 `true`로 설정하고 전달된 날짜를 `private datePublished` 필드에 대한 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 구분하는데 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)
	PublicationType
	Publisher
	Title
	CopyrightDate
	CopyrightName
	Pages
	Copyright()
	GetPublicationDate()
	Publish()

**Key**

Unique member	
Inherited member	
Overridden member	

### Book 클래스

Book 클래스는 책을 특수한 출판문 형식으로 나타냅니다. 다음 예제에서는 Book 클래스에 대한 소스 코드를 보여 줍니다.

```

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.");
            ulong nISBN = 0;
            if (!UInt64.TryParse(isbn, out nISBN))
                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("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)
    {
        Book book = obj as Book;
        if (book == null)
            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 같은 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 보여 줍니다.

## 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.

```

using System;
using static System.Console;

public class Example
{
    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");
        WriteLine($"{book.Title} and {book2.Title} are the same publication: " +
                 $""\{(Publication) book\}.Equals({book2})\}");
    }

    public static void ShowPublicationInfo(Publication pub)
    {
        string pubDate = pub.GetPublicationDate();
        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 달힌 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 둘레(또는 원주)를 계산하는 공식은 삼각형의 둘레를 계산하는 공식과 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.

```
using System;

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 `Triangle`, `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.

```

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 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.

```

using System;

public class Example
{
    public static void Main()
    {
        Shape[] shapes = { new Rectangle(10, 12), new Square(5),
                           new Circle(3) };
        foreach (var shape in shapes) {
            Console.WriteLine($"{shape}: area, {Shape.GetArea(shape)}; " +
                $"perimeter, {Shape.GetPerimeter(shape)}");
            var rect = shape as Rectangle;
            if (rect != null) {
                Console.WriteLine($"    Is Square: {rect.IsSquare()}, Diagonal: {rect.Diagonal}");
                continue;
            }
            var sq = shape as Square;
            if (sq != null) {
                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
}

```

## 참조

- [상속\(C# 프로그래밍 가이드\)](#)

# LINQ(Language-Integrated Query) 작업

2020-11-02 • 41 minutes to read • [Edit Online](#)

## 소개

이 자습서에서는 .NET Core 및 C# 언어의 기능에 대해 설명합니다. 다음을 수행하는 방법을 알아봅니다.

- 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. 빌드 할 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.

## 필수 구성 요소

.NET Core를 실행 하려면 컴퓨터에 설정해야 합니다. [.NET Core 다운로드](#) 페이지에서 설치 지침을 확인할 수 있습니다. Windows, Ubuntu Linux나 OS X 또는 Docker 컨테이너에서 이 애플리케이션을 실행 할 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 프롬프트에 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.

```
// 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.

```
// 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.

```
// 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.

```
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:\> C:\Windows\system32\cmd.exe
C:\> console-linq>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.

```
// 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);
    }

    // 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 프로세스

의 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.

```
// 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.

```
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` 한 다음, 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`를 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.

```

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.

```

// 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);
    }
}

```

## 비|교

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.

```

public static bool SequenceEquals<T>
    (this IEnumerable<T> first, IEnumerable<T> second)
{
    var firstIter = first.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
    {
        if (!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.

```

// 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.

## 초적화

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 되도

록 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.

```
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 계산 되는 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. 52번 째 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.

```
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.

```
using System.IO;
```

이렇게 하면 문제가 해결되고 빨간색 오류가 사라집니다.

그런 후 로그 메시지를 사용하여 각 쿼리의 정의를 계측합니다.

```

// 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 캔 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.

```

public static void Main(string[] args)
{
    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](#)

# C#에서 특성 사용

2020-11-02 • 21 minutes to read • [Edit Online](#)

특성은 선언적으로 정보를 코드와 연결하는 방법을 제공합니다. 다양한 대상에 적용할 수 있는 재사용 가능 요소를 제공할 수도 있습니다.

`[Obsolete]` 특성을 고려하세요. 이 특성은 클래스, 구조체, 메서드, 생성자 등에 적용될 수 있으며 요소가 더 이상 필요하지 않는 사실을 \_선언\_합니다. 이 특성을 찾고 응답으로 특정 작업을 수행하는 것은 모두 C# 컴파일러가 진행합니다.

이 자습서에서는 코드에 특성을 추가하는 방법, 사용자 지정 특성을 만들고 사용하는 방법, .NET Core로 빌드되는 일부 특성을 사용하는 방법을 소개합니다.

## 사전 요구 사항

.NET Core를 실행 하려면 컴퓨터에 설정해야 합니다. [.NET Core 다운로드](#) 페이지에서 설치 지침을 찾을 수 있습니다. Windows, Ubuntu Linux, macOS 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

이제 모든 도구를 설치했으므로 새로운 .NET Core 애플리케이션을 만들어 보겠습니다. 명령줄 생성기를 사용하려면 즐겨 사용하는 셸에서 다음 명령을 실행합니다.

```
dotnet new console
```

이 명령은 기본 .NET Core 프로젝트 파일을 만듭니다. `dotnet restore`를 실행하여 이 프로젝트를 컴파일하는데 필요한 종속성을 복원해야 합니다.

`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 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.

이 프로그램을 실행하려면 `dotnet run`을 사용합니다. 콘솔에 "Hello, World" 출력이 표시됩니다.

## 코드에 특성을 추가하는 방법

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 사용할 수 없음을 알리기 위해 사용됩니다. 예를 들어 대괄호를 사용하여 클래스에 이 특성을 배치할 수 있습니다.

```
[Obsolete]  
public class MyClass  
{  
}
```

이 클래스는 `ObsoleteAttribute`로 지칭되지만 코드에서 `[Obsolete]`를 사용하는 데만 필요합니다. 이것이 C#에서 준수하는 규칙입니다. 원할 경우 전체 이름 `[ObsoleteAttribute]`를 사용할 수 있습니다.

클래스를 더 이상 사용되지 않는 것으로 표시할 경우 더 이상 사용되지 않는 이유 및/또는 대상 사용할 항목에 대한 정보를 제공하는 것이 좋습니다. 이 작업을 위해 `Obsolete` 특성에 문자열 매개 변수를 전달합니다.

```
[Obsolete("ThisClass is obsolete. Use ThisClass2 instead.")]  
public class ThisClass  
{  
}
```

이 문자열은 `var attr = new ObsoleteAttribute("some string")`를 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. 식 또는 변수는 사용할 수 없습니다. 위치 또는 명명된 매개 변수는 얼마든지 사용할 수 있습니다.

## 고유한 특성을 만드는 방법

특성을 만드는 과정은 `Attribute` 기본 클래스에서 상속하는 것만큼 간단합니다.

```
public class MySpecialAttribute : Attribute  
{  
}
```

위에 따라, 이제 코드베이스의 어디에서든지 `[MySpecial]` (또는 `[MySpecialAttribute]`)을 특성으로 사용할 수 있습니다.

```
[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 임의 위치에서 해당 메타데이터에 대해 작업을 수행할 수 있습니다(이 자습서의 뒷부분에 좀 더 자세히 설명되어 있음).

여기에는 확인해 볼만한 '과제'가 제공됩니다. 위에서 설명한 것처럼 특성을 사용할 때는 특정 형식만 인수로 전달되도록 허용됩니다. 그러나 특성 유형을 만들 때 C# 컴파일러는 사용자가 해당 매개 변수를 만들지 못하게 하지 않습니다. 아래 예제에서는 적절히 컴파일을 수행하는 생성자로 특성을 만들었습니다.

```
public class GotchaAttribute : Attribute  
{  
    public GotchaAttribute(Foo myClass, string str) {  
    }  
}
```

그러나 이 생성자를 특성 구문에서 사용할 수는 없습니다.

```
[Gotcha(new Foo(), "test")] // does not compile
public class AttributeFail
{
}
```

위에서는

```
Attribute constructor parameter 'myClass' has type 'Foo', which is not a valid attribute parameter type
```

과 같은 컴파일러 오류가 발생합니다.

## 특성 사용을 제한하는 방법

특성은 다양한 "대상"에 사용할 수 있습니다. 위의 예제에서는 클래스에 대한 특성만 표시하지만 다음에도 사용될 수 있습니다.

- 어셈블리
- 클래스
- 생성자
- 대리자
- 열거형
- 이벤트
- 필드
- GenericParameter
- 인터페이스
- 메서드
- 모듈
- 매개 변수
- 속성
- ReturnValue
- 구조체

특성 클래스를 만들 때 기본적으로 C#은 허용 가능한 특성 대상 중 하나에 대해 해당 특성을 사용할 수 있습니다. 특성을 특정 대상으로 제한하려는 경우 특성 클래스에 대해 `AttributeUsageAttribute`를 사용하면 됩니다. 맞습니다. 특성에 대한 특성입니다.

```
[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.

```
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. 외부 영향이 없으면 어떤 작업도 수행하지 않습니다.

특성을 찾아 작업을 수행하려면 일반적으로 [리플렉션](#)이 필요합니다. 이 자습서에서는 리플렉션을 자세히 다루지 않겠지만, 기본 개념은 리플렉션을 사용하면 다른 코드를 검사하는 코드를 C#에서 작성할 수 있다는 것입니다.

예를 들어 리플렉션을 사용하여 클래스에 대한 정보(코드 헤드에 `using System.Reflection;` 추가)를 가져올 수 있습니다.

```
TypeInfo typeInfo = typeof(MyClass).GetTypeInfo();
Console.WriteLine("The assembly qualified name of MyClass is " + typeInfo.AssemblyQualifiedName);
```

다음과 같이 출력됩니다.

```
The assembly qualified name of MyClass is ConsoleApplication.MyClass, attributes, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null
```

`TypeInfo` 개체(또는 `MethodInfo`, `FieldInfo` 등)가 있으면 `GetCustomAttributes` 메서드를 사용할 수 있습니다. 그러면 `Attribute` 개체 컬렉션이 반환됩니다. `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).

```
var attrs = typeInfo.GetCustomAttributes();
foreach(var attr in attrs)
    Console.WriteLine("Attribute on MyClass: " + attr.GetType().Name);
```

해당 내용은 콘솔에 `Attribute on MyClass: ObsoleteAttribute`와 같이 표시됩니다. `MyClass`에 다른 특성을 추가해 보세요.

이러한 `Attribute` 개체가 지연되어 인스턴스화되는 것에 유의해야 합니다. 즉, `GetCustomAttribute` 또는 `GetCustomAttributes`를 사용해야만 인스턴스화됩니다. 또한 매번 인스턴스화되기도 합니다. `GetCustomAttributes`를 연속해서 2번 호출하면 2개의 다른 `ObsoleteAttribute` 인스턴스가 반환됩니다.

## BCL(기본 클래스 라이브러리)의 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. PostSharp은 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 않는 경우 해당 메서드에 대한 호출(메서드 자체는 아님)이 C# 컴파일러에 의해 제거됩니다. 일반적으로 이 특성은 디버깅(진단) 목적으로 사용됩니다.
- `[CallerMemberName]`. 이 특성은 매개 변수에 사용될 수 있으며 `System.Runtime.CompilerServices` 네임스페이스에 있습니다. 이 특성은 메서드의 매개 변수에 적용됩니다. 해당 매개 변수는 해당 메서드가 호출된 때마다 호출된 메서드의 이름으로 대체됩니다.

이스에 있습니다. 다른 메서드를 호출하는 메서드의 이름을 삽입하는 데 사용되는 특성입니다. 일반적으로 다양한 UI 프레임워크에서 INotifyPropertyChanged를 구현하는 경우 '매직 문자열'을 제거하는 방법으로 사용됩니다. 예를 들어

```
public class MyUIClass : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    public void RaisePropertyChanged([CallerMemberName] string propertyName = null)
    {
        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. 이 경우 입력 관련 버그가 방지되며 좀 더 매끄러운 리팩터링/이름 바꾸기가 가능해집니다.

## 요약

특성은 C#에 선언적 기능을 제공하지만 코드의 메타데이터 형식이며 단독으로 동작하지 않습니다.

# C# 9.0의 새로운 기능

2021-02-18 • 49 minutes to read • [Edit Online](#)

C# 9.0은 다음 기능과 개선 사항을 C# 언어에 추가합니다.

- [레코드](#)
- [Init 전용 setter](#)
- [최상위 문](#)
- [패턴 일치 개선 사항](#)
- 원시 크기 정수
- 함수 포인터
- localsinit 플래그 내보내기 무시
- 대상으로 형식화된 새 식
- 정적 무명 함수
- 대상으로 형식화된 조건식
- 공변 반환 형식
- `foreach` 루프에 대한 확장 `GetEnumerator` 지원
- 람다 무시 항목 매개 변수
- 로컬 함수의 특성
- 모듈 이니셜라이저
- 부분 메서드에 대한 새로운 기능

C# 9.0은 .NET 5에서 지원됩니다. 자세한 내용은 [C# 언어 버전 관리](#)를 참조하세요.

[.NET 다운로드 페이지](#)에서 최신 .NET SDK를 다운로드할 수 있습니다.

## 레코드 유형

C# 9.0에서는 같음에 대한 값 의미 체계를 제공하는 합성 메서드를 제공하는 참조 형식인 '\*'레코드 종류'\*'가 도입되었습니다. 레코드는 기본적으로 변경할 수 없습니다.

레코드 종류를 사용하면 .NET에서 변경할 수 없는 참조 형식을 쉽게 만들 수 있습니다. 지금까지 .NET 형식은 크게 참조 형식(클래스 및 무명 형식 포함)과 값 형식(구조체 및 튜플 포함)으로 분류되었습니다. 변경할 수 없는 값 형식이 권장되지만, 변경 가능한 값 형식을 사용해도 오류가 자주 발생하지는 않습니다. 값 형식 변수에는 값이 포함되므로 값 형식을 메서드에 전달할 때 원래 데이터의 복사본이 변경됩니다.

변경할 수 없는 참조 형식에도 많은 이점이 있습니다. 이러한 이점은 공유 데이터를 사용하는 동시 프로그램에서 더욱 두드러지게 나타납니다. 아쉽게도 C#에서는 변경할 수 없는 참조 형식을 만들기 위한 상당한 추가 코드를 작성해야 했습니다. 레코드는 같음에 대한 값 의미 체계를 사용하는, 변경할 수 없는 참조 형식의 형식 선언을 제공합니다. 같음 및 해시 코드에 대한 합성 메서드는 해당 속성이 모두 동일한 경우 두 레코드가 같다고 간주합니다. 다음과 같은 정의를 고려해 보세요.

```
public record Person
{
    public string LastName { get; }
    public string FirstName { get; }

    public Person(string first, string last) => (FirstName, LastName) = (first, last);
}
```

레코드 정의에서 `FirstName` 및 `LastName`이라는 두 개의 읽기 전용 속성을 포함하는 `Person` 형식을 만듭니다.

`Person` 형식은 참조 형식입니다. IL을 확인했다면 이 형식은 클래스입니다. 생성된 후에는 속성을 수정할 수 없다는 점에서 변경할 수 없는 형식입니다. 레코드 종류를 정의하면 컴파일러에서 다른 여러 메서드를 자동으로 합성합니다.

- 값 기반 같음 비교 메서드
- `GetHashCode()` 재정의
- 멤버 복사 및 복제
- `PrintMembers` 및 `ToString()`

레코드는 상속을 지원합니다. `Person`에서 파생된 새 레코드를 다음과 같이 선언할 수 있습니다.

```
public record Teacher : Person
{
    public string Subject { get; }

    public Teacher(string first, string last, string sub)
        : base(first, last) => Subject = sub;
}
```

추가 파생을 방지하기 위해 레코드를 봉인할 수도 있습니다.

```
public sealed record Student : Person
{
    public int Level { get; }

    public Student(string first, string last, int level) : base(first, last) => Level = level;
}
```

컴파일러는 위 메서드의 여러 버전을 합성합니다. 메서드 시그니처는 레코드 종류가 `sealed`인지 여부와 직접 기본 클래스가 `object`인지 여부에 따라 달라집니다. 레코드에는 다음 기능이 있어야 합니다.

- 같음은 값을 기반으로 하며 형식 일치 검사를 포함합니다. 예를 들어 `Student`는 두 레코드가 동일한 이름을 공유하는 경우에도 `Person`과 같을 수 없습니다.
- 레코드의 문자열 표현이 일관성 있게 자동으로 생성됩니다.
- 레코드에서 복사본 생성을 지원합니다. 올바른 복사본 생성에는 상속 계층 구조와 개발자가 추가한 속성이 포함되어야 합니다.
- 수정 내용과 함께 레코드를 복사할 수 있습니다. 복사 및 수정 작업에서 비파괴적 변경을 지원합니다.

컴파일러는 익숙한 `Equals` 오버로드, `operator ==`, `operator !=` 외에도 새 `EqualityContract` 속성을 합성합니다. 이 속성은 레코드 종류와 일치하는 `Type` 개체를 반환합니다. 기본 형식이 `object`이면 속성은 `virtual`이 됩니다. 기본 형식이 다른 레코드 종류이면 속성은 `override`가 됩니다. 레코드 종류가 `sealed`이면 속성은 `sealed`가 됩니다. 합성된 `GetHashCode`는 기본 형식 및 레코드 종류에 선언된 모든 속성과 필드의 `GetHashCode`를 사용합니다. 해당 합성 메서드는 상속 계층 구조 전체에 값 기반 같음을 적용합니다. 즉, `Student`는 동일한 이름을 가진 `Person`과 같다고 간주하지 않습니다. 레코드 종류 간에 공유되는 모든 속성이 같은 뿐 아니라 두 레코드 종류도 일치해야 합니다.

또한 레코드에는 합성 생성자와 복사본을 만들기 위한 “clone” 메서드가 있습니다. 합성 생성자에는 레코드 형식의 단일 매개 변수가 있습니다. 합성 생성자는 레코드의 모든 속성에 동일한 값을 사용하여 새 레코드를 생성합니다. 레코드가 `sealed`이면 해당 생성자는 `private`이 되고, 그렇지 않으면 `protected`가 됩니다. 합성된 “clone” 메서드는 레코드 계층 구조의 복사본 생성을 지원합니다. 실제 이름이 컴파일러에서 생성되기 때문에 “clone” 용어는 따옴표 안에 표시됩니다. 레코드 종류에 `Clone`이라는 메서드를 만들 수는 없습니다. 합성된 “clone” 메서드는 가상 디스패치를 사용하여 복사되는 레코드 종류를 반환합니다. 컴파일러는 `record`의 액세스 한정자에 따라 “clone” 메서드에 대해 다른 한정자를 추가합니다.

- 레코드 종류가 `abstract` 이면 “clone” 메서드도 `abstract` 가 됩니다. 기본 형식이 `object` 가 아니면 메서드도 `override` 가 됩니다.
- 기본 형식이 `object` 일 때 `abstract` 가 아닌 레코드 종류의 경우:
  - 레코드가 `sealed` 이면 “clone” 메서드에 한정자가 추가되지 않습니다(즉, `virtual` 이 아님).
  - 레코드가 `sealed` 가 아니면 “clone” 메서드는 `virtual` 이 됩니다.
- 기본 형식이 `object` 가 아닐 때 `abstract` 가 아닌 레코드 종류의 경우:
  - 레코드가 `sealed` 이면 “clone” 메서드도 `sealed` 가 됩니다.
  - 레코드가 `sealed` 가 아니면 “clone” 메서드는 `override` 가 됩니다.

모든 규칙의 결과로, 모든 레코드 종류 계층 구조에서 같음이 일관되게 구현됩니다. 다음 예제와 같이 해당 속성과 종류가 동일하면 두 레코드는 같습니다.

```
var person = new Person("Bill", "Wagner");
var student = new Student("Bill", "Wagner", 11);

Console.WriteLine(student == person); // false
```

컴파일러는 인쇄 출력을 지원하는 두 가지 메서드인 `ToString()` 재정의와 `PrintMembers` 를 합성합니다.

`PrintMembers` 는 `System.Text.StringBuilder` 을 인수로 사용합니다. 레코드 종류의 모든 속성에 대해 쉼표로 구분된 속성 이름 및 값 목록이 추가됩니다. `PrintMembers` 는 다른 레코드에서 파생된 모든 레코드에 대해 기본 구현을 호출합니다. `ToString()` 재정의는 `PrintMembers` 에서 생성된 문자열을 { } 및 } 로 묶어 반환합니다. 예를 들어 `Student` 의 `ToString()` 메서드는 다음 코드와 같이 `string` 을 반환합니다.

```
"Student { LastName = Wagner, FirstName = Bill, Level = 11 }"
```

지금까지 살펴본 예제에서는 일반적인 구문을 사용하여 속성을 선언합니다. ‘위치 레코드’라는 보다 간결한 형식이 있습니다. 다음은 앞에서 위치 레코드로 정의된 세 가지 레코드 종류입니다.

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

public record Teacher(string FirstName, string LastName,
    string Subject)
    : Person(FirstName, LastName);

public sealed record Student(string FirstName,
    string LastName, int Level)
    : Person(FirstName, LastName);
```

이 선언에서는 이전 버전과 동일한 기능을 만듭니다(다음 섹션에서 설명하는 몇 가지 추가 기능 포함). 레코드에서 메서드를 추가하지 않으므로 이 선언은 대괄호가 아닌 세미콜론으로 끝납니다. 본문을 추가하고 다른 메서드도 포함할 수 있습니다.

```

public record Pet(string Name)
{
    public void ShredTheFurniture() =>
        Console.WriteLine("Shredding furniture");
}

public record Dog(string Name) : Pet(Name)
{
    public void WagTail() =>
        Console.WriteLine("It's tail wagging time");

    public override string ToString()
    {
        StringBuilder s = new();
        base.PrintMembers(s);
        return $"{s.ToString()} is a dog";
    }
}

```

컴파일러는 위치 레코드의 `Deconstruct` 메서드를 생성합니다. `Deconstruct` 메서드에는 레코드 종류의 모든 `public` 속성 이름과 일치하는 매개 변수가 있습니다. `Deconstruct` 메서드를 사용하여 레코드를 구성 요소 속성으로 분해할 수 있습니다.

```

var person = new Person("Bill", "Wagner");

var (first, last) = person;
Console.WriteLine(first);
Console.WriteLine(last);

```

마지막으로 레코드는 `with` 식을 지원합니다. \* `with expression` 은 레코드 복사본을 만들지만 `with`에 지정된 속성을 수정하도록 컴파일러에 지시합니다.

```
Person brother = person with { FirstName = "Paul" };
```

위의 줄은 `Lastname` 속성을 `person`의 복사본이고 `FirstName`은 "Paul" 인 새 `Person` 레코드를 만듭니다. `with` 식에서는 속성을 원하는 개수만큼 설정할 수 있습니다. `with` 식을 사용하여 정확한 복사본을 만들 수도 있습니다. 수정할 속성에 대해 빈 집합을 지정합니다.

```
Person clone = person with { };
```

"clone" 메서드를 제외한 모든 합성 멤버를 직접 작성할 수 있습니다. 레코드 종류에 합성 메서드의 시그니처와 일치하는 메서드가 있는 경우 해당 메서드는 컴파일러에서 합성되지 않습니다. 앞의 `Dog` 레코드 예제에는 수동 코딩된 `ToString()` 메서드가 예제로 포함되어 있습니다.

이 [레코드 탐색](#) 자습서에서 레코드 종류에 대해 자세히 알아보세요.

## Init 전용 setter

'Init 전용 setter'는 개체의 멤버를 초기화하는 일관성 있는 구문을 제공합니다. 속성 이니셜라이저를 사용하면 어떤 값이 어떤 속성을 설정하는지 명확하게 파악할 수 있습니다. 단점은 해당 속성이 설정 가능해야 한다는 것입니다. C# 9.0부터 속성 및 인덱서에 대해 `set` 접근자 대신 `init` 접근자를 만들 수 있습니다. 호출자는 속성 이니셜라이저 구문을 사용하여 생성 식에서 해당 값을 설정할 수 있지만, 생성이 완료되고 나면 `readonly` 속성이 됩니다. Init 전용 setter는 상태 변경 창을 제공합니다. 이 창은 생성 단계가 끝날 때 닫힙니다. 속성 이니셜라이저 및 `with-expression`을 비롯한 모든 초기화가 완료되면 생성 단계가 사실상 끝납니다.

작성하는 모든 형식에서 `init` 전용 setter를 선언할 수 있습니다. 예를 들어 다음 구조체는 기상 관측 구조체를

정의합니다.

```
public struct WeatherObservation
{
    public DateTime RecordedAt { get; init; }
    public decimal TemperatureInCelsius { get; init; }
    public decimal PressureInMillibars { get; init; }

    public override string ToString() =>
        $"At {RecordedAt:h:mm tt} on {RecordedAt:M/d/yyyy}: " +
        $"Temp = {TemperatureInCelsius}, with {PressureInMillibars} pressure";
}
```

호출자는 불변성을 유지하면서 속성 이니셜라이저 구문을 사용하여 값을 설정할 수 있습니다.

```
var now = new WeatherObservation
{
    RecordedAt = DateTime.Now,
    TemperatureInCelsius = 20,
    PressureInMillibars = 998.0m
};
```

하지만 초기화 후 관측을 변경할 경우 초기화 외부에서 init 전용 속성에 할당하여 오류가 발생합니다.

```
// Error! CS8852.
now.TemperatureInCelsius = 18;
```

init 전용 setter는 파생 클래스에서 기본 클래스 속성을 설정하는 데 유용할 수 있습니다. 기본 클래스의 도우미를 통해 파생 속성을 설정할 수도 있습니다. 위치 레코드는 init 전용 setter를 사용하여 속성을 선언합니다. 해당 setter는 with-expression에 사용됩니다. 정의하는 모든 `class` 또는 `struct`에 대해 Init 전용 setter를 선언할 수 있습니다.

## 최상위 문

'최상위 문'은 많은 애플리케이션에서 불필요한 공식 절차를 제거합니다. 정식 "Hello World!" 프로그램을 고려해 보세요.

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

작업을 수행하는 코드 줄은 1줄뿐입니다. 최상위 문을 사용하면 모든 상용구를 `using` 문과 작업을 수행하는 1줄로 바꿀 수 있습니다.

```
using System;

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

1줄 프로그램을 원하는 경우 `using` 지시문을 제거하고 정규화된 형식 이름을 사용할 수 있습니다.

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

애플리케이션의 한 파일에서만 최상위 문을 사용할 수 있습니다. 컴파일러가 여러 소스 파일에서 최상위 문을 발견할 경우 오류가 발생합니다. 일반적으로 `Main` 메서드인 선언된 프로그램 진입점 메서드와 최상위 문을 결합하는 경우에도 오류가 발생합니다. 따라서 일반적으로 `Program` 클래스의 `Main` 메서드에 있는 문이 한 파일에 포함되어 있다고 생각할 수 있습니다.

이 기능의 가장 일반적인 용도 중 하나는 교육 자료를 만드는 경우입니다. 초급 C# 개발자가 1~2줄의 코드로 정식 "Hello World!"를 작성할 수 있습니다. 추가 공식 절차가 필요하지 않습니다. 그러나 숙련된 개발자도 이 기능의 다양한 용도를 발견하게 됩니다. 최상위 수준 문을 통해 Jupyter Notebook에서 제공하는 것과 비슷한 스크립트 유사 환경을 실험용으로 사용할 수 있습니다. 최상위 수준 문은 작은 콘솔 프로그램 및 유ти리티에 적합합니다. Azure Functions는 최상위 문에 이상적인 사용 사례입니다.

가장 중요한 점은 최상위 문이 애플리케이션의 범위나 복잡성을 제한하지 않는다는 것입니다. 해당 문은 모든 .NET 클래스에 액세스하거나 사용할 수 있습니다. 또한 명령줄 인수나 반환 값의 사용을 제한하지 않습니다. 최상위 문은 `args`라는 문자열 배열에 액세스할 수 있습니다. 최상위 문에서 정수 값을 반환하는 경우 해당 값은 합성된 `Main` 메서드의 정수 반환 코드가 됩니다. 최상위 문에 비동기식을 포함할 수 있습니다. 이 경우 합성 진입점은 `Task` 또는 `Task<int>`를 반환합니다.

## 패턴 일치 개선 사항

C# 9에는 새로운 패턴 일치 개선 사항이 포함되어 있습니다.

- '형식 패턴'은 변수를 형식과 일치시킵니다.
- '괄호로 묶인 패턴'은 패턴 조합의 우선 순위를 적용하거나 강조합니다.
- '결합' `and` '패턴'에서는 두 패턴이 모두 일치해야 합니다.
- '분리' `or` '패턴'에서는 패턴 중 하나가 일치해야 합니다.
- '부정' `not` '패턴'에서는 패턴이 일치하지 않아야 합니다.
- '관계형 패턴'에서는 입력과 지정된 상수 간에 보다 작은, 보다 큼, 작거나 같음, 크거나 같음 관계가 있어야 합니다.

새로운 패턴은 패턴 구문을 보강합니다. 이 예제를 참조하십시오.

```
public static bool IsLetter(this char c) =>
    c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';
```

또는 선택적 괄호를 사용하여 `and`에 `or`보다 높은 우선 순위가 있음을 명확하게 지정합니다.

```
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.

```
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.

## 성능 및 interop

원시 크기 정수, 함수 포인터, `localsinit` 플래그 생략의 세 가지 새로운 기능을 통해 고성능이 필요한 하위 수준 라이브러리 및 네이티브 interop 지원이 향상되었습니다.

원시 크기 정수인 `nint` 및 `nuint`는 정수 형식입니다. 기본 형식 `System.IntPtr` 및 `System.UIntPtr`으로 표현됩니다. 컴파일러는 해당 형식의 추가 변환과 연산을 네이티브 정수 형식으로 표시합니다. 원시 크기 정수는 `.MaxValue` 또는 `.MinValue`의 속성을 정의합니다. 이러한 값은 대상 머신의 정수 원시 크기에 따라 달라지므로 컴파일 시간 상수로 표현할 수 없습니다. 이러한 값은 런타임에 읽기 전용입니다. 다음 범위에서 `nint`의 상수 값을 사용할 수 있습니다. `[ int.MinValue .. int.MaxValue ]`. 다음 범위에서 `nuint`의 상수 값을 사용할 수 있습니다. `[ uint.MinValue .. uint.MaxValue ]`. 컴파일러는 `System.Int32` 및 `System.UInt32` 형식을 사용하여 모든 단항 및 이진 연산자에 대해 상수 정리를 수행합니다. 결과가 32비트에 맞지 않으면 런타임에 연산이 실행되고 상수로 간주하지 않습니다. 정수 연산이 광범위하게 사용되고 가장 빠른 성능이 필요한 시나리오에서는 원시 크기 정수를 사용하여 성능을 향상할 수 있습니다.

함수 포인터는 IL opcode `ldftn` 및 `calli`에 액세스하는 편리한 구문을 제공합니다. 새로운 `delegate_` 구문을 사용하여 함수 포인터를 선언할 수 있습니다. `delegate*` 형식은 포인터 형식입니다. `delegate*` 형식을 호출하면 `Invoke()` 메서드에서 `callvirt`을 사용하는 대리자와는 달리 `calli`가 사용됩니다. 호출 구문은 동일합니다. 함수 포인터 호출은 `managed` 호출 규칙을 사용합니다. `delegate*` 구문 뒤에 `unmanaged` 키워드를 추가하여 `unmanaged` 호출 규칙을 사용하도록 선언합니다. `delegate*` 선언에서 특성을 사용하여 다른 호출 규칙을 지정할 수 있습니다.

마지막으로, `System.Runtime.CompilerServices.SkipLocalsInitAttribute`를 추가하여 컴파일러에 `localsinit` 플래그를 내보내지 않도록 지시할 수 있습니다. 이 플래그는 모든 지역 변수를 0으로 초기화하도록 CLR에 지시합니다. `localsinit` 플래그는 1.0부터 C#의 기본 동작이었습니다. 그러나 0으로 초기화를 추가로 수행할 경우 일부 시나리오에서 성능에 상당한 영향을 미칠 수 있습니다. 특히, `stackalloc`을 사용하는 경우에 해당합니다. 이 경우에는 `SkipLocalsInitAttribute`를 추가할 수 있습니다. 단일 메서드 또는 속성이나 `class`, `struct`, `interface` 또는 모듈에 추가할 수도 있습니다. 이 특성은 `abstract` 메서드에는 영향을 주지 않고, 구현과 관련해서 생성된 코드에 적용됩니다.

위의 기능은 일부 시나리오에서 성능을 향상할 수 있습니다. 채택 전과 후에 모두 신중하게 벤치마킹한 후에만 사용해야 합니다. 원시 크기 정수를 사용하는 코드는 여러 대상 플랫폼에서 여러 정수 크기로 테스트해야 합니다. 다른 기능에는 안전하지 않은 코드가 필요합니다.

## 기능 마무리

다른 많은 기능을 통해 코드를 보다 효율적으로 작성할 수 있습니다. C# 9.0에서는 만든 개체의 형식을 이미 알고 있는 경우 `new` 식에서 형식을 생략할 수 있습니다. 가장 일반적인 용도는 필드 선언입니다.

```
private List<WeatherObservation> _observations = new();
```

인수로 메서드에 전달할 새 개체를 만들어야 하는 경우 대상으로 형식화된 `new`를 사용할 수도 있습니다. 다음 시그니처를 사용하는 `ForecastFor()` 메서드를 고려해 보세요.

```
public WeatherForecast ForecastFor(DateTime forecastDate, WeatherForecastOptions options)
```

다음과 같이 메서드를 호출할 수 있습니다.

```
var forecast = station.ForecastFor(DateTime.Now.AddDays(2), new());
```

이 기능의 다른 유용한 용도는 init 전용 속성과 결합하여 새 개체를 초기화하는 것입니다.

```
WeatherStation station = new() { Location = "Seattle, WA" };
```

`return new();` 문을 사용하여 기본 생성자가 만든 인스턴스를 반환할 수 있습니다.

유사한 기능은 [조건식](#)의 대상 유형 확인을 개선합니다. 이 변경 내용이 도입되면서 두 식 간에 암시적 변환을 포함할 수는 없지만 두 식에 모두 대상 유형으로의 암시적 변환을 사용할 수 있습니다. 이 변경 내용을 발견하지 못할 수도 있습니다. 이전에 캐스트가 필요했거나 컴파일되지 않던 일부 조건식이 이제 작동할 뿐입니다.

C# 9.0부터 [람다 식](#) 또는 [무명 메서드](#)에 `static` 한정자를 추가할 수 있습니다. 정적 람다 식은 `static` 로컬 함수와 유사합니다. 정적 람다 또는 무명 메서드는 지역 변수나 인스턴스 상태를 캡처할 수 없습니다. `static` 한정자는 실수에 의한 다른 변수 캡처를 방지합니다.

공변 반환 형식은 [재정의](#) 메서드의 반환 형식에 유연성을 제공합니다. 재정의 메서드는 재정의된 기본 메서드의 반환 형식에서 파생된 형식을 반환할 수 있습니다. 이 메서드는 레코드나 가상 클론 또는 팩터리 메서드를 지원하는 기타 형식에 유용할 수 있습니다.

또한 [foreach](#) 루프는 `foreach` 패턴을 충족하는 확장 메서드 `GetEnumerator`를 인식하고 사용합니다. 이렇게 변경함으로써 `foreach`는 비동기 패턴과 같은 다른 패턴 기반 생성 및 패턴 기반 분해와 일치하게 됩니다. 실제로 이 변경은 모든 형식에 `foreach` 지원을 추가할 수 있음을 의미합니다. 설계상 개체를 열거하는 것이 적합한 경우로 사용을 제한해야 합니다.

다음으로, 무시 항목을 람다 식에 대한 매개 변수로 사용할 수 있습니다. 이 편리한 기능을 사용하면 인수 이름을 지정할 필요가 없으며, 컴파일러에서 인수를 사용하지 않을 수 있습니다. 모든 인수에 `_`을 사용합니다. 자세한 내용은 [람다 식](#) 문서의 [람다 식 입력 매개 변수](#) 섹션을 참조하세요.

마지막으로, 이제 [로컬 함수](#)에 특성을 적용할 수 있습니다. 예를 들어 로컬 함수에 [null 허용 특성 주석](#)을 적용할 수 있습니다.

## 코드 생성기 지원

두 가지 최종 기능은 C# 코드 생성기를 지원합니다. C# 코드 생성기는 Roslyn 분석기나 코드 수정 사항과 유사하게, 직접 작성할 수 있는 구성 요소입니다. 차이점은 코드 생성기의 경우 컴파일 프로세스의 일부로 코드를 분석하고 새 소스 코드 파일을 작성한다는 것입니다. 일반적인 코드 생성기는 코드에서 특성이나 다른 규칙을 검색합니다.

코드 생성기는 Roslyn 분석 API를 사용하여 특성이나 다른 코드 요소를 읽습니다. 해당 정보를 통해 컴파일에 새 코드를 추가합니다. 소스 생성기는 코드를 추가할 수만 있고 컴파일의 기존 코드를 수정할 수는 없습니다.

코드 생성기에 대해 추가된 두 가지 기능은 '부분 메서드 구문'에 대한 확장과 '모듈 이니셜라이저'입니다. 먼저 부분 메서드 변경 내용입니다. C# 9.0 이전에는 부분 메서드가 `private` 이지만 액세스 한정자를 지정하거나 `void` 반환 또는 `out` 매개 변수를 사용할 수 없습니다. 이 제한 사항 때문에, 메서드 구현을 제공하지 않을 경우 컴파일러에서 부분 메서드 호출을 모두 제거합니다. C# 9.0에서는 해당 제한 사항이 제거되었지만 부분 메서드 선언에 구현이 필요합니다. 코드 생성기에서 이 구현을 제공할 수 있습니다. 호환성이 손상되는 변경을 방지하기 위해 컴파일러에서 액세스 한정자가 없는 부분 메서드는 이전 규칙을 따른다고 간주합니다. 부분 메서드에 `private` 액세스 한정자가 포함되어 있으면 새 규칙에 따라 부분 메서드가 제어됩니다.

코드 생성기의 두 번째 새로운 기능은 `*모듈 이니셜라이저**`입니다. 모듈 이니셜라이저는 [ModuleInitializerAttribute](#) 특성이 연결된 메서드입니다. 이러한 메서드는 전체 모듈 내의 다른 필드 액세스 또는 메서드 호출 전에 런타임에서 호출됩니다. 모듈 이니셜라이저 메서드는 다음과 같아야 합니다.

- 정적이어야 함
- 매개 변수가 없어야 함
- `void`를 반환해야 함
- 제네릭 메서드가 아니어야 함
- 제네릭 클래스에 포함되지 않아야 함

- 포함하는 모듈에서 액세스할 수 있어야 함

마지막 글머리 기호 사항은 사실상, 메서드와 메서드를 포함하는 클래스가 `internal` 또는 `public`이어야 함을 의미합니다. 메서드는 로컬 함수가 될 수 없습니다.

# C# 8.0의 새로운 기능

2020-11-02 • 42 minutes to read • [Edit Online](#)

C#8.0은 다음 기능 및 향상된 기능을 C# 언어에 추가합니다.

- [읽기 전용 멤버](#)
- [기본 인터페이스 메서드](#)
- [패턴 일치 개선 사항:](#)
  - [Switch 식](#)
  - [속성 패턴](#)
  - [튜플 패턴](#)
  - [위치 패턴](#)
- [using 선언](#)
- [정적 로컬 함수](#)
- [삭제 가능한 ref struct](#)
- [nullable 참조 형식](#)
- [비동기 스트림](#)
- [비동기 삭제 가능](#)
- [인덱스 및 범위](#)
- [null 병합 할당](#)
- [관리되지 않는 생성 형식](#)
- [중첩 식의 stackalloc](#)
- [보간된 약어 문자열의 향상된 기능](#)

C#8.0은 .NET Core 3.x 및 .NET Standard 2.1에서 지원됩니다. 자세한 내용은 [C# 언어 버전 관리](#)를 참조하세요.

이 문서의 나머지 부분에서는 이러한 기능에 대해 간략하게 설명합니다. 심화 문서가 공개되면 해당 자습서에 대한 링크 및 개요가 제공됩니다. `dotnet try` 글로벌 도구를 사용하여 환경에서 다음과 같은 기능을 탐색할 수 있습니다.

1. [dotnet-try](#) 글로벌 도구를 설치합니다.
2. [dotnet/try-samples](#) 리포지토리를 복제합니다.
3. 현재 디렉터리를 `try-samples` 리포지토리의 `csharp8` 하위 디렉터리로 설정합니다.
4. `dotnet try`를 실행합니다.

## 읽기 전용 멤버

구조체의 멤버에 `readonly` 한정자를 적용할 수 있습니다. 이 한정자는 멤버가 상태를 수정하지 않음을 나타냅니다. 이것이 `readonly` 한정자를 `struct` 선언에 적용하는 것보다 더 세부적입니다. 다음과 같이 변경 가능한 구조체를 고려합니다.

```

public struct Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Distance => Math.Sqrt(X * X + Y * Y);

    public override string ToString() =>
        $"({X}, {Y}) is {Distance} from the origin";
}

```

대부분의 구조체와 마찬가지로 `ToString()` 메서드는 상태를 수정하지 않습니다. `readonly` 한정자를 `ToString()`의 선언에 추가하여 다음을 나타낼 수 있습니다.

```

public readonly override string ToString() =>
    $"({X}, {Y}) is {Distance} from the origin";

```

`ToString`은 `readonly`로 표시되지 않은 `Distance` 속성에 액세스하므로 위와 같이 변경할 경우 컴파일러 경고가 생성됩니다.

```

warning CS8656: Call to non-readonly member 'Point.Distance.get' from a 'readonly' member results in an
implicit copy of 'this'

```

컴파일러는 방어 복사본을 만들 필요가 있을 때 사용자에게 경고합니다. `Distance` 속성은 상태를 변경하지 않으므로 `readonly` 한정자를 선언에 추가하여 이 경고를 수정할 수 있습니다.

```

public readonly double Distance => Math.Sqrt(X * X + Y * Y);

```

읽기 전용 속성에는 `readonly` 한정자가 필요합니다. 컴파일러는 `get` 접근자가 상태를 수정하지 않는다고 가정하지 않습니다. 명시적으로 `readonly`를 선언해야 합니다. 단, 자동 구현 속성은 예외입니다. 컴파일러에서 모든 자동 구현 getter를 `readonly`로 처리하므로 예제의 `X` 및 `Y` 속성에는 `readonly` 한정자를 추가할 필요가 없습니다.

컴파일러는 `readonly` 멤버가 상태를 수정하지 않는다는 규칙을 적용합니다. `readonly` 한정자를 제거하지 않을 경우 다음 메서드는 컴파일되지 않습니다.

```

public readonly void Translate(int xOffset, int yOffset)
{
    X += xOffset;
    Y += yOffset;
}

```

이 기능을 사용하여 디자인 의도를 지정할 수 있으므로 컴파일러는 이를 적용하고 디자인 의도에 따라 최적화를 수행할 수 있습니다.

자세한 내용은 [구조체 형식](#) 문서의 [readonly 인스턴스 멤버](#) 섹션을 참조하세요.

## 기본 인터페이스 메서드

이제 인터페이스에 멤버를 추가하고 해당 멤버에 대한 구현을 제공할 수 있습니다. 이 언어 기능을 사용하여 API 작성자는 소스 또는 이진과 해당 인터페이스의 기존 구현과의 호환성을 영향을 미치지 않고 후속 버전에서 인터페이스에 멤버를 추가할 수 있습니다. 기존 구현은 기본 구현을 상속합니다. 또한 이 기능을 사용하여 C#은 유사한 기능을 지원하는 Android 또는 Swift를 대상으로 하는 API와 상호 운용됩니다. 또한 기본 인터페이스 멤버는 “특성” 언어 기능과 유사한 시나리오를 사용하도록 설정합니다.

기본 인터페이스 메서드는 많은 시나리오와 언어 요소에 영향을 줍니다. 첫 번째 자습서에서는 [기본 구현으로 인터페이스 업데이트](#)에 대해 다룹니다. 다른 자습서 및 참조 업데이트는 일반 릴리스 시점에 제공됩니다.

## 더 많은 곳에서 더 많은 패턴 사용

패턴 일치는 관련이 있으나 유형이 다른 데이터에서 세이프 종속 기능을 사용할 수 있도록 지원하는 도구를 제공합니다. C# 7.0부터 `is` 식과 `switch` 문을 사용하는 형식 패턴 및 상수 패턴을 위한 구문이 추가되었습니다. 이 기능은 데이터와 기능을 각각 별도로 구분하는 프로그래밍 패러다임을 지원하기 위한 조심스러운 첫 번째 단계입니다. 업계에서 마이크로 서비스와 기타 클라우드 기반 아키텍처를 도입하는 사례가 늘어남에 따라 다른 언어 도구에 대한 요구가 높아졌습니다.

C# 8.0은 코드의 더 많은 곳에서 더 많은 패턴 식을 사용할 수 있도록 이 용어를 확장합니다. 데이터와 기능을 각각 별도로 구분하는 경우 이러한 기능을 고려해 보시기 바랍니다. 알고리즘이 개체의 런타임 형식이 아닌 다른 요소에 종속되는 경우 패턴 일치를 고려해 보시기 바랍니다. 이러한 기법은 디자인을 표현할 추가적인 방법을 제공합니다.

C# 8.0에는 더 많은 곳에서 패턴을 사용할 수 있도록 지원하는 기능에 더해 재귀 패턴도 추가되었습니다. 패턴 식의 결과는 식입니다. 재귀 패턴은 다른 패턴 식의 출력에 적용된 패턴 식입니다.

### Switch 식

`switch` 문의 결과 각각의 `case` 블록에서 값이 생성되는 경우가 종종 있습니다. **Switch 식**을 사용하면 더욱더 간결한 식 구문을 사용할 수 있습니다. `case` 및 `break` 키워드를 반복적으로 사용하고 중괄호를 적용해야 하는 경우가 줄어듭니다. 다음과 같이 무지개의 색을 나열하는 enum 예를 살펴보겠습니다.

```
public enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

애플리케이션이 `R`, `G` 및 `B` 구성 요소에서 생성된 `RGBColor` 형식을 정의한 경우 `switch` 식을 포함하는 다음 메서드를 사용하여 `Rainbow` 값을 RGB 값으로 변환할 수 있습니다.

```
public static RGBColor FromRainbow(Rainbow colorBand) =>
    colorBand switch
    {
        Rainbow.Red      => new RGBColor(0xFF, 0x00, 0x00),
        Rainbow.Orange   => new RGBColor(0xFF, 0x7F, 0x00),
        Rainbow.Yellow   => new RGBColor(0xFF, 0xFF, 0x00),
        Rainbow.Green    => new RGBColor(0x00, 0xFF, 0x00),
        Rainbow.Blue     => new RGBColor(0x00, 0x00, 0xFF),
        Rainbow.Indigo   => new RGBColor(0x4B, 0x00, 0x82),
        Rainbow.Violet   => new RGBColor(0x94, 0x00, 0xD3),
        _                  => throw new ArgumentException(message: "invalid enum value", paramName:
nameof(colorBand)),
    };
}
```

여기에서 몇 가지 구문 개선 사항을 확인할 수 있습니다.

- 변수가 `switch` 키워드 앞에 옵니다. 이처럼 순서가 변경된 결과 `switch` 식과 `switch` 문을 눈으로 구분하기가 쉬워졌습니다.
- `case` 및 `:` 요소가 `=>`으로 대체되어 간결성과 직관성이 향상되었습니다.

- `default` 케이스가 `_` 무시 항목으로 대체되었습니다.
- 본문이 문이 아닌 식입니다.

기존의 `switch` 문을 사용하는 동일한 코드와 비교해 보세요.

```
public static RGBColor FromRainbowClassic(Rainbow colorBand)
{
    switch (colorBand)
    {
        case Rainbow.Red:
            return new RGBColor(0xFF, 0x00, 0x00);
        case Rainbow.Orange:
            return new RGBColor(0xFF, 0x7F, 0x00);
        case Rainbow.Yellow:
            return new RGBColor(0xFF, 0xFF, 0x00);
        case Rainbow.Green:
            return new RGBColor(0x00, 0xFF, 0x00);
        case Rainbow.Blue:
            return new RGBColor(0x00, 0x00, 0xFF);
        case Rainbow.Indigo:
            return new RGBColor(0x4B, 0x00, 0x82);
        case Rainbow.Violet:
            return new RGBColor(0x94, 0x00, 0xD3);
        default:
            throw new ArgumentException(message: "invalid enum value", paramName: nameof(colorBand));
    };
}
```

## 속성 패턴

속성 패턴을 사용하면 검사 대상 개체의 속성을 일치시킬 수 있습니다. 구매자의 주소를 기준으로 판매세를 계산하는 전자상거래 사이트를 예로 들어 보겠습니다. 판매세 계산은 `Address` 클래스의 주요 임무가 아닙니다. 판매세는 시간이 흐름에 따라 주소 형식이 변경되는 빈도보다 자주 변경될 가능성이 큽니다. 판매세의 금액은 주소의 `State` 속성에 종속됩니다. 다음 메서드는 속성 패턴을 사용하여 주소 및 가격으로부터 판매세를 컴퓨팅합니다.

```
public static decimal ComputeSalesTax(Address location, decimal salePrice) =>
    location switch
    {
        { State: "WA" } => salePrice * 0.06M,
        { State: "MN" } => salePrice * 0.075M,
        { State: "MI" } => salePrice * 0.05M,
        // other cases removed for brevity...
        _ => 0M
    };
}
```

패턴 일치는 이 알고리즘을 표현하기 위한 더 간결한 구문을 만듭니다.

## 튜플 패턴

일부 알고리즘은 여러 입력을 사용합니다. 튜플 패턴을 사용하면 튜플로 표현되는 여러 값에 따라 전환할 수 있습니다. 다음 코드는 게임 '가위, 바위, 보'에 대한 전환 식을 보여 줍니다.

```

public static string RockPaperScissors(string first, string second)
=> (first, second) switch
{
    ("rock", "paper") => "rock is covered by paper. Paper wins.",
    ("rock", "scissors") => "rock breaks scissors. Rock wins.",
    ("paper", "rock") => "paper covers rock. Paper wins.",
    ("paper", "scissors") => "paper is cut by scissors. Scissors wins.",
    ("scissors", "rock") => "scissors is broken by rock. Rock wins.",
    ("scissors", "paper") => "scissors cuts paper. Scissors wins.",
    (_, _) => "tie"
};

```

메시지는 승자를 나타냅니다. 버리기 사례는 동률의 세 가지 조합 또는 기타 텍스트 입력을 나타냅니다.

### 위치 패턴

일부 형식에는 해당 속성을 불연속 변수로 분해하는 `Deconstruct` 메서드가 포함됩니다. `Deconstruct` 메서드에 액세스할 수 있는 경우 **위치 패턴**을 사용하여 개체의 속성을 검사하고 패턴에 해당 속성을 사용할 수 있습니다. `x` 및 `y`의 불연속 변수를 만들려면 `Deconstruct` 메서드를 포함하는 다음 `Point` 클래스를 사용하는 것이 좋습니다.

```

public class Point
{
    public int X { get; }
    public int Y { get; }

    public Point(int x, int y) => (X, Y) = (x, y);

    public void Deconstruct(out int x, out int y) =>
        (x, y) = (X, Y);
}

```

또한 사분면의 다양한 위치를 나타내는 다음 열거형을 고려하세요.

```

public enum Quadrant
{
    Unknown,
    Origin,
    One,
    Two,
    Three,
    Four,
    OnBorder
}

```

다음 메서드는 위치 패턴을 사용하여 `x`와 `y`의 값을 추출합니다. 그런 다음, `when` 절을 사용하여 점의 `Quadrant`를 확인합니다.

```

static Quadrant GetQuadrant(Point point) => point switch
{
    (0, 0) => Quadrant.Origin,
    var (x, y) when x > 0 && y > 0 => Quadrant.One,
    var (x, y) when x < 0 && y > 0 => Quadrant.Two,
    var (x, y) when x < 0 && y < 0 => Quadrant.Three,
    var (x, y) when x > 0 && y < 0 => Quadrant.Four,
    var (_, _) => Quadrant.OnBorder,
    _ => Quadrant.Unknown
};

```

이전 `switch`의 버리기 패턴은 `x` 또는 `y` 중 하나만 0인 경우에 일치합니다. `expression` 식은 항상 값을 생성하

거나 예외를 throw해야 합니다. 일치하는 케이스가 없으면 switch 식이 예외를 throw합니다. switch 식이 가능한 모든 케이스를 포함하지 않으면 컴파일러에서 경고가 생성됩니다.

이 [패턴 일치에 대한 고급 자습서](#)에서 패턴 일치 기법을 탐색할 수 있습니다.

## Using 선언

using 선언은 `using` 키워드 뒤에 오는 변수 선언입니다. using 선언은 선언되는 변수를 바깥쪽 범위의 끝에서 삭제하라고 컴파일러에 알립니다. 텍스트 파일을 쓰는 다음 코드를 예로 들어 보겠습니다.

```
static int WriteLinesToFile(IEnumerable<string> lines)
{
    using var file = new System.IO.StreamWriter("WriteLines2.txt");
    int skippedLines = 0;
    foreach (string line in lines)
    {
        if (!line.Contains("Second"))
        {
            file.WriteLine(line);
        }
        else
        {
            skippedLines++;
        }
    }
    // Notice how skippedLines is in scope here.
    return skippedLines;
    // file is disposed here
}
```

위 예에서 메서드의 닫는 중괄호에 도달하면 파일이 삭제됩니다. 이 지점이 바로 `file`이 선언된 범위의 끝입니다. 위 코드는 기존의 [using 문](#) 문을 사용하는 다음 코드와 동일합니다.

```
static int WriteLinesToFile(IEnumerable<string> lines)
{
    using (var file = new System.IO.StreamWriter("WriteLines2.txt"))
    {
        int skippedLines = 0;
        foreach (string line in lines)
        {
            if (!line.Contains("Second"))
            {
                file.WriteLine(line);
            }
            else
            {
                skippedLines++;
            }
        }
        return skippedLines;
    } // file is disposed here
}
```

위 예에서 `using` 문의 닫는 중괄호에 도달하면 파일이 삭제됩니다.

두 경우 모두 컴파일러가 `Dispose()`를 호출합니다. `using` 문의 식을 삭제할 수 없는 경우 컴파일러에서 오류를 생성합니다.

## 정적 로컬 함수

이제 로컬 함수가 바깥쪽 범위의 변수를 캡처(참조)하지 않도록 [로컬 함수](#)에 `static` 한정자를 추가할 수 있습니다.

니다. 이렇게 하면 `CS8421`, "A static local function can't contain a reference to <variable>"(정적 로컬 함수는 <variable> 참조를 포함할 수 없습니다.)이(가) 생성됩니다.

다음과 같은 코드를 생각해 볼 수 있습니다. 여기서 로컬 함수 `LocalFunction`은 바깥쪽 범위(메서드 `M`)에서 선언된 변수 `y`에 액세스합니다. 따라서 `LocalFunction`을 `static` 한정자와 함께 선언할 수 없습니다.

```
int M()
{
    int y;
    LocalFunction();
    return y;

    void LocalFunction() => y = 0;
}
```

다음 코드에는 정적 로컬 함수가 포함되어 있습니다. 여기서 로컬 함수는 바깥쪽 범위의 변수에 액세스하지 않으므로 정적이 될 수 있습니다.

```
int M()
{
    int y = 5;
    int x = 7;
    return Add(x, y);

    static int Add(int left, int right) => left + right;
}
```

## 삭제 가능한 ref struct

`ref` 한정자를 사용하여 선언된 `struct`는 인터페이스를 구현할 수 없으므로 `IDisposable`을 구현할 수 없습니다. 따라서 `ref struct`가 삭제 가능해지려면 액세스 가능한 `void Dispose()` 메서드를 가져야 합니다. 이 기능은 `readonly ref struct` 선언에도 적용됩니다.

## nullable 참조 형식

`nullable` 주석 컨텍스트에서 참조 형식의 변수는 모두 `nullable`이 아닌 참조 형식으로 간주됩니다. 변수가 `null`이 될 수 있음을 나타내려면 형식 이름 뒤에 `?>`를 추가하여 해당 변수를 `nullable` 참조 형식으로 선언해야 합니다.

`nullable`이 아닌 참조 형식의 경우, 로컬 변수가 선언될 때 `null`이 아닌 값으로 초기화되도록 컴파일러가 흐름 분석을 사용합니다. 필드는 생성 시점에 초기화되어야 합니다. 사용 가능한 생성자 호출이나 이니셜라이저를 통해 변수가 설정되지 않으면 컴파일러에서 경고를 생성합니다. 또한, `nullable`이 아닌 참조 형식은 `null`이 될 수 있는 값에 할당할 수 없습니다.

`nullable` 참조 형식은 할당되지 않았는지 또는 `null`로 초기화되었는지 검사되지 않습니다. 단, `nullable` 참조 형식의 변수가 액세스되거나 `nullable`이 아닌 참조 형식에 할당되기 전에 `null`에 대해 검사되도록 컴파일러가 흐름 분석을 사용합니다.

[nullable 참조 형식](#) 개요에서 이 기능에 대해 자세히 알아볼 수 있습니다. [nullable 참조 형식 자습서](#)를 활용하여 새 애플리케이션에서 직접 사용해 보세요. [자습서: nullable 참조 형식이 있는 기존 코드 마이그레이션에서는 nullable 참조 형식을 사용할 수 있도록 기존 코드베이스를 마이그레이션하는 방법](#)을 알아볼 수 있습니다.

## 비동기 스트림

C# 8.0부터 스트림을 비동기식으로 만들고 사용할 수 있습니다. 비동기 스트림을 반환하는 메서드는 다음과 같은 세 가지 속성을 갖습니다.

1. `async` 한정자를 사용하여 선언되었습니다.
2. `IAsyncEnumerable<T>`를 반환합니다.
3. 비동기 스트림의 연속적인 요소를 반환하기 위해 메서드가 `yield return` 문을 포함합니다.

비동기 스트림을 사용하려면 스트림의 요소를 열거할 때 `foreach` 키워드 앞에 `await` 키워드를 사용해야 합니다. `await` 키워드를 추가하려면 비동기 스트림을 열거하는 메서드가 `async` 한정자와 함께 선언되어야 하며, 이 메서드가 `async` 메서드를 허용하는 형식을 반환해야 합니다. 일반적으로 이는 `Task` 또는 `Task<TResult>` 형식을 반환해야 함을 의미합니다. `ValueTask` 또는 `ValueTask<TResult>` 형식도 가능합니다. 메서드는 비동기 스트림을 사용할 수도 있고 생성할 수 있습니다. 비동기 스트림을 생성한다는 것은 메서드가 `IAsyncEnumerable<T>`를 반환한다는 것을 의미합니다. 다음 코드는 0에서 19까지의 시퀀스를 생성합니다. 숫자가 생성되는 각 시점 사이에는 100ms의 대기 시간이 있습니다.

```
public static async System.Collections.Generic.IAsyncEnumerable<int> GenerateSequence()
{
    for (int i = 0; i < 20; i++)
    {
        await Task.Delay(100);
        yield return i;
    }
}
```

`await foreach` 문을 사용하여 시퀀스를 열거합니다.

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

**비동기 스트림 생성 및 사용** 자습서에서 직접 비동기 스트림을 사용해 볼 수 있습니다. 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 삭제 가능

C# 8.0부터 언어는 `System.IAsyncDisposable` 인터페이스를 구현하는 비동기 삭제 가능 형식을 지원합니다. `await using` 문을 사용하여 삭제 가능한 객체를 비동기적으로 사용합니다. 자세한 내용은 [DisposeAsync](#) 메서드 구현 문서를 참조하세요.

## 인덱스 및 범위

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.Index`는 인덱스를 시퀀스로 표현합니다.
- 인덱스가 시퀀스의 끝을 기준으로 하도록 지정하는 끝부터 인덱스 연산자 `^`입니다.
- `System.Range`는 시퀀스의 하위 범위를 나타냅니다.
- 범위의 시작과 끝을 피연산자로 지정하는 범위 연산자 `..`입니다.

인덱스에 대한 규칙을 사용하여 시작하겠습니다. `sequence` 배열을 고려합니다. `0` 인덱스는 `sequence[0]`과 동일합니다. `~0` 인덱스는 `sequence[sequence.Length]`와 동일합니다. `sequence[^0]`은 `sequence[sequence.Length]`처럼 예외를 throw합니다. `n`이 어떤 숫자이든, 인덱스 `^n`은 `sequence.Length - n`과 동일합니다.

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.

몇 가지 예를 살펴보겠습니다. 다음과 같은 배열이 있습니다. 앞에서부터의 인덱스와 뒤에서부터의 인덱스가 주석으로 처리되어 있습니다.

```
var words = new string[]
{
    // index from start      index from end
    "The",      // 0            ^9
    "quick",    // 1            ^8
    "brown",    // 2            ^7
    "fox",      // 3            ^6
    "jumped",   // 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.

```
Console.WriteLine($"The last word is {words[^1]}");
// writes "dog"
```

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.

```
var quickBrownFox = words[1..4];
```

다음 코드는 "lazy"와 "dog"를 포함하는 하위 범위를 만듭니다. 이 하위 범위에는 `words[^2]` 과 `words[^1]` 이 포함되며. 끝 인덱스 `words[^0]` 은 포함되지 않습니다.

```
var lazyDog = words[^2..^0];
```

다음 예제는 시작만, 끝만, 그리고 시작과 끝이 모두 열린 범위를 만듭니다.

```
var allWords = words[..]; // contains "The" through "dog".
var firstPhrase = words[..4]; // contains "The" through "fox"
var lastPhrase = words[6..]; // contains "the", "lazy" and "dog"
```

범위를 변수로 선언할 수도 있습니다.

```
Range phrase = 1..4;
```

이렇게 변수로 선언된 범위는 `[ ]` 문자와 `]`  문자 사이에 사용할 수 있습니다.

```
var text = words[phrase];
```

배열만 인덱스와 범위를 지원합니다. [문자열](#), [Span<T>](#) 또는 [ReadOnlySpan<T>](#)에서 인덱스 및 범위를 사용할 수도 있습니다. 자세한 내용은 [인덱스 및 범위에 대한 형식 지원](#)을 참조하세요.

인덱스와 범위에 대한 자세한 내용은 [인덱스 및 범위에 대한 자습서](#)에서 확인할 수 있습니다.

## null 병합 할당

C# 8.0에서 null 병합 할당 연산자 `??=` 가 도입되었습니다. `??=` 연산자를 사용하여 왼쪽 피연산자가 `null`로 계산되는 경우에만 오른쪽 피연산자의 값을 왼쪽 피연산자에 대입할 수 있습니다.

```
List<int> numbers = null;
int? i = null;

numbers ??= new List<int>();
numbers.Add(i ??= 17);
numbers.Add(i ??= 20);

Console.WriteLine(string.Join(" ", numbers)); // output: 17 17
Console.WriteLine(i); // output: 17
```

자세한 내용은 [?? 및 ??= 연산자](#) 문서를 참조하세요.

## 관리되지 않는 생성 형식

C# 7.3 및 이전 버전에서 생성 형식(하나 이상의 형식 인수를 포함하는 형식)은 [관리되지 않는 형식](#)일 수 없습니다. C# 8.0부터는 관리되지 않는 형식의 필드만 포함된 경우 생성된 값 형식이 관리되지 않습니다.

예를 들어 다음과 같은 제네릭 `Coords<T>` 형식의 정의를 살펴보겠습니다.

```
public struct Coords<T>
{
    public T X;
    public T Y;
}
```

`Coords<int>` 형식은 C# 8.0 이상에서 관리되지 않는 형식입니다. 관리되지 않는 형식과 마찬가지로 이 형식의 변수에 대한 포인터를 만들거나 이 형식의 인스턴스에 대해 [스택에서 메모리 블록을 할당](#)할 수 있습니다.

```
Span<Coords<int>> coordinates = stackalloc[]
{
    new Coords<int> { X = 0, Y = 0 },
    new Coords<int> { X = 0, Y = 3 },
    new Coords<int> { X = 4, Y = 0 }
};
```

자세한 내용은 [관리되지 않는 형식](#)을 참조하세요.

## 중첩 식의 `stackalloc`

C# 8.0부터 `stackalloc` 식의 결과가 `System.Span<T>` 또는 `System.ReadOnlySpan<T>` 형식이면 다른 식에서 `stackalloc` 식을 사용할 수 있습니다.

```
Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };
var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6, 8 });
Console.WriteLine(ind); // output: 1
```

## 보간된 약어 문자열의 향상된 기능

[보간된](#) 약어 문자열에서 `$` 및 `@` 토큰은 순서에 관계없이 사용할 수 있습니다. `$$..."` 및 `@$..."` 모두 유효한 보간된 약어 문자열입니다. 이전 C# 버전에서는 `$` 토큰이 `@` 토큰 앞에 나타나야 했습니다.

# C# 7.0~C# 7.3의 새로운 기능

2021-02-18 • 66 minutes to read • [Edit Online](#)

C# 7.0~C# 7.3에서는 C#을 사용한 개발 환경에 다양한 기능과 증분 개선 사항을 도입했습니다. 이 문서에서는 새로운 언어 기능과 컴파일러 옵션에 대해 간략하게 설명합니다..NET Framework 기반 애플리케이션에 대해 지원되는 최신 버전인 C# 7.3의 동작을 설명합니다.

C# 7.1에 추가된 언어 버전 선택 구성 요소를 사용하면 프로젝트 파일에서 컴파일러 언어 버전을 지정할 수 있습니다.

C# 7.0~7.3에서는 C# 언어에 다음 기능과 테마를 추가합니다.

- **튜플 및 무시 항목**
  - 여러 public 필드가 포함된 간단한 명명되지 않은 형식을 만들 수 있습니다. 컴파일러 및 IDE 도구는 이러한 형식의 의미 체계를 이해합니다.
  - 삭제는 할당된 값에 신경 쓰지 않을 때 할당에서 사용되는 임시 쓰기 전용 변수입니다. 매개 변수는 `out` 매개 변수를 사용하여 메서드를 호출할 때뿐만 아니라 튜플 및 사용자 정의 형식을 분해할 때 특히 유용합니다.
- **패턴 일치**
  - 임의 형식 및 해당 형식의 멤버 값에 따라 분기 논리를 만들 수 있습니다.
- **`async` `Main` 메서드**
  - 애플리케이션에 대한 진입점은 `async` 한정자를 가질 수 있습니다.
- **로컬 함수**
  - 함수를 다른 함수 내부에 중첩하여 범위와 표시 여부를 제한할 수 있습니다.
- **추가 식 본문 멤버**
  - 식을 사용하여 작성할 수 있는 멤버 목록이 증가했습니다.
- **`throw` 식**
  - `throw` 문이었기 때문에 이전에 허용되지 않은 코드 구문에서 예외를 throw할 수 있습니다.
- **`default` 리터럴 식**
  - 대상 형식을 유추할 수 있는 경우 기본 값 식에서 기본 리터럴 식을 사용할 수 있습니다.
- **숫자 리터럴 구문 개선 사항**
  - 새로운 토큰으로 숫자 상수의 가독성이 향상됩니다.
- **`out` 변수**
  - `out` 값을 사용되는 메서드에 대한 인수로 인라인으로 선언할 수 있습니다.
- **뒤에 오지 않는 명명된 인수**
  - 명명된 인수 뒤에는 위치 인수가 올 수 있습니다.
- **`private protected` 액세스 한정자**
  - `private protected` 액세스 한정자는 동일한 어셈블리의 파생된 클래스에 대해 액세스를 사용합니다.
- **향상된 오버로드 확인**
  - 오버로드 확인 모호성을 해결하는 새로운 규칙
- **안전하고 효율적인 코드를 작성하는 방법**
  - 참조 의미 체계를 사용하는 값 유형으로 작동할 수 있는 구문 개선의 조합입니다.

마지막으로, 컴파일러에는 다음과 같은 새 옵션이 있습니다.

- `-refout` 및 `-refonly` - 참조 어셈블리 생성을 제어합니다.
- `-publicsign` - OSS(오픈 소스 소프트웨어) 시그니처를 사용하도록 설정합니다.

- `-pathmap` - 소스 디렉터리에 대한 매핑을 제공합니다.

이 문서의 나머지 부분에서는 해당 기능에 대한 개요를 제공합니다. 각 기능의 논리적 근거와 구문을 알아봅니다. `dotnet try` 글로벌 도구를 사용하여 환경에서 다음과 같은 기능을 탐색할 수 있습니다.

1. `dotnet-try` 글로벌 도구를 설치합니다.
2. `dotnet/try-samples` 리포지토리를 복제합니다.
3. 현재 디렉터리를 `try-samples` 리포지토리의 `csharp7` 하위 디렉터리로 설정합니다.
4. `dotnet try`를 실행합니다.

## 튜플 및 무시 항목

C#에서는 디자인 의도를 설명하는 데 사용되는 클래스 및 구조체에 대한 다양한 구문을 제공합니다. 하지만 다양한 구문에는 이점이 거의 없는데 추가 작업이 필요한 경우가 있습니다. 일반적으로 두 개 이상의 데이터 요소가 포함된 간단한 구조체가 필요한 메서드를 작성할 수 있습니다. 이러한 시나리오를 지원하기 위해 튜플이 C#에 추가되었습니다. 튜플은 데이터 멤버를 나타내는 여러 필드가 포함된 간단한 데이터 구조입니다. 필드의 유효성이 검사되지 않고 고유한 메서드를 정의할 수 없습니다. C# 튜플 형식은 `==` 및 `!=`를 지원합니다. 자세한 내용은

### NOTE

튜플은 C# 7.0 이전부터 사용할 수 있었지만 비효율적이었고 언어 지원이 없었습니다. 즉, 튜플 요소는 `Item1`, `Item2` 등으로만 참조될 수 있었습니다. C# 7.0은 새롭고 보다 효율적인 튜플 유형을 사용하여 튜플의 필드에 대해 의미론적 이름을 사용할 수 있는 튜플에 대한 언어 지원을 소개합니다.

각 멤버에 값을 할당하고 필요에 따라 튜플의 각 멤버에 의미 체계 이름을 입력하여 튜플을 만들 수 있습니다.

```
(string Alpha, string Beta) namedLetters = ("a", "b");
Console.WriteLine($"{namedLetters.Alpha}, {namedLetters.Beta}");
```

`namedLetters` 튜플에는 `Alpha` 및 `Beta`라는 필드가 포함됩니다. 이러한 이름은 컴파일 시간에만 존재하며 런타임 시 리플렉션을 통해 튜플을 검사하는 경우 보존되지 않습니다.

튜플 할당에서 할당의 오른쪽에 필드의 이름을 지정할 수도 있습니다.

```
var alphabetStart = (Alpha: "a", Beta: "b");
Console.WriteLine($"{alphabetStart.Alpha}, {alphabetStart.Beta}");
```

메서드에서 반환된 튜플의 멤버를 패키지 해제하려는 경우가 있을 수 있습니다. 이 작업을 수행하려면 튜플에서 각 값에 대한 개별 변수를 선언합니다. 이 패키지 해제 작업을 튜플 `분해`라고 합니다.

```
(int max, int min) = Range(numbers);
Console.WriteLine(max);
Console.WriteLine(min);
```

.NET에 있는 형식에 대한 비슷한 분해를 제공할 수도 있습니다. `Deconstruct` 메서드를 클래스의 멤버로 작성합니다. 해당 `Deconstruct` 메서드는 추출하려는 각 속성에 대한 `out` 인수 집합을 제공합니다. `X` 및 `Y` 좌표를 추출하는 분해자 메서드를 제공하는 이 `Point` 클래스를 살펴봅니다.

```
public class Point
{
    public Point(double x, double y)
        => (X, Y) = (x, y);

    public double X { get; }
    public double Y { get; }

    public void Deconstruct(out double x, out double y) =>
        (x, y) = (X, Y);
}
```

튜플에 `Point`을 할당하여 개별 필드를 추출할 수 있습니다.

```
var p = new Point(3.14, 2.71);
(double X, double Y) = p;
```

여러 번 튜플을 초기화할 때 할당의 오른쪽에 사용되는 변수는 튜플 요소에 대해 원하는 이름과 동일합니다. 튜플 요소의 이름은 튜플을 초기화하는 데 사용된 변수를 통해 유추할 수 있습니다.

```
int count = 5;
string label = "Colors used in the map";
var pair = (count, label); // element names are "count" and "label"
```

**튜플 형식** 문서에서 해당 기능을 자세히 알아볼 수 있습니다.

종종 튜플을 분해하거나 `out` 매개 변수로 메서드를 호출할 때 값을 신경 쓰지 않고 사용하지 않을 변수를 정의해야 합니다. C#은 버림에 대한 지원을 추가하여 이 시나리오를 처리합니다. 무시는 이름이 `_`(밑줄 문자)인 쓰기 전용 변수입니다. 단일 변수에 버리려는 모든 값을 할당할 수 있습니다. 버림은 할당 문과 별도로 분리되는 할당되지 않은 변수와 같습니다. 코드에서 버림을 사용할 수 없습니다.

다음 시나리오에서는 버림이 지원되지 않습니다.

- 튜플이나 사용자 정의 형식을 분해할 때.
- `out` 매개 변수로 메서드를 호출할 때.
- `is` 및 `switch` 문을 사용한 패턴 일치 작업에서.
- 할당 값을 버림으로 명시적으로 지정할 때 독립 실행형 식별자인 경우.

다음 예제에서는 서로 다른 2년간의 도시 데이터가 포함된 6 튜플을 반환하는 `QueryCityDataForYears` 메서드를 정의합니다. 예제의 메서드 호출은 메서드에 의해 반환된 두 개의 채우기 값에만 관련되어 있으므로 튜플을 해체할 때 튜플의 나머지 값을 버림으로 처리합니다.

```

using System;
using System.Collections.Generic;

public class Example
{
    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

```

자세한 내용은 [버림](#)을 참조하세요.

## 패턴 일치

'패턴 일치'는 코드에서 제어 흐름을 표현하는 새로운 방법을 제공하는 기능 세트입니다. 형식, 값 또는 속성 값에 대해 변수를 테스트할 수 있습니다. 해당 기술은 좀 더 읽기 쉬운 코드 흐름을 만듭니다.

패턴 일치는 `is` 식과 `switch` 식을 지원합니다. 각 식을 통해 개체 및 관련 속성을 검사하여 해당 개체가 검색된 패턴을 충족하는지 확인할 수 있습니다. `when` 키워드를 사용하여 패턴에 대한 추가 규칙을 지정합니다.

`is` 패턴 식은 친숙한 `is 연산자`를 확장하여 해당 형식에 대한 개체를 쿼리하고 하나의 명령에서 결과를 할당합니다. 다음 코드는 변수가 `int`인지 검사하고, 그럴 경우 현재 합계에 추가합니다.

```

if (input is int count)
    sum += count;

```

앞서 간단한 예제에서는 `is` 식의 개선 사항을 보여줍니다. 참조 형식뿐만 아니라 값 형식에 대해 테스트할 수 있고, 올바른 형식의 새로운 변수에 성공적인 결과를 할당할 수 있습니다.

`switch` 일치 식에는 이미 C# 언어의 일부인 `switch` 문에 기반을 둔 친숙한 구문이 있습니다. 업데이트된 `switch` 문에는 몇 가지 새로운 구문이 포함됩니다.

- `switch` 식의 관리 형식은 더 이상 정수 형식, `Enum` 형식, `string` 또는 해당 형식 중 하나에 해당하는

nullable 형식으로 제한되지 않습니다. 모든 형식을 사용할 수 있습니다.

- 각 `case` 레이블에서 `switch` 식 형식을 테스트할 수 있습니다. `is` 식과 마찬가지로 해당 형식에 새 변수를 할당할 수 있습니다.
- `when` 절을 해당 변수의 추가 테스트 조건에 추가할 수 있습니다.
- 이제 `case` 레이블의 순서가 중요합니다. 일치하는 첫 번째 분기가 실행됩니다. 다른 분기는 건너뜁니다.

다음 코드에서는 이 새로운 기능을 보여줍니다.

```
public static int SumPositiveNumbers(IEnumerable<object> sequence)
{
    int sum = 0;
    foreach (var i in sequence)
    {
        switch (i)
        {
            case 0:
                break;
            case IEnumerable<int> childSequence:
            {
                foreach(var item in childSequence)
                    sum += (item > 0) ? item : 0;
                break;
            }
            case int n when n > 0:
                sum += n;
                break;
            case null:
                throw new NullReferenceException("Null found in sequence");
            default:
                throw new InvalidOperationException("Unrecognized type");
        }
    }
    return sum;
}
```

- `case 0:` 은 친숙한 상수 패턴입니다.
- `case IEnumerable<int> childSequence:` 는 형식 패턴입니다.
- `case int n when n > 0:` 은 추가 `when` 조건을 포함한 형식 패턴입니다.
- `case null:` 은 null 패턴입니다.
- `default:` 는 친숙한 기본 사례입니다.

C# 7.1부터 `is` 및 `switch` 형식 패턴의 패턴 식에는 제네릭 형식 매개 변수의 형식이 포함될 수 있습니다. 이 방법은 `struct` 또는 `class` 형식 중 하나일 수 있는 형식을 확인하고 boxing을 방지하려는 경우에 가장 유용합니다.

[C#의 패턴 일치](#)에서 패턴 일치에 대해 자세히 알아볼 수 있습니다.

## 비동기 기본

비동기 기본 메서드를 통해 `Main` 메서드에서 `await`를 사용할 수 있습니다. 이전에 다음을 작성해야 했습니다.

```
static int Main()
{
    return DoAsyncWork().GetAwaiter().GetResult();
}
```

이제 다음을 작성할 수 있습니다.

```
static async Task<int> Main()
{
    // This could also be replaced with the body
    // DoAsyncWork, including its await expressions:
    return await DoAsyncWork();
}
```

프로그램이 종료 코드를 반환하지 않는 경우 `Task`를 반환하는 `Main` 메서드를 선언할 수 있습니다.

```
static async Task Main()
{
    await SomeAsyncMethod();
}
```

프로그래밍 가이드의 [비동기 기본](#) 문서에서 세부 정보에 대해 자세히 읽어볼 수 있습니다.

## 로컬 함수

클래스에 대한 대부분의 디자인에는 한 위치에서만 호출되는 메서드가 포함됩니다. 이러한 추가 `private` 메서드는 각 메서드를 작고 집중되게 유지합니다. 로컬 함수를 사용하면 다른 메서드의 컨텍스트 내부에서 메서드를 선언할 수 있습니다. 로컬 함수를 통해 클래스 `readers`는 로컬 메서드가 선언된 컨텍스트에서만 호출된다는 것을 더 쉽게 알 수 있습니다.

로컬 함수는 일반적으로 공용 반복기 메서드 및 공용 비동기 메서드에 사용됩니다. 두 메서드 형식 모두 프로그래머가 예상한 것보다 늦게 오류를 보고하는 코드를 생성합니다. 반복기 메서드에서 모든 예외는 반환된 시퀀스를 열거하는 코드를 호출할 경우에만 관찰됩니다. 비동기 메서드에서 모든 예외는 반환된 `Task`가 대기 상태일 경우에만 관찰됩니다. 다음 예제에서는 로컬 함수를 사용하여 반복기 구현으로부터 매개 변수 유효성 검사를 분리하는 방법을 보여줍니다.

```
public static IEnumerable<char> AlphabetSubset3(char start, char end)
{
    if (start < 'a' || start > 'z')
        throw new ArgumentOutOfRangeException(paramName: nameof(start), message: "start must be a letter");
    if (end < 'a' || end > 'z')
        throw new ArgumentOutOfRangeException(paramName: nameof(end), message: "end must be a letter");

    if (end <= start)
        throw new ArgumentException($"{nameof(end)} must be greater than {nameof(start)}");

    return alphabetSubsetImplementation();

    IEnumerable<char> alphabetSubsetImplementation()
    {
        for (var c = start; c < end; c++)
            yield return c;
    }
}
```

`async` 메서드에서 같은 방법을 사용하면 인수 유효성 검사에서 발생하는 예외가 비동기 작업이 시작되기 전에 `throw`됩니다.

```

public 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 longRunningWorkImplementation();

    async Task<string> longRunningWorkImplementation()
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}. Enjoy.";
    }
}

```

이제 다음 구문이 지원됩니다.

```

[field: SomeThingAboutFieldAttribute]
public int SomeProperty { get; set; }

```

`SomeThingAboutFieldAttribute` 특성은 `SomeProperty`에 대한 컴파일러 생성 지원 필드에 적용됩니다. 자세한 내용은 C# 프로그래밍 가이드의 [특성](#)을 참조하세요.

#### NOTE

로컬 함수가 지원하는 일부 디자인은 람다 식을 사용하여 수행할 수도 있습니다. 자세한 내용은 [로컬 함수와 람다 식 비교](#)를 참조하세요.

## 추가 식 본문 멤버

C# 6에서는 멤버 함수의 식 본문 멤버 및 읽기 전용 속성을 추가했습니다. C# 7.0에서는 식으로 구현될 수 있는 허용 멤버를 확장합니다. C# 7.0에서는 속성 및 인덱서에 대한 생성자, 종료자 및 `get` / `set` 접근자를 구현할 수 있습니다. 다음 코드는 각각에 대한 예제를 보여 줍니다.

```

// Expression-bodied constructor
public ExpressionMembersExample(string label) => this.Label = label;

// Expression-bodied finalizer
~ExpressionMembersExample() => Console.Error.WriteLine("Finalized!");

private string label;

// Expression-bodied get / set accessors.
public string Label
{
    get => label;
    set => this.label = value ?? "Default label";
}

```

#### NOTE

이 예제에는 종료자가 필요하지 않지만 구문을 보여 주기 위해 표시됩니다. 관리되지 않는 리소스를 릴리스해야 하는 경우가 아니면 클래스에서 종료자를 구현하면 안 됩니다. 관리되지 않는 리소스를 직접 관리하지 않고 [SafeHandle](#) 클래스를 사용하는 것도 고려해야 합니다.

식 본문 멤버의 새 위치는 C# 언어의 중요한 마일스톤을 나타냅니다. 이러한 기능은 오픈 소스 [Roslyn](#) 프로젝트에서 작업하는 커뮤니티 멤버에 의해 구현되었습니다.

메서드를 식 본문 멤버로 변경하는 것은 [이진 호환 가능 변경](#)입니다.

## Throw 식

C#에서 `throw`는 항상 문이었습니다. `throw`는 식이 아닌 명령문이므로 이 명령문을 사용할 수 없는 C# 구문이 있었습니다. 이러한 구문에는 조건식, null 병합 식 및 몇몇 람다 식이 포함되었습니다. 식 본문 멤버가 추가됨에 따라 `throw` 식이 유용할 수 있는 추가 위치도 추가됩니다. 이러한 구문을 작성할 수 있도록 C# 7.0에서는 [throw 식](#)을 추가합니다.

이렇게 추가하면 자세한 식 기반 코드를 쉽게 작성할 수 있습니다. 오류 검사를 위해 명령문을 추가할 필요는 없습니다.

## 기본 리터럴 식

기본 리터럴 식은 기본값 식에 대한 향상된 기능입니다. 이러한 식은 변수를 기본 값으로 초기화합니다. 여기서 이전에 다음을 작성했습니다.

```
Func<string, bool> whereClause = default(Func<string, bool>);
```

이제 초기화의 오른쪽에서 형식을 생략할 수 있습니다.

```
Func<string, bool> whereClause = default;
```

자세한 내용은 [기본 연산자](#) 문서의 [기본 리터럴](#) 섹션을 참조하세요.

## 숫자 리터럴 구문 개선 사항

숫자 상수를 잘못 읽으면 코드를 처음 읽을 때 이해하기가 더 어려울 수 있습니다. 비트 마스크 또는 다른 기호 값은 잘못 이해하기 쉽습니다. C# 7.0에는 의도한 용도에 맞게 가장 읽기 쉬운 방식으로 숫자를 작성할 수 있는 [이진 리터럴](#) 및 숫자 구분 기호라는 두 가지 새로운 기능이 포함됩니다.

비트 마스크를 만들 때 또는 숫자의 이진 표현이 가장 읽기 쉬운 코드를 만들 때마다 해당 숫자를 이진으로 작성하세요.

```
public const int Sixteen = 0b0001_0000;
public const int ThirtyTwo = 0b0010_0000;
public const int SixtyFour = 0b0100_0000;
public const int OneHundredTwentyEight = 0b1000_0000;
```

상수의 시작 부분에 있는 `0b`는 숫자가 이진 숫자로 작성되었음을 나타냅니다. 이진 숫자는 길어질 수 있으므로, 이전 예제의 이진 상수와 같이 `_`을 숫자 구분 기호로 추가하면 일반적으로 비트 패턴을 더 쉽게 확인할 수 있습니다. 숫자 구분 기호는 상수 내의 어디에나 나타날 수 있습니다. 10진수 숫자의 경우 숫자 구분 기호를 천 단위 구분 기호로 사용하는 것이 일반적입니다. 16진수 및 이진 숫자 리터럴은 `_`로 시작할 수도 있습니다.

```
public const long BillionsAndBillions = 100_000_000_000;
```

숫자 구분 기호는 `decimal`, `float` 및 `double` 형식에서도 사용할 수 있습니다.

```
public const double AvogadroConstant = 6.022_140_857_747_474e23;
public const decimal GoldenRatio = 1.618_033_988_749_894_848_204_586_834_365_638_117_720_309_179M;
```

함께 사용하면 숫자 상수를 훨씬 더 읽기 쉽게 선언할 수 있습니다.

## out 변수

`out` 매개 변수를 지원하는 기존 구문이 C# 7에서 개선되었습니다. 이제 개별 선언 문을 작성하는 대신 메서드 호출의 인수 목록에서 `out` 변수를 선언할 수 있습니다.

```
if (int.TryParse(input, out int result))
    Console.WriteLine(result);
else
    Console.WriteLine("Could not parse input");
```

이전 예제와 같이 `out` 변수의 형식을 명확하게 지정하는 것이 좋습니다. 그러나 이 언어는 암시적 형식 지역 변수를 지원하지 않습니다.

```
if (int.TryParse(input, out var answer))
    Console.WriteLine(answer);
else
    Console.WriteLine("Could not parse input");
```

- 코드를 읽기가 더 쉽습니다.
  - `out` 변수는 앞의 코드 줄이 아니라 변수를 사용하는 곳에서 선언합니다.
- 초기 값을 할당할 필요가 없습니다.
  - 메서드 호출에서 사용되는 위치에 `out` 변수를 선언하여 변수가 할당되기 전에 실수로 사용할 수 없습니다.

`out` 변수 선언이 허용하기 위해 C# 7.0에 추가된 구문은 필드 이니셜라이저, 속성 이니셜라이저, 생성자 이니셜라이저 및 쿼리 절을 포함하도록 확장되었습니다. 이를 통해 다음 예제와 같은 코드를 사용할 수 있습니다.

```
public class B
{
    public B(int i, out int j)
    {
        j = i;
    }
}

public class D : B
{
    public D(int i) : base(i, out var j)
    {
        Console.WriteLine($"The value of 'j' is {j}");
    }
}
```

## 뒤에 오지 않는 명명된 인수

명명된 인수가 올바른 위치에 있을 때 메서드 호출은 이제 위치 인수보다 앞에 있는 명명된 인수를 사용할 수 있습니다. 자세한 내용은 [명명된 인수 및 선택적 인수](#)를 참조하세요.

## *private protected* 액세스 한정자

새로운 복합 액세스 한정자인 `private protected`는 동일한 어셈블리에 선언된 클래스 또는 파생 클래스를 포함하여 멤버에 액세스할 수 있음을 나타냅니다. `protected internal`은 동일한 어셈블리에 있는 파생 클래스나 클래스에 의한 액세스를 허용하지만 `private protected`는 동일한 어셈블리에서 선언된 파생 유형에 대한 액세스를 제한합니다.

자세한 내용은 언어 참조에서 [액세스 한정자](#)를 참조하세요.

## 향상된 오버로드 후보

모든 릴리스에서 오버로드 해결 규칙은 모호한 메서드 호출에 “분명한” 선택이 있는 상황을 해결하도록 업데이트됩니다. 이 릴리스는 컴파일러가 분명한 선택 항목을 선택하도록 도와주는 세 가지 새로운 규칙을 추가합니다.

1. 메서드 그룹에 인스턴스와 정적 멤버가 둘 다 포함되어 있으면 인스턴스 수신기 또는 컨텍스트 없이 호출되는 경우 컴파일러는 인스턴스 멤버를 버립니다. 메서드가 인스턴스 수신기를 통해 호출된 경우 컴파일러는 정적 멤버를 버립니다. 수신기가 없는 경우 컴파일러는 정적 컨텍스트에 정적 멤버만 포함하고, 이외의 경우에는 정적 및 인스턴스 멤버를 둘 다 포함합니다. 수신기가 모호하게 인스턴스 또는 형식인 경우에는 컴파일러가 둘 다 포함합니다. 암시적 `this` 인스턴스 수신기를 사용할 수 없는 정적 컨텍스트에는 정적 멤버와 같이 `this`가 정의되지 않은 멤버의 본문과 필드 이니셜라이저 및 생성자 이니셜라이저와 같이 `this`를 사용할 수 없는 위치가 포함됩니다.
2. 형식 인수가 해당 제약 조건을 충족하지 않는 제네릭 메서드가 메서드 그룹에 포함된 경우 이러한 멤버는 후보 집합에서 제거됩니다.
3. 메서드 그룹 변환의 경우 반환 형식이 대리자의 반환 형식과 일치하지 않는 후보 메서드가 집합에서 제거됩니다.

어떤 메서드가 더 나은지 알고 있으면 모호한 메서드 오버로드에 대한 컴파일러 오류가 감소하므로 이 변경을 알 수 있습니다.

## 좀 더 효율적인 안전한 코드 사용

안전하게 실행할 C# 코드 및 안전하지 않은 코드를 작성할 수 있어야 합니다. 안전한 코드를 사용하면 버퍼 오버런, 이상 포인터 및 기타 메모리 액세스 오류와 같은 오류 클래스를 피할 수 있습니다. 이러한 새 기능은 확인 가능한 안전한 코드의 기능을 확장합니다. 안전한 구문을 사용하여 더 많은 코드를 작성하도록 노력하세요. 이러한 기능을 사용하면 이 작업이 더 쉬워집니다.

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.
- 추가적인 제네릭 제약 조건을 사용할 수 있습니다.
- 매개 변수의 `in` 한정자는 인수가 참조에 의해 전달되지만 호출된 메서드에 의해 수정되지 않도록 지정합니다. `in` 한정자를 인수에 추가하는 것은 [소스 호환 가능 변경](#)입니다.
- 메서드의 `ref readonly` 한정자는 메서드가 참조별 값을 반환하지만 해당 개체에 대한 쓰기를 허용하지 않음을 나타내기 위해 반환합니다. 반환된 값에 할당되는 경우 `ref readonly` 한정자를 추가하는 것은 [소스 호환 가능 변경](#)입니다. 기존 `ref` 반환 문에 `readonly` 한정자를 추가하는 것은 [호환되지 않는 변경](#)입니다. 호출자가 `readonly` 한정자를 포함하도록 `ref` 지역 변수의 선언을 업데이트하도록 합니다.

- `readonly struct` 선언은 구조체가 변경 가능하지 않으며 `in` 매개 변수로서 멤버 메서드로 전달되어야 함을 나타냅니다. 기존 구조체 선언에 `readonly` 한정자를 추가하는 것은 [이진 호환 가능 변경](#)입니다.
- `ref struct` 선언은 구조체 유형이 관리되는 메모리에 직접 액세스하고 항상 스택에 할당되어야 함을 나타냅니다. 기존 `struct` 선언에 `ref` 한정자를 추가하는 것은 [호환되지 않는 변경](#)입니다. `ref struct`는 클래스의 멤버가 되거나 힙에 할당될 수 있는 다른 위치에서 사용될 수 없습니다.

[안전하고 효율적인 코드 작성](#)에서 모든 변경 내용을 자세히 읽을 수 있습니다.

### Ref local 및 return

이 기능을 통해 다른 곳에 정의된 변수에 대한 참조를 사용 및 반환하는 알고리즘이 가능해집니다. 한 가지 예는 큰 매트릭스를 사용하고 특정 특징을 가진 하나의 위치를 찾는 것입니다. 다음 메서드는 참조를 행렬의 해당 스토리지에 반환합니다.

```
public static ref int Find(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

다음 코드에서처럼 반환 값을 `ref`로 선언하고 행렬에서 해당 값을 수정할 수 있습니다.

```
ref var item = ref MatrixSearch.Find(matrix, (val) => val == 42);
Console.WriteLine(item);
item = 24;
Console.WriteLine(matrix[4, 2]);
```

C# 언어에는 `ref`로컬 및 반환을 잘못 사용하지 않도록 방지하는 몇 가지 규칙이 있습니다.

- `ref` 키워드를 메서드 서명 및 메서드의 모든 `return` 문에 추가해야 합니다.
  - 그러면 메서드가 메서드 전체에서 참조별로 반환되도록 합니다.
- `ref return`은 값 변수 또는 `ref` 변수에 할당될 수 있습니다.
  - 호출자는 반환 값을 복사할지 여부를 제어합니다. 반환 값을 할당할 때 `ref` 한정자를 생략하면 호출자가 스토리지에 대한 참조가 아닌 값의 복사본을 요청한다는 것을 나타냅니다.
- 표준 메서드 반환 값을 `ref`로컬 변수에 할당할 수 없습니다.
  - 이로 인해 `ref int i = sequence.Count();` 같은 문이 허용되지 않습니다.
- 메서드 실행보다 길게 수명이 연장되지 않는 변수에 `ref`를 반환할 수 없습니다.
  - 즉, 로컬 변수 또는 비슷한 범위의 변수에 참조를 반환할 수 없습니다.
- `ref local` 및 `return`은 비동기 메서드와 함께 사용할 수 없습니다.
  - 컴파일러는 비동기 메서드가 반환될 때 참조된 변수가 최종 값으로 설정되었는지 여부를 알 수 없습니다.

`ref local` 및 `ref return`을 추가하면 값을 복사하거나 역참조 작업을 여러 번 수행하는 경우를 방지하여 더 효율적인 알고리즘이 가능해집니다.

`ref`를 반환 값에 추가하는 것은 [소스 호환 가능 변경](#)입니다. 기존 코드는 컴파일되지만, 참조 반환 값은 할당 시 복사됩니다. 호출자는 반환 값 스토리지를 `ref` 지역 변수로 업데이트하여 반환을 참조로 저장해야 합니다.

이제 `ref` 지역 변수를 다시 할당하여 초기화된 후 다른 인스턴스를 참조할 수 있습니다. 이제 다음 코드가 컴파일됩니다.

```
ref VeryLargeStruct refLocal = ref veryLargeStruct; // initialization  
refLocal = ref anotherVeryLargeStruct; // reassigned, refLocal refers to different storage.
```

자세한 내용은 [ref 반환 및 ref 지역](#)에 대한 아티클 및 [foreach](#)에 대한 아티클을 참조하세요.

자세한 내용은 [ref 키워드](#) 문서를 참조하세요.

조건부 [ref](#) 식

마지막으로, 조건식은 값 결과 대신 참조 결과를 생성할 수 있습니다. 예를 들어 다음을 작성하여 두 배열 중 하나의 첫 번째 요소에 대한 참조를 검색합니다.

```
ref var r = ref (arr != null ? ref arr[0] : ref otherArr[0]);
```

[r](#) 변수는 [arr](#) 또는 [otherArr](#)의 첫 번째 값에 대한 참조입니다.

자세한 내용은 언어 참조에서 [조건부 연산자\(?\)](#)를 참조하세요.

[in](#) 매개 변수 한정자

[in](#) 키워드는 기존 [ref](#) 및 [out](#) 키워드를 보완하여 참조로 인수를 전달합니다. [in](#) 키워드는 인수를 참조로 전달하도록 지정하지만 호출된 메서드는 값을 수정하지 않습니다.

다음 코드와 같이 값 또는 읽기 전용 참조로 전달되는 오버로드를 선언할 수 있습니다.

```
static void M(S arg);  
static void M(in S arg);
```

값으로 전달(이전 예제의 첫 번째 경우) 오버로드가 읽기 전용 참조로 전달 버전보다 더 효율적입니다. 읽기 전용 참조 인수를 사용하여 버전을 호출하려면 메서드를 호출할 때 [in](#) 한정자를 포함해야 합니다.

자세한 내용은 [in 매개 변수 한정자](#)에 대한 문서를 참조하세요.

더 많은 형식이 [fixed](#) 문을 지원함

[fixed](#) 문은 제한된 형식 집합을 지원했습니다. C# 7.3부터 [ref T](#) 또는 [ref readonly T](#)를 반환하는 [GetPinnableReference\(\)](#) 메서드를 포함하는 모든 형식이 [fixed](#) 일 수 있습니다. 이 기능을 추가하면 [fixed](#) 를 [System.Span<T>](#) 및 관련 형식과 함께 사용할 수 있습니다.

자세한 내용은 언어 참조에서 [fixed 문](#) 문서를 참조하세요.

[fixed](#) 필드 인덱싱에 고정이 필요하지 않음

다음 구조체를 고려합니다.

```
unsafe struct S  
{  
    public fixed int myFixedField[10];  
}
```

이전 버전의 C#에서는 [myFixedField](#)의 일부인 정수 중 하나에 액세스하려면 변수를 고정해야 했습니다. 이제 다음 코드는 별도의 [fixed](#) 문 안에 [p](#) 변수를 고정하지 않고 컴파일됩니다.

```
class C
{
    static S s = new S();

    unsafe public void M()
    {
        int p = s.myFixedField[5];
    }
}
```

p 변수는 `myFixedField`의 한 요소에 액세스합니다. 별도의 `int*` 변수를 선언할 필요가 없습니다. `unsafe` 키워드는 여전히 필요합니다. 이전 버전의 C#에서는 두 번째 고정된 포인터를 선언해야 합니다.

```
class C
{
    static S s = new S();

    unsafe public void M()
    {
        fixed (int* ptr = s.myFixedField)
        {
            int p = ptr[5];
        }
    }
}
```

자세한 내용은 [fixed 문](#)에 대한 문서를 참조하세요.

`stackalloc` 배열이 이니셜라이저를 지원함

초기화할 때 배열의 요소 값을 지정할 수 있었습니다.

```
var arr = new int[3] {1, 2, 3};
var arr2 = new int[] {1, 2, 3};
```

이제 `stackalloc`로 선언된 배열에 동일한 구문을 적용할 수 있습니다.

```
int* pArr = stackalloc int[3] {1, 2, 3};
int* pArr2 = stackalloc int[] {1, 2, 3};
Span<int> arr = stackalloc [] {1, 2, 3};
```

자세한 내용은 [stackalloc 연산자](#) 문서를 참조하세요.

향상된 제네릭 제약 조건

이제 형식 매개 변수에 대한 기본 클래스 제약 조건으로 `System.Enum` 또는 `System.Delegate` 형식을 지정할 수 있습니다.

또한 새 `unmanaged` 제약 조건을 사용하여 형식 매개 변수가 nullable이 아닌 [비관리형 형식](#)이 되도록 지정할 수 있습니다.

자세한 내용은 [where 제네릭 제약 조건](#) 및 [형식 매개 변수 제약 조건](#)에 대한 문서를 참조하세요.

기존 형식에 이러한 제약 조건을 추가하는 것은 [호환되지 않는 변경](#)입니다. 폐쇄형 제네릭 형식은 더 이상 이러한 새 제약 조건을 충족할 수 없습니다.

일반화된 비동기 반환 형식

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 반환

하거나 동기적으로 완료된 경우 코드의 성능이 중요한 섹션에서 추가 할당에 상당한 시간이 소요될 수 있습니다. 연속 루프에서 이러한 할당이 발생하면 부담이 될 수 있습니다.

새로운 언어 기능을 통해 비동기 메서드 반환 형식이 `Task`, `Task<T>` 및 `void`에 제한되지 않게 되었습니다. 반환된 형식은 비동기 패턴을 충족해야 합니다. 즉, `GetAwaiter` 메서드에 액세스할 수 있어야 합니다. 한 가지 구체적인 예로 이 새로운 언어 기능을 사용할 수 있도록 `ValueTask` 형식이 .NET에 추가되었습니다.

```
public async ValueTask<int> Func()
{
    await Task.Delay(100);
    return 5;
}
```

#### NOTE

`ValueTask<TResult>` 형식을 사용하려면 NuGet 패키지 `System.Threading.Tasks.Extensions`를 추가해야 합니다.

이 개선 사항은 라이브러리 작성자가 성능이 중요한 코드에서 `Task`를 할당하지 않도록 방지하는 데 유용합니다.

## 새로운 컴파일러 옵션

새로운 컴파일러 옵션은 C# 프로그램에 대한 새로운 빌드 및 DevOps 시나리오를 지원합니다.

### 참조 어셈블리 생성

참조 전용 어셈블리 인 `-refout` 및 `-refonly`를 생성하는 두 개의 새로운 컴파일러 옵션이 있습니다. 연결된 문서에서는 이러한 옵션과 참조 어셈블리를 보다 자세히 설명합니다.

### 공개 또는 오픈 소스 서명

`-publicsign` 컴파일러 옵션은 공개 키를 사용하여 어셈블리에 서명하도록 컴파일러에 지시합니다. 어셈블리는 서명됨으로 표시되지만 시그니처는 공개 키에서 가져옵니다. 이 옵션을 사용하면 공개 키를 사용하여 오픈 소스 프로젝트에서 서명된 어셈블리를 빌드할 수 있습니다.

자세한 내용은 [-publicsign 컴파일러 옵션](#) 문서를 참조하세요.

### pathmap

`-pathmap` 컴파일러 옵션은 빌드 환경의 소스 경로를 맵핑된 소스 경로로 바꾸도록 컴파일러에 지시합니다.  
`-pathmap` 옵션은 컴파일러에서 작성된 PDB 파일의 소스 경로 또는 `CallerFilePathAttribute`를 제어합니다.

자세한 내용은 [-pathmap 컴파일러 옵션](#) 문서를 참조하세요.

# C# 컴파일러의 호환성이 손상되는 변경에 대해 알 아봅니다

2020-11-02 • 2 minutes to read • [Edit Online](#)

Roslyn 팀은 C# 및 Visual Basic 컴파일러의 호환성이 손상되는 변경 목록을 유지 관리합니다. 변경 내용에 대한 정보는 GitHub 리포지토리의 다음 링크에서 확인할 수 있습니다.

- .NET 5.0 및 C# 9.0용으로 도입되는 VS2019 버전 16.8의 호환성이 손상되는 변경
- VS2019 대비 VS2019 업데이트 1 이상의 호환성이 손상되는 변경
- VS2017 이후 호환성이 손상되는 변경(C# 7)
- Roslyn 2.0(VS2017)과 달라진 Roslyn 3.0(VS2019)의 호환성이 손상되는 변경
- Roslyn 1.0(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

2021-02-18 • 35 minutes to read • [Edit Online](#)

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 리포지토리에서](#) 확인할 수 있습니다.

## IMPORTANT

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# 빌드 도구는 최신 주요 언어 릴리스가 기본 언어 버전으로 고려합니다. 주요 릴리스 사이에는 이 섹션의 다른 문서에서 상세히 설명한 포인트 릴리스가 있을 수 있습니다. 포인트 릴리스에서 최신 기능을 사용하려면 [컴파일러 언어 버전을 구성](#)하고 해당 버전을 선택해야 합니다. C# 7.0 이후 세 가지 포인트 릴리스가 있습니다.

- C# 7.3:
  - C# 7.3은 [Visual Studio 2017 버전 15.7](#) 및 [.NET Core 2.1 SDK](#)부터 사용할 수 있습니다.
- C# 7.2:
  - C# 7.2는 [Visual Studio 2017 버전 15.5](#) 및 [.NET Core 2.0 SDK](#)부터 사용할 수 있습니다.
- C# 7.1:
  - C# 7.1은 [Visual Studio 2017 버전 15.3](#) 및 [.NET Core 2.0 SDK](#)부터 사용할 수 있습니다.

## C# 버전 1.0

돌이켜보면 Visual Studio.NET 2002와 함께 릴리스된 C# 버전 1.0은 Java와 매우 비슷했습니다. [ECMA에 대해 명시된 설계 목표의 일부](#)로 "단순하고 현대적인 범용 개체 지향 언어"를 추구했습니다. 당시에 Java와 같은 형태는 이러한 초기 설계 목표를 달성한 것을 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

## C# 버전 1.2

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 2.0

이제 흥미로운 일이 시작됩니다. 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 일반적인 OO(개체 지향) 언어로 시작되었지만 C# 버전 2.0을 통해 급격히 바뀌었습니다. C#이 자리를 잡은 후 개발자들은 몇 가지 심각한 고민을 겪었습니다. 이후 대대적인 문제가 발생했습니다.

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. 모든 작업에서 `ArrayList`에서 파생된 `ListInt` 형식을 만들거나 `Object`에서 캐스팅하는 것보다 제네릭을 사용하는 것이 좋습니다.

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.

그리고 이때까지 C#은 Java를 따라잡으려는 노력을 계속했습니다. Java는 이미 제네릭 및 반복기가 포함된 버전을 출시했습니다. 하지만 언어가 계속 발전함에 따라 곧 변경될 것입니다.

## C# 버전 3.0

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. 일반적으로 C# 버전의 핵심 기능은 LINQ(Language-Integrated Query)라고도 하는 쿼리 식이라 생각합니다.

더 미묘한 뷰는 LINQ가 생성되는 기본인 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 조합은 정수 목록을 훨씬 더 효율적으로 보이게 해줍니다.

실제로 개념을 파악하고 통합하는 데는 시간이 걸렸지만 점진적으로 그 개념이 완성되었습니다. 그리고 몇 년이 지난 지금, 코드는 훨씬 더 간결하고 간단하며 기능적입니다.

## C# 버전 4.0

Visual Studio 2010과 함께 릴리스된 C# 버전 4.0은 버전 3.0의 혁신에 따른 기대에 부응하느라 어려움을 겪을 것으로 예상되었습니다. 버전 3.0에서 C#은 Java의 그림자를 확실히 벗어나 두각을 나타내었습니다. 금세 정교해졌습니다.

다음 버전에서는 몇 가지 흥미로운 새 기능이 도입되었습니다.

- 동적 바인딩
- 명명된/선택적 인수
- 제네릭 공변(covariant) 및 반공변(contravariant)
- 포함된 interop 형식

포함된 interop 형식은 배포의 어려움을 완화합니다. 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 5.0

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

관련 항목

- [코드 프로젝트: C# 5.0의 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 언어 첫 번째 클래스 참여자로 비동기를 적용하여 다시 업계의 판도를 바꾸었습니다. 장기 실행 작업 및 콜백 웹 구현을 처리한 적이 있다면 이 언어 기능을 좋아할 것입니다.

## C# 버전 6.0

버전 3.0과 5.0에서 C#은 개체 지향 언어의 주요 새 기능을 추가했습니다. Visual Studio 2015와 함께 릴리스된 버전 6.0은 주요 핵심 기능 대신 C# 프로그래밍을 보다 생산적으로 만드는 많은 작은 기능을 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 각 기능은 그 자체로 흥미롭습니다. 그러나 전체적으로 살펴보면 흥미로운 패턴을 볼 수 있습니다. 이 버전에서 C#은 언어 상용구를 제거하여 코드를 더 간결하고 읽기 쉽게 만들었습니다. 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 7.0

C# 버전 7.0은 Visual Studio 2017과 함께 릴리스되었습니다. 이 버전에는 C# 6.0 방식의 혁신적이고 유용한 기능이 있지만 서비스형 컴파일러는 없습니다. 다음은 새 기능 중 일부입니다.

- 외부 변수
- 튜플 및 분해
- 패턴 일치
- 로컬 함수
- 확장된 식 본문 멤버
- 참조 로컬 및 반환

이러한 기능에는 다음이 포함됩니다.

- 삭제
- 이진 리터럴 및 자릿수 구분 기호
- Throw 식

이러한 모든 기능은 개발자에게 멋진 새 기능과 이전보다 훨씬 깔끔한 코드를 작성할 수 있는 기회를 제공합니다.

다. 하이라이트는 `out` 키워드와 함께 사용할 변수의 선언을 압축하고 튜플을 통해 여러 개의 반환 값을 허용하는 것입니다.

그러나 C#은 더욱 광범위하게 사용되고 있습니다. .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 7.1

C#은 C# 7.1과 함께 '포인트 릴리스'를 제공하기 시작했습니다. 이 버전은 언어 버전 선택 구성 요소, 세 개의 새로운 언어 기능 및 새로운 컴파일러 동작을 추가했습니다.

이 릴리스의 새로운 언어 기능은 다음과 같습니다.

- `async Main` 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 변수의 패턴 일치
  - 형식이 제네릭 형식 매개 변수인 변수에서 패턴 일치 식을 사용할 수 있습니다.

마지막으로 컴파일러에는 참조 어셈블리 생성을 제어하는 두 가지 옵션 `-refout` 및 `-refonly`가 있습니다.

## C# 버전 7.2

C# 7.2는 몇 가지 작은 언어 기능을 추가했습니다.

- 안전하고 효율적인 코드를 작성하는 방법
  - 참조 의미 체계를 사용하는 값 유형으로 작동할 수 있는 구문 개선의 조합입니다.
- 뒤에 오지 않는 명명된 인수
  - 명명된 인수 뒤에는 위치 인수가 올 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.
- 조건부 `ref` 식
  - 이제 조건식(`?:`)의 결과가 참조일 수 있습니다.

## C# 버전 7.3

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.
- 추가적인 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.
- 인수에서 `in` 만 다른 경우 메서드 해결이 향상되었습니다.
- 이제 오버로드 해결에 모호한 사례가 감소했습니다.

새 컴파일러 옵션은 다음과 같습니다.

- `-publicsign` - OSS(오픈 소스 소프트웨어) 시그니처를 사용하도록 설정합니다.
- `-pathmap` - 소스 디렉터리에 대한 매핑을 제공합니다.

## C# 버전 8.0

C# 8.0은 특히 .NET Core C#을 대상으로 하는 첫 번째 주 릴리스입니다. 일부 기능은 새 CLR 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 개선 사항:
  - Switch 식
  - 속성 패턴
  - 튜플 패턴
  - 위치 패턴
- using 선언
- 정적 로컬 함수
- 삭제 가능한 ref struct
- nullable 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 3.0용 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.

'이 문서는 [NDepend 블로그에 최초로 게시되었습니다.](#) Erik Dietrich 및 Patrick Smacchia 제공.'

# 언어 기능 및 라이브러리 형식 간의 관계

2020-11-02 • 6 minutes to read • [Edit Online](#)

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. 필요한 경우 해당 형식이나 멤버가 아직 배포되지 않은 환경에 대한 코드를 작성할 때 최신 버전의 언어에 필요한 형식을 포함하는 NuGet 패키지가 있습니다.

표준 라이브러리 기능에 대한 이 종속성은 첫 번째 버전부터 C# 언어의 일부였습니다. 해당 버전에서 예제가 포함되어 있습니다.

- [Exception](#) - 모든 컴파일러 생성 예외에 사용됩니다.
- [String](#) - C# `string` 형식은 [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# 7.0은 [ValueTuple](#)에 종속성을 추가하여 튜플 언어 기능을 계속해서 구현합니다.

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. 작업에서 해당 종속성을 관리하는 방법을 이해하는 것이 중요합니다.

## 종속성 관리

C# 컴파일러 도구는 이제 지원되는 플랫폼의 .NET 라이브러리의 릴리스 주기에서 분리됩니다. 사실상 서로 다른 .NET 라이브러리는 다른 릴리스 주기를 갖습니다. Windows에서 .NET Framework는 Windows 업데이트로 릴리스되고, .NET Core는 별도 일정으로 제공되며, Xamarin 버전의 라이브러리 업데이트는 각 대상 플랫폼에 대한 Xamarin 도구로 제공됩니다.

대부분의 시간에서 이러한 변경 내용이 발견되지 않습니다. 그러나 해당 플랫폼의 .NET 라이브러리에 아직 없는 기능을 필요로 하는 언어의 최신 버전을 사용하는 경우 이러한 새 유형을 제공하는 NuGet 패키지를 참조할 수 있습니다. 플랫폼으로서 앱 지원은 새 프레임워크를 설치로 업데이트되며 추가 참조를 제거할 수 있습니다.

이러한 분리는 해당 프레임워크가 없을 수도 있는 컴퓨터를 대상으로 하는 경우에도 새로운 언어 기능을 사용할 수 있음을 의미합니다.

# C# 개발자를 위한 버전 및 업데이트 고려 사항

2020-11-02 • 6 minutes to read • [Edit Online](#)

C# 언어에 새로운 기능이 추가되므로 호환성은 매우 중요한 목표입니다. 대부분의 경우 문제없이 새 컴파일러 버전으로 기존 코드를 다시 컴파일할 수 있습니다.

라이브러리에서 새 언어 기능을 채택할 때는 더욱 주의해야 할 수 있습니다. 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. 새 기능을 채택하기로 결정하면 소스 호환 가능 및 이진 호환 가능이라는 두 가지 호환성 변형을 고려해야 합니다.

## 이진 호환 가능 변경

라이브러리를 사용하는 애플리케이션 및 라이브러리를 다시 빌드하지 않고 업데이트된 라이브러리를 사용할 수 있는 경우 라이브러리 변경은 **이진 호환 가능**합니다. 종속 어셈블리를 다시 빌드하거나 소스 코드를 변경하지 않아도 됩니다. 이진 호환 가능 변경은 소스 호환 가능 변경이기도 합니다.

## 소스 호환 가능 변경

라이브러리를 사용하는 애플리케이션 및 라이브러리에 대해 소스 코드를 변경하지 않아도 되지만 올바르게 작동하려면 새 버전에 대해 소스를 다시 컴파일해야 하는 경우 라이브러리 변경은 **소스 호환 가능**합니다.

## 호환되지 않는 변경

변경이 소스 호환 가능 또는 이진 호환 가능하지 않은 경우 종속 라이브러리 및 애플리케이션에서 소스 코드를 변경하고 다시 컴파일해야 합니다.

## 라이브러리 평가

이러한 호환성 개념은 라이브러리에 대한 public 및 protected 선언에 영향을 주며 내부 구현에는 영향을 주지 않습니다. 내부적으로 새로운 기능을 채택하면 항상 **이진 호환 가능**합니다.

이진 호환 가능 변경은 public 선언에 대해 이전 구문과 동일한 컴파일된 코드를 생성하는 새 구문을 제공합니다. 예를 들어 메서드를 식 본문 멤버로 변경하는 것은 **이진 호환 가능** 변경입니다.

원래 코드:

```
public double CalculateSquare(double value)
{
    return value * value;
}
```

새로운 코드:

```
public double CalculateSquare(double value) => value * value;
```

소스 호환 가능 변경은 public 멤버에 대해 컴파일된 코드를 변경하지만 기존 호출 사이트와 호환되는 방식으로 변경하는 구문을 도입합니다. 예를 들어 값 기준 매개 변수에서 `in` 참조 기준 매개 변수로 메서드 시그니처를 변경하는 것은 소스 호환 가능하지만 이진 호환 가능하지는 않습니다.

원래 코드:

```
public double CalculateSquare(double value) => value * value;
```

새로운 코드:

```
public double CalculateSquare(in double value) => value * value;
```

[새로운 기능](#) 문서에서는 public 선언에 영향을 주는 기능을 도입하는 것이 소스 호환 가능 또는 이전 호환 가능 한지 설명합니다.

# 형식(C# 프로그래밍 가이드)

2021-02-18 • 35 minutes to read • [Edit Online](#)

## 형식, 변수 및 값

C#은 강력한 형식의 언어입니다. 모든 변수 및 상수에는 값으로 계산되는 모든 식을 실행하는 형식이 있습니다. 모든 메서드 선언은 각 입력 매개 변수와 반환 값에 대해 이름, 매개 변수 수, 형식 및 종류(값, 참조 또는 출력)를 지정합니다. .NET 클래스 라이브러리는 기본 제공 숫자 형식 집합 및 파일 시스템, 네트워크 연결, 컬렉션, 개체 배열, 날짜 등의 다양한 논리 구문을 나타내는 더 복잡한 형식을 정의합니다. 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.
- 런타임에 변수에 대한 메모리가 할당될 위치.
- 허용되는 작업 유형.

컴파일러는 형식 정보를 사용하여 코드에서 수행되는 모든 작업의 형식이 안전한지 확인합니다. 예를 들어 `int` 형식의 변수를 선언할 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.

```
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;
```

### NOTE

C 및 C++ 개발자는 C#에서 `bool`이 `int`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.

```

// 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`가 필요하고 문자열을 반환하는 메서드를 보여 줍니다.

```

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`를 선언하고 `true`의 부울 값을 여기에 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#에서는 정수, 부동 소수점 값, 부울 식, 텍스트 문자, 10진수 값 및 기타 데이터 형식을 표현하는 기본 제공 형식의 표준 세트를 제공합니다. 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` 및 `enum` 구문을 사용하여 자체 사용자 지정 형식을 만듭니다. .NET 클래스 라이브러리 자체는 자체 애플리케이션에서 사용할 수 있는 Microsoft에서 제공되는 사용자 지정 형식의 컬렉션입니다. 기본적으로 클래스 라이브러리의 가장 자주 사용되는 형식을 모든 C# 프로그램에서 사용할 수 있습니다. 기타 형식은 정의되어 있는 어셈블리에 대한 프로젝트 참조를 명시적으로 추가할 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)

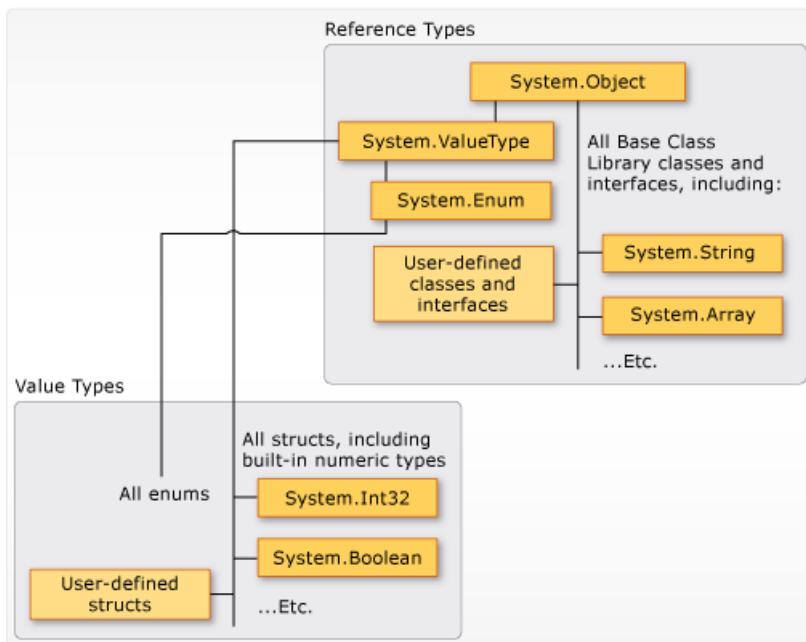
.NET의 형식 시스템에 대한 다음과 같은 두 가지 기초 사항을 이해해야 합니다.

- 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`)와 같은 기본 제공 숫자 형식을 포함한 모든 형식은 기본적으로 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`를 사용하여 정의한 형식은 값 형식이고, 모든 기본 제공 숫자 형식은 `structs`입니다. `class` 키워드를 사용하여 정의한 형식은 참조 형식입니다. 참조 형식과 값 형식의 컴파일 타입 규칙 및 런타임 동작은 서로 다릅니다.

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



#### NOTE

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.

### 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 포함되므로, 변수가 선언된 컨텍스트에 관계없이 메모리가 인라인으로 할당됩니다. 값 형식 변수에 대한 별도 힙 할당이나 가비지 수집 오버헤드는 없습니다.

값 형식에는 `struct` 및 `enum`의 두 가지 범주가 있습니다.

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

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

하지만 단순 비집계 형식처럼 값을 선언하고 변수에 할당합니다.

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

값 형식은 *sealed*입니다. 즉, `System.Int32`와 같은 값 형식에서 형식을 도출할 수 없습니다. 구조체는 `System.ValueType`에서만 상속할 수 있기 때문에 사용자 정의 클래스 또는 구조체에서 상속하는 구조체를 정의

할 수 없습니다. 그러나 구조체는 하나 이상의 인터페이스를 구현할 수 있습니다. 구조체 형식을 구조체가 구현하는 인터페이스 형식으로 캐스트할 수 있습니다. 이 캐스트로 인해 *boxing* 작업이 관리되는 힙의 참조 형식 객체 내에 구조체를 래핑합니다. *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.

```
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.

```
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

[클래스](#), [대리자](#), 배열 또는 [인터페이스](#)로 정의되는 형식은 참조 형식입니다. 런타임에 참조 형식의 변수를 선언하면 `new` 연산자를 사용하여 개체를 명시적으로 만들거나 다음 예제와 같이 `new`를 사용하여 다른 곳에서 만들어진 개체를 할당할 때까지 변수에는 `null` 값이 포함됩니다.

```
MyClass mc = new MyClass();
MyClass mc2 = mc;
```

인터페이스는 구현하는 클래스 개체와 함께 초기화되어야 합니다. `MyClass`가 `IMyInterface`를 구현하는 경우 다음 예제와 같이 `IMyInterface`의 인스턴스를 만듭니다.

```
IMyInterface iface = new MyClass();
```

개체가 만들어지면 관리되는 힙에 메모리가 할당되고 변수에는 개체 위치에 대한 참조만 포함됩니다. 관리되는

힙의 형식은 할당될 때, 그리고 가비지 수집이라는 CLR의 자동 메모리 관리 기능에 의해 회수될 때 오버헤드가 필요합니다. 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 파생되지만 다음 예제와 같이 C#에서 제공하는 단순한 구문을 사용하여 배열을 선언하고 사용할 수 있습니다.

```
// 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 상속될 수 있고 기타 클래스는 직접 만든 클래스에서 상속되고 가상 메서드를 재정의할 수 있습니다. 클래스를 직접 만드는 방법에 대한 자세한 내용은 [클래스 및 구조체](#)를 참조하세요. 상속 및 가상 메서드에 대한 자세한 내용은 [상속](#)을 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 4.56이 float로 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.

```
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

클라이언트 코드가 형식의 인스턴스를 만들 때 제공하는 실제 형식(구체적 형식)에 대한 자리 표시자로 사용되는 하나 이상의 [형식 매개 변수](#)를 사용하여 형식을 선언할 수 있습니다. 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: 문자열)을 지정합니다.

```
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 [개체](#)로 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

앞에서 설명한 대로 `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.

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

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

```
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.

- [캐스팅 및 형식 변환](#)
- [boxing 및 unboxing](#)
- [dynamic 형식 사용](#)
- [값 형식](#)
- [참조 형식](#)
- [클래스 및 구조체](#)
- [익명 형식](#)
- [제네릭](#)

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [XML 데이터 형식 변환](#)
- [정수 형식](#)

# nullable 참조 형식

2021-02-18 • 27 minutes to read • [Edit Online](#)

C# 8.0에는 참조 형식 변수의 속성에 대한 중요한 설명을 할 수 있는 nullable 참조 형식 과 nullable이 아닌 참조 형식 이 도입되었습니다.

- 참조는 null이 아니어야 합니다. 변수에 null이 허용되지 않는 경우 컴파일러는 null이 아닌지 먼저 확인하지 않고 이러한 참조를 역참조하기에 안전한지 확인하는 규칙을 적용합니다.
  - 이 변수는 null이 아닌 값으로 초기화되어야 합니다.
  - 이 변수에는 null 값을 절대로 할당할 수 없습니다.
- 참조에 null이 허용됩니다. 변수에 null이 허용되는 경우 컴파일러는 null 참조 검사를 제대로 했는지 확인하는 다른 규칙을 적용합니다.
  - 컴파일러가 해당 값이 null이 아님을 보장할 수 있는 경우에만 변수를 역참조할 수 있습니다.
  - 이러한 변수는 기본 null 값으로 초기화할 수 있으며 다른 코드에서 null 값을 할당할 수 있습니다.

이 새로운 기능은 변수 선언에서 설계 의도를 파악할 수 없었던 이전 버전의 C#에 비해 참조 변수 처리에 관한 상당한 이점을 제공합니다. 기존 컴파일러는 참조 형식의 null 참조 예외에 대해 안전성을 제공하지 않았습니다.

- 참조가 null일 수 있습니다. 참조 형식 변수가 null로 초기화되거나 나중에 null이 할당되는 경우 컴파일러는 경고를 발생하지 않습니다. 컴파일러는 이러한 변수가 null 검사 없이 역참조될 때 경고를 발생시킵니다.
- 참조가 null이 아닌 것으로 간주됩니다. 컴파일러는 참조 형식이 역참조될 때 어떠한 경고도 발생시키지 않습니다. 변수가 null일 수 있는 식으로 설정된 경우 컴파일러에서 경고를 발생시킵니다.

이러한 경고는 컴파일 시에 내보내집니다. 컴파일러는 null 허용 컨텍스트에 null 검사 또는 다른 런타임 구성을 추가하지 않습니다. 런타임 시 null 허용 참조와 null을 허용하지 않는 참조는 동일합니다.

Nullable 참조 형식을 추가함으로써 의도를 보다 명확하게 선언할 수 있습니다. null 값은 변수가 값을 참조하지 않음을 나타내는 올바른 방법입니다. 코드에서 모든 null 값을 제거하는 데 이 기능을 사용하지 마세요. 대신, 컴파일러 및 코드를 읽는 다른 개발자에게 의도를 선언해야 합니다. 의도를 선언해 놓으면 컴파일러는 해당 의도와 일치하지 않는 코드 작성 시 이를 알려줍니다.

nullable 참조 형식은 nullable 값 형식과 동일한 구문을 사용하여 작성합니다. 변수 형식에 ? 가 추가됩니다. 예를 들어 다음 변수 선언은 nullable 문자열 변수 name 을 나타냅니다.

```
string? name;
```

형식 이름에 ? 가 추가되지 않은 모든 변수는 null을 허용하지 않는 참조 형식입니다. 여기에는 이 기능을 설정한 기존 코드에 있는 모든 참조 형식 변수가 포함됩니다.

컴파일러는 정적 분석을 사용하여 nullable 참조가 null이 아닌 것으로 알려져 있는지 확인합니다. 컴파일러는 null일 수 있는 nullable 참조를 역참조하는 경우 경고를 표시합니다. 변수 이름 뒤에 null 허용 연산자 ! 를 사용하여 이 동작을 재정의할 수 있습니다. 예를 들어 name 변수가 null이 아닌 것으로 알고 있는데 컴파일러 경고가 발생하는 경우 다음 코드를 작성하여 컴파일러 분석을 재정의할 수 있습니다.

```
name!.Length;
```

## 형식의 Null 허용 여부

모든 참조 형식은 경고가 생성되는 경우를 설명하는 4가지 *nullabilities*(Null 허용 여부) 중 하나일 수 있습니다.

- *Nonnullable*(Null 허용 안 함): 이 형식의 변수에는 null을 할당할 수 없습니다. 이 형식의 변수는 역참조되기 전에 null 검사가 필요하지 않습니다.
- *Nullable*(Null 허용): 이 형식의 변수에는 null을 할당할 수 있습니다. `null`에 대한 사전 검사 없이 이 형식의 변수를 역참조하면 경고가 발생합니다.
- *Oblivious*(모호한): C# 8.0 이전 상태는 명확하지 않습니다. 이 형식의 변수는 경고 없이 역참조되거나 할당될 수 있습니다.
- *Unknown*(알 수 없음): Unknown(알 수 없음)은 형식이 nullable 또는 nonnullable이어야 하는지 제약 조건으로 컴파일러에 명시하지 않은 형식 매개 변수에 일반적입니다.

변수 선언에서 형식의 null 허용 여부는 변수가 선언된 nullable 컨텍스트로 제어됩니다.

## Nullable 컨텍스트

Nullable 컨텍스트를 통해 컴파일러가 참조 형식 변수를 해석하는 방식을 미세하게 제어할 수 있습니다. 지정된 소스 줄의 nullable 주석 컨텍스트는 enabled 또는 disabled입니다. C# 8.0 이전 컴파일러는 사용할 수 없는 nullable 컨텍스트에서 모든 코드를 컴파일하는 것으로 생각할 수 있습니다. 모든 참조 형식은 null일 수 있습니다. nullable 경고 컨텍스트도 enabled 또는 disabled일 수 있습니다. nullable 경고 컨텍스트는 컴파일러가 해당 흐름 분석을 사용하여 생성한 경고를 지정합니다.

.csproj 파일에 nullable 요소를 사용하여 프로젝트에 대한 nullable 주석 컨텍스트 및 nullable 경고 컨텍스트를 설정할 수 있습니다. 이 요소는 컴파일러가 형식의 null 허용 여부를 해석하는 방법 및 생성되는 경고를 구성합니다. 유효한 설정은 다음과 같습니다.

- `enable`: nullable 주석 컨텍스트가 **enabled**입니다. nullable 경고 컨텍스트가 **enabled**입니다.
  - 예를 들어 참조 형식 변수 `string`은 null이 아닙니다. 모든 null 허용 여부 경고가 enabled입니다.
- `warnings`: nullable 주석 컨텍스트가 **disabled**입니다. nullable 경고 컨텍스트가 **enabled**입니다.
  - 참조 형식의 변수가 모호합니다. 모든 null 허용 여부 경고가 enabled입니다.
- `annotations`: nullable 주석 컨텍스트가 **enabled**입니다. nullable 경고 컨텍스트가 **disabled**입니다.
  - 참조 형식의 변수(예: 문자열)는 null을 허용하지 않습니다. 모든 null 허용 여부 경고가 disabled입니다.
- `disable`: nullable 주석 컨텍스트가 **disabled**입니다. nullable 경고 컨텍스트가 **disabled**입니다.
  - 참조 형식의 변수는 이전 버전의 C#과 마찬가지로 모호할 수 있습니다. 모든 null 허용 여부 경고가 disabled입니다.

예제:

```
<Nullable>enable</Nullable>
```

또한 지시문을 사용하여 프로젝트의 아무 곳에나 이러한 동일한 컨텍스트를 설정할 수도 있습니다.

- `#nullable enable`: nullable 주석 컨텍스트와 nullable 경고 컨텍스트를 **enabled**로 설정합니다.
- `#nullable disable`: nullable 주석 컨텍스트와 nullable 경고 컨텍스트를 **disabled**로 설정합니다.
- `#nullable restore`: nullable 주석 컨텍스트와 nullable 경고 컨텍스트를 프로젝트 설정으로 복원합니다.
- `#nullable disable warnings`: nullable 경고 컨텍스트를 **disabled**로 설정합니다.
- `#nullable enable warnings`: nullable 경고 컨텍스트를 **enabled**로 설정합니다.
- `#nullable restore warnings`: nullable 경고 컨텍스트를 프로젝트 설정으로 복원합니다.
- `#nullable disable annotations`: nullable 주석 컨텍스트를 **disabled**로 설정합니다.
- `#nullable enable annotations`: nullable 주석 컨텍스트를 **enabled**로 설정합니다.

- `#nullable restore annotations`: 주석 경고 컨텍스트를 프로젝트 설정으로 복원합니다.

### IMPORTANT

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 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.

기본적으로 null 허용 주석 및 경고 컨텍스트는 새 프로젝트를 포함하여 사용 안 함입니다. 이는 기존 코드가 변경 없이 새로운 경고를 생성하지 않고 컴파일됨을 의미합니다.

이러한 옵션은 null 허용 참조 형식을 사용하도록 [기존 코드베이스를 업데이트](#)하는 두 가지 고유한 전략을 제공합니다.

## nullable 주석 컨텍스트

컴파일러는 disabled nullable 주석 컨텍스트에 다음 규칙을 사용합니다.

- disabled 컨텍스트에서 nullable 참조를 선언할 수 없습니다.
- 모든 참조 변수에는 null 값을 할당할 수 있습니다.
- 참조 형식의 변수가 역참조되면 경고가 생성되지 않습니다.
- disabled 컨텍스트에는 null 허용 연산자를 사용할 수 없습니다.

동작은 이전 버전의 C#과 같습니다.

컴파일러는 enabled nullable 주석 컨텍스트에 다음 규칙을 사용합니다.

- 모든 참조 형식의 변수는 nullable이 아닌 참조입니다.
- nullable이 아닌 모든 참조는 안전하게 역참조될 수 있습니다.
- 모든 nullable 참조 형식(변수 선언에서 형식 뒤에 `?` 표시)은 null일 수 있습니다. 정적 분석은 값이 역참조될 때 null이 아닌 것으로 알려져 있는지 확인합니다. 아닌 경우 컴파일러가 경고를 생성합니다.
- null 허용 연산자를 사용하여 nullable 참조가 null이 아님을 선언할 수 있습니다.

enabled nullable 주석 컨텍스트에서 참조 형식에 추가된 `?` 문자는 nullable 참조 형식을 선언합니다. null 비 허용 연산자 `!` 를 식 뒤에 추가하여 식이 null이 아님을 선언할 수 있습니다.

## Nullable 경고 컨텍스트

nullable 경고 컨텍스트는 nullable 주석 컨텍스트와 다릅니다. 새 주석이 해제(disabled)되어도 경고는 설정(enabled)할 수 있습니다. 컴파일러는 정적 흐름 분석을 사용하여 모든 참조에 대한 null 상태를 확인합니다. null 상태는 nullable 경고 컨텍스트가 disabled 가 아닌 경우 nullable이 아님 또는 null일 수 있음입니다. 컴파일러가 null일 수 있음으로 확인한 경우 역참조하면 경고가 발생합니다. 참조 상태는 컴파일러가 다음 두 조건 중 하나를 확인할 수 없으면 null일 수 있음입니다.

1. 변수에 null이 아닌 값이 명확하게 할당되었습니다.
2. 변수 또는 식이 역참조되기 전에 null 검사되었습니다.

컴파일러는 nullable 경고 컨텍스트에서 null일 수 있는 변수 또는 식을 역참조할 때 경고를 생성합니다. 또한

컴파일러는 활성화된 null 허용 주석 컨텍스트에서 null을 허용하지 않는 참조 형식에 null일 수 있음 변수 또는 식이 할당되면 경고를 생성합니다.

## API를 설명하는 특성

인수 또는 반환 값이 null일 수 있거나 null이 될 수 없는 경우에 대한 자세한 정보를 컴파일러에 제공하는 API에 특성을 추가합니다. [null 허용 특성](#)을 다루는 언어 참조의 문서에서 이러한 특성에 대해 자세히 알아볼 수 있습니다. 이러한 특성은 현재 및 이후 릴리스에서 .NET 라이브러리에 추가됩니다. 가장 일반적으로 사용되는 API가 먼저 업데이트됩니다.

## 알려진 문제

참조 유형을 포함하는 배열 및 구조체는 nullable 참조 형식 기능의 알려진 문제입니다.

### 구조체

null을 허용하지 않는 참조 형식을 포함하는 구조에서는 경고 없이 `default`를 할당할 수 있습니다. 다음 예제를 참조하세요.

```
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.

```
#nullable enable

public struct Foo<T>
{
    public T Bar { get; set; }
}

public static class Program
{
    public static void Main()
    {
        string s = default(Foo<string>).Bar;
    }
}
```

앞의 예제에서 `Bar` 속성은 런타임 시 `null`이 되고 경고 없이 `null`을 허용하지 않는 문자열에 할당됩니다.

## 배열

배열은 nullable 참조 형식의 알려진 문제가 되기도 합니다. 경고를 생성하지 않는 다음 예제를 고려하세요.

```
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.

## 참조

- [Nullable 참조 형식 사양 초안](#)
- [Nullable 참조 소개 자습서](#)
- [기존 코드베이스를 nullable 참조로 마이그레이션](#)
- [-nullable\(C# 컴파일러 옵션\)](#)

# Nullable 참조 형식을 사용하기 위해 라이브러리를 업데이트하고 호출자에 nullable 규칙 전달

2021-02-18 • 32 minutes to read • [Edit Online](#)

Nullable 참조 형식을 추가하면 모든 변수에 `null` 값이 허용되거나 필요한지를 선언할 수 있습니다. 또한 `AllowNull`, `DisallowNull`, `MaybeNull`, `NotNull`, `NotNullWhen`, `MaybeNullWhen`, `NotNullIfNotNull` 등의 여러 특성을 적용하여 인수와 반환 값의 null 상태를 완전히 설명할 수 있습니다. 따라서 코드를 작성할 때 뛰어난 환경을 제공합니다. Null을 허용하지 않는 변수를 `null`로 설정할 수 있으면 경고가 표시됩니다. Nullable 변수를 역참조하기 전에 `null` 검사하지 않은 경우 경고가 발생합니다. 라이브러리 업데이트는 시간이 걸릴 수 있지만 얻는 결과를 보면 가치가 있습니다. 컴파일러에 `null` 값이 허용되거나 금지되는 '경우'에 관해 더 많은 정보를 제공하면 API 사용자에게 더 유용한 정보가 제공됩니다. 친숙한 예제부터 살펴보겠습니다. 라이브러리에 리소스 문자열을 검색하는 다음 API가 있다고 가정합니다.

```
bool TryGetMessage(string key, out string message)
```

앞의 예제는 .NET의 친숙한 `Try*` 패턴을 따릅니다. 이 API에 대한 두 개의 참조 인수인 `key` 및 `message` 매개 변수가 있습니다. 이 API에는 해당 인수의 nullness에 관련된 다음 규칙이 있습니다.

- 호출자는 `null`을 `key`의 인수로 전달하지 않아야 합니다.
- 호출자는 값이 `null`인 변수를 `message`의 인수로 전달할 수 있습니다.
- `TryGetMessage` 메서드가 `true`를 반환하면 `message` 값은 null이 아닙니다. 반환 값이 `false`, 이면 `message`의 값과 해당 null 상태는 null입니다.

`key`에 대한 규칙은 변수 형식으로 완전히 표현될 수 있습니다. `key`는 `null`을 허용하지 않는 참조 형식이어야 합니다. `message` 매개 변수는 더 복잡합니다. `null`을 인수로 허용하지만, 성공 시 해당 `out` 인수가 null이 아님을 보장합니다. 이 시나리오의 경우 기대치를 설명하는 더 다양한 어휘가 필요합니다.

Nullable 참조를 위해 라이브러리를 업데이트하려면 일부 변수 및 형식 이름에 `?`를 포함하는 것 이상의 작업이 필요합니다. 앞의 예제에서는 API를 검토하고 각 입력 인수의 기대 사항을 고려해야 함을 보여 줍니다. 반환 값의 보장과 메서드 반환의 `out` 또는 `ref` 인수를 고려합니다. 그런 다음 해당 규칙을 컴파일러에 전달하면 컴파일러에서 호출자가 해당 규칙을 준수하지 않는 경우 경고를 제공합니다.

이 작업에는 시간이 걸립니다. 우선 라이브러리나 애플리케이션에서 nullable을 인식하도록 하면서 다른 요구 사항도 적절하게 충족하는 전략을 따르겠습니다. 진행 중인 개발을 균형 있게 조정하여 nullable 참조 형식을 사용하도록 설정하는 방법을 살펴봅니다. 제네릭 형식 정의의 문제를 알아봅니다. 개별 API의 사전 및 사후 조건을 설명하는 특성을 적용하는 방법을 알아봅니다.

## nullable 참조 형식을 위한 전략 선택

우선 nullable 참조 형식을 기본적으로 설정하거나 해제할지를 선택합니다. 두 가지 전략이 있습니다.

- 전체 프로젝트에서는 nullable 참조 형식을 사용하도록 설정하고 준비되지 않은 코드에서는 사용하지 않도록 설정합니다.
- Nullable 참조 형식에 대한 주석이 추가된 코드에만 nullable 참조 형식을 사용하도록 설정합니다.

첫 번째 전략은 nullable 참조 형식을 위해 라이브러리를 업데이트할 때 라이브러리에 다른 기능을 추가하는 경우에 가장 효과적입니다. 새로 개발하는 코드는 모두 nullable을 인식합니다. 기존 코드를 업데이트할 때는 해당 클래스에서 nullable 참조 형식을 사용하도록 설정합니다.

이 첫 번째 전략을 따르면 다음 단계를 수행합니다.

1. .csproj 파일에 `<Nullable>enable</Nullable>` 요소를 추가하여 전체 프로젝트에서 nullable 참조 형식을 사용하도록 설정합니다.
2. 프로젝트의 모든 소스 파일에 `#nullable disable` pragma를 추가합니다.
3. 각 파일을 작업할 때 pragma를 제거하고 경고가 있으면 해결합니다.

이 첫 번째 전략은 모든 파일에 pragma를 추가하는 사전 작업이 더 많이 필요합니다. 장점이라면 프로젝트에 추가되는 모든 새 코드 파일에서 nullable을 사용할 수 있습니다. 모든 새 작업은 nullable 인식이므로 기존 코드만 업데이트만 업데이트하면 됩니다.

두 번째 전략은 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.

이 두 번째 전략을 따르면 다음 단계를 수행합니다.

1. Nullable을 인식하도록 할 파일에 `#nullable enable` pragma를 추가합니다.
2. 경고가 표시되면 해결합니다.
3. 전체 라이브러리가 nullable을 인식하도록 지정될 때까지 처음 두 단계를 계속합니다.
4. .csproj 파일에 `<Nullable>enable</Nullable>` 요소를 추가하여 전체 프로젝트에서 nullable 형식을 사용하도록 설정합니다.
5. 더 이상 필요하지 않으므로 `#nullable enable` pragma를 제거합니다.

이 두 번째 전략은 사전 작업이 덜 필요합니다. 단점이라면 새 파일을 만들 때 첫 번째 작업으로 pragma를 추가하고 여기에서 nullable을 인식하도록 설정해야 한다는 것입니다. 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.

#### IMPORTANT

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 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.

## Nullable 경고에서 호환성이 손상되는 변경을 도입해야 하나요?

Nullable 참조 형식을 사용하도록 설정하기 전에 변수는 'nullable 인식 불가능'으로 간주됩니다. Nullable 참조 형식을 사용하도록 설정하면 해당 변수는 모두 'null을 허용하지 않습니다'. 컴파일러에서는 해당 변수가 null이 아닌 값으로 초기화되지 않은 경우 경고를 발생시킵니다.

또한 값이 초기화되지 않은 경우의 반환 값 때문에도 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. 참조 변수가 null이 될 수 없는 경우 원래 선언이 올바릅니다. 이 작업을

수행할 때 목표는 경고를 해결하는 것만이 아닙니다. 무엇보다 컴파일러에서 잠재적 null 값을 사용하는 사용자의 의도로 이해하게 만들어야 합니다. 경고를 검토할 때 라이브러리와 관련하여 다음으로 중요한 결정을 내리게 됩니다. 디자인 의도를 더 명확하게 전달하기 위해 API 시그니처를 수정하시겠습니까? 이전에 살펴본 TryGetMessage 메서드를 위한 더 나은 API 시그니처는 다음과 같을 수 있습니다.

```
string? TryGetMessage(string key);
```

반환 값은 성공이나 실패를 나타내며 값을 찾은 경우 값을 전달합니다. 대부분의 경우 API 시그니처를 변경하면 null 값이 전달되는 방식이 개선될 수 있습니다.

그러나 퍼블릭 라이브러리나 사용자 기반이 대규모인 라이브러리의 경우 API 시그니처 변경을 도입하지 않는 것이 좋습니다. 해당 사례와 기타 일반적인 패턴의 경우 특성을 적용하여 인수나 반환 값이 null 일 수 있는 경우를 더 명확히 정의할 수 있습니다. API의 표면 변경을 고려할지 여부와 관계없이 형식 주석만으로는 인수나 반환 값의 null 값을 설명하는 데 충분하지 않을 가능성이 큽니다. 해당 경우 특성을 적용하여 API를 더 명확하게 설명할 수 있습니다.

## 특성을 통한 형식 주석 확장

변수의 null 상태에 대한 추가 정보를 표현하기 위해 여러 가지 특성이 추가되었습니다. C# 8에 null 허용 참조 형식을 도입하기 전에 작성한 모든 코드는 'null 인식 불가능'이었습니다. 즉, 모든 참조 형식 변수가 null일 수 있지만 null 검사가 필요하지 않습니다. 코드가 'null 허용 인식'이 되면 해당 규칙이 변경됩니다. 참조 형식은 null 값이 아니어야 하며 null 허용 참조 형식은 참조되기 전에 null에 대해 검사되어야 합니다.

API에 대한 규칙은 TryGetValue API 시나리오에서 살펴본 것처럼 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 nullable 분석에 영향을 주는 특성에 관한 문서에서 확인할 수 있습니다.

## 제네릭 정의 및 null 허용 여부

제네릭 형식과 제네릭 메서드의 null 상태를 올바로 전달하려면 특별한 주의가 필요합니다. 각별한 주의가 필요한 이유는 nullable 값 형식과 nullable 참조 형식이 근본적으로 다르다는 점 때문입니다. int? 는 Nullable<int>의 동의어이지만 string? 는 컴파일러에서 추가한 특성을 갖는 string입니다. 결과적으로 컴파일러에서는 T 가 class 인지 struct 인지를 모르면 T? 의 올바른 코드를 생성할 수 없습니다.

그렇다고 nullable 형식(값 형식 또는 참조 형식 중 하나)을 폐쇄형 제네릭 형식의 형식 인수로 사용할 수 없다는 의미는 아닙니다. List<string?> 와 List<int?> 는 모두 List<T> 의 유효한 인스턴스화입니다.

따라서 T? 를 제네릭 클래스 또는 메서드 선언에서 제약 조건이 없이 사용할 수는 없습니다. 예를 들어 Enumerable.FirstOrDefault<TSource>(IEnumerable<TSource>) 는 T? 를 반환하도록 변경되지 않습니다. struct 또는 class 제약 조건을 추가하여 이 제한을 극복할 수 있습니다. 해당 제약 조건 중 하나가 있으면 컴파일러는 T 와 T? 모두의 코드를 생성하는 방법을 알 수 있습니다.

제네릭 형식 인수에 사용되는 형식을 null을 허용하지 않는 형식으로 제한하는 것이 좋습니다. 이렇게 하려면 해당 형식 인수에 notnull 제약 조건을 추가합니다. 해당 제약 조건을 적용하면 형식 인수는 nullable 형식이 될 수 없습니다.

## 런타임에 초기화되는 속성, 데이터 전송 개체, null 허용 여부

생성 후 설정됨을 의미하는 런타임에 초기화되는 속성의 null 허용 여부를 나타내려면 클래스에서 계속 원래 디자인 의도를 올바르게 표현하도록 특별히 고려해야 할 수 있습니다.

DTO(데이터 전송 개체)와 같이 런타임에 초기화되는 속성을 포함하는 형식은 데이터베이스 ORM(개체 관계형 매퍼), 역직렬 변환기, 다른 소스의 속성을 자동으로 채우는 일부 다른 구성 요소 등과 같은 외부 라이브러리에

의해 인스턴스화되는 경우가 많습니다.

Nullable 참조 형식을 사용하도록 설정하기 전에 학생을 나타내는 다음 DTO 클래스를 고려합니다.

```
class Student
{
    [Required]
    public string FirstName { get; set; }

    [Required]
    public string LastName { get; set; }

    public string VehicleRegistration { get; set; }
}
```

이 경우 `Required` 특성으로 표시되는 디자인 의도에서는 이 시스템에서 `FirstName` 및 `LastName` 속성이 필수이고 따라서 null이 아님을 나타냅니다.

`VehicleRegistration` 속성은 필수가 아니므로 null일 수 있습니다.

Nullable 참조 형식을 사용하도록 설정할 때 원래 의도와 일치하도록 DTO의 속성 중 nullable일 수 있는 속성을 나타내려고 합니다.

```
class Student
{
    [Required]
    public string FirstName { get; set; }

    [Required]
    public string LastName { get; set; }

    public string? VehicleRegistration { get; set; }
}
```

이 DTO 경우 null일 수 있는 속성은 `VehicleRegistration` 뿐입니다.

그러나 컴파일러는 `FirstName` 및 `LastName` 모두에 대해 null을 허용하지 않는 속성이 초기화되지 않았음을 나타내는 `CS8618` 경고를 발생시킵니다.

원래 의도를 유지하면서 컴파일러 경고를 해결하는 세 가지 옵션을 사용할 수 있습니다. 해당 옵션은 모두 유효하지만, 코딩 스타일과 디자인 요구 사항에 가장 적합한 옵션을 선택해야 합니다.

생성자에서 초기화

초기화되지 않음 경고를 해결하는 데 적합한 방법은 생성자에서 속성을 초기화하는 것입니다.

```

class Student
{
    public Student(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    [Required]
    public string FirstName { get; set; }

    [Required]
    public string LastName { get; set; }

    public string? VehicleRegistration { get; set; }
}

```

이 접근 방식은 클래스를 인스턴스화하기 위해 사용하는 라이브러리가 생성자에서 매개 변수 전달을 지원하는 경우에만 적용됩니다.

라이브러리는 생성자에서 전부는 아니지만 '일부' 속성의 전달을 지원할 수 있습니다. 예를 들어 EF Core에서는 일반 열 속성은 [생성자 바인딩](#)을 지원하지만 탐색 속성의 경우 지원하지 않습니다.

클래스를 인스턴스화하는 라이브러리 관련 설명서를 확인하여 생성자 바인딩을 지원하는 범위를 파악하세요.

#### Nullable 지원 필드가 있는 속성

생성자 바인딩이 적합하지 않은 경우 이 문제를 해결하는 한 가지 방법으로 nullable 지원 필드가 있는 null을 허용하지 않는 속성을 사용할 수 있습니다.

```

private string? _firstName;

[Required]
public string FirstName
{
    set => _firstName = value;
    get => _firstName
        ?? throw new InvalidOperationException("Uninitialized " + nameof(FirstName))
}

```

해당 시나리오에서는 `FirstName` 속성을 초기화되기 전에 액세스하는 경우 API 계약이 잘못 사용되었으므로 코드에서 `InvalidOperationException` 을 `throw`합니다.

지원 필드를 사용할 경우 일부 라이브러리에 특별히 고려할 사항이 있을 수 있습니다. 예를 들어 EF Core에서는 [지원 필드](#) 을 바로 사용하도록 구성해야 할 수 있습니다.

#### 속성을 `null`로 초기화

Nullable 지원 필드 사용의 간단한 대안으로 또는 클래스를 인스턴스화하는 라이브러리가 해당 접근 방식과 호환되지 않는 경우 `null` 허용 연산자(`!`)를 통해 속성을 직접 `null`로 초기화할 수 있습니다.

```

[Required]
public string FirstName { get; set; } = null!;

[Required]
public string LastName { get; set; } = null!;

public string? VehicleRegistration { get; set; }

```

프로그래밍 버그의 결과인 경우를 제외하고는 속성이 올바로 초기화되기 전에 속성에 액세스하여 런타임에 실제 `null` 값을 확인하지는 못합니다.

## 참조

- 기존 코드베이스를 nullable 참조로 마이그레이션
- EF Core에서 nullable 참조 형식 사용

# 네임스페이스(C# 프로그래밍 가이드)

2021-02-18 • 3 minutes to read • [Edit Online](#)

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

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

`System`은 네임스페이스이고 `Console`은 해당 네임스페이스의 클래스입니다. 전체 키워드가 필요하지 않으므로 다음 예처럼 `using` 키워드를 사용할 수 있습니다.

```
using System;
```

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

자세한 내용은 [using 지시문](#)을 참조하세요.

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

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

네임스페이스 이름은 유효한 C# [식별자 이름](#)이어야 합니다.

## 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.

## 참고 항목

- C# 프로그래밍 가이드
- 네임스페이스 사용
- My 네임스페이스를 사용하는 방법
- 식별자 이름
- using 지시문
- :: 연산자

# 형식, 변수 및 값

2021-02-18 • 17 minutes to read • [Edit Online](#)

C#은 강력한 형식의 언어입니다. 모든 변수 및 상수에는 값으로 계산되는 모든 식을 실행하는 형식이 있습니다. 모든 메서드 시그니처는 각 입력 매개 변수 및 반환 값의 형식을 지정합니다. .NET 클래스 라이브러리는 기본 제공 숫자 형식 집합 및 파일 시스템, 네트워크 연결, 컬렉션, 개체 배열, 날짜 등의 다양한 논리 구문을 나타내는 더 복잡한 형식을 정의합니다. 일반 C# 프로그램에서는 클래스 라이브러리의 형식 및 프로그램의 문제 도메인에 관련된 개념을 모델링하는 사용자 정의 형식을 사용합니다.

형식에 저장된 정보에는 다음이 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.
- 런타임에 변수에 대한 메모리가 할당될 위치.
- 허용되는 작업 유형.

컴파일러는 형식 정보를 사용하여 코드에서 수행되는 모든 작업이 형식이 안전한지 확인합니다. 예를 들어 `int` 형식의 변수를 선언할 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.

```
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;
```

## NOTE

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

컴파일러는 형식 정보를 실행 파일에 메타데이터로 포함합니다. CLR(공용 언어 런타임)은 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.

```

// 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 시그니처에서 지정됩니다. 다음 시그니처는 입력 인수로 `int`가 필요하고 문자열을 반환하는 메서드를 보여 줍니다.

```

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

```

변수가 선언된 후에는 새 형식으로 다시 선언될 수 없고 선언된 형식과 호환되지 않는 값이 할당될 수 없습니다. 예를 들어 `int`를 선언하고 `true`의 부울 값을 여기에 할당할 수 없습니다. 그러나 같은 새 변수에 할당되거나 메서드 인수로 전달될 경우 다른 형식으로 변환할 수 있습니다. 데이터 손실을 일으키지 않는 형식 `변환`은 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#에서는 정수, 부동 소수점 값, 부울 식, 텍스트 문자, 10진수 값 및 기타 데이터 형식을 표현하는 기본 제공 숫자 형식의 표준 집합을 제공합니다. 기본 제공 `string` 및 `object` 형식도 있습니다. 이러한 형식을 모든 C# 프로그램에서 사용할 수 있습니다. 기본 제공 형식의 전체 목록은 [기본 제공 형식](#)을 참조하세요.

## 사용자 지정 형식

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

## 제네릭 형식

클라이언트 코드가 형식의 인스턴스를 만들 때 제공하는 실제 형식(구체적 형식)에 대한 자리 표시자로 사용되는 하나 이상의 형식 매개 변수를 사용하여 형식을 선언할 수 있습니다. 해당 형식을 `제네릭 형식`이라고 합니다. 예를 들어 `List<T>`에는 규칙에 따라 이름 `T`가 지정된 형식 매개 변수 1개가 있습니다. 형식의 인스턴스를 만들 때 목록에 포함될 개체의 형식(예: 문자열)을 지정합니다.

```

List<string> strings = new List<string>();

```

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

## 암시적 형식, 무명 형식 및 튜플 형식

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

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

메서드에서 값을 두 개 이상 반환하려는 것이 일반적입니다. 단일 메서드 호출에서 여러 값을 반환하는 [튜플 형식](#)을 만들 수 있습니다. 자세한 내용은 [튜플 형식](#)을 참조하세요.

## 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. `Int32` (C# 키워드: `int`)와 같은 기본 제공 숫자 형식을 포함한 모든 형식은 기본적으로 단일 기본 형식 [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 형식](#)으로 정의됩니다. 여기에는 .NET 클래스 라이브러리의 모든 사용자 지정 형식과 자체 사용자 정의 형식도 포함됩니다. `struct` 또는 `enum` 키워드를 사용하여 정의하는 형식은 [값 형식](#)입니다. [값 형식](#)에 대한 자세한 내용은 [값 형식](#)을 참조하세요. `class` 키워드를 사용하여 정의한 형식은 [참조 형식](#)입니다. [참조 형식](#)에 대한 자세한 내용은 [클래스](#)를 참조하세요. 참조 형식과 [값 형식](#)의 컴파일 시간 규칙 및 런타임 동작은 서로 다릅니다.

## 참조

- [구조체 형식](#)
- [열거형 형식](#)
- [클래스](#)

# 클래스(C# 프로그래밍 가이드)

2021-02-18 • 13 minutes to read • [Edit Online](#)

## 참조 형식

클래스로 정의된 형식은 참조 형식입니다. 런타임에 참조 형식의 변수를 선언하면 `new` 연산자를 사용하여 클래스의 인스턴스를 명시적으로 만들거나 다음 예제와 같이 다른 곳에서 만들어진 호환성 있는 형식의 개체를 할당할 때까지 변수에는 `null` 값이 포함됩니다.

```
//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. 관리되는 힙의 형식은 할당될 때, 그리고 *가비지 수집*이라는 CLR의 자동 메모리 관리 기능에 의해 회수될 때 오버헤드가 필요합니다. 그러나 가비지 수집은 고도로 최적화되고 대부분 시나리오에서 성능 문제를 일으키지 않습니다. 가비지 수집에 대한 자세한 내용은 [자동 메모리 관리 및 가비지 수집](#)을 참조하세요.

## 클래스 선언

클래스는 다음 예제와 같이 `class` 키워드 다음에 고유 식별자를 사용하여 선언됩니다.

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

`class` 키워드는 액세스 수준 뒤에 옵니다. 이 경우 `public`이 사용되므로 누구나 이 클래스의 인스턴스를 만들 수 있습니다. 클래스 이름은 `class` 키워드 뒤에 옵니다. 클래스의 이름은 유효한 C# [식별자 이름](#)이어야 합니다. 정의의 나머지 부분은 동작과 데이터가 정의되는 클래스 본문입니다. 클래스의 필드, 속성, 메서드 및 이벤트를 모두 [클래스 멤버](#)라고 합니다.

## 개체 만들기

클래스와 개체는 바꿔 사용되기도 하지만 서로 다른 항목입니다. 클래스는 개체의 형식을 정의하지만 개체 자체는 아닙니다. 개체는 클래스에 기반을 둔 구체적 엔터티이고 클래스의 인스턴스라고도 합니다.

개체를 만들려면 다음과 같이 `new` 키워드 뒤에 개체의 기반이 되는 클래스의 이름을 사용합니다.

```
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 자체를 포함하지 않습니다. 실제로 개체를 만들지 않고도 개체 참조를 만들 수 있습니다.

```
Customer object2;
```

런타임에는 참조를 통한 개체 액세스 시도에 실패하므로 개체를 참조하지 않는 이와 같은 개체 참조는 만들지 않는 것이 좋습니다. 그러나 새 개체를 만들거나 다음과 같이 기존 개체를 할당하여 개체를 참조하는 참조를 만들 수 있습니다.

```
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 참조되므로 클래스를 참조 형식이라고 합니다.

## 클래스 상속

클래스는 개체 지향 프로그래밍의 기본적인 특성인 '상속'을 완전히 지원합니다. 클래스를 만들 때 [sealed](#)로 정의되지 않은 다른 클래스에서 상속될 수 있으며 다른 클래스는 직접 만든 클래스에서 상속되고 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.

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

기본 클래스를 선언하는 클래스는 생성자를 제외하고 기본 클래스의 모든 멤버를 상속합니다. 자세한 내용은 [상속](#)을 참조하세요.

C++와 달리 C#의 클래스는 하나의 기본 클래스에서만 직접 상속될 수 있습니다. 그러나 기본 클래스 자체는 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](#)로 선언될 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. 이와 달리 [sealed](#) 클래스에서는 다른 클래스가 파생될 수 없습니다. 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.

## 예제

다음 예제에서는 [자동 구현 속성](#), 메서드 및 생성자라는 특수 메서드를 포함하는 공용 클래스를 정의합니다. 자세한 내용은 [속성, 메서드 및 생성자](#) 항목을 참조하세요. 그런 다음, 클래스의 인스턴스는 `new` 키워드를 사용하여 인스턴스화됩니다.

```

using System;

public class Person
{
    // Constructor that takes no arguments:
    public Person()
    {
        Name = "unknown";
    }

    // Constructor that takes one argument:
    public Person(string name)
    {
        Name = name;
    }

    // Auto-implemented readonly property:
    public string Name { get; }

    // Method that overrides the base class (System.Object) implementation.
    public override string ToString()
    {
        return Name;
    }
}
class TestPerson
{
    static void Main()
    {
        // Call the constructor that has no parameters.
        var person1 = new Person();
        Console.WriteLine(person1.Name);

        // Call the constructor that has one parameter.
        var person2 = new Person("Sarah Jones");
        Console.WriteLine(person2.Name);
        // Get the string representation of the person2 instance.
        Console.WriteLine(person2);

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// unknown
// Sarah Jones
// Sarah Jones

```

## C# 언어 사양

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

## 참조

- [C# 프로그래밍 가이드](#)
- [개체 지향 프로그래밍](#)
- [다형성](#)
- [식별자 이름](#)
- [멤버](#)
- [메서드](#)

- 생성자
- 종료자
- 개체

# 튜플 및 기타 형식 분해

2021-02-18 • 19 minutes to read • [Edit Online](#)

튜플은 메서드 호출에서 여러 값을 검색할 수 있는 간단한 방법을 제공합니다. 하지만 튜플을 검색한 후 튜플의 개별 요소를 처리해야 합니다. 다음 예제와 같이 요소별로 이 작업을 수행하면 번거롭습니다. `QueryCityData` 메서드는 3 튜플을 반환하며 각 튜플 요소가 별도의 작업에서 변수에 할당됩니다.

```
using System;

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.

C# 7.0부터는 한 번의 `분해` 작업으로 튜플에서 여러 요소를 검색하거나 개체에서 여러 필드, 속성 및 계산 값을 검색할 수 있습니다. 튜플을 분해할 때는 튜플 요소를 개별 변수에 할당하고, 개체를 분해할 때는 선택한 값을 개별 변수에 할당합니다.

## 튜플 분해

C#에서는 튜플 분해를 기본적으로 지원하므로 한 작업에서 튜플의 모든 항목을 패키지 해제할 수 있습니다. 튜플을 분해하는 일반 구문은 정의하는 구문과 유사합니다. 즉, 대입문 왼쪽에서 각 요소가 할당되는 변수를 괄호로 묶습니다. 예를 들어 다음 문은 4 튜플의 요소를 네 개의 개별 변수에 할당합니다.

```
var (name, address, city, zip) = contact.GetAddressInfo();
```

다음과 같은 세 가지 방법으로 튜플을 분해합니다.

- 괄호 안에 각 필드의 형식을 명시적으로 선언할 수 있습니다. 다음 예제에서는 이 방법을 사용하여 `QueryCityData` 메서드에서 반환된 3 튜플을 분해합니다.

```

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 예제에서는 `QueryCityData` 메서드에서 반환된 3 튜플을 분해할 때 형식 유추를 사용합니다.

```

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.

```

public static void Main()
{
    (string city, var population, var area) = QueryCityData("New York City");

    // Do something with the data.
}

```

이 방법은 번거로우므로 사용하지 않는 것이 좋습니다.

- 마지막으로, 튜플을 이미 선언된 변수로 분해할 수 있습니다.

```

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.
}

```

튜플에 있는 모든 필드의 형식이 같은 경우에도 괄호 밖에 특정 형식을 지정할 수 없습니다. 이 경우 컴파일러 오류 CS8136, "Deconstruction 'var (...)'" 양식에서는 'var'에 특정 형식을 사용할 수 없습니다."가 생성됩니다.

튜플의 각 요소도 변수에 할당해야 합니다. 생략하는 요소가 있으면 컴파일러에서 CS8132 오류, "'x' 요소의 튜플을 'y' 변수로 분해할 수 없습니다."가 생성됩니다.

분해의 왼쪽에 있는 기존 변수에 할당 및 선언을 혼합할 수 없습니다. 컴파일러가 "분해는 왼쪽에 선언과 식을 혼합할 수 없습니다"라는 오류 CS8184를 생성합니다. 구성원이 새로 선언된 변수 및 기존 변수를 포함할 경우.

## 무시 항목을 사용한 튜플 요소 분해

튜플을 분해할 때 일부 요소 값에만 관심이 있는 경우가 종종 있습니다. C# 7.0부터는 C#에서 지원하는 무시 항목 즉, 무시하도록 선택한 값을 갖는 쓰기 전용 변수를 활용할 수 있습니다. 무시 항목은 할당에서 밑줄 문자("\_")로 지정됩니다. 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

의 이름, 도시의 면적, 연도, 해당 연도의 도시 인구, 두 번째 연도, 해당 두 번째 연도의 도구 인구를 포함하는 6 튜플을 반환합니다. 이 예제는 이러한 두 연도 사이의 인구 변화를 보여 줍니다. 튜플에서 사용 가능한 데이터 중 도시 면적에는 관심이 없고 디자인 타입에 도시 이름과 두 날짜를 알고 있습니다. 따라서 튜플에 저장된 두 가지 인구 값에만 관심이 있고 나머지 값은 무시 항목으로 처리할 수 있습니다.

```
using System;
using System.Collections.Generic;

public class Example
{
    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 분해

C#은 튜플이 아닌 형식의 분해에 대해 기본 제공 지원을 제공하지 않습니다. 그러나 클래스, 구조체 또는 인터페이스의 만든 이는 하나 이상의 `Deconstruct` 메서드를 구현하여 형식의 인스턴스를 분해하도록 허용할 수 있습니다. 이 메서드는 `void`를 반환하며 분해할 각 값은 메서드 시그니처에서 `out` 매개 변수로 표시됩니다. 예를 들어 다음 `Person` 클래스의 `Deconstruct` 메서드는 이름, 중간 이름 및 성을 반환합니다.

```
public void Deconstruct(out string fname, out string mname, out string lname)
```

그러면 다음과 같은 할당을 사용하여 `p`라는 `Person` 클래스의 인스턴스를 분해할 수 있습니다.

```
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

```

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 Example
{
    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 항목을 사용한 사용자 정의 형식 분해

튜플에서와 마찬가지로 무시 항목을 사용하여 `Deconstruct` 메서드에서 반환된 항목 중 선택한 항목을 무시할 수 있습니다. 각 무시 항목은 "\_"이라는 변수로 정의하며 단일 분해 작업에 여러 무시 항목을 포함할 수 있습니다.

다음 예제에서는 `Person` 객체를 4개의 문자열(이름, 성, 도시 및 주)로 분해하지만 성과 주는 무시합니다.

```
// Deconstruct the person object.  
var (fName, _, city, _) = p;  
Console.WriteLine($"Hello {fName} of {city}!");  
// The example displays the following output:  
// Hello John of Boston!
```

## 확장 메서드를 사용한 사용자 정의 형식 분해

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 접근성이 동일한 경우 `access` 변수는 속성 전체의 접근성을 나타냅니다. 그러지 않으면 `get` 및 `set` 접근자의 접근성이 `getAccess` 및 `setAccess` 변수로 표시됩니다.

```
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 Example
{
    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.WriteLine($"\\nAccessibility of the {listType.FullName}.{prop.Name} property: {accessibility}");

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
```

## 참조

- [삭제](#)
- [튜플 형식](#)

# 인터페이스(C# 프로그래밍 가이드)

2020-11-02 • 14 minutes to read • [Edit Online](#)

인터페이스에는 비추상 [클래스](#) 또는 [구조체](#)에서 구현해야 하는 관련 기능 그룹에 대한 정의가 포함되어 있습니다. 인터페이스에서는 구현이 있어야 하는 `static` 메서드를 정의할 수 있습니다. C# 8.0부터 인터페이스는 구성원에 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 상속을 시뮬레이트하려는 경우 인터페이스를 사용해야 합니다.

다음 예제와 같이 `interface` 키워드를 사용하여 인터페이스를 정의합니다.

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

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

`IEquatable<T>` 인터페이스를 구현하는 모든 클래스나 구조체에는 인터페이스에서 지정한 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 구조체는 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 포함될 수 있습니다. 예제에 대한 링크는 [관련 단원](#)을 참조하세요. 인터페이스에는 인스턴스 필드, 인스턴스 생성자 또는 종료자가 포함될 수 없습니다. 인터페이스 멤버는 기본적으로 `public`입니다.

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

클래스 또는 구조체에서 인터페이스를 구현하는 경우 클래스 또는 구조체에서 인터페이스에서 선언하는 모든 멤버의 구현을 제공해야 하지만 기본 구현은 제공하지 않습니다. 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.

```

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 인터페이스의 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. 인터페이스에서 메서드의 기본 구현을 선언하는 경우 해당 인터페이스를 구현하는 모든 클래스가 해당 구현을 상속합니다. 인터페이스에 정의된 구현은 가상이며 구현하는 클래스가 해당 구현을 재정의할 수 있습니다.

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.

- 인터페이스는 일반적으로 추상 멤버만 갖는 추상 기본 클래스와 같습니다. 인터페이스를 구현하는 모든 클래스 또는 구조체는 모든 멤버를 구현해야 합니다. 필요에 따라 인터페이스에서 해당 멤버 일부 또는 모두의 기본 구현을 정의할 수 있습니다. 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.

## 관련 섹션

- [인터페이스 속성](#)
- [인터페이스의 인덱서](#)
- [인터페이스 이벤트를 구현하는 방법](#)
- [클래스 및 구조체](#)
- [상속](#)

- 인터페이스
- 메서드
- 다형성
- 추상/봉인된 클래스 및 클래스 멤버
- 속성
- 이벤트
- 인덱서

## 참조

- C# 프로그래밍 가이드
- 상속
- 식별자 이름

# 메서드(C#)

2021-02-18 • 52 minutes to read • [Edit Online](#)

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.

## NOTE

이 항목에서는 명명된 메서드에 대해 설명합니다. 익명 함수에 대한 자세한 내용은 [익명 함수](#)를 참조하세요.

## 메서드 시그니처

메서드는 다음을 지정하여 `class` 또는 `struct`에서 선언됩니다.

- `public` 또는 `private`와 같은 선택적 액세스 수준입니다. 기본값은 `private`입니다.
- `abstract` 또는 `sealed`와 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.

## IMPORTANT

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.

```
using System;

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`가 있습니다. 두 메서드는 이름이 같지만 해당 매개 변수 형

식으로 구별되어야 합니다.

## 메서드 호출

메서드는 인스턴스 또는 정적일 수 있습니다. 인스턴스 메서드를 호출하려면 개체를 인스턴스화하고 해당 개체에서 메서드를 호출해야 합니다. 인스턴스 메서드는 인스턴스 및 해당 데이터에 대해 작동합니다. 메서드가 속하는 형식의 이름을 참조하여 정적 메서드를 호출합니다. 정적 메서드는 인스턴스 데이터에 대해 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) 뒤에 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 인수 이름(호출하는 코드에 사용된 경우)은 메서드에 정의된 명명된 매개 변수와 동일할 필요가 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.

```
public class Example
{
    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 번째 인수는 `miles` 매개 변수의 값이 되고, 두 번째 인수는 `speed` 매개 변수의 값이 됩니다.

```

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);
    }
}

```

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 있지만 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.

```

using System;

class TestMotorcycle : Motorcycle
{
    public override int Drive(int miles, int speed)
    {
        return (int) Math.Round( ((double)miles) / speed, 0);
    }

    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {

        TestMotorcycle moto = new TestMotorcycle();
        moto.StartEngine();
        moto.AddGas(15);
        var travelTime = moto.Drive(speed: 60, miles: 170);
        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.

```
var 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](#) 개체를 인스턴스화하고, [Person.Equals](#) 메서드를 호출하여 두 개체가 같은지 여부를 확인합니다. 그러나 [Equals](#) 메서드는 [Person](#) 클래스에서 정의되지 않고 [Object](#)에서 상속됩니다.

```
using System;

public class Person
{
    public String FirstName;
}

public class Example
{
    public static void Main()
    {
        var p1 = new Person();
        p1.FirstName = "John";
        var p2 = new Person();
        p2.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.

```

using System;

public class Person
{
    public String FirstName;

    public override bool Equals(object obj)
    {
        var p2 = obj as Person;
        if (p2 == null)
            return false;
        else
            return FirstName.Equals(p2.FirstName);
    }

    public override int GetHashCode()
    {
        return FirstName.GetHashCode();
    }
}

public class Example
{
    public static void Main()
    {
        var p1 = new Person();
        p1.FirstName = "John";
        var p2 = new Person();
        p2.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 형식](#)입니다. 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.

```

using System;

public class Example
{
    public static void Main()
    {
        int 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` 메서드에 전달합니다. 이 예제는 인수를 값으로 메서드에 전달한다는 점에서 기본적으로 이전 예제와 같은 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.

```

using System;

public class SampleRefType
{
    public int value;
}

public class Example
{
    public static void Main()
    {
        var rt = new SampleRefType();
        rt.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 방지하

지만 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.

```
using System;

public class Example
{
    public static void Main()
    {
        int value = 20;
        Console.WriteLine("In Main, value = {0}", value);
        ModifyValue(ref value);
        Console.WriteLine("Back in Main, value = {0}", value);
    }

    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.

```
using System;

public class Example
{
    static void Main()
    {
        int i = 2, j = 3;
        System.Console.WriteLine("i = {0}  j = {1}" , i, j);

        Swap(ref i, ref j);

        System.Console.WriteLine("i = {0}  j = {1}" , i, j);
    }

    static void Swap(ref int x, ref int y)
    {
        int temp = x;
        x = y;
        y = temp;
    }
}

// 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 배열

경우에 따라 메서드의 인수 개수를 정확하게 지정하라는 요구 사항은 제한적입니다. `params` 키워드를 사용하

여 매개 변수를 매개 변수 배열로 지정하면 가변 개수의 인수로 메서드를 호출할 수 있습니다. `params` 키워드로 태그가 지정된 매개 변수는 배열 형식이어야 하며, 메서드의 매개 변수 목록에서 마지막 매개 변수여야 합니다.

그러면 호출자가 다음 네 가지 방법 중 하나로 메서드를 호출할 수 있습니다.

- 원하는 개수의 요소를 포함하는 적절한 형식의 배열 전달
- 적절한 형식의 개별 인수가 포함된 쉼표로 구분된 목록을 메서드에 전달
- `null`을 전달
- 매개 변수 배열에 인수 제공 안 함

다음 예제에서는 매개 변수 배열의 모든 모음을 반환하는 `GetVowels` 메서드를 정의합니다. `Main` 메서드는 해당 메서드를 호출하는 네 가지 방법을 모두 보여 줍니다. 호출자가 `params` 한정자를 포함하는 매개 변수의 인수를 제공할 필요는 없습니다. 이 경우 매개 변수는 빈 배열입니다.

```
using System;
using System.Linq;

class Example
{
    static void Main()
    {
        string fromArray = GetVowels(new[] { "apple", "banana", "pear" });
        Console.WriteLine($"Vowels from array: '{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 string[] input)
    {
        if (input == null || input.Length == 0)
        {
            return string.Empty;
        }

        var vowels = new char[] { '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: ''
```

## 선택적 매개 변수 및 인수

메서드 정의에서 해당 매개 변수를 필수 또는 선택 사항으로 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 하나로 매개 변수의 기본값을 할당해야 합니다.

- 리터럴 문자열이나 숫자와 같은 상수
- `new ValType()` 형태의 식. 여기서 `ValType`은 값 형식입니다. 이 경우 형식의 실제 멤버가 아닌 값 형식의 매개 변수가 없는 암시적 생성자가 호출됩니다.
- `default(ValType)` 형태의 식. 여기서 `ValType`은 값 형식입니다.

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.

```
using System;

public class Options
{
    public void ExampleMethod(int required, int optionalInt = default(int),
                             string description = "Optional Description")
    {
        Console.WriteLine("{0}: {1} + {2} = {3}", description, required,
                          optionalInt, required + optionalInt);
    }
}
```

여러 선택적 인수가 있는 메서드를 위치 인수로 호출하는 경우 호출자가 첫 번째 매개 변수부터 인수가 제공되는 마지막 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.

```
public class Example
{
    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:
//     Optional Description: 10 + 0 = 10
//     Optional Description: 10 + 2 = 12
//     Addition with zero:: 12 + 0 = 12
```

선택적 매개 변수를 사용하면 오버로드 확인 또는 C# 컴파일러가 메서드 호출에서 호출해야 하는 특정 오버로드를 확인하는 방법이 다음과 같이 영향을 받습니다.

- 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. 이는 매개 변수가 적은 후보에 대한 오버로드 확인에서 일반적인 기본 설정의 결과입니다.

## 반환 값

메서드는 호출자에 값을 반환할 수 있습니다. 메서드 이름 앞에 나열된 반환 형식이 `void`가 아니면 메서드는 `return` 키워드를 사용하여 값을 반환할 수 있습니다. `return` 키워드에 이어 반환 형식과 일치하는 변수, 상수 또는 식을 포함하는 문은 메서드 호출자에 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` 키워드를 사용하지 않으면 메서드는 코드 블록 끝에 도달할 때 실행을 중지합니다.

예를 들어 이들 두 메서드에서는 `return` 키워드를 사용하여 정수를 반환합니다.

```
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.

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```
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 둘 이상의 값을 반환하려는 경우도 있습니다. C# 7.0부터 튜플 형식 및 튜플 리터럴을 사용하면 이 작업을 쉽게 수행할 수 있습니다. 튜플 형식은 튜플 요소의 데이터 형식을 정의합니다. 튜플 리터럴은 반환된 튜플의 실제 값을 제공합니다. 다음 예제에서 `(string, string, string, int)` 는 `GetPersonalInfo` 메서드에서 반환되는 튜플 형식을 정의합니다. `(per.FirstName, per.MiddleName, per.LastName, per.Age)` 식은 튜플 리터럴입니다. 메서드는 `PersonInfo` 개체와 함께 이름, 중간 이름 및 성을 반환합니다.

```

public (string, string, string, int) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}

```

호출자는 반환된 튜플을 코드에서 다음과 같이 사용할 수 있습니다.

```

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.

```

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.

```

var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.FName} {person.LName}: age = {person.Age}");

```

메서드에 배열이 인수로 전달되고 메서드가 개별 요소의 값을 수정하는 경우 값의 스타일 또는 기능 흐름 개선을 위해 메서드에서 배열을 반환하도록 선택할 수도 있지만 필수는 아닙니다. 이는 C#에서 모든 참조 형식을 값으로 전달하고 배열 참조의 값이 배열에 대한 포인터이기 때문입니다. 다음 예제에서는 `DoubleValues` 메서드에서 수행한 `values` 배열 내용의 변경 사항을 배열에 대한 참조가 있는 모든 코드에서 관찰할 수 있습니다.

```

using System;

public class Example
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8 };
        DoubleValues(values);
        foreach (var value in values)
            Console.Write("{0} ", value);
    }

    public static void DoubleValues(int[] arr)
    {
        for (int ctr = 0; ctr <= arr.GetUpperBound(0); ctr++)
            arr[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. 물론, 형식의 소스 코드를 소유하지 않은 경우에는 이 작업을 수

행 할 수 없습니다. 이는 메서드를 지원하기 위해 전용 데이터 필드를 추가하는 경우에도 주요 변경 내용이 됩니다.

- 파생 클래스에서 새 메서드를 정의합니다. 구조체, 열거형 등의 다른 형식에 대해 상속을 사용하여 이러한 방식으로 메서드를 추가할 수는 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 있지 않아도 됩니다. 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 람다 식에 코드를 분할하지 않고도 비동기 메서드를 호출할 수 있습니다.

메서드에 `async` 한정자를 표시하면 메서드에서 `await` 연산자를 사용할 수 있습니다. 제어가 비동기 메서드의 `await` 식에 도달하면 대기된 작업이 완료되지 않은 경우 제어가 호출자로 반환되고, 대기된 작업이 완료될 때 까지 `await` 키워드가 있는 메서드의 진행이 일시 중단됩니다. 작업이 완료되면 메서드가 실행이 다시 시작될 수 있습니다.

### NOTE

비동기 메서드는 아직 완료되지 않은 첫 번째 대기된 개체를 검색할 때나 비동기 메서드의 끝에 도달할 때 중에서 먼저 발생하는 시점에 호출자에게 반환됩니다.

비동기 메서드의 반환 형식은 `Task<TResult>`, `Task` 또는 `void`일 수 있습니다. `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. C# 7.0부터 비동기 메서드에는 [작업과 유사한 반환 형식](#)이 있을 수 있습니다.

다음 예제에서 `DelayAsync`는 정수를 반환하는 `return` 문을 포함하는 비동기 메서드입니다. 비동기 메서드이기 때문에 해당 메서드 선언의 반환 형식은 `Task<int>`여야 합니다. 반환 형식이 `Task<int>`이므로 `DoSomethingAsync`의 `await` 식 계산에서 다음 `int result = await delayTask` 문과 같이 정수가 생성됩니다.

```

using System;
using System.Threading.Tasks;

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

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.

```

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>`, `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

# 속성

2021-02-18 • 22 minutes to read • [Edit Online](#)

속성은 C#의 주요 구성 요소입니다. 언어는 개발자가 디자인 의도를 정확하게 표현하는 코드를 작성할 수 있는 구문을 정의합니다.

속성은 액세스 시 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.

```
public class Person
{
    public string FirstName;
    // remaining implementation removed from listing
}
```

속성 정의에는 해당 속성의 값을 검색하고 할당하는 `get` 및 `set` 접근자에 대한 선언이 포함됩니다.

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

    // remaining implementation removed from listing
}
```

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 종괄호 뒤에 속성의 값을 설정합니다. `FirstName` 속성의 초기 값을 `null` 대신 빈 문자열로 설정할 수도 있습니다. 아래와 같이 지정하면 됩니다.

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;

    // remaining implementation removed from listing
}
```

이 아티클의 뒷부분에서 살펴보겠지만 특정 초기화는 읽기 전용 속성에 가장 유용합니다.

아래 표시된 대로 스토리지를 직접 정의할 수도 있습니다.

```

public class Person
{
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }
    private string firstName;
    // remaining implementation removed from listing
}

```

속성 구현이 단일 식인 경우 getter 또는 setter에 대해 식 본문 멤버를 사용할 수 있습니다.

```

public class Person
{
    public string FirstName
    {
        get => firstName;
        set => firstName = value;
    }
    private string firstName;
    // remaining implementation removed from listing
}

```

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 구문을 지원하는 여러 가지 변환이 있습니다. 각 변환에 대한 구문 옵션을 살펴보고 알아봅니다.

## 시나리오

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.

### 유효성 검사

`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.

```

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;
    // remaining implementation removed from listing
}

```

속성 setter 유효성 검사의 일부인 `throw` 식을 사용하여 앞의 예제를 간소화할 수 있습니다.

```

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;
    // remaining implementation removed from listing
}

```

위의 예제에서는 첫 번째 이름을 비워 두거나 공백일 수 없다는 규칙을 적용합니다. 개발자가 작성하는 경우

```
hero.FirstName = "";
```

해당 할당으로 인해 `ArgumentException`이 `throw`됩니다. 속성 `set` 접근자의 반환 형식이 `void`여야 하므로 예외를 `throw`하여 `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.

### 읽기 전용

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 한다고 가정합니다. `set` 접근자에 `public` 대신 `private` 액세스 가능성을 제공할 수 있습니다.

```

public class Person
{
    public string FirstName { get; private set; }

    // remaining implementation removed from listing
}

```

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.

생성자 또는 속성 이니셜라이저에서만 설정할 수 있도록 속성 수정을 제한할 수도 있습니다. 다음과 같이 `Person` 클래스를 수정할 수 있습니다.

```
public class Person
{
    public Person(string firstName) => this.FirstName = firstName;

    public string FirstName { get; }

    // remaining implementation removed from listing
}
```

이 기능은 읽기 전용 속성으로 노출되는 컬렉션을 초기화하는 데 주로 사용됩니다.

```
public class Measurements
{
    public ICollection<DataPoint> points { get; } = new List<DataPoint>();
```

### 계산된 속성

속성은 멤버 필드의 값을 반환할 필요가 없습니다. 계산된 값을 반환하는 속성을 만들 수 있습니다. `Person` 개체를 확장하여 이름과 성을 연결해서 계산된 전체 이름을 반환하겠습니다.

```
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.

```
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.

```

public class Person
{
    public string FirstName { get; set; }

    public string LastName { get; set; }

    private string fullName;
    public string FullName
    {
        get
        {
            if (fullName == 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.

```

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 == 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 구현 세부 정보를 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

C# 7.3부터 필드 특성은 자동 구현된 속성의 컴파일러에서 생성된 지원 필드에 연결될 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.

```
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.

```
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)
            {
                PropertyChanged?.Invoke(this,
                    new PropertyChangedEventArgs(nameof(FirstName)));
            }
            firstName = value;
        }
    }
    private string firstName;

    public event PropertyChangedEventHandler PropertyChanged;
    // remaining implementation removed from listing
}
```

? . 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 입력하는 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, 지연 평가 또는 시나리오에 필요한 모든 요구 사항을 제공할 수 있습니다.

# 인덱서

2020-11-02 • 21 minutes to read • [Edit Online](#)

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.

```
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.

```
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 폴더](#)에 액세스할 수 있습니다. 다운로드 카드는 [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 장점

이 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 필요 한 페이지의 공간을 만들기 위해 메모리에서 페이지를 제거합니다.

```
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
                {
                    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 모델링 할 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 예제에서는 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.

```

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 맵핑됩니다. `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, `public` 메서드를 옵션 설정에 사용하는 디자인입니다.

### 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.

```
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.

```

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.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.

```

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 인덱싱 하는 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` 객체가 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](#) 개념을 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.

# 무시 항목 - C# 가이드

2021-02-18 • 19 minutes to read • [Edit Online](#)

C# 7.0부터 C#에서는 애플리케이션 코드에서 의도적으로 사용되지 않는 자리 표시자 변수인 무시 항목을 지원합니다. 무시 항목은 할당되지 않은 변수에 해당하므로 값을 가지지 않습니다. 무시 항목은 컴파일러 및 코드를 읽는 다른 사용자에게 의도를 전달합니다. 식의 결과를 무시하라는 것입니다. 식의 결과, 하나 이상의 튜플 식 멤버, 메서드에 대한 `out` 매개 변수 또는 패턴 일치 식의 대상을 무시해야 할 수 있습니다.

단일 무시 변수만 있으므로 해당 변수를 스토리지에 할당하지 못할 수도 있습니다. 무시 항목은 메모리 할당을 줄일 수 있습니다. 무시 항목은 코드의 의도를 명확하게 합니다. 또한 코드의 가독성 및 유지 관리를 향상시킵니다.

변수가 무시 항목임을 지정하려면 변수에 밀출(`_`)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.

```
(_, _, area) = city.GetCityInformation(cityName);
```

C# 9.0부터 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 유효한 무시 항목인 경우, 해당 값을 검색하거나 할당 작업에서 사용하려고 하면 “이름 '\_'이 현재 컨텍스트에 없습니다.”라는 컴파일러 오류 CS0301이 생성됩니다. 이 오류는 `_`에 값이 할당되어 있지 않고 스토리지 위치도 할당되어 있지 않을 수 있기 때문입니다. 실제 변수인 경우에는 이전 예제에서처럼 2개 이상의 값을 무시 할 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.

```
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.

```

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.

## switch 를 사용한 패턴 일치

무시 패턴은 [switch](#) 키워드를 사용한 패턴 일치에서 사용할 수 있습니다. 모든 식은 무시 패턴과 항상 일치됩니다.

다. (**is** 식과 함께 사용할 수 있습니다. 그러나 해당 의미를 변경하지 않고 무시 항목을 제거할 수 있으므로 그렇게 사용하는 경우는 드뭅니다.)

다음 예제에서는 **is** 문을 사용하여 개체가 [IFormatProvider](#) 구현을 제공하고 개체가 **null**인지 테스트하는지를 결정하는 **ProvidesFormatInfo** 메서드를 정의합니다. 또한 무시 패턴을 사용하여 다른 형식의 **null**이 아닌 개체도 처리합니다.

```
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.

```
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.

```
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` 객체를 무시합니다. 작업을 할당하면 작업이 완료되려고 할 때 `throw`되는 예외가 표시되지 않습니다. 그러므로 `Task`를 무시하고 해당 비동기 작업에서 생성되는 모든 오류를 무시하려고 하는 의도를 명확하게 합니다.

```
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.

```
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");
```

## NOTE

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. 예들 들어 다음과 같습니다.

```
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. 예들 들어 다음과 같습니다.

```
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 들면 다음과 같습니다.

```
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

- 튜플 및 기타 형식 분해
- `is` 키워드
- `switch` 키워드

# 제네릭(C# 프로그래밍 가이드)

2020-11-02 • 9 minutes to read • [Edit Online](#)

제네릭에서 .NET에 도입한 형식 매개 변수 개념은 클라이언트 코드에서 클래스 또는 메서드를 선언하고 인스턴스화할 때까지 하나 이상의 형식 지정을 지원하는 클래스 및 메서드를 디자인할 수 있도록 합니다. 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.

```
// 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 클래스 및 메서드는 제네릭이 아닌 클래스 및 메서드에서는 결합할 수 없는 방식으로 재사용성, 형식 안전성 및 효율성을 결합합니다. 제네릭은 컬렉션 및 해당 컬렉션에서 작동하는 메서드에서 가장 자주 사용됩니다. [System.Collections.Generic](#) 네임스페이스에는 몇 가지 제네릭 기반 컬렉션 클래스가 있습니다. [ArrayList](#)와 같은 제네릭이 아닌 컬렉션은 권장되지 않으며 호환성을 위해 유지 관리됩니다. 자세한 내용은 [.NET의 제네릭](#)을 참조하세요.

물론, 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 `List<T>` 클래스를 사용해야 합니다.) 형식 매개 변수 `T`는 일반적으로 구체적인 형식을 사용하여 목록에 저장된 항목의 형식을 나타내는 여러 위치에서 사용되며, 다음과 같은 방법으로 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>`가 `GenericList<int>`와 같이 구체적인 형식으로 인스턴스화되면 `T`가 나타날 때마다 `int`로 바뀝니다.

```

// 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 IEnumarator<T> GetEnumarator()
    {
        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 변경하기만 하면 다음 코드를 쉽게 수정하여 문자열이나 다른 모든 사용자 지정 형식 목록을 만들 수 있습니다.

```

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");
    }
}

```

## 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. 이러한 제네릭 컬렉션 클래스는 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.
- 제네릭 데이터 형식에 사용되는 형식에 대한 정보는 리플렉션을 사용하여 런타임 시 얻을 수 있습니다.

## 관련 단원

- [제네릭 형식 매개 변수](#)
- [형식 매개 변수에 대한 제약 조건](#)
- [제네릭 클래스](#)
- [제네릭 인터페이스](#)
- [제네릭 메서드](#)
- [제네릭 대리자](#)
- [C++ 템플릿과 C# 제네릭의 차이점](#)
- [제네릭 및 리플렉션](#)
- [런타임의 제네릭](#)

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)을 참조하세요.

## 참조

- [System.Collections.Generic](#)
- [C# 프로그래밍 가이드](#)
- [유형](#)
- [`<typeparam>`](#)
- [`<typeparamref>`](#)

- .NET의 제네릭

# 반복기

2020-11-02 • 15 minutes to read • [Edit Online](#)

작성하는 거의 모든 프로그램에서 컬렉션을 반복해야 하는 경우가 있습니다. 컬렉션에 있는 모든 항목을 조사하는 코드를 작성합니다.

또한 해당 클래스의 요소에 대해 반복기(컨테이너, 특히 목록을 트래버스하는 개체)를 생성하는 메서드인 반복기 메서드를 만듭니다. 이러한 메서드는 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 이러한 두 시나리오에 대한 기능을 제공합니다. 이 문서에서는 해당 기능에 대한 개요를 제공합니다.

이 자습서는 여러 단계로 구성됩니다. 각 단계 후에 애플리케이션을 실행하고 진행 상황을 확인할 수 있습니다. 이 항목에 대한 전체 샘플을 보거나 [다운로드](#)할 수도 있습니다. 다운로드 지침은 [샘플 및 자습서](#)를 참조하세요.

## foreach로 반복 처리

컬렉션 열거는 간단합니다. `foreach` 키워드는 컬렉션을 열거하여 컬렉션의 각 요소에 대해 포함된 문을 한 번 실행합니다.

```
foreach (var item in collection)
{
    Console.WriteLine(item.ToString());
}
```

아주 간단합니다. 컬렉션의 모든 내용을 반복하려면 `foreach` 문만 있으면 됩니다. 하지만 `foreach` 문이 마법은 아닙니다. 이 문은 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.

## 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.

```

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.

```

public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;
}

```

둘 중 하나를 결정할 필요가 없습니다. 메서드의 요구를 충족하는데 필요한 만큼 `yield return` 문을 사용할 수 있습니다.

```

public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    index = 100;
    while (index < 110)
        yield return index++;
}

```

이 구문은 기본 구문입니다. 반복기 메서드를 작성하는 실제 예제를 살펴보겠습니다. IoT 프로젝트를 진행하고 있고 디바이스 센서는 매우 큰 데이터 스트림을 생성한다고 가정합니다. 데이터를 파악하려면 N번째 데이터 요소마다 샘플링하는 메서드를 작성할 수 있습니다. 이 작은 반복기 메서드면 충분합니다.

```

public static IEnumerable<T> Sample(this IEnumerable<T> sourceSequence, int interval)
{
    int index = 0;
    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.

```

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.

```

public IEnumerable<int> GetSingleDigitNumbers()
{
    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 헤드를 반환하려는 상황을 가정해 보세요. 다음과 같은 두 메서드로 작성할 수 있습니다.

```

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 표

준 관용구로 확장됩니다. 또한 개발자가 리소스를 제대로 관리하지 못해 발생하는 오류를 최소화합니다.

컴파일러는 첫 번째 예제에 표시된 `foreach` 루프를 다음과 유사한 구문으로 변환합니다.

```
IEnumerator<int> enumerator = collection.GetEnumerator();
while (enumerator.MoveNext())
{
    var item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

위의 구문은 버전 5 이상의 C# 컴파일러에서 생성된 코드를 나타냅니다. 버전 5 이전에는 `item` 변수의 범위가 달랐습니다.

```
// C# versions 1 through 4:
IEnumerator<int> enumerator = collection.GetEnumerator();
int item = default(int);
while (enumerator.MoveNext())
{
    item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

이전 동작에서는 람다 식과 관련된 버그를 진단하기가 어렵고 미묘할 수 있기 때문에 변경되었습니다. 람다 식에 대한 자세한 내용은 [람다 식](#)을 참조하세요.

컴파일러에서 생성되는 정확한 코드는 약간 더 복잡하며 `GetEnumerator()`에서 반환된 객체가 `IDisposable` 인터페이스를 구현하는 상황을 처리합니다. 전체 확장에서는 다음과 더 유사한 코드를 생성합니다.

```
{
    var enumerator = collection.GetEnumerator();
    try
    {
        while (enumerator.MoveNext())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    } finally
    {
        // dispose of enumerator.
    }
}
```

열거자가 삭제되는 방식의 `enumerator` 형식의 특성에 따라 달라집니다. 일반적인 경우 `finally` 절은 다음과 같이 확장됩니다.

```
finally
{
    (enumerator as IDisposable)?.Dispose();
}
```

그러나 `enumerator`의 형식이 sealed 형식이고 `enumerator`의 형식에서 `IDisposable`로의 암시적 변환이 없는 경우 `finally` 절은 빈 블록으로 확장됩니다.

```
finally
{
}
```

`enumerator`의 형식에서 `IDisposable`로의 암시적 변환이 있고 `enumerator` 형식이 nullable이 아닌 값 형식인 경우 `finally` 절은 다음과 같이 확장됩니다.

```
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.

# 대리자 소개

2021-02-18 • 7 minutes to read • [Edit Online](#)

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 살펴보겠지만 이러한 알고리즘에 대해 작성하는 C# 코드는 형식이 안전하며 언어 규칙 및 컴파일러를 사용하여 형식이 인수 및 반환 형식과 일치하도록 합니다.

[함수 포인터](#)는 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.

## 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 대리자나 지역 바인딩 알고리즘이 매우 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 섹션의 나머지 문서에서는 대리자를 사용하여 작업할 때 사용되는 언어 기능, 라이브러리 지원 및 관용구에 대해 설명합니다.

`delegate` 키워드와 이 키워드에서 생성하는 코드에 대해 알아봅니다. `System.Delegate` 클래스의 기능 및 해당 기능을 사용하는 방법에 대해서도 살펴봅니다. 형식이 안전한 대리자를 만드는 방법 및 대리자를 통해 호출할 수 있는 메서드를 만드는 방법에 대해 알아봅니다. 또한 람다 식을 사용하여 대리자 및 이벤트로 작업하는 방법을 알아봅니다. 대리자가 LINQ 구성 요소 중 하나가 되는 위치를 확인합니다. 어떻게 대리자가 .NET 이벤트 패턴의 기반이 되는지와 어떻게 다른지도 살펴봅니다.

이제 시작해 보겠습니다.

다음

# System.Delegate 및 delegate 키워드

2020-11-02 • 18 minutes to read • [Edit Online](#)

## 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.

```
// 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 [여기](#)를 참조하세요.

구문이 변수를 선언하는 것처럼 보일 수 있지만 실제로는 형식을 선언합니다. 클래스 내부, 직접 네임스페이스 내부 또는 전역 네임스페이스에 대리자 형식을 정의할 수 있습니다.

### NOTE

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.

```
// 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.

```
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 `throw`합니다. 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 개체의 호출 목록에 메서드를 추가합니다.

길이별로 문자열 목록을 정렬한다고 가정합니다. 비교 함수는 다음과 같을 수 있습니다.

```
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 개체의 대상 목록에 연결하고 해당 대리자를 통해 메서드에 액세스할 수 있습니다.

해당 메서드를 `List.Sort()` 메서드에 전달하여 관계를 만듭니다.

```
phrases.Sort(CompareLength);
```

메서드 이름은 꽤나 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.

```
Comparison<string> comparer = CompareLength;
phrases.Sort(comparer);
```

대리자 대상으로 사용되는 메서드가 작은 메서드인 경우에는 일반적으로 [람다 식](#) 구문을 사용하여 할당을 수행합니다.

```
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 것이었습니다. 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 대리를 호출합니다. [이 시리즈의 뒷부분](#)에서 살펴보겠지만 이러한 메서드에서 직접 사용되는 패턴이 있습니다.

이제 언어 구문 및 대리자 지원 클래스를 확인했으므로 강력한 형식의 대리를 사용하고, 만들고, 호출하는 방법을 살펴보겠습니다.

[다음](#)

# 강력한 형식의 대리자

2020-05-20 • 8 minutes to read • [Edit Online](#)

## 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.

```
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.

```
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.

```
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.

```
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

2020-11-02 • 23 minutes to read • [Edit Online](#)

## 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.

```
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.

```
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 매우 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.

```
public static class Logger
{
    public static Action<string> WriteMessage;

    public static void LogMessage(string msg)
    {
        WriteMessage(msg);
    }
}
```

위의 정적 클래스는 사용할 수 있는 가장 간단한 클래스입니다. 메시지를 콘솔에 기록하는 메서드에 대한 단일 구현을 작성해야 합니다.

```
public static class LoggingMethods
{
    public static void LogToConsole(string message)
    {
        Console.Error.WriteLine(message);
    }
}
```

마지막으로 로거에 선언된 WriteMessage 대리자에 대리자를 연결함으로써 대리자를 연결해야 합니다.

```
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.

## 출력의 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.

```
public enum Severity
{
    Verbose,
    Trace,
    Information,
    Warning,
    Error,
    Critical
}
```

```
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}";
        WriteMessage(outputMsg);
    }
}
```

그런 다음 해당 `Severity` 인수를 사용하여 로그의 출력으로 전송되는 메시지를 필터링해 보겠습니다.

```
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}";
        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.

다음 해당 파일 기반 로거입니다.

```

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.

```
var file = new FileLogger("log.txt");
```

이러한 두 메서드는 함께 사용할 수 있습니다. 두 로그 메서드를 연결하고 메시지를 콘솔 및 파일에 생성할 수 있습니다.

```

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.

```
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 IDisposable을 구현하여 파일을 닫도록 선택할 수 있습니다. 두 메서드에는 장점과 단점이 있습니다. 둘 다 클래스 간 결합을 약간 더 많이 만듭니다.

두 시나리오를 지원하기 위해 로거 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 NullReferenceException 을 throw 합니다. 메서드가 연결되지 않은 경우에도 자동으로 계속되는 디자인을 원할 수 있습니다. Delegate.Invoke() 메서드와 결합된 null 조건부 연산자를 사용하면 간단합니다.

```
public static void LogMessage(string msg)
{
    WriteMessage?.Invoke(msg);
}
```

왼쪽 피연산자(이 경우 WriteMessage )가 null이면 null 조건부 연산자(?.) 가 단락됩니다. 즉, 메시지를 기록하고 시도하지 않는다는 의미입니다.

System.Delegate 또는 System.MulticastDelegate 에 대한 설명서에 나열된 Invoke() 메서드를 찾을 수 없습니다. 컴파일러는 선언된 모든 대리자 형식에 대해 형식이 안전한 Invoke 메서드를 생성합니다. 따라서 이 예제에서 Invoke 는 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 매우 느슨하게 결합됩니다. 그러면 여러 가지 장점을 제공합니다. 새 출력 메커니즘을 만들어 시스템에 연결하기가 매우 쉽습니다. 이러한 다른 메커니즘에는 로그 메시지를 기록하는 메서드 하나만 있으면 됩니다. 이 디자인은 새 기능을 추가할 때 매우 복원력이 있습니다. 모든 기록기에 필요한 계약은 하나의 메서드를 구현하는 것입니다. 해당 메서드는 정적 메서드이거나 인스턴스 메서드일 수 있습니다. 공용, 개인 또는 기타 법적 액세스 할 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

2020-11-02 • 10 minutes to read • [Edit Online](#)

## 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

```
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` 반환값이 있습니다. 이벤트 선언은 동사 또는 동사 구여야 합니다. 이벤트가 발생한 문제를 보고할 때는 과거 시제를 사용합니다. 발생하려고 하는 어떤 문제를 보고하려면 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 강제로 지정할 수 있습니다.

이벤트를 발생시키려면 대리자 호출 구문을 사용하여 이벤트 처리기를 호출합니다.

```
Progress?.Invoke(this, new FileListArgs(file));
```

[대리자](#)에 대한 섹션에서 설명한 대로 ?. 연산자를 사용하면 해당 이벤트에 대한 구독자가 없을 때 이벤트를 발생시키지 않도록 하기가 쉽습니다.

+ = 연산자를 사용하여 이벤트를 구독합니다.

```
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.

```
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

2020-03-18 • 23 minutes to read • [Edit Online](#)

## 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.

```
void OnEventRaised(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 올바른 더 많이 파생된 형식을 알고 있을 수도 있습니다. 규칙에 따라 `object`를 사용합니다.

두 번째 인수는 일반적으로 `System.EventArgs`에서 파생된 형식이었습니다. 다음 섹션에서 이 규칙이 더 이상 적용되지 않음을 확인할 수 있습니다. 이벤트 형식에 추가 인수가 필요하지 않은 경우 두 인수를 계속 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.

```
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.

```
public class FileSearcher
{
    public event EventHandler<FileEventArgs> FileFound;

    public void Search(string directory, string searchPattern)
    {
        foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
        {
            FileFound?.Invoke(this, new FileEventArgs(file));
        }
    }
}
```

## 필드와 유사한 이벤트 정의 및 발생

이벤트를 클래스에 추가하는 가장 간단한 방법은 이전 예제와 같이 해당 이벤트를 `public` 필드로 선언하는 것입니다.

```
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.

```
EventHandler<FileEventArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    filesFound++;
};

fileLister.FileFound += onFileFound;
```

또한 처리기 제거도 가능합니다.

```
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 EventArguments에 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 `FileEventArgs` 형식에 추가해야 합니다.

```
public class FileEventArgs : EventArgs
{
    public string FoundFile { get; }
    public bool CancelRequested { get; set; }

    public FileEventArgs(string fileName)
    {
        FoundFile = fileName;
    }
}
```

이 새 필드는 자동으로 부울 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.

```
public void List(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        var args = new FileEventArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}
```

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.

```
EventHandler<FileEventArgs> 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 업데이트 할 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.

```
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.

```
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 디렉터리를 트래버스하고 두 이벤트를 발생시키는 `Search` 메서드의 오버로드를 추가해 보겠습니다. 이 작업을 수행하는 가장 쉬운 방법은 기본 인수를 사용하여 모든 디렉터리를 검색하도록 지정하는 것입니다.

```

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)
        {
            directoryChanged?.Invoke(this,
                new SearchDirectoryArgs(dir, totalDirs, completedDirs++));
            // Search 'dir' and its subdirectories for files that match the search pattern:
            SearchDirectory(dir, searchPattern);
        }
        // Include the Current Directory:
        directoryChanged?.Invoke(this,
            new SearchDirectoryArgs(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))
    {
        var args = new FileFoundArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}

```

이제 모든 하위 디렉터리를 검색하기 위해 오버로드를 호출하는 애플리케이션을 실행할 수 있습니다. 새 `ChangeDirectory` 이벤트에는 구독자가 없지만 `?.Invoke()` 관용구를 사용하여 이 작업이 제대로 작동하도록 합니다.

콘솔 창에 진행률을 표시하는 줄을 작성하는 처리기를 추가해 보겠습니다.

```

fileLister.DirectoryChanged += (sender, eventArgs) =>
{
    Console.WriteLine($"Entering '{eventArgs.CurrentSearchDirectory}'.");
    Console.WriteLine($"{eventArgs.CompletedDirs} of {eventArgs.TotalDirs} completed... ");
};

```

.NET에 코스터 전체에 적용되는 패턴을 살펴보았습니다. 이러한 패턴 및 규칙을 학습하면 자연스러운 C# 및 .NET을 신속하게 작성할 수 있습니다.

다음으로 .NET의 최신 릴리스에서 이러한 패턴의 일부 변경 내용을 확인합니다.

[다음](#)

# 업데이트된 .NET Core 이벤트 패턴

2020-03-18 • 11 minutes to read • [Edit Online](#)

## 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 `FileFoundArgs` 및 `SearchDirectoryArgs`의 정의를 변경할 수 있습니다. 프로그램은 동일하게 작동합니다.

한 가지 더 변경하는 경우 `SearchDirectoryArgs`를 구조체로 변경할 수 있습니다.

```
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.

`FileFoundArgs`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

학습할 하나의 최종 패턴이 있습니다. 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 권장되는 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.

```
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 `throw`할 수 없습니다. 호출하는 메서드가 `async` 이므로 실행을 계속했습니다. 실제 런타임 동작은 각 환경마다 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. 작업 할 라이브러리에 이전 버전의 예제가 많이 표시됩니다. 그러나 최신 패턴이 무엇인지 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

2020-11-02 • 10 minutes to read • [Edit Online](#)

## 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 런타임에 바인딩 시나리오를 제공합니다. 즉, 런타임에만 알려지는 메서드를 호출하여 구성 요소가 통신하는 시나리오가 가능합니다. 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 관계 없이 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 관용구가 있음을 확인했습니다. 이러한 관용구도 작업을 수행하기는 하지만 메서드에서 값을 반환하는 것만큼 자연스럽지 않습니다.

이러한 두 가지 추론은 종종 존재할 수 있습니다. 대리자 메서드가 값을 반환하는 경우 어떤 방식으로든 알고리즘에 영향을 줄 수 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 발

생시킬 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.

# LINQ(Language-Integrated Query)

2020-11-02 • 8 minutes to read • [Edit Online](#)

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 데이터 원본의 각 유형에 대해 다른 쿼리 언어를 배워야 합니다. SQL 데이터베이스, XML 문서, 다양한 웹 서비스 등. LINQ를 사용할 경우 쿼리는 클래스, 메서드, 이벤트와 같은 고급 언어 구문이 됩니다.

쿼리를 작성하는 개발자의 경우 LINQ에서 가장 눈에 잘 띠는 "언어 통합" 부분은 쿼리 식입니다. 쿼리 식은 선언적 쿼리 구문으로 작성됩니다. 쿼리 구문을 사용하면 최소한의 코드로 데이터 소스에 대해 필터링, 정렬 및 그룹화 작업을 수행할 수 있습니다. 동일한 기본 쿼리 식 패턴을 사용하여 SQL 데이터베이스, ADO .NET 데이터 세트, XML 문서 및 스트림, .NET 컬렉션에서 데이터를 쿼리하고 변환할 수 있습니다.

다음 예제에서는 전체 쿼리 작업을 보여 줍니다. 전체 작업에는 데이터 소스 만들기, 쿼리 식 정의 및 `foreach` 문의 쿼리 실행이 포함됩니다.

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

        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 97 92 81
```

## 쿼리 식 개요

- 쿼리 식은 LINQ 사용 데이터 소스에서 데이터를 쿼리하고 변환하는 데 사용할 수 있습니다. 예를 들어 단일 쿼리로 SQL 데이터베이스에서 데이터를 검색하고 XML 스트림을 출력으로 생성할 수 있습니다.
- 쿼리 식은 익숙한 C# 언어 구문을 많이 사용하기 때문에 쉽게 마스터할 수 있습니다.
- 대부분의 경우 컴파일러가 형식을 유추할 수 있기 때문에 명시적으로 형식을 제공할 필요는 없지만 쿼리 식의 변수는 모두 강력한 형식을 갖습니다. 자세한 내용은 [LINQ 쿼리 작업의 형식 관계](#)를 참조하세요.
- 쿼리는 쿼리 변수를 반복할 때까지(예: `foreach` 문) 실행되지 않습니다. 자세한 내용은 [LINQ 쿼리 소개](#)를 참조하세요.
- 컴파일 타임에 쿼리 식은 C# 사양에 명시된 규칙에 따라 표준 쿼리 연산자 메서드 호출으로 변환됩니다. 쿼리 구문을 사용하여 표현할 수 있는 모든 쿼리는 메서드 구문으로도 표현할 수 있습니다. 그러나 대부분의 경우 쿼리 구문이 더 읽기 쉽고 간결합니다. 자세한 내용은 [C# 언어 사양](#) 및 [표준 쿼리 연산자 개](#)

[오](#)를 참조하세요.

- 일반적으로 LINQ 쿼리를 작성하는 경우 가능하면 쿼리 구문을 사용하고 필요한 경우 메서드 구문을 사용하는 것이 좋습니다. 두 개의 다른 품 간에 의미 체계 또는 성능상의 차이는 없습니다. 쿼리 식이 메서드 구문으로 작성된 동급의 식보다 읽기 쉬운 경우가 많습니다.
- [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. 자세한 내용은 [쿼리 구문과 메서드 구문 비교](#)를 참조하세요.
- 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.

## 다음 단계

LINQ에 대한 자세한 내용을 알아보려면 [쿼리 식 기본 사항](#)에서 몇 가지 기본 개념을 익힌 후 관심 있는 LINQ 기술에 대한 설명서를 읽어보세요.

- XML 문서: [LINQ to XML](#)
- ADO.NET Entity Framework: [LINQ to Entities](#)
- .NET 컬렉션, 파일, 문자열 등: [LINQ to Objects](#)

LINQ를 보다 깊이 있게 이해하려면 [C#의 LINQ](#)를 참조하세요.

C#에서 LINQ를 사용하려면 자습서 [LINQ 작업](#)을 참조하세요.

# 쿼리 식 기본 사항

2020-03-18 • 30 minutes to read • [Edit Online](#)

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 Database 테이블은 행 시퀀스를 포함합니다. XML 파일에는 XML 요소의 "시퀀스"가 있습니다(트리 구조에서는 계층적으로 구성되지만). 메모리 내 컬렉션은 개체의 시퀀스를 포함합니다.

애플리케이션의 관점에서 소스 데이터의 특정 형식과 구조는 중요하지 않습니다. 애플리케이션에는 소스 데이터가 항상 `IEnumerable<T>` 또는 `IQueryable<T>` 컬렉션으로 표시됩니다. 예를 들어 LINQ to XML에서 소스 데이터는 `IEnumerable< XElement >`로 표시됩니다.

이 소스 시퀀스에서 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 여러 가지 방법으로 반환된 시퀀스를 정렬 또는 그룹화할 수 있습니다(`scores` 를 `int[]` 로 가정).

```
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 쿼리를 통해 데이터 소스에 있는 특정 고객 레코드에서 성만 검색할 수 있습니다. 또는 전체 레코드를 검색한 다음 이를 사용하여 최종 결과 시퀀스를 생성하기 전에 다른 메모리 내 개체 유형 또는 XML 데이터를 구성할 수 있습니다. 다음 예제는 `int`에서 `string`으로의 프로젝션을 보여 줍니다. `highScoresQuery`의 새 형식을 참조하세요.

```
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.

```
int highScoreCount =  
    (from score in scores  
     where score > 80  
     select score)  
    .Count();
```

이전 예제에서는 `Count` 메서드를 호출하기 전에 쿼리 식 주변에 괄호를 사용했습니다. 구체적인 결과를 저장하기 위한 새 변수를 사용하여 식을 표현할 수도 있습니다. 이 기술은 결과를 저장하는 쿼리와 별도로 쿼리를 저장하는 변수를 유지하기 때문에 더 읽기 쉽습니다.

```
IEnumerable<int> highScoresQuery3 =  
    from score in scores  
    where score > 80  
    select score;  
  
int 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, 다른 식과 같으며 C# 식이 유 효한 모든 컨텍스트에서 사용할 수 있습니다. 쿼리 식은 SQL 또는 XQuery와 유사한 선언적 구문으로 작성된 절집합으로 구성됩니다. 각 절에는 하나 이상의 C# 식이 포함되며, 이러한 식은 자체가 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 사이에 선택적으로 `where, orderby, join, let` 절과 심지어 추가 `from` 절 중에서 하나 이상을 포함할 수 있습니다. 또한 `into` 키워드를 사용하여, `join` 또는 `group` 절의 결과를 동일한 쿼리 식의 추가 쿼리 절에 대한 소스로 사용할 수 있습니다.

### 쿼리 변수

LINQ에서 쿼리 변수는 쿼리의 결과 대신 쿼리를 저장하는 변수입니다. 좀 더 구체적으로, 쿼리 변수는 항상 `foreach` 문이나 `IEnumerator.MoveNext` 메서드에 대한 직접 호출에서 반복될 때 요소 시퀀스를 생성하는 `Enumerable` 형식입니다.

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.

```

static void Main()
{
    // Data source.
    int[] scores = { 90, 71, 82, 93, 75, 82 };

    // Query Expression.
    IEnumerable<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 (int testScore in scoreQuery)
    {
        Console.WriteLine(testScore);
    }
}
// Outputs: 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 실행될 때, 쿼리 변수 `scoreQuery`를 통해 쿼리 결과가 반환되지 않습니다. 오히려 반복 변수 `testScore`를 통해 반환됩니다. `scoreQuery` 변수는 두 번째 `foreach` 루프에서 반복될 수 있습니다. 변수 또는 데이터 소스를 수정하지 않는 한 동일한 결과가 생성됩니다.

쿼리 변수는 쿼리 구문이나 메서드 구문 또는 이 두 가지 조합으로 표현된 쿼리를 저장할 수 있습니다. 다음 예제에서는 `queryMajorCities` 및 `queryMajorCities2` 모두 쿼리 변수입니다.

```

//Query syntax
IQueryable<City> queryMajorCities =
    from city in cities
    where city.Population > 100000
    select city;

// Method-based syntax
IQueryable<City> queryMajorCities2 = cities.Where(c => c.Population > 100000);

```

반면에 다음 두 예제는 각각의 변수가 쿼리로 초기화되더라도 쿼리 변수가 아닌 변수를 보여 줍니다. 이들은 결과를 저장하기 때문에 쿼리 변수가 아닙니다.

```

int highestScore =
    (from score in scores
     select score)
    .Max();

// or split the expression
IEnumerable<int> scoreQuery =
    from score in scores
    select score;

int highScore = scoreQuery.Max();
// the following returns the same result
int highScore = scores.Max();

List<City> 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;

List<City> largeCitiesList2 = largeCitiesQuery.ToList();

```

쿼리를 표현하는 다양한 방법에 대한 자세한 내용은 [LINQ의 쿼리 구문 및 메서드 구문을 참조하세요.](#)

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 항목의 앞에 나온 쿼리 예제를 암시적 형식을 사용하여 표현할 수도 있습니다.

```

// Use of var is optional here and in all queries.
// queryCities is an IEnumerable<City> just as
// when it is explicitly typed.
var queryCities =
    from city in cities
    where city.Population > 100000
    select city;

```

자세한 내용은 [암시적으로 형식화된 지역 변수 및 LINQ 쿼리 작업의 형식 관계](#)를 참조하세요.

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 형식이 `Country`로 지정됩니다. 범위 변수의 형식은 강력하게 지정되므로 사용 가능한 형식 멤버에 액세스하기 위해 점 연산자를 사용할 수 있습니다.

```

IQueryable<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 `from` 절을 추가로 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.

```
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` 값인 그룹의 시퀀스를 만듭니다.

```
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.

```
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 필드의 하위 집합만 포함하는 일련의 무명 형식을 프로젝션합니다. 새 개체는 개체 이니셜라이저를 사용하여 초기화됩니다.

```
// Here var is required because the query
// produces an anonymous type.
var queryNameAndPop =
    from country in countries
    select new { Name = country.Name, Pop = country.Population };
```

소스 데이터를 변환하기 위해 `select` 절을 사용하는 모든 방법에 대한 자세한 내용은 [select 절](#)을 참조하세요.

"**into**"를 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 이렇게 할 수 있습니다. 다음 예제에

서 `countries` 는 인구 1,000만 명 범위에 따라 그룹화됩니다. 이러한 그룹을 만든 후에는 추가 절이 일부 그룹을 필터링한 다음 오름차순으로 그룹을 정렬합니다. 추가 작업을 수행하려면 `countryGroup` 으로 표현되는 연속이 필요합니다.

```
// 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. 선택 사항인 절은 쿼리 본문에서 0번 또는 여러 번 사용할 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.

```
IEnumerable<City> queryCityPop =
    from city in cities
    where city.Population < 200000 && city.Population > 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.

```
IEnumerable<Country> querySortedCountries =
    from country in countries
    orderby country.Area, country.Population descending
    select country;
```

`descending` 키워드는 선택 사항이며, 순서가 지정되지 않은 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 개의 시퀀스를 조인한 후 `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` 개체

를 연결합니다. `Category` 가 `categories` 의 문자열과 일치하지 않는 제품은 필터링됩니다. `select` 문은 속성을 `cat` 및 `prod`에서 가져오는 새 형식을 프로젝션합니다.

```
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 `Split`에서 반환하는 문자열 배열의 첫 번째 요소를 저장합니다.

```
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 (string s in queryFirstNames)
    Console.WriteLine(s + " ");
//Output: Svetlana Claire Sven Cesar
```

자세한 내용은 [let 절](#)을 참조하세요.

#### 쿼리 식의 하위 쿼리

쿼리 절 자체에 쿼리 식을 포함할 수 있는데, 이를 **하위 쿼리**라고도 합니다. 각 하위 쿼리는 자체 `from` 절로 시작되는데, 이것이 반드시 첫 번째 `from` 절에 있는 동일한 데이터 소스를 가리킬 필요는 없습니다. 예를 들어 다음 쿼리는 그룹화 작업의 결과를 검색하기 위해 `select` 문에서 사용되는 쿼리 식을 보여 줍니다.

```
var queryGroupMax =
    from student in students
    group student by student.GradeLevel into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore =
            (from student2 in studentGroup
            select student2.Scores.Average())
            .Max()
    };
```

자세한 내용은 [그룹화 작업에서 하위 쿼리를 수행하는 방법](#)을 참조하세요.

## 참조

- [C# 프로그래밍 가이드](#)
- [LINQ\(Language-Integrated Query\)](#)
- [쿼리 키워드\(LINQ\)](#)
- [표준 쿼리 연산자 개요](#)

# C#의 LINQ

2020-03-18 • 2 minutes to read • [Edit Online](#)

이 섹션에는 LINQ에 대한 자세한 정보를 제공하는 항목의 링크가 포함되어 있습니다.

## 섹션 내용

### [LINQ 쿼리 소개](#)

모든 언어 및 데이터 소스에서 공통된 기본 LINQ 쿼리 작업의 세 가지 부분에 대해 설명합니다.

### [LINQ 및 제네릭 형식](#)

LINQ에서 사용되는 제네릭 형식을 간략하게 소개합니다.

### [LINQ를 통한 데이터 변환](#)

쿼리에서 검색된 데이터를 변환할 수 있는 다양한 방법을 설명합니다.

### [LINQ 쿼리 작업의 형식 관계](#)

LINQ 쿼리 작업의 세 부분에서 형식이 유지 및/또는 변환되는 방법을 설명합니다.

### [LINQ의 쿼리 구문 및 메서드 구문](#)

LINQ 쿼리를 표현하는 두 가지 방법인 메서드 구문과 쿼리 구문을 비교합니다.

### [LINQ를 지원하는 C# 기능](#)

C#에서 LINQ를 지원하는 언어 구문을 설명합니다.

## 관련 단원

### [LINQ 쿼리 식](#)

LINQ의 쿼리에 대한 개요를 포함하고 추가 리소스에 대한 링크를 제공합니다.

### [표준 쿼리 연산자 개요](#)

LINQ에서 사용되는 표준 메서드를 소개합니다.

# C#에서 LINQ 쿼리 작성하기

2020-11-02 • 9 minutes to read • [Edit Online](#)

이 문서에서는 C#에서 LINQ 쿼리를 작성할 수 있는 세 가지 방법을 보여 줍니다.

1. 쿼리 구문을 사용합니다.
2. 메서드 구문을 사용합니다.
3. 쿼리 구문과 메서드 구문을 조합해서 사용합니다.

다음 예제에서는 앞서 나열한 각 방법을 사용하여 몇몇 간단한 LINQ 쿼리를 보여 줍니다. 일반적인 규칙은 가능한 경우 항상 (1)을 사용하고, 필요할 때마다 (2) 및 (3)을 사용하는 것입니다.

## NOTE

이러한 쿼리는 간단한 메모리 내 컬렉션에서 작동합니다. 그러나 기본 구문은 LINQ to Entities 및 LINQ to XML에서 사용되는 구문과 동일합니다.

## 예제 - 쿼리 구문

대부분의 쿼리를 작성하는 권장 방법은 쿼리/ 구문을 사용하여 쿼리/ 식을 만드는 것입니다. 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.

```
// Query #1.  
List<int> numbers = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };  
  
// The query variable can also be implicitly typed by using var  
IQueryable<int> filteringQuery =  
    from num in numbers  
    where num < 3 || num > 7  
    select num;  
  
// Query #2.  
IQueryable<int> orderingQuery =  
    from num in numbers  
    where num < 3 || num > 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 다른 문에서 쿼리 변수를 반복할 때까지 실제로 쿼리가 실행되지 않습니다.

니다. 자세한 내용은 [LINQ 쿼리 소개](#)를 참조하세요.

## 예제 - 메서드 구문

일부 쿼리 작업은 메서드 호출로 표현해야 합니다. 가장 일반적인 메서드는 singleton 숫자 값을 반환하는 [Sum](#), [Max](#), [Min](#), [Average](#) 등입니다. 이러한 메서드는 단일 값만 나타내며 추가 쿼리 작업의 소스로 사용할 수 없기 때문에 항상 모든 쿼리에서 마지막에 호출해야 합니다. 다음 예제는 쿼리 식에서의 메서드 호출을 보여 줍니다.

```
List<int> numbers1 = new List<int>() { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
List<int> numbers2 = new List<int>() { 15, 14, 11, 13, 19, 18, 16, 17, 12, 10 };
// Query #4.
double average = numbers1.Average();

// Query #5.
IQueryable<int> concatenationQuery = numbers1.Concat(numbers2);
```

메서드에 Action 또는 Func 매개 변수가 있는 경우 다음 예제와 같이 [lambda](#) 식의 형식으로 제공됩니다.

```
// Query #6.
IQueryable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
```

이전 쿼리에서 쿼리 #4만 즉시 실행됩니다. 그 이유는 해당 쿼리가 제네릭 [IQueryable<T>](#) 컬렉션이 아니라 단일 값을 반환하기 때문입니다. 메서드 자체는 값을 계산하기 위해 `foreach`를 사용해야 합니다.

다음 예제와 같이 `var`과 함께 암시적 형식을 사용하여 각각의 이전 쿼리를 작성할 수 있습니다.

```
// var is used for convenience in these queries
var 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.

```
// Query #7.

// Using a query expression with method syntax
int numCount1 =
    (from num in numbers1
     where num < 3 || num > 7
     select num).Count();

// Better: Create a new variable to store
// the method call result
IQueryable<int> numbersQuery =
    from num in numbers1
    where num < 3 || num > 7
    select num;

int 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.

```
var numCount = (from num in numbers...
```

다음과 같이 메서드 구문에서 작성할 수 있습니다.

```
var numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

다음과 같이 명시적 형식을 사용하여 작성할 수 있습니다.

```
int numCount = numbers.Where(n => n < 3 || n > 7).Count();
```

## 참고 항목

- [연습: C#에서 쿼리 작성](#)
- [LINQ\(Language-Integrated Query\)](#)
- [where 절](#)

# 개체의 컬렉션 쿼리

2020-03-18 • 3 minutes to read • [Edit Online](#)

다음 예제는 `Student` 개체의 목록에 대해 간단한 쿼리를 수행하는 방법을 보여 줍니다. 각 `Student` 개체에는 학생에 대한 몇 가지 기본 정보와 네 개 시험에서 학생의 점수를 나타내는 목록이 있습니다.

이 애플리케이션은 이 섹션에서 동일한 `students` 데이터 소스를 사용하는 많은 다른 예제의 프레임워크 역할을 합니다.

## 예제

다음 쿼리는 첫 번째 시험에서 90점 이상의 점수를 받은 학생을 반환합니다.

```
public class Student
{
    #region data
    public enum GradeLevel { FirstYear = 1, SecondYear, ThirdYear, FourthYear };

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Id { get; set; }
    public GradeLevel Year;
    public List<int> ExamScores;

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", Id = 120,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 99, 82, 81, 79}},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", Id = 116,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 99, 86, 90, 94}},
        new Student {FirstName = "Hanying", LastName = "Feng", Id = 117,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 93, 92, 80, 87}},
        new Student {FirstName = "Cesar", LastName = "Garcia", Id = 114,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int> { 97, 89, 85, 82}},
        new Student {FirstName = "Debra", LastName = "Garcia", Id = 115,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 35, 72, 91, 70}},
        new Student {FirstName = "Hugo", LastName = "Garcia", Id = 118,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 92, 90, 83, 78}},
        new Student {FirstName = "Sven", LastName = "Mortensen", Id = 113,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 88, 94, 65, 91}},
        new Student {FirstName = "Claire", LastName = "O'Donnell", Id = 112,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int> { 75, 84, 91, 39}},
        new Student {FirstName = "Svetlana", LastName = "Omelchenko", Id = 111,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int> { 97, 92, 81, 60}},
        new Student {FirstName = "Lance", LastName = "Tucker", Id = 119,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 68, 79, 88, 92}},
        new Student {FirstName = "Michael", LastName = "Tucker", Id = 122,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int> { 94, 92, 91, 91}},
        new Student {FirstName = "Eugene", LastName = "Zabokritski", Id = 121,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int> { 85, 88, 93, 89}}
    };
}
```

```

        Year = GradeLevel.FourthYear,
        ExamScores = new List<int> { 96, 85, 91, 60}};

};

#endregion

// Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

public static void QueryHighScores(int exam, int score)
{
    var highScores = from student in students
                     where student.ExamScores[exam] > score
                     select new {Name = student.FirstName, Score = student.ExamScores[exam]};

    foreach (var item in highScores)
    {
        Console.WriteLine($"{item.Name,-15}{item.Score}");
    }
}

public class Program
{
    public static void Main()
    {
        Student.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}

```

이 쿼리는 사용자가 실험할 수 있도록 의도적으로 간단하게 만든 쿼리입니다. 예를 들어, `where` 절에서 더 많은 조건을 시도하거나 `orderby` 절을 사용하여 결과를 정렬할 수 있습니다.

## 참고 항목

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

# 메서드에서 쿼리를 반환하는 방법(C# 프로그래밍 가이드)

2020-11-02 • 4 minutes to read • [Edit Online](#)

이 예제는 메서드에서 반환 값 및 `out` 매개 변수로서 쿼리를 반환하는 방법을 보여 줍니다.

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 반환하는 것으로 간주됩니다. 메서드에서 반환되는 쿼리 변수는 여전히 구성 또는 수정 가능합니다.

## 예제

다음 예제에서 첫 번째 메서드는 쿼리를 반환 값으로 반환하고 두 번째 메서드는 쿼리를 `out` 매개 변수로 반환합니다. 두 경우 모두 쿼리 결과가 아니라 쿼리가 반환됩니다.

```
class MQ
{
    // QueryMethod1 returns a query as its value.
    IEnumerable<string> QueryMethod1(ref int[] ints)
    {
        var intsToStrings = from i in ints
                            where i > 4
                            select i.ToString();
        return intsToStrings;
    }

    // QueryMethod2 returns a query as the value of parameter returnQ.
    void QueryMethod2(ref int[] ints, out IEnumerable<string> returnQ)
    {
        var intsToStrings = from i in ints
                            where i < 4
                            select i.ToString();
        returnQ = intsToStrings;
    }

    static void Main()
    {
        MQ app = new MQ();

        int[] nums = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

        // QueryMethod1 returns a query as the value of the method.
        var myQuery1 = app.QueryMethod1(ref nums);

        // Query myQuery1 is executed in the following foreach loop.
        Console.WriteLine("Results of executing myQuery1:");
        // Rest the mouse pointer over myQuery1 to see its type.
        foreach (string s in myQuery1)
        {
            Console.WriteLine(s);
        }

        // You also can execute the query returned from QueryMethod1
        // directly, without using myQuery1.
        Console.WriteLine("\nResults of executing myQuery1 directly:");
    }
}
```

```

// Rest the mouse pointer over the call to QueryMethod1 to see its
// return type.
foreach (string s in app.QueryMethod1(ref nums))
{
    Console.WriteLine(s);
}

IEnumerable<string> myQuery2;
// QueryMethod2 returns a query as the value of its out parameter.
app.QueryMethod2(ref nums, out myQuery2);

// Execute the returned query.
Console.WriteLine("\nResults of executing myQuery2:");
foreach (string s in myQuery2)
{
    Console.WriteLine(s);
}

// You can modify a query by using query composition. A saved query
// is nested inside a new query definition that revises the results
// of the first query.
myQuery1 = from item in myQuery1
            orderby item descending
            select item;

// Execute the modified query.
Console.WriteLine("\nResults of executing modified myQuery1:");
foreach (string s in myQuery1)
{
    Console.WriteLine(s);
}

// Keep console window open in debug mode.
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}
}

```

## 참고 항목

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

# 쿼리 결과를 메모리에 저장

2020-03-18 • 2 minutes to read • [Edit Online](#)

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](#)

쿼리 결과를 저장하는 경우 다음 예제와 같이 반환된 컬렉션 개체를 새 변수에 할당하는 것이 좋습니다.

## 예제

```
class StoreQueryResults
{
    static List<int> numbers = new List<int>() { 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };
    static void Main()
    {

        I Enumerable<int> queryFactorsOfFour =
            from num in numbers
            where num % 4 == 0
            select num;

        // Store the results in a new variable
        // without executing a foreach loop.
        List<int> factorsofFourList = queryFactorsOfFour.ToList();

        // Iterate the list just to prove it holds data.
        Console.WriteLine(factorsofFourList[2]);
        factorsofFourList[2] = 0;
        Console.WriteLine(factorsofFourList[2]);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key");
        Console.ReadKey();
    }
}
```

## 참고 항목

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

# 쿼리 결과 그룹화

2020-03-18 • 13 minutes to read • [Edit Online](#)

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.

## 예제

이 항목의 모든 예제에는 다음 도우미 클래스 및 데이터 소스가 사용됩니다.

```
public class StudentClass
{
    #region data
    protected enum GradeLevel { FirstYear = 1, SecondYear, ThirdYear, FourthYear };
    protected class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
        public GradeLevel Year;
        public List<int> ExamScores;
    }

    protected static List<Student> students = new List<Student>
    {
        new Student {FirstName = "Terry", LastName = "Adams", ID = 120,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 99, 82, 81, 79}},
        new Student {FirstName = "Fadi", LastName = "Fakhouri", ID = 116,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 99, 86, 90, 94}},
        new Student {FirstName = "Hanying", LastName = "Feng", ID = 117,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 93, 92, 80, 87}},
        new Student {FirstName = "Cesar", LastName = "Garcia", ID = 114,
            Year = GradeLevel.FourthYear,
            ExamScores = new List<int>{ 97, 89, 85, 82}},
        new Student {FirstName = "Debra", LastName = "Garcia", ID = 115,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 35, 72, 91, 70}},
        new Student {FirstName = "Hugo", LastName = "Garcia", ID = 118,
            Year = GradeLevel.SecondYear,
            ExamScores = new List<int>{ 92, 90, 83, 78}},
        new Student {FirstName = "Sven", LastName = "Mortensen", ID = 113,
            Year = GradeLevel.FirstYear,
            ExamScores = new List<int>{ 88, 94, 65, 91}},
        new Student {FirstName = "Claire", LastName = "O'Donnell", ID = 112,
            Year = GradeLevel.ThirdYear,
            ExamScores = new List<int>{ 96, 85, 77, 89}}
    }
}
```

```

        Year = GradeLevel.FourthYear,
        ExamScores = new List<int>{ 75, 84, 91, 39}),
new Student {FirstName = "Svetlana", LastName = "Omelchenko", ID = 111,
    Year = GradeLevel.SecondYear,
    ExamScores = new List<int>{ 97, 92, 81, 60}},
new Student {FirstName = "Lance", LastName = "Tucker", ID = 119,
    Year = GradeLevel.ThirdYear,
    ExamScores = new List<int>{ 68, 79, 88, 92}},
new Student {FirstName = "Michael", LastName = "Tucker", ID = 122,
    Year = GradeLevel.FirstYear,
    ExamScores = new List<int>{ 94, 92, 91, 91}},
new Student {FirstName = "Eugene", LastName = "Zabokritski", ID = 121,
    Year = GradeLevel.FourthYear,
    ExamScores = new List<int>{ 96, 85, 91, 60}}
};

#endregion

//Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

public void QueryHighScores(int exam, int score)
{
    var highScores = from student in students
                     where student.ExamScores[exam] > score
                     select new {Name = student.FirstName, Score = student.ExamScores[exam]};

    foreach (var item in highScores)
    {
        Console.WriteLine($"{item.Name,-15}{item.Score}");
    }
}

public class Program
{
    public static void Main()
    {
        StudentClass sc = new StudentClass();
        sc.QueryHighScores(1, 90);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}

```

## 예제

다음 예제에서는 요소의 단일 속성을 그룹 키로 사용하여 소스 요소를 그룹화하는 방법을 보여 줍니다. 이 경우 키는 학생의 성인 `string`입니다. 키에 부분 문자열을 사용할 수도 있습니다. 그룹화 작업에는 형식에 대한 기본 같은 비교자가 사용됩니다.

`StudentClass` 클래스에 다음 메서드를 붙여넣습니다. `Main` 메서드에서 호출 문을 `sc.GroupBySingleProperty()`으로 변경합니다.

```

public void GroupBySingleProperty()
{
    Console.WriteLine("Group by a single property in an object:");

    // Variable queryLastNames is an IEnumerable<IGrouping<string,
    // DataClass.Student>>.
    var queryLastNames =
        from student in students
        group student by student.LastName into newGroup
        orderby newGroup.Key
        select newGroup;

    foreach (var nameGroup in queryLastNames)
    {
        Console.WriteLine($"Key: {nameGroup.Key}");
        foreach (var student in nameGroup)
        {
            Console.WriteLine($"{student.LastName}, {student.FirstName}");
        }
    }
}

/* Output:
Group by a single property in an object:
Key: Adams
    Adams, Terry
Key: Fakhouri
    Fakhouri, Fadi
Key: Feng
    Feng, Hanying
Key: Garcia
    Garcia, Cesar
    Garcia, Debra
    Garcia, Hugo
Key: Mortensen
    Mortensen, Sven
Key: O'Donnell
    O'Donnell, Claire
Key: Omelchenko
    Omelchenko, Svetlana
Key: Tucker
    Tucker, Lance
    Tucker, Michael
Key: Zabokritski
    Zabokritski, Eugene
*/

```

## 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.

`StudentClass` 클래스에 다음 메서드를 붙여넣습니다. `Main` 메서드에서 호출 문을 `sc.GroupBySubstring()` 으로 변경합니다.

```

public void GroupBySubstring()
{
    Console.WriteLine("\r\nGroup by something other than a property of the object:");

    var queryFirstLetters =
        from student in students
        group student by student.LastName[0];

    foreach (var studentGroup in queryFirstLetters)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        // Nested foreach is required to access group items.
        foreach (var student in studentGroup)
        {
            Console.WriteLine($"{student.LastName}, {student.FirstName}");
        }
    }
}

/* Output:
Group by something other than a property of the object:
Key: A
    Adams, Terry
Key: F
    Fakhouri, Fadi
    Feng, Hanying
Key: G
    Garcia, Cesar
    Garcia, Debra
    Garcia, Hugo
Key: M
    Mortensen, Sven
Key: O
    O'Donnell, Claire
    Omelchenko, Svetlana
Key: T
    Tucker, Lance
    Tucker, Michael
Key: Z
    Zabokritski, Eugene
*/

```

## 예제

다음 예제에서는 숫자 범위를 그룹 키로 사용하여 소스 요소를 그룹화하는 방법을 보여 줍니다. 그런 다음 쿼리는 이름과 성 및 학생이 속한 백분위수 범위만 포함된 무명 형식으로 결과를 프로젝션합니다. 결과를 표시하는데 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.

```

//Helper method, used in GroupByRange.
protected static int GetPercentile(Student s)
{
    double avg = s.ExamScores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

```

`StudentClass` 클래스에 다음 메서드를 붙여넣습니다. `Main` 메서드에서 호출 문을 `sc.GroupByRange()`으로 변경합니다.

```

public void GroupByRange()
{
    Console.WriteLine("\r\nGroup by numeric range and project into a new anonymous type:");

    var queryNumericRange =
        from student in students
        let percentile = GetPercentile(student)
        group new { student.FirstName, student.LastName } by percentile into percentGroup
        orderby percentGroup.Key
        select percentGroup;

    // Nested foreach required to iterate over groups and group items.
    foreach (var studentGroup in queryNumericRange)
    {
        Console.WriteLine($"Key: {studentGroup.Key * 10}");
        foreach (var item in studentGroup)
        {
            Console.WriteLine($"{item.LastName}, {item.FirstName}");
        }
    }
}

/* Output:
Group by numeric range and project into a new anonymous type:
Key: 60
    Garcia, Debra
Key: 70
    O'Donnell, Claire
Key: 80
    Adams, Terry
    Feng, Hanying
    Garcia, Cesar
    Garcia, Hugo
    Mortensen, Sven
    Omelchenko, Svetlana
    Tucker, Lance
    Zabokritski, Eugene
Key: 90
    Fakhouri, Fadi
    Tucker, Michael
*/

```

## 예제

다음 예제에서는 부울 비교식을 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. 무명 형식의 속성은 `Key` 멤버에 대한 속성이 되고 쿼리가 실행될 때 이름으로 액세스할 수 있습니다.

`StudentClass` 클래스에 다음 메서드를 붙여넣습니다. `Main` 메서드에서 호출 문을 `sc.GroupByBoolean()` 으로 변경합니다.

```

public void GroupByBoolean()
{
    Console.WriteLine("\r\nGroup by a Boolean into two groups with string keys");
    Console.WriteLine("\"True\" and \"False\" and project into a new anonymous type:");
    var queryGroupByAverages = from student in students
                                group new { student.FirstName, student.LastName }
                                by student.ExamScores.Average() > 75 into studentGroup
                                select studentGroup;

    foreach (var studentGroup in queryGroupByAverages)
    {
        Console.WriteLine($"Key: {studentGroup.Key}");
        foreach (var student in studentGroup)
            Console.WriteLine($"{student.FirstName} {student.LastName}");
    }
}
/* Output:
Group by a Boolean into two groups with string keys
"True" and "False" and project into a new anonymous type:
Key: True
    Terry Adams
    Fadi Fakhouri
    Hanying Feng
    Cesar Garcia
    Hugo Garcia
    Sven Mortensen
    Svetlana Omelchenko
    Lance Tucker
    Michael Tucker
    Eugene Zabokritski
Key: False
    Debra Garcia
    Claire O'Donnell
*/

```

## 예제

다음 예제에서는 무명 형식을 사용하여 여러 값이 포함된 키를 캡슐화하는 방법을 보여 줍니다. 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.

`StudentClass` 클래스에 다음 메서드를 붙여넣습니다. `Main` 메서드에서 호출 문을 `sc.GroupByCompositeKey()` 으로 변경합니다.

```

public void GroupByCompositeKey()
{
    var queryHighScoreGroups =
        from student in students
        group student by new { FirstLetter = student.LastName[0],
            Score = student.ExamScores[0] > 85 } into studentGroup
        orderby studentGroup.Key.FirstLetter
        select studentGroup;

    Console.WriteLine("\r\nGroup and order by a compound key:");
    foreach (var scoreGroup in queryHighScoreGroups)
    {
        string s = scoreGroup.Key.Score == true ? "more than" : "less than";
        Console.WriteLine($"Name starts with {scoreGroup.Key.FirstLetter} who scored {s} 85");
        foreach (var item in scoreGroup)
        {
            Console.WriteLine($"{item.FirstName} {item.LastName}");
        }
    }
}

/* Output:
Group and order by a compound key:
Name starts with A who scored more than 85
    Terry Adams
Name starts with F who scored more than 85
    Fadi Fakhouri
    Hanying Feng
Name starts with G who scored more than 85
    Cesar Garcia
    Hugo Garcia
Name starts with G who scored less than 85
    Debra Garcia
Name starts with M who scored more than 85
    Sven Mortensen
Name starts with O who scored less than 85
    Claire O'Donnell
Name starts with O who scored more than 85
    Svetlana Omelchenko
Name starts with T who scored less than 85
    Lance Tucker
Name starts with T who scored more than 85
    Michael Tucker
Name starts with Z who scored more than 85
    Eugene Zabokritski
*/

```

## 참고 항목

- [GroupBy](#)
- [IGrouping<TKey,TElement>](#)
- [LINQ\(Language-Integrated Query\)](#)
- [group 절](#)
- [익명 형식](#)
- [그룹화 작업에서 하위 쿼리 수행](#)
- [중첩 그룹 만들기](#)
- [데이터 그룹화](#)

# 중첩 그룹 만들기

2020-03-18 • 2 minutes to read • [Edit Online](#)

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.

## 예제

### NOTE

이 예제에는 [개체의 컬렉션 쿼리](#)에서 샘플 코드에 정의된 개체에 대한 참조가 포함됩니다.

```

public void QueryNestedGroups()
{
    var queryNestedGroups =
        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;

    // Three nested foreach loops are required to iterate
    // over all elements of a grouped group. Hover the mouse
    // cursor over the iteration variables to see their actual type.
    foreach (var outerGroup in queryNestedGroups)
    {
        Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
        foreach (var innerGroup in outerGroup)
        {
            Console.WriteLine($"    Names that begin with: {innerGroup.Key}");
            foreach (var innerGroupElement in innerGroup)
            {
                Console.WriteLine($"        {innerGroupElement.LastName} {innerGroupElement.FirstName}");
            }
        }
    }
}

/*
Output:
DataClass.Student Level = SecondYear
    Names that begin with: Adams
        Adams Terry
    Names that begin with: Garcia
        Garcia Hugo
    Names that begin with: Omelchenko
        Omelchenko Svetlana
DataClass.Student Level = ThirdYear
    Names that begin with: Fakhouri
        Fakhouri Fadi
    Names that begin with: Garcia
        Garcia Debra
    Names that begin with: Tucker
        Tucker Lance
DataClass.Student Level = FirstYear
    Names that begin with: Feng
        Feng Hanying
    Names that begin with: Mortensen
        Mortensen Sven
    Names that begin with: Tucker
        Tucker Michael
DataClass.Student Level = FourthYear
    Names that begin with: Garcia
        Garcia Cesar
    Names that begin with: O'Donnell
        O'Donnell Claire
    Names that begin with: Zabokritski
        Zabokritski Eugene
*/

```

중첩 그룹의 내부 요소를 반복하려면 세 개의 중첩 `foreach` 루프가 필요합니다.

## 참고 항목

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

# 그룹화 작업에서 하위 쿼리 수행

2020-05-20 • 3 minutes to read • [Edit Online](#)

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 생성되는 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.

## 예제

### NOTE

이 예제에는 [개체의 컬렉션 쿼리](#)에서 샘플 코드에 정의된 개체에 대한 참조가 포함됩니다.

```
public void QueryMax()
{
    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()
        };

    int 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.

```
public void QueryMaxUsingMethodSyntax()
{
    var queryGroupMax = students
        .GroupBy(student => student.Year)
        .Select(studentGroup => new
    {
        Level = studentGroup.Key,
        HighestScore = studentGroup.Select(student2 => student2.ExamScores.Average()).Max()
    });

    int count = queryGroupMax.Count();
    Console.WriteLine($"Number of groups = {count}");

    foreach (var item in queryGroupMax)
    {
        Console.WriteLine($" {item.Level} Highest Score={item.HighestScore}");
    }
}
```

## 참고 항목

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

# 연속 키를 기준으로 결과 그룹화

2020-03-18 • 9 minutes to read • [Edit Online](#)

다음 예제에서는 연속 키의 하위 시퀀스를 나타내는 청크로 요소를 그룹화하는 방법을 보여 줍니다. 예를 들어 키-값 쌍의 다음 시퀀스가 제공된다고 가정합니다.

KEY	값
A	We
A	think
A	that
B	Linq
C	is
A	really
B	cool
B	!

다음 그룹이 이 순서로 만들어집니다.

1. We, think, that
2. Linq
3. is
4. really
5. cool, !

솔루션은 스레드로부터 안전하고 결과를 스트리밍 방식으로 반환하는 확장 메서드로 구현됩니다. 즉, 솔루션은 소스 시퀀스를 거치면서 그룹을 생성합니다. `group` 또는 `orderby` 연산자와 달리 모든 시퀀스를 다 읽기 전에 그룹을 호출자에 반환하기 시작할 수 있습니다.

스레드로부터의 안전성을 달성하려면 소스 코드 요소에 설명된 대로 소스 시퀀스가 반복됨에 따라 각 그룹 또는 청크의 복사본을 만듭니다. 소스 시퀀스에 연속 항목의 큰 시퀀스가 포함된 경우 공용 언어 런타임이 `OutOfMemoryException`을 throw할 수 있습니다.

## 예제

다음 예제에서는 확장 메서드 및 이를 사용하는 클라이언트 코드를 보여 줍니다.

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace ChunkIt
```

```

{
    // Static class to contain the extension methods.
    public static class MyExtensions
    {
        public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource, TKey>(this IEnumerable<TSource>
source, Func<TSource, TKey> keySelector)
        {
            return source.ChunkBy(keySelector, EqualityComparer<TKey>.Default);
        }

        public static IEnumerable<IGrouping<TKey, TSource>> ChunkBy<TSource, TKey>(this IEnumerable<TSource>
source, Func<TSource, TKey> keySelector, IEqualityComparer<TKey> comparer)
        {
            // Flag to signal end of source sequence.
            const bool noMoreSourceElements = true;

            // Auto-generated iterator for the source array.
            var enumerator = source.GetEnumerator();

            // Move to the first element in the source sequence.
            if (!enumerator.MoveNext()) yield break;

            // Iterate through source sequence and create a copy of each Chunk.
            // On each pass, the iterator advances to the first element of the next "Chunk"
            // in the source sequence. This loop corresponds to the outer foreach loop that
            // executes the query.
            Chunk<TKey, TSource> current = null;
            while (true)
            {
                // Get the key for the current Chunk. The source iterator will churn through
                // the source sequence until it finds an element with a key that doesn't match.
                var key = keySelector(enumerator.Current);

                // Make a new Chunk (group) object that initially has one GroupItem, which is a copy of the
                // current source element.
                current = new Chunk<TKey, TSource>(key, enumerator, value => comparer.Equals(key,
keySelector(value)));

                // Return the Chunk. A Chunk is an IGrouping<TKey,TSource>, which is the return value of the
                // ChunkBy method.
                // At this point the Chunk only has the first element in its source sequence. The remaining
                // elements will be
                // returned only when the client code foreach's over this chunk. See Chunk.GetEnumerator for
                // more info.
                yield return current;

                // Check to see whether (a) the chunk has made a copy of all its source elements or
                // (b) the iterator has reached the end of the source sequence. If the caller uses an inner
                // foreach loop to iterate the chunk items, and that loop ran to completion,
                // then the Chunk.GetEnumerator method will already have made
                // copies of all chunk items before we get here. If the Chunk.GetEnumerator loop did not
                // enumerate all elements in the chunk, we need to do it here to avoid corrupting the
                iterator
                // for clients that may be calling us on a separate thread.
                if (current.CopyAllChunkElements() == noMoreSourceElements)
                {
                    yield break;
                }
            }
        }

        // A Chunk is a contiguous group of one or more source elements that have the same key. A Chunk
        // has a key and a list of ChunkItem objects, which are copies of the elements in the source
        sequence.
        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 = null;
    }

    // The value that is used to determine matching elements
    private readonly TKey key;

    // Stores a reference to the enumerator for the source sequence
    private IEnumrator<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 = false;

    // Private object for thread synchronization
    private object m_Lock;

    // REQUIRES: enumerator != null && predicate != null
    public Chunk(TKey key, IEnumrator<TSource> enumerator, Func<TSource, bool> predicate)
    {
        this.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,
    // and the source enumerator is either at the end, or else on an element with a new key.
    // the tail of the linked list is set to null in the CopyNextChunkElement method if the
    // key of the next element does not match the current chunk's key, or there are no more elements
    in the source.
    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.
        // If MoveNext returns false we are at the end, and isLastSourceElement is set to true
        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 = null;
            predicate = null;
        }
        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. It first checks whether
    // there are more elements in the source sequence. If there are, it
    // Returns true if enumerator for this chunk was exhausted.
    internal bool CopyAllChunkElements()
    {
        while (true)
        {
            lock (_Lock)
            {
                if (DoneCopyingChunk)
                {
                    // If isLastSourceElement is false,
                    // it signals to the outer iterator
                    // to continue iterating.
                    return isLastSourceElement;
                }
                else
                {
                    CopyNextChunkElement();
                }
            }
        }
    }

    public TKey Key => key;

    // Invoked by the inner foreach loop. This method stays just one step ahead
    // of the client requests. It adds the next element of the chunk only after
    // the clients requests the last element in the list so far.
    public IEnumrator<TSource> GetEnumerator()
    {
        //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 (_Lock)
            {
                if (current == tail)
                {
                    CopyNextChunkElement();
                }
            }

            // Move to the next ChunkItem in the list.
            current = current.Next;
        }
    }
}

```

```

        System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() =>
GetEnumerator();
    }
}

// A simple named type is used for easier viewing in the debugger. Anonymous types
// work just as well with the ChunkBy operator.
public class KeyValuePair
{
    public string Key { get; set; }
    public string Value { get; set; }
}

class Program
{
    // The source sequence.
    public static IEnumerable<KeyValuePair> list;

    // Query variable declared as class member to be available
    // on different threads.
    static IGrouping<string, KeyValuePair> query;

    static void Main(string[] args)
    {
        // Initialize the source sequence with an array initializer.
        list = new[]
        {
            new KeyValuePair{ Key = "A", Value = "We" },
            new KeyValuePair{ Key = "A", Value = "think" },
            new KeyValuePair{ Key = "A", Value = "that" },
            new KeyValuePair{ Key = "B", Value = "Linq" },
            new KeyValuePair{ Key = "C", Value = "is" },
            new KeyValuePair{ Key = "A", Value = "really" },
            new KeyValuePair{ Key = "B", Value = "cool" },
            new KeyValuePair{ Key = "B", Value = "!" }
        };
        // Create the query by using our user-defined query operator.
        query = list.ChunkBy(p => p.Key);

        // 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}");
            }
        }

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
}

```

프로젝트에서 확장 메서드를 사용하려면 `MyExtensions` 정적 클래스를 신규 또는 기존 소스 코드 파일에 복사하고, 필요한 경우 해당 메서드가 있는 네임스페이스에 대한 `using` 지시문을 추가합니다.

## 참고 항목

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

# 런타임에 동적으로 조건자 필터 지정

2020-05-20 • 5 minutes to read • [Edit Online](#)

경우에 따라 `where` 절의 소스 요소에 적용해야 하는 조건자 수를 런타임 할 때까지 알 수 없습니다. 다음 예제와 같이 여러 조건자 필터를 동적으로 지정하는 한 가지 방법은 `Contains` 메서드를 사용하는 것입니다. 이 예제는 두 가지 방법으로 구성됩니다. 먼저 프로그램에서 제공되는 값을 필터링하여 프로젝트를 실행합니다. 그런 다음 런타임에 제공된 입력을 사용하여 프로젝트를 다시 실행합니다.

## Contains 메서드를 사용하여 필터링 하려면

1. 새 콘솔 애플리케이션을 열고 이름을 `PredicateFilters`로 지정합니다.
2. [개체의 컬렉션 쿼리](#)에서 `StudentClass` 클래스를 복사하여 `Program` 클래스 아래의 `PredicateFilters` 네 임스페이스에 붙여 넣습니다. `StudentClass` 는 `Student` 개체 목록을 제공합니다.
3. `StudentClass` 에서 `Main` 메서드를 주석으로 처리합니다.
4. `Program` 클래스를 다음 코드로 바꿉니다.

```
class DynamicPredicates : StudentClass
{
    static void Main(string[] args)
    {
        string[] ids = { "111", "114", "112" };

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryByID(string[] ids)
    {
        var queryNames =
            from student in students
            let i = student.ID.ToString()
            where ids.Contains(i)
            select new { student.LastName, student.ID };

        foreach (var name in queryNames)
        {
            Console.WriteLine($"{name.LastName}: {name.ID}");
        }
    }
}
```

5. `ids` 의 선언 아래에서 `DynamicPredicates` 클래스의 `Main` 메서드에 다음 줄을 추가합니다.

```
QueryById(ids);
```

6. 프로젝트를 실행합니다.

7. 다음 출력이 콘솔 창에 표시됩니다.

Garcia: 114

O'Donnell: 112

Omelchenko: 111

8. 다음 단계로 프로젝트를 다시 실행합니다. 이번에는 `ids` 배열 대신 런타임에 입력된 입력을 사용합니다. `Main` 메서드에서 `QueryByID(ids)` 를 `QueryByID(args)` 로 변경합니다.
9. 명령줄 인수 `122 117 120 115` 를 사용하여 프로젝트를 실행합니다. 프로젝트가 실행되면 해당 값은 `Main` 메서드의 매개 변수인 `args` 의 요소가 됩니다.
10. 다음 출력이 콘솔 창에 표시됩니다.

Adams: 120

Feng: 117

Garcia: 115

Tucker: 122

## switch 문을 사용하여 필터링 하려면

1. `switch` 문을 사용하여 미리 결정된 대체 쿼리 중에서 선택할 수 있습니다. 다음 예제에서 `studentQuery` 는 런타임에 지정되는 성적 수준 또는 학년에 따라 다른 `where` 절을 사용합니다.
2. 다음 메서드를 복사하여 `DynamicPredicates` 클래스에 붙여 넣습니다.

```

// To run this sample, first specify an integer value of 1 to 4 for the command
// line. This number will be converted to a GradeLevel value that specifies which
// set of students to query.
// Call the method: QueryByYear(args[0]);

static void QueryByYear(string level)
{
    GradeLevel year = (GradeLevel)Convert.ToInt32(level);
    IEnumerable<Student> studentQuery = null;
    switch (year)
    {
        case GradeLevel.FirstYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.FirstYear
                           select student;
            break;
        case GradeLevel.SecondYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.SecondYear
                           select student;
            break;
        case GradeLevel.ThirdYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.ThirdYear
                           select student;
            break;
        case GradeLevel.FourthYear:
            studentQuery = from student in students
                           where student.Year == GradeLevel.FourthYear
                           select student;
            break;
        default:
            break;
    }
    Console.WriteLine($"The following students are at level {year}");
    foreach (Student name in studentQuery)
    {
        Console.WriteLine($"{name.LastName}: {name.ID}");
    }
}

```

3. `Main` 메서드에서 `QueryByID` 호출을 `QueryByYear(args[0])` 호출로 바꿔 `args` 배열의 첫 번째 요소를 인수로 전송합니다.

4. 1에서 4 사이의 정수 값인 명령줄 인수를 사용하여 프로젝트를 실행합니다.

## 참고 항목

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

# 내부 조인 수행

2020-05-20 • 17 minutes to read • [Edit Online](#)

관계형 데이터베이스 용어에서 **내부 조인**은 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.

이 문서에서는 내부 조인의 네 가지 변환을 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

## 예제 - 단순 키 조인

다음 예제에서는 두 개의 사용자 정의 형식인 `Person` 및 `Pet`의 개체를 포함하는 두 개의 컬렉션을 만듭니다. 쿼리는 C#의 `join` 절을 사용하여 `Person` 개체를 `Owner` 가 해당 `Person` 인 `Pet` 개체와 일치시킵니다. C#의 `select` 절은 결과 개체의 모양을 정의합니다. 이 예제에서 결과 개체는 소유자의 이름과 애완 동물의 이름으로 구성된 무명 형식입니다.

```

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

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Simple inner join.
/// </summary>
public static void InnerJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = rui };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene, rui };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create a collection of person-pet pairs. Each element in the collection
    // is an anonymous type containing both the person's name and their pet's name.
    var query = from person in people
               join pet in pets on person equals pet.Owner
               select new { OwnerName = person.FirstName, PetName = pet.Name };

    foreach (var ownerAndPet in query)
    {
        Console.WriteLine($"\"{ownerAndPet.PetName}\\" is owned by {ownerAndPet.OwnerName}");
    }
}

// This code produces the following output:
//
// "Daisy" is owned by Magnus
// "Barley" is owned by Terry
// "Boots" is owned by Terry
// "Whiskers" is owned by Charlotte
// "Blue Moon" is owned by Rui

```

`Lastname`이 "Huff"인 `Person` 개체는 `Pet.Owner`가 `Person`과 같은 `Pet` 개체가 없기 때문에 결과 집합에 표시되지 않습니다.

## 예제 - 복합 키 조인

속성 하나만 기준으로 요소를 상호 연결하는 대신 복합 키를 사용하여 여러 속성을 기준으로 요소를 비교할 수 있습니다. 이렇게 하려면 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 `Employee` 개체 목록과 `Student` 개체 목록을 사용하여 학생이기도 한 직원을 확인합니다. 이

여러 형식에는 둘 다 `String` 형식의 `FirstName` 및 `LastName` 속성이 있습니다. 각 목록의 요소에서 조인 키를 만드는 함수는 각 요소의 `FirstName` 및 `LastName` 속성으로 구성된 무명 형식을 반환합니다. 조인 작업은 이러한 복합 키가 같은지 비교하고 각 목록에서 이름과 성이 둘 다 일치하는 개체 쌍을 반환합니다.

```
class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int EmployeeID { get; set; }
}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int StudentID { get; set; }
}

/// <summary>
/// Performs a join operation using a composite key.
/// </summary>
public static void CompositeKeyJoinExample()
{
    // Create a list of employees.
    List<Employee> employees = new List<Employee> {
        new Employee { FirstName = "Terry", LastName = "Adams", EmployeeID = 522459 },
        new Employee { FirstName = "Charlotte", LastName = "Weiss", EmployeeID = 204467 },
        new Employee { FirstName = "Magnus", LastName = "Hedland", EmployeeID = 866200 },
        new Employee { FirstName = "Vernette", LastName = "Price", EmployeeID = 437139 } };

    // Create a list of students.
    List<Student> students = new List<Student> {
        new Student { FirstName = "Vernette", LastName = "Price", StudentID = 9562 },
        new Student { FirstName = "Terry", LastName = "Earls", StudentID = 9870 },
        new Student { FirstName = "Terry", LastName = "Adams", StudentID = 9913 } };

    // 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 employee in employees
                                join student in students
                                on new { employee.FirstName, employee.LastName }
                                equals new { student.FirstName, student.LastName }
                                select employee.FirstName + " " + employee.LastName;

    Console.WriteLine("The following people are both employees and students:");
    foreach (string name in query)
        Console.WriteLine(name);
}

// This code produces the following output:
//
// The following people are both employees and students:
// Terry Adams
// Vernette Price
```

## 예제 - 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.

다음 예제에서는 `Person` 개체 목록, `Cat` 개체 목록, `Dog` 개체 목록의 세 가지 컬렉션을 만듭니다.

C#의 첫 번째 `join` 절은 `Cat.Owner` 와 일치하는 `Person` 을 기준으로 사람과 고양이를 일치시킵니다. `Person` 개체 및 `Cat.Name` 을 포함하는 무명 형식의 시퀀스를 반환합니다.

C#의 두 번째 `join` 절은 `Person` 형식의 `Owner` 속성과 동물 이름의 첫 글자로 구성된 복합 키를 기준으로 제공된 개 목록의 `Dog` 개체와 첫 번째 조인에서 반환된 무명 형식을 상호 연결합니다. 일치하는 각 쌍의 `Cat.Name` 및 `Dog.Name` 속성을 포함하는 무명 형식의 시퀀스를 반환합니다. 내부 조인이기 때문에 두 번째 데이터 소스에 일치 항목이 있는 첫 번째 데이터 소스의 개체만 반환됩니다.

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

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

class Cat : Pet
{ }

class Dog : Pet
{ }

public static void MultipleJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };
    Person rui = new Person { FirstName = "Rui", LastName = "Raposo" };
    Person phyllis = new Person { FirstName = "Phyllis", LastName = "Harris" };

    Cat barley = new Cat { Name = "Barley", Owner = terry };
    Cat boots = new Cat { Name = "Boots", Owner = terry };
    Cat whiskers = new Cat { Name = "Whiskers", Owner = charlotte };
    Cat bluemoon = new Cat { Name = "Blue Moon", Owner = rui };
    Cat daisy = new Cat { Name = "Daisy", Owner = magnus };

    Dog fourwheeldrive = new Dog { Name = "Four Wheel Drive", Owner = phyllis };
    Dog duke = new Dog { Name = "Duke", Owner = magnus };
    Dog denim = new Dog { Name = "Denim", Owner = terry };
    Dog wiley = new Dog { Name = "Wiley", Owner = charlotte };
    Dog snoopy = new Dog { Name = "Snoopy", Owner = rui };
    Dog snickers = new Dog { Name = "Snickers", Owner = arlene };

    // Create three lists.
    List<Person> people =
        new List<Person> { magnus, terry, charlotte, arlene, rui, phyllis };
    List<Cat> cats =
        new List<Cat> { barley, boots, whiskers, bluemoon, daisy };
    List<Dog> dogs =
        new List<Dog> { fourwheeldrive, duke, denim, wiley, snoopy, snickers };

    // The first join matches Person and Cat.Owner from the list of people and
    // cats, based on a common Person. The second join matches dogs whose names start
    // with the same letter as the cats that have the same owner.
    var query = from person in people
               join cat in cats on person equals cat.Owner
               join dog in dogs on
                   new { Owner = person, Letter = cat.Name.Substring(0, 1) }
                   equals new { dog.Owner, Letter = dog.Name.Substring(0, 1) }
               select new { CatName = cat.Name, DogName = dog.Name };

    foreach (var obj in query)
    {
        Console.WriteLine(
            $"The cat '{obj.CatName}' shares a house, and the first letter of their name, with '{obj.DogName}'");
    }
}
```

```

    + the cat \ {obj.DogName} \ shares a house, and the first letter of their name, with \
{obj.DogName}\".");
}
}

// This code produces the following output:
//
// The cat "Daisy" shares a house, and the first letter of their name, with "Duke".
// The cat "Whiskers" shares a house, and the first letter of their name, with "Wiley".

```

## 예제 - 그룹화된 조인을 사용하는 내부 조인

다음 예제에서는 그룹 조인을 사용하여 내부 조인을 구현하는 방법을 보여 줍니다.

`query1`에서 `Person` 개체 목록은 `Pet.Owner` 속성과 일치하는 `Person`을 기준으로 `Pet` 개체 목록에 그룹 조인 됩니다. 그룹 조인은 각 그룹이 `Person` 개체 및 일치하는 `Pet` 개체 시퀀스로 구성된 중간 그룹 컬렉션을 만듭니다.

두 번째 `from` 절을 쿼리에 추가하여 이 시퀀스의 시퀀스를 더 긴 시퀀스에 결합(또는 평면화)할 수 있습니다. 최종 시퀀스의 요소 형식은 `select` 절에 의해 지정됩니다. 이 예제에서 해당 형식은 일치하는 각 쌍의 `Person.FirstName` 및 `Pet.Name` 속성으로 구성된 무명 형식입니다.

`query1`의 결과는 `into` 절 없이 `join` 절을 사용해서 내부 조인을 수행하여 얻은 결과 집합과 같습니다.

`query2` 변수는 이와 동일한 쿼리를 보여 줍니다.

```

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

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// Performs an inner join by using GroupJoin().
/// </summary>
public static void InnerGroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    var query1 = from person in people
                join pet in pets on person equals pet.Owner into gj
                from subpet in gj
                select new { OwnerName = person.FirstName, PetName = subpet.Name };

    Console.WriteLine("Inner join using GroupJoin():");
    foreach (var v in query1)
        ;
}

```

```

    }
        Console.WriteLine($"{v.OwnerName} - {v.PetName}");
    }

var query2 = from person in people
            join pet in pets on person equals pet.Owner
            select new { OwnerName = person.FirstName, PetName = pet.Name };

Console.WriteLine("\nThe equivalent operation using Join():");
foreach (var v in query2)
    Console.WriteLine($"{v.OwnerName} - {v.PetName}");
}

// This code produces the following output:
//
// Inner join using GroupJoin():
// Magnus - Daisy
// Terry - Barley
// Terry - Boots
// Terry - Blue Moon
// Charlotte - Whiskers
//
// The equivalent operation using Join():
// Magnus - Daisy
// Terry - Barley
// Terry - Boots
// Terry - Blue Moon
// Charlotte - Whiskers

```

## 참고 항목

- [Join](#)
- [GroupJoin](#)
- [그룹화 조인 수행](#)
- [왼쪽 우선 외부 조인 수행](#)
- [무명 형식](#)

# 그룹화 조인 수행

2020-11-02 • 9 minutes to read • [Edit Online](#)

그룹 조인은 계층적 데이터 구조를 생성하는 데 유용합니다. 첫 번째 컬렉션의 각 요소와 두 번째 컬렉션에서 상관 관계가 지정된 요소 집합을 쌍으로 구성합니다.

예를 들어 `Student`라는 클래스 또는 관계형 데이터베이스 테이블에 `Id` 및 `Name`이라는 두 개의 필드가 있을 수 있습니다. `Course`라는 두 번째 클래스 또는 관계형 데이터베이스 테이블에 `StudentId` 및 `CourseTitle`이라는 두 개의 필드가 있을 수 있습니다. 일치하는 `Student.Id` 및 `Course.StudentId`를 기반으로 하는 이러한 두 데이터 소스의 그룹 조인은 각 `Student`를 `Course` 개체 컬렉션(비어 있을 수 있음)과 그룹화합니다.

## NOTE

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.

## WARNING

`Enumerable.GroupJoin`에는 기존 관계형 데이터베이스 용어에 직접적으로 해당하는 항목이 없습니다. 그러나 이 메서드는 내부 조인 및 원쪽 우선 외부 조인의 상위 집합을 구현합니다. 이러한 작업은 모두 그룹화된 조인과 관련하여 작성할 수 있습니다. 자세한 내용은 [조인 작업 및 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.

## 예제 - 그룹 조인

다음 예제에서는 `Pet.Owner` 속성과 일치하는 `Person`에 따라 `Person` 및 `Pet` 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 `Person` 개체)를 생성합니다. 두 번째 컬렉션의 해당 요소(이 예제에서는 `Pet` 개체)는 컬렉션으로 그룹화됩니다. 마지막으로, 결과 선택기 함수는 `Person.FirstName` 및 `Pet` 개체 컬렉션으로 구성된 각 일치 항목에 대해 무명 형식을 만듭니다.

```

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

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example performs a grouped join.
/// </summary>
public static void GroupJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create a list where each element is an anonymous type
    // that contains the person's first name and a collection of
    // pets that are owned by them.
    var query = from person in people
               join pet in pets on person equals pet.Owner into gj
               select new { OwnerName = person.FirstName, Pets = gj };

    foreach (var v in query)
    {
        // Output the owner's name.
        Console.WriteLine($"{v.OwnerName}:");
        // Output each of the owner's pet's names.
        foreach (Pet pet in v.Pets)
            Console.WriteLine($"  {pet.Name}");
    }
}

// This code produces the following output:
//
// Magnus:
//   Daisy
// Terry:
//   Barley
//   Boots
//   Blue Moon
// Charlotte:
//   Whiskers
// Arlene:

```

## 예제 - XML을 만들기 위한 그룹 조인

그룹 조인은 LINQ to XML을 사용하여 XML을 만드는 데 적합합니다. 다음 예제는 무명 형식을 만드는 대신 결과 선택기 함수가 조인된 개체를 나타내는 XML 요소를 만든다는 점을 제외하고 앞의 예제와 비슷합니다.

```

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

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

/// <summary>
/// This example creates XML output from a grouped join.
/// </summary>
public static void GroupJoinXMLExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    // Create XML to display the hierarchical organization of people and their pets.
    XElement ownersAndPets = new XElement("PetOwners",
        from person in people
        join pet in pets on person equals pet.Owner into gj
        select new XElement("Person",
            new XAttribute("FirstName", person.FirstName),
            new XAttribute("LastName", person.LastName),
            from subpet in gj
            select new XElement("Pet", subpet.Name)));
}

Console.WriteLine(ownersAndPets);
}

// This code produces the following output:
//
// <PetOwners>
//   <Person FirstName="Magnus" LastName="Hedlund">
//     <Pet>Daisy</Pet>
//   </Person>
//   <Person FirstName="Terry" LastName="Adams">
//     <Pet>Barley</Pet>
//     <Pet>Boots</Pet>
//     <Pet>Blue Moon</Pet>
//   </Person>
//   <Person FirstName="Charlotte" LastName="Weiss">
//     <Pet>Whiskers</Pet>
//   </Person>
//   <Person FirstName="Arlene" LastName="Huff" />
// </PetOwners>

```

## 참조

- [Join](#)

- GroupJoin
- 내부 조인 수행
- 왼쪽 우선 외부 조인 수행
- 무명 형식

# 왼쪽 우선 외부 조인 수행

2020-03-18 • 5 minutes to read • [Edit Online](#)

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.

## 예제

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 `Person` 개체 목록은 `Pet.Owner`와 일치하는 `Person` 개체를 기준으로 `Pet` 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 `Pet` 개체의 각 시퀀스에서 `DefaultIfEmpty`를 호출합니다. 메서드는 `Person` 개체에 대해 일치하는 `Pet` 개체의 시퀀스가 비어 있는 경우 단일 기본값을 포함하는 컬렉션을 반환하여 각 `Person` 개체가 결과 컬렉션에 반환되도록 합니다.

### NOTE

참조 형식의 기본값은 `null` 이므로 예제에서는 각 `Pet` 컬렉션의 각 요소에 액세스하기 전에 `null` 참조를 확인합니다.

```

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

class Pet
{
    public string Name { get; set; }
    public Person Owner { get; set; }
}

public static void LeftOuterJoinExample()
{
    Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    Person terry = new Person { FirstName = "Terry", LastName = "Adams" };
    Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };
    Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

    Pet barley = new Pet { Name = "Barley", Owner = terry };
    Pet boots = new Pet { Name = "Boots", Owner = terry };
    Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };
    Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };
    Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

    // Create two lists.
    List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };
    List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

    var query = from person in people
               join pet in pets on person equals pet.Owner into gj
               from subpet in gj.DefaultIfEmpty()
               select new { person.FirstName, PetName = subpet?.Name ?? String.Empty };

    foreach (var v in query)
    {
        Console.WriteLine($"{v.FirstName} : {v.PetName}");
    }
}

// This code produces the following output:
//
// Magnus:      Daisy
// Terry:       Barley
// Terry:       Boots
// Terry:       Blue Moon
// Charlotte:   Whiskers
// Arlene:

```

## 참조

- [Join](#)
- [GroupJoin](#)
- [내부 조인 수행](#)
- [그룹화 조인 수행](#)
- [무명 형식](#)

# join 절 결과를 순서대로 정렬

2020-03-18 • 2 minutes to read • [Edit Online](#)

이 예제에서는 조인 작업 결과를 순서대로 정렬하는 방법을 보여 줍니다. 조인 후 순서대로 정렬합니다. 조인 전에 하나 이상의 소스 시퀀스와 함께 `orderby` 절을 사용할 수도 있지만 일반적으로 권장되지는 않습니다. 일부 LINQ 공급자는 조인 후에 정렬 순서를 유지하지 않을 수도 있습니다.

## 예제

이 쿼리는 그룹 조인을 만든 후 범위 내에 있는 범주 요소에 따라 그룹을 정렬합니다. 무명 형식 이니셜라이저 내의 하위 쿼리는 제품 시퀀스에서 일치하는 모든 요소를 순서대로 정렬합니다.

```
class HowToOrderJoins
{
    #region Data
    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
    {
        new Category(){Name="Beverages", ID=001},
        new Category(){ Name="Condiments", ID=002},
        new Category(){ Name="Vegetables", ID=003},
        new Category() { Name="Grains", ID=004},
        new Category() { Name="Fruit", ID=005}
    };

    // Specify the second data source.
    List<Product> products = new List<Product>()
    {
        new Product{Name="Cola", CategoryID=001},
        new Product{Name="Tea", CategoryID=001},
        new Product{Name="Mustard", CategoryID=002},
        new Product{Name="Pickles", CategoryID=002},
        new Product{Name="Carrots", CategoryID=003},
        new Product{Name="Bok Choy", CategoryID=003},
        new Product{Name="Peaches", CategoryID=005},
        new Product{Name="Melons", CategoryID=005},
    };
    #endregion
    static void Main()
    {
        HowToOrderJoins app = new HowToOrderJoins();
        app.OrderJoin1();

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

```

void OrderJoin1()
{
    var groupJoinQuery2 =
        from category in categories
        join prod in products on category.ID equals prod.CategoryID into prodGroup
        orderby category.Name
        select new
        {
            Category = category.Name,
            Products = from prod2 in prodGroup
                        orderby prod2.Name
                        select prod2
        };
}

foreach (var productGroup in groupJoinQuery2)
{
    Console.WriteLine(productGroup.Category);
    foreach (var prodItem in productGroup.Products)
    {
        Console.WriteLine($" {prodItem.Name,-10} {prodItem.CategoryID}");
    }
}
/* Output:
   Beverages
      Cola      1
      Tea       1
   Condiments
      Mustard   2
      Pickles   2
   Fruit
      Melons    5
      Peaches   5
   Grains
   Vegetables
      Bok Choy  3
      Carrots   3
*/
}

```

## 참고 항목

- [LINQ\(Language-Integrated Query\)](#)
- [orderby 절](#)
- [join 절](#)

# 복합 키를 사용하여 조인

2020-03-18 • 2 minutes to read • [Edit Online](#)

이 예제에서는 둘 이상의 키를 사용하여 일치를 정의하려는 조인 작업을 수행하는 방법을 보여 줍니다. 이 작업은 복합 키를 사용하여 수행됩니다. 무명 형식이나 비교할 값이 포함된 명명된 형식으로 복합 키를 만듭니다. 쿼리 변수가 메서드 경계를 넘어 전달되는 경우 키의 [Equals](#) 및 [GetHashCode](#)를 재정의하는 명명된 형식을 사용합니다. 속성의 이름과 속성이 나타나는 순서는 각 키에서 동일해야 합니다.

## 예제

다음 예제에서는 복합 키를 사용하여 세 테이블의 데이터를 조인하는 방법을 보여 줍니다.

```
var query = from o in db.Orders
            from p in db.Products
            join d in db.OrderDetails
                on new {o.OrderID, p.ProductID} equals new {d.OrderID, d.ProductID} into details
                from d in details
            select new {o.OrderID, p.ProductID, d.UnitPrice};
```

복합 키에 대한 형식 유추는 키에 있는 속성의 이름 및 속성이 나타나는 순서에 따라 달라집니다. 소스 시퀀스에 있는 속성에 동일한 이름이 없는 경우 키에서 새 이름을 할당해야 합니다. 예를 들어 `orders` 테이블과 `OrderDetails` 테이블이 각각 해당 열에 다른 이름을 사용한 경우 무명 형식에서 동일한 이름을 지정하여 복합 키를 만들 수 있습니다.

```
join...on new {Name = o.CustomerName, ID = o.CustID} equals
        new {Name = d.CustName, ID = d.CustID }
```

복합 키는 `group` 절에서도 사용할 수 있습니다.

## 참고 항목

- [LINQ\(Language-Integrated Query\)](#)
- [join 절](#)
- [group 절](#)

# 사용자 지정 조인 작업 수행

2020-03-18 • 8 minutes to read • [Edit Online](#)

이 예제에서는 `join` 절에 사용할 수 없는 조인 작업을 수행하는 방법을 보여 줍니다. 쿼리 식에서 `join` 절은 조인 작업의 가장 일반적인 형식인 동등 조인으로 제한되고 최적화되었습니다. 동등 조인을 수행하는 경우 `join` 절을 사용하면 항상 최상의 성능을 얻을 수 있습니다.

그러나 `join` 절은 다음과 같은 경우에 사용할 수 없습니다.

- 조인이 같지 않음(비동등 조인)의 식에서 서술된 경우
- 조인이 둘 이상의 같음 또는 같지 않음에서 서술된 경우
- 조인 작업 앞에서 오른쪽(내부) 시퀀스에 대한 임시 범위 변수를 도입해야 하는 경우

조인 동등이 아닌 조인을 수행하려면 여러 개의 `from` 절을 사용하여 각 데이터 소스를 독립적으로 도입할 수 있습니다. 그런 다음 `where` 절의 조건자 식을 각 소스에 대한 범위 변수에 적용합니다. 식은 메서드 호출의 형식을 사용할 수도 있습니다.

## NOTE

여러 `from` 절을 사용하여 내부 컬렉션에 액세스하는 경우와 이러한 종류의 사용자 지정 조인 작업을 혼동하지 마세요. 자세한 내용은 [join 절](#)을 참조하세요.

## 예제

다음 예제의 첫 번째 메서드는 간단한 크로스 조인을 보여 줍니다. 크로스 조인은 매우 큰 결과 집합을 생성할 수 있으므로 주의해서 사용해야 합니다. 그러나 추가 쿼리가 실행되는 소스 시퀀스를 만들기 위해 일부 시나리오에서 사용할 수 있습니다.

두 번째 메서드는 범주 ID가 왼쪽의 범주 목록에 나열된 모든 제품의 시퀀스를 생성합니다. `let` 절과 `Contains` 메서드를 사용하여 임시 배열을 만듭니다. 또한 쿼리 앞에서 배열을 만들고 첫 번째 `from` 절을 제거할 수도 있습니다.

```
class CustomJoins
{
    #region Data

    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
    {
        new Category(){Name="Beverages", ID=001},
        new Category(){Name="Condiments", ID=002}.
    }
}
```

```

    new Category(){ Name="Vegetables", ID=003},
};

// Specify the second data source.
List<Product> products = new List<Product>()
{
    new Product{Name="Tea", CategoryID=001},
    new Product{Name="Mustard", CategoryID=002},
    new Product{Name="Pickles", CategoryID=002},
    new Product{Name="Carrots", CategoryID=003},
    new Product{Name="Bok Choy", CategoryID=003},
    new Product{Name="Peaches", CategoryID=005},
    new Product{Name="Melons", CategoryID=005},
    new Product{Name="Ice Cream", CategoryID=007},
    new Product{Name="Mackerel", CategoryID=012},
};
#endifregion

static void Main()
{
    CustomJoins app = new CustomJoins();
    app.CrossJoin();
    app.NonEquijoin();

    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}

void CrossJoin()
{
    var crossJoinQuery =
        from c in categories
        from p in products
        select new { c.ID, p.Name };

    Console.WriteLine("Cross Join Query:");
    foreach (var v in crossJoinQuery)
    {
        Console.WriteLine($"{v.ID,-5}{v.Name}");
    }
}

void NonEquijoin()
{
    var nonEquijoinQuery =
        from p in products
        let catIds = from c in categories
                    select c.ID
        where catIds.Contains(p.CategoryID) == true
        select new { Product = p.Name, CategoryID = p.CategoryID };

    Console.WriteLine("Non-equijoin query:");
    foreach (var v in nonEquijoinQuery)
    {
        Console.WriteLine($"{v.CategoryID,-5}{v.Product}");
    }
}

/* Output:
Cross Join Query:
1   Tea
1   Mustard
1   Pickles
1   Carrots
1   Bok Choy
1   Peaches
1   Melons
1   Ice Cream
1   Mackerel

```

```
1      MATCHED 1
2      Tea
2      Mustard
2      Pickles
2      Carrots
2      Bok Choy
2      Peaches
2      Melons
2      Ice Cream
2      Mackerel
3      Tea
3      Mustard
3      Pickles
3      Carrots
3      Bok Choy
3      Peaches
3      Melons
3      Ice Cream
3      Mackerel
Non-equijoin query:
1      Tea
2      Mustard
2      Pickles
3      Carrots
3      Bok Choy
Press any key to exit
*/
```

## 예제

다음 예제에서는 쿼리가 내부(오른쪽) 시퀀스의 경우 조인 절 앞에서 가져올 수 없는 일치 키에 따라 두 시퀀스를 조인해야 합니다. `join` 절을 사용하여 이 조인을 수행하는 경우 각 요소에 대해 `Split` 메서드를 호출해야 합니다. 여러 개의 `from` 절을 사용하면 쿼리에서 반복된 메서드 호출의 오버헤드를 방지할 수 있습니다. 그러나 `join` 이 최적화되었으므로 이 특정 사례에서는 여러 개의 `from` 절을 사용하는 것보다 더 빠를 수 있습니다. 결과는 주로 메서드 호출이 얼마나 광범위한지에 따라 달라집니다.

```

// Optional. Store the newly created student objects in memory
// for faster access in future queries
List<Student> students = queryNamesScores.ToList();

foreach (var student in students)
{
    Console.WriteLine($"The average score of {student.FirstName} {student.LastName} is
{student.ExamScores.Average()}.");
}

//Keep console window open in debug mode
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public List<int> ExamScores { get; set; }
}

/* 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.
*/

```

## 참고 항목

- [LINQ\(Language-Integrated Query\)](#)
- [join 절](#)
- [Join 절 결과를 서순대로 정렬](#)

# 쿼리 식의 Null 값 처리

2020-04-02 • 2 minutes to read • [Edit Online](#)

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 null이거나 값이 null인 요소를 포함하고 사용 중인 쿼리가 null 값을 처리하지 않는 경우 쿼리를 실행하면 `NullReferenceException`이 throw됩니다.

## 예제

다음 예제와 같이 null 참조 예외를 피하도록 방어적으로 코딩할 수 있습니다.

```
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.

## 예제

`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.

```
void TestMethod(Northwind db)
{
    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 };
}
```

## 참고 항목

- [Nullable<T>](#)
- [LINQ\(Language-Integrated Query\)](#)
- [Nullable 값 형식](#)

# 쿼리 식의 예외 처리

2020-03-18 • 5 minutes to read • [Edit Online](#)

쿼리 식의 컨텍스트에서 모든 메서드를 호출할 수 있습니다. 그러나 데이터 소스의 내용을 수정하거나 예외를 throw하는 것과 같은 부작용이 생길 수 있는 쿼리 식에서는 메서드를 호출하지 않는 것이 좋습니다. 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 throw된 이유를 이해할 경우 이 예외를 catch할 수 있습니다. 자세한 내용은 [최선의 예외 구현 방법](#)을 참조하세요.

마지막 예제에서는 쿼리 실행 중에 예외를 throw해야 할 경우 사례를 처리하는 방법을 보여 줍니다.

## 예제

다음 예제에서는 예외 처리 코드를 쿼리 식 외부로 이동하는 방법을 보여 줍니다. 이 작업은 메서드가 쿼리에 로컬인 변수에 의존하지 않는 경우에만 가능합니다.

```
class ExceptionsOutsideQuery
{
    static void Main()
    {
        // DO THIS with a datasource that might
        // throw an exception. It is easier to deal with
        // outside of the query expression.
        IEnumerable<int> dataSource;
        try
        {
            dataSource = GetData();
        }
        catch (InvalidOperationException)
        {
            // Handle (or don't handle) the exception
            // in the way that is appropriate for your application.
            Console.WriteLine("Invalid operation");
            goto Exit;
        }

        // 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());

        //Keep the console window open in debug mode
        Exit:
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // A data source that is very likely to throw an exception!
    static IEnumerable<int> GetData()
    {
        throw new InvalidOperationException();
    }
}
```

## 예제

몇몇 경우에는 쿼리 내에서 throw된 예외에 대한 가장 좋은 응답은 쿼리 실행을 즉시 종지하는 것입니다. 다음 예제에서는 쿼리 본문 내부에서 throw된 예외를 처리하는 방법을 보여 줍니다. `SomeMethodThatMightThrow` 가 잠재적으로 쿼리 실행을 종지해야 하는 예외를 일으킬 수 있다고 가정합니다.

`try` 블록은 쿼리 자체가 아니라 `foreach` 루프를 포함합니다. 이는 쿼리가 실제로 실행되는 지점이 `foreach` 루프이기 때문입니다. 자세한 내용은 [LINQ 쿼리 소개](#)를 참조하세요.

```
class QueryThatThrows
{
    static void Main()
    {
        // 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;

        // Runtime exceptions are thrown when query is executed.
        // Therefore they must be handled in the foreach loop.
        try
        {
            foreach (var item in exceptionDemoQuery)
            {
                Console.WriteLine($"Processing {item}");
            }
        }

        // Catch whatever exception you expect to raise
        // and/or do any necessary cleanup in a finally block
        catch (InvalidOperationException e)
        {
            Console.WriteLine(e.Message);
        }

        //Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Not very useful as a general purpose method.
    static string SomeMethodThatMightThrow(string s)
    {
        if (s[4] == 'C')
            throw new InvalidOperationException();
        return @"C:\newFolder\" + s;
    }
}
/* Output:
   Processing C:\newFolder\fileA.txt
   Processing C:\newFolder\fileB.txt
   Operation is not valid due to the current state of the object.
*/
```

## 참고 항목

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

# 비동기 프로그래밍

2021-02-18 • 29 minutes to read • [Edit Online](#)

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. 이 작업은 다음과 같이 구현할 수 있습니다.

```
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.

```

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

비동기 작업과 관련하여 많은 작업이 수행됩니다. `Task` 및 `Task<T>`의 백그라운드에서 수행되는 작업이 궁금하면 [세부 비동기](#) 문서에서 자세한 내용을 확인하세요.

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 약속 모델](#)입니다.

## I/O 해야 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 모델링 하는 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 있을 경우 `Task.Run` 없이 `async` 및 `await`를 사용합니다. 작업 병렬 라이브러리를 사용하면 안 됩니다. 그 이유는 [세부 비동기](#)에 설명되어 있습니다.

CPU 바인딩된 작업이 있고 빠른 응답이 필요할 경우 `async` 및 `await`를 사용하지만 `Task.Run`을 사용하여 또 다른 스레드에서 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 비해 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.

#### NOTE

프로덕션 코드에서 HTML 구문 분석을 수행하려는 경우 정규식을 사용하지 마세요. 대신 구문 분석 라이브러리를 사용하세요.

```
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.

```

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.

```

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.

```

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));
    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.

## 중요한 정보 및 조언

비동기 프로그래밍을 사용하는 경우 예기치 않은 동작을 방지할 수 있는 몇 가지 세부 사항을 염두에 두어야 합니다.

- `async` 메서드에는 본문에 `await` 키워드가 있어야 합니다. 키워드가 없으면 일시 중단되지 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 추가해야 합니다.

이 규칙을 .NET에서 사용하여 동기 및 비동기 메서드를 더 쉽게 구별할 수 있습니다. 코드에서 명시적으로 호출되지 않은 특정 메서드(예: 이벤트 처리기 또는 웹 컨트롤러 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	수행 할 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

- 가능하면 `ValueTask`를 사용

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 경우 코드의 성능이 중요한 셙션에서 추가 할당에 상당한 시간이 소요될 수 있습니다. 연속 루프에서 이러한 할당이 발생하면 부담이 될 수 있습니다. 자세한 내용은 [일반화된 비동기 반환 형식](#)을 참조하세요.

- `ConfigureAwait(false)`를 사용

일반적인 질문은 "언제 `Task.ConfigureAwait(Boolean)` 메서드를 사용해야 하는가"입니다. 이 메서드를 사용하면 `Task` 인스턴스가 awainer를 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 확실히 예측 가능하고, 테스트 가능하고, 유지 관리 가능한 코드베이스가 생성됩니다.

## 기타 리소스

- [세부 비동기](#)에서는 작업이 어떻게 작동하는지 자세히 설명합니다.
- [async 및 await를 사용한 비동기 프로그래밍\(C#\)](#)
- Lucian Wischik의 [Six Essential Tips for Async](#)(비동기에 대한 6가지 필수 팁)는 비동기 프로그래밍에 대한 훌륭한 리소스입니다.

# 패턴 일치

2020-11-02 • 32 minutes to read • [Edit Online](#)

패턴은 값에 특정 모양이 있는지 테스트하고 모양이 일치하는 경우 값에서 정보를 추출할 수 있습니다. 패턴 일치는 이미 사용하고 있는 알고리즘에 대해 더 간결한 구문을 제공합니다. 이미 기존 구문을 사용하여 패턴 일치 알고리즘을 만듭니다. 값을 테스트하는 `if` 또는 `switch` 문을 작성합니다. 그런 다음 이러한 문이 일치할 경우 해당 값에서 정보를 추출하고 사용합니다. 새 구문 요소는 이미 친숙한 `is` 및 `switch` 문에 대한 확장입니다. 이러한 새 확장은 값 테스트와 해당 정보 추출을 결합합니다.

이 문서에서는 읽을 수 있고 간결한 코드를 사용하는 방법을 보여 주는 새로운 구문을 살펴보겠습니다. 패턴 일치를 통해 데이터와 데이터를 조작하는 메서드가 긴밀하게 연결된 개체 지향 디자인과 달리 데이터와 코드가 구분된 구문을 사용할 수 있습니다.

이러한 새 구문을 보여 주기 위해 패턴 일치 문을 사용하여 도형을 나타내는 구조체를 살펴보겠습니다. 클래스 계층 구조를 작성하고 [가상 메서드 및 재정의된 메서드](#)를 만들어 개체의 런타임 형식에 따라 개체 동작을 사용자 지정하는 작업에 대해 이미 잘 알고 있을 것입니다.

이러한 기술은 클래스 계층 구조로 구성되지 않은 데이터에는 사용할 수 없습니다. 데이터와 메서드가 분리된 경우 다른 도구가 필요합니다. 새 [패턴 일치](#) 구문에서는 더 명확한 구문으로 데이터를 검사하고 해당 데이터의 조건에 따라 제어 흐름을 조작할 수 있습니다. 이미 변수 값을 테스트하는 `if` 및 `switch` 문을 작성합니다. 변수 형식을 테스트하는 `is` 문을 작성합니다. 패턴 일치는 이러한 문에 새로운 기능을 추가합니다.

이 문서에서는 여러 도형의 면적을 계산하는 메서드를 작성합니다. 그러나 이 작업을 위해 개체 지향 기술을 사용하여 다양한 세이프에 대한 클래스 계층 구조를 작성하지 않습니다. 대신 [패턴 일치](#)를 사용합니다. 이 샘플을 진행하면서 이 코드를 개체 계층 구조로 구성된 방식과 대조합니다. 쿼리 및 조작해야 하는 데이터가 클래스 계층 구조가 아닌 경우 패턴 일치를 통해 세련된 디자인을 사용할 수 있습니다.

추상 세이프 정의로 시작하고 다른 특정 세이프 클래스를 추가하는 대신 각 도형에 대한 간단한 데이터 전용 정의로 시작하겠습니다.

```

public class Square
{
    public double Side { get; }

    public Square(double side)
    {
        Side = side;
    }
}

public class Circle
{
    public double Radius { get; }

    public Circle(double radius)
    {
        Radius = radius;
    }
}

public struct Rectangle
{
    public double Length { get; }
    public double Height { get; }

    public Rectangle(double length, double height)
    {
        Length = length;
        Height = height;
    }
}

public class Triangle
{
    public double Base { get; }
    public double Height { get; }

    public Triangle(double @base, double height)
    {
        Base = @base;
        Height = height;
    }
}

```

이러한 구조체에서 일부 셰이프의 면적을 계산하는 메서드를 작성하겠습니다.

## is 형식 패턴 식

C# 7.0 이전에는 일련의 `if` 및 `is` 문에서 각 형식을 테스트해야 했습니다.

```

public static double ComputeArea(object shape)
{
    if (shape is Square)
    {
        var s = (Square)shape;
        return s.Side * s.Side;
    }
    else if (shape is Circle)
    {
        var c = (Circle)shape;
        return c.Radius * c.Radius * Math.PI;
    }
    // elided
    throw new ArgumentException(
        message: "shape is not a recognized shape",
        paramName: nameof(shape));
}

```

위의 코드는 '형식 패턴'의 클래식 식입니다. 변수를 테스트하여 해당 형식을 확인하고 해당 형식에 따라 다른 작업을 수행합니다.

`is` 식에 대한 확장을 사용하여 테스트에 성공할 경우 변수를 할당하면 이 코드가 더 간단해집니다.

```

public static double ComputeAreaModernIs(object shape)
{
    if (shape is Square s)
        return s.Side * s.Side;
    else if (shape is Circle c)
        return c.Radius * c.Radius * Math.PI;
    else if (shape is Rectangle r)
        return r.Height * r.Length;
    // elided
    throw new ArgumentException(
        message: "shape is not a recognized shape",
        paramName: nameof(shape));
}

```

이 업데이트된 버전에서 `is` 식은 변수를 테스트하고 적절한 형식의 새로운 변수에 할당합니다. 또한 이 버전에는 `struct` 인 `Rectangle` 형식이 포함되어 있습니다. 새 `is` 식은 참조 형식뿐 아니라 값 형식에서도 작동합니다.

패턴 일치 식에 대한 언어 규칙은 일치 식의 결과를 잘못 사용하는 경우를 방지하는 데 도움이 됩니다. 위의 예제에서 `s`, `c` 및 `r` 변수는 범위 내에만 있고 해당 패턴 일치 식에 `true` 결과가 있는 경우 무기한 할당됩니다. 다른 위치에 있는 변수 중 하나를 사용하려는 경우 코드에서 컴파일러 오류를 생성합니다.

범위부터 시작하여 해당 규칙을 자세히 살펴보겠습니다. `c` 변수는 첫 번째 `if` 문의 `else` 분기에서만 범위 내에 있습니다. `s` 변수는 `ComputeAreaModernIs` 메서드에서 범위 내에 있습니다. 이는 `if` 문의 각 분기가 변수에 대해 별도 범위를 설정하기 때문입니다. 그러나 `if` 문 자체는 별도 범위를 설정하지 않습니다. 즉, `if` 문에서 선언된 변수는 `if` 문(이 경우 메서드)과 동일한 범위에 있습니다. 이 동작은 패턴 일치와 관련은 없지만 변수 범위와 `if` 및 `else` 문에 대해 정의된 동작입니다.

`c` 및 `s` 변수는 한정적으로 할당된 `true` 시 메커니즘 때문에 해당 `if` 문이 `true`일 때 할당됩니다.

### TIP

이 항목의 샘플에서는 패턴 일치 `is` 식이 `if` 문의 `true` 분기에 있는 일치 변수를 한정적으로 할당하는 권장 구문을 사용합니다. `if (!(shape is Square s))` 및 `s` 변수가 `false` 분기에서만 한정적으로 할당된다고 지정하면 논리를 반전시킬 수 있습니다. 이 논리는 C#에서 유효하지만 논리를 따르는 것이 더 혼동되기 때문에 따르지 않는 것이 좋습니다.

이 규칙은 해당 패턴이 충족되지 않은 경우 패턴 일치 식의 결과에 실수로 액세스할 가능성성이 없음을 의미합니다.

## 패턴 일치 `switch` 문 사용

시간이 흐르면서 다른 세이프 형식을 지원해야 할 수도 있습니다. 테스트하는 조건 수가 증가함에 따라 `is` 패턴 일치 식 사용이 불편해질 수 있습니다. 확인하려는 각 형식에 대한 `if` 문이 필요할 뿐 아니라 `is` 식은 입력이 단일 형식과 일치하는지 여부의 테스트로 제한됩니다. 이 경우 `switch` 패턴 일치 식을 선택하는 것이 더 나을 수 있습니다.

기존의 `switch` 문은 패턴 식이었으며 상수 패턴을 지원했습니다. `case` 문에서 사용된 상수와 변수를 비교할 수 있습니다.

```
public static string GenerateMessage(params string[] parts)
{
    switch (parts.Length)
    {
        case 0:
            return "No elements to the input";
        case 1:
            return $"One element: {parts[0]}";
        case 2:
            return $"Two elements: {parts[0]}, {parts[1]}";
        default:
            return $"Many elements. Too many to write";
    }
}
```

`switch` 문이 지원하는 패턴은 상수 패턴뿐이었습니다. 숫자 형식과 `string` 형식으로 더욱 제한되었습니다. 이러한 제한 사항이 제거되었으며, 이제 형식 패턴을 사용하여 `switch` 문을 작성할 수 있습니다.

```
public static double ComputeAreaModernSwitch(object shape)
{
    switch (shape)
    {
        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        case Rectangle r:
            return r.Height * r.Length;
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}
```

패턴 일치 `switch` 문은 기존의 C 스타일 `switch` 문을 사용한 개발자에게 친숙한 구문을 사용합니다. 각 `case` 가 평가되고, 입력 변수와 일치하는 조건 아래에 있는 코드가 실행됩니다. 코드 실행은 `case` 식 간에 “이동”할 수

없습니다. `case` 문의 구문에서는 각 `case` 가 `break`, `return` 또는 `goto`로 끝나야 합니다.

#### NOTE

다른 레이블로 이동하는 `goto` 문은 클래식 switch 문인 상수 패턴에만 유효합니다.

`switch` 문을 제어하는 중요한 새 규칙이 있습니다. `switch` 식의 변수 형식에 대한 제한 사항이 제거되었습니다. 이 예제의 `object`와 같은 모든 형식을 사용할 수 있습니다. `case` 식이 더 이상 상수 값으로 제한되지 않습니다. 이러한 제한이 제거되면서 `switch` 섹션을 다시 정렬할 경우 프로그램의 동작이 변경될 수 있습니다.

상수 값으로 제한될 경우 최대 한 개의 `case` 레이블만 `switch` 식과 일치할 수 있습니다. 모든 `switch` 섹션이 다음 섹션으로 이동해서는 안 되는 규칙과 결합되어 동작에 영향을 주지 않고 `switch` 섹션을 임의의 순서로 다시 정렬할 수 있습니다. 이제 보다 일반화된 `switch` 식을 사용하므로 각 섹션의 순서가 중요합니다. `switch` 식은 텍스트 순서로 계산됩니다. `switch` 식과 일치하는 첫 번째 `switch` 레이블로 실행이 전송됩니다.

`default` `case`는 일치하는 다른 `case` 레이블이 없는 경우에만 실행됩니다. `default` 사례는 텍스트 순서에 관계 없이 마지막으로 평가됩니다. `default` `case`가 없고 일치하는 다른 `case` 문이 없는 경우 `switch` 문 뒤의 문에서 실행이 계속됩니다. `case` 레이블 코드는 실행되지 않습니다.

## case 식의 when 절

`case` 레이블에 `when` 절을 사용하여 면적이 0인 해당 세이프에 대한 특수 사례를 만들 수 있습니다. 측면 길이가 0인 사각형 또는 반지름이 0인 원은 면적이 0입니다. `case` 레이블에 `when` 절을 사용하여 해당 조건을 지정할 수 있습니다.

```
public static double ComputeArea_Version3(object shape)
{
    switch (shape)
    {
        case Square s when s.Side == 0:
        case Circle c when c.Radius == 0:
            return 0;

        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}
```

이 변경 내용은 새 구문에 대한 몇 가지 중요한 사항을 보여 줍니다. 첫째, 여러 `case` 레이블을 하나의 `switch` 섹션에 적용할 수 있습니다. 문 블록은 이러한 레이블이 `true`인 경우에 실행됩니다. 이 인스턴스에서 `switch` 식이 면적이 0인 원 또는 사각형인 경우 메서드가 상수 0을 반환합니다.

이 예제에서는 첫 번째 `switch` 블록에 대한 두 개의 `case` 레이블에 있는 두 개의 변수를 소개합니다. 이 `switch` 블록의 문은 `c`(원) 또는 `s`(사각형) 변수를 사용하지 않습니다. 이러한 변수는 이 `switch` 블록에서 한 정적으로 할당되지 않습니다. 이 사례 중 하나가 일치하면 변수 중 하나가 명확하게 할당된 것입니다. 그러나 `case` 중 하나가 런타임에 일치할 수 있기 때문에 컴파일 시간에 어떤 사례가 할당되었는지 알 수 없습니다. 이런 이유로 동일한 블록에 여러 `case` 레이블을 사용하는 경우 대부분 `case` 문에서 새 변수를 도입하지 않거나 `when` 절에서만 변수를 사용합니다.

면적이 0인 세이프를 추가한 경우 사각형과 삼각형인 세이프 유형 두 개를 더 추가하겠습니다.

```

public static double ComputeArea_Version4(object shape)
{
    switch (shape)
    {
        case Square s when s.Side == 0:
        case Circle c when c.Radius == 0:
        case Triangle t when t.Base == 0 || t.Height == 0:
        case Rectangle r when r.Length == 0 || r.Height == 0:
            return 0;

        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        case Triangle t:
            return t.Base * t.Height / 2;
        case Rectangle r:
            return r.Length * r.Height;
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}

```

이 변경 내용은 degenerate 사례에 대한 `case` 레이블과 새로운 각 세이프에 대한 레이블 및 블록을 추가합니다.

마지막으로, `null` case를 추가하여 인수가 `null`이 아닌지 확인할 수 있습니다.

```

public static double ComputeArea_Version5(object shape)
{
    switch (shape)
    {
        case Square s when s.Side == 0:
        case Circle c when c.Radius == 0:
        case Triangle t when t.Base == 0 || t.Height == 0:
        case Rectangle r when r.Length == 0 || r.Height == 0:
            return 0;

        case Square s:
            return s.Side * s.Side;
        case Circle c:
            return c.Radius * c.Radius * Math.PI;
        case Triangle t:
            return t.Base * t.Height / 2;
        case Rectangle r:
            return r.Length * r.Height;
        case null:
            throw new ArgumentNullException(paramName: nameof(shape), message: "Shape must not be null");
        default:
            throw new ArgumentException(
                message: "shape is not a recognized shape",
                paramName: nameof(shape));
    }
}

```

`null` 패턴의 특수 동작은 패턴에서 형식이 없는 `null` 상수를 모든 참조 형식 또는 `null` 허용 값 형식으로 변환할 수 있기 때문에 흥미롭습니다. `null`을 임의 형식으로 변환하는 대신, 언어에서 `null` 값은 변수의 컴파일 시간 형식과 관계없이 어떠한 형식 패턴과도 일치하지 않는다고 정의합니다. 이 동작을 통해 새 `switch` 기반 형식 패턴이 `is` 문과 일치하게 됩니다. `is` 문은 확인되는 값이 `null` 일 경우 항상 `false`를 반환합니다. 더 간단하기도 합니다. 형식을 확인한 후에는 추가 `null` 검사가 필요하지 않습니다. 위 샘플의 `case` 블록에 `null` 검사가 없

다는 사실에서 이를 확인할 수 있습니다. 형식 패턴 일치를 통해 null이 아닌 값이 보장되므로 null 검사가 필요하지 않습니다.

## case 식의 var 선언

match 식 중 하나인 var이 소개되면서 패턴 일치에 대한 새 규칙이 도입됩니다.

첫 번째 규칙은 var 선언이 일반 형식 유추 규칙을 따른다는 것입니다. 형식은 switch 식의 정적 형식으로 유추됩니다. 이 규칙에 따라 형식이 항상 일치합니다.

두 번째 규칙은 var 선언에 다른 형식 패턴 식이 포함하는 null 검사가 없다는 것입니다. 즉, 변수가 null일 수 있고 이 경우 null 검사가 필요합니다.

이러한 두 규칙은 여러 인스턴스에서 case 식의 var 선언이 default 식과 동일한 조건과 일치함을 의미합니다. default 사례보다 기본값이 아닌 사례가 선호되므로 default 사례는 실행되지 않습니다.

### NOTE

컴파일러는 default 사례가 작성되었지만 실행되지 않을 경우에 경고를 표시하지 않습니다. 이것은 모든 가능한 사례가 나열된 현재 switch 문 동작과 일치합니다.

세 번째 규칙은 var 사례가 유용할 수 있는 사용법을 소개합니다. 입력이 문자열이고 알려진 명령 값을 검색하는 패턴 일치를 수행한다고 가정해 봅니다. 다음과 같이 작성할 수 있습니다.

```
static object CreateShape(string shapeDescription)
{
    switch (shapeDescription)
    {
        case "circle":
            return new Circle(2);

        case "square":
            return new Square(4);

        case "large-circle":
            return new Circle(12);

        case var o when (o?.Trim().Length ?? 0) == 0:
            // white space
            return null;
        default:
            return "invalid shape description";
    }
}
```

var 사례는 null, 빈 문자열, 또는 공백만 포함하는 문자열과 일치합니다. 앞의 코드에서는 ?. 연산자를 사용하여 실수로 NullReferenceException을 throw하지 않도록 합니다. default case는 이 명령 파서에서 인식되지 않는 기타 문자열 값을 처리합니다.

이는 default 식과 구별되는 var 사례 식을 고려할 수 있는 하나의 예제입니다.

## 결론

'패턴 일치 구문'을 사용하면 상속 계층 구조와 관련이 없는 다양한 변수 및 형식 간의 제어 흐름을 쉽게 관리할 수 있습니다. 논리를 제어하여 변수에 테스트하는 조건을 사용할 수도 있습니다. 데이터 및 해당 데이터를 조작하는 메서드가 별개인 더욱 분산된 애플리케이션을 빌드하는 경우 더 자주 필요한 패턴과 구문을 사용할 수 있습니다. 이 샘플에 사용된 셰이프 구조체에는 메서드가 없고 읽기 전용 속성만 있습니다. 패턴 일치는 모든 데이터 형식에서 작동합니다. 개체를 검사하는 식을 작성하고 해당 조건에 따라 제어 흐름 결정을 내립니다.

추상 `Shape` 및 각각 고유한 가상 메서드 구현으로 면적을 계산하는 특정 파생 셰이프에 대한 클래스 계층 구조를 만들어 수행하는 디자인과 이 샘플의 코드를 비교합니다. 데이터로 작업하고 데이터 스토리지 문제와 동작 문제를 구분하려는 경우 패턴 일치 식은 매우 유용한 도구일 수 있습니다.

## 참고 항목

- [자습서: 패턴 일치를 사용하여 형식 기반 및 데이터 기반 알고리즘 빌드](#)

# 안전하고 효율적인 C# 코드 작성

2020-04-27 • 46 minutes to read • [Edit Online](#)

C#의 새 기능을 사용하면 향상된 성능으로 안정형의 안전 코드를 작성할 수 있습니다. 이러한 기술을 신중하게 적용한다면 안전하지 않은 코드가 필요한 시나리오는 더 적어집니다. 이러한 기능을 통해 값 형식에 대한 참조를 메서드 인수 및 메서드 반환 값으로 사용하기가 더 쉬워집니다. 안전하게 수행하면 이러한 기술로 값 형식 복사가 최소화됩니다. 값 형식을 사용하면 할당 수 및 가비지 수집 단계를 최소화할 수 있습니다.

이 문서의 샘플 코드 상당수는 C# 7.2에 추가된 기능을 사용합니다. 이러한 기능을 사용하려면 C# 7.2 이상을 사용하도록 프로젝트를 구성해야 합니다. 언어 버전 설정에 대한 자세한 내용은 [언어 버전 구성](#)을 참조하세요.

이 문서에서는 효율적인 리소스 관리를 위한 기술에 중점을 둡니다. 값 형식을 사용할 경우 한 가지 장점은 대개 힙 할당을 할 필요가 없다는 것입니다. 단점은 값으로 복사된다는 점입니다. 이러한 장단점 간에 균형을 잡으려고 하니 많은 양의 데이터에서 작동하는 알고리즘을 최적화하기가 어렵습니다. C# 7.2의 새로운 언어 기능은 값 형식에 대한 참조를 사용하여 안전하고 효율적인 코드를 가능하게 하는 메커니즘을 제공합니다. 이러한 기능을 현명하게 사용하여 할당 및 복사 작업을 최소화하세요. 이 문서에서는 이러한 새로운 기능을 살펴봅니다.

이 문서에서는 다음과 같은 리소스 관리 기술에 중점을 둡니다.

- `readonly struct`를 선언하여 형식이 변경 불가능임을 표현합니다. 이를 통해 컴파일러는 `in` 매개 변수를 사용할 때 방어형 복사본을 저장할 수 있습니다.
- 형식을 변경할 수 없는 경우 `struct` 멤버를 `readonly`로 선언하여 멤버가 상태를 수정하지 않음을 나타냅니다.
- 반환 값이 `IntPtr.Size`보다 큰 `struct`이고 스토리지 수명이 값을 반환하는 메서드보다 클 경우 `ref readonly` 반환을 사용합니다.
- 성능 상의 이유로 `readonly struct`의 크기가 `IntPtr.Size`보다 큰 경우 `in` 매개 변수로 전달해야 합니다.
- `readonly` 한정자로 선언되었거나 메서드가 구조체의 `readonly` 멤버만 호출하는 경우를 제외하고 `struct`를 `in` 매개 변수로 전달하면 안 됩니다. 이 지침을 위반하면 성능이 저하되고 모호한 동작이 발생할 수 있습니다.
- 메모리를 바이트의 시퀀스로 사용하도록 `ref struct` 또는 `readonly ref struct` (예: `Span<T>` 또는 `ReadOnlySpan<T>`)를 사용합니다.

이러한 기술을 통해 참조 및 값에 관한 상충적인 두 가지 목표 간에 균형을 유지할 수 있습니다. 참조 형식인 변수는 메모리의 위치에 대한 참조를 유지합니다. 값 형식인 변수는 직접 해당 값을 포함합니다. 이러한 차이는 메모리 리소스를 관리하는 데 중요한 차이점을 강조합니다. 값 형식은 일반적으로 메서드에 전달되거나 메서드에서 반환된 경우 복사됩니다. 이 동작에는 값 형식의 멤버를 호출하는 경우 `this`의 값을 복사하는 동작이 포함됩니다. 복사의 비용은 형식의 크기와 관련이 있습니다. 참조 형식은 관리형 힙에 할당됩니다. 각 새 개체는 새로 할당해야 하고 이후에 회수되어야 합니다. 이러한 두 작업은 모두 시간이 걸립니다. 참조 형식이 메서드에 인수로 전달되거나 메서드에서 반환되면 참조가 복사됩니다.

이 문서에서는 이러한 권장 사항을 설명하기 위해 3D 요소 구조체의 다음 예제 개념을 사용합니다.

```
public struct Point3D
{
    public double X;
    public double Y;
    public double Z;
}
```

다른 예제에서는 이 개념의 다른 구현을 사용합니다.

## 변경할 수 없는 값 형식에 대해 읽기 전용 구조체 선언

`readonly` 한정자를 사용하여 `struct`를 선언하면 변경할 수 없는 형식을 만들려는 의도를 컴파일러에 알립니다. 컴파일러는 다음 규칙을 사용하여 해당 디자인 결정을 적용합니다.

- 모든 필드 멤버는 `readonly`여야 합니다.
- 모든 속성은 자동으로 구현된 속성을 포함하여 읽기 전용이어야 합니다.

이러한 두 규칙으로 `readonly struct`의 멤버가 해당 구조체 상태를 수정하지 않도록 할 수 있습니다. `struct`는 변경할 수 없습니다. `Point3D` 구조체는 다음 예제에 나온 것처럼 변경할 수 없는 구조체로 정의할 수 있습니다.

```
readonly public struct ReadonlyPoint3D
{
    public ReadonlyPoint3D(double x, double y, double z)
    {
        this.X = x;
        this.Y = y;
        this.Z = z;
    }

    public double X { get; }
    public double Y { get; }
    public double Z { get; }
}
```

디자인 의도가 변경할 수 없는 값 형식을 만드는 것이라면 이 권장 사항을 따릅니다. 성능 개선 사항은 추가적인 혜택입니다. `readonly struct`는 디자인 의도를 명확하게 표현합니다.

## 구조체를 변경할 수 없는 경우 readonly 멤버 선언

C# 8.0 이상에서는 구조체 형식이 변경 가능한 경우 `readonly`로 변경되지 않는 멤버를 선언해야 합니다. 3D 요소 구조체가 필요하지만 가변성을 지원해야 하는 다른 애플리케이션을 생각해 보겠습니다. 다음 버전의 3D 요소 구조체는 구조체를 수정하지 않는 멤버에만 `readonly` 한정자를 추가합니다. 디자인에서 일부 멤버의 구조체에 대한 수정을 지원해야 하는 경우 이 예제를 따릅니다. 하지만 여전히 일부 멤버에 `readonly`를 적용하는 이점이 필요할 수 있습니다.

```

public struct Point3D
{
    public Point3D(double x, double y, double z)
    {
        _x = x;
        _y = y;
        _z = z;
    }

    private double _x;
    public double X
    {
        readonly get => _x;
        set => _x = value;
    }

    private double _y;
    public double Y
    {
        readonly get => _y;
        set => _y = value;
    }

    private double _z;
    public double Z
    {
        readonly get => _z;
        set => _z = value;
    }

    public readonly double Distance => Math.Sqrt(X * X + Y * Y + Z * Z);

    public readonly override string ToString() => $"{{X}}, {{Y}}, {{Z}}";
}

```

위의 샘플에서는 `readonly` 한정자를 적용할 수 있는 여러 위치(메서드, 속성, 속성 접근자)를 보여 줍니다. 자동 구현 속성을 사용하는 경우, 컴파일러에서 읽기/쓰기 속성의 `get` 접근자에 `readonly` 한정자를 추가합니다. 컴파일러는 `get` 접근자만 있는 속성의 자동 구현 속성 선언에 `readonly` 한정자를 추가합니다.

상태를 변경하지 않는 멤버에 `readonly` 한정자를 추가하면 두 가지 관련 혜택이 있습니다. 첫째, 컴파일러에서 의도를 적용합니다. 해당 멤버는 구조체의 상태를 변경할 수 없습니다. 둘째, 컴파일러에서 `readonly` 멤버에 액세스할 때 `in` 매개 변수의 방어형 복사본을 만들지 않습니다. 컴파일러는 `readonly` 멤버가 `struct`를 수정하지 않도록 하여 이 최적화를 안전하게 지원할 수 있습니다.

## 가능하면 큰 구조체에 `ref readonly return` 문 사용

반환되는 값이 반환 메서드에 로컬이 아닐 경우 참조로 값을 반환할 수 있습니다. 참조로 반환하는 것은 구조체가 아니라 참조만 복사된다는 것을 의미합니다. 다음 예제에서는 반환되는 값이 로컬 변수이므로 `Origin` 속성은 `ref` 반환을 사용할 수 없습니다.

```
public Point3D Origin => new Point3D(0,0,0);
```

단, 반환된 값이 정적 멤버이므로 다음 속성 정의를 참조로 반환할 수 있습니다.

```

public struct Point3D
{
    private static Point3D origin = new Point3D(0,0,0);

    // Dangerous! returning a mutable reference to internal storage
    public ref Point3D Origin => ref origin;

    // other members removed for space
}

```

호출자가 원점을 수정하지 않도록 하려면 `ref readonly`로 값을 반환해야 합니다.

```

public struct Point3D
{
    private static Point3D origin = new Point3D(0,0,0);

    public static ref readonly Point3D Origin => ref origin;

    // other members removed for space
}

```

`ref readonly`를 반환하면 더 큰 구조체 복사본을 저장하고 내부 데이터 멤버의 불변성을 유지할 수 있습니다.

호출 사이트에서 호출자가 `Origin` 속성을 `ref readonly` 또는 값으로 사용하도록 선택할 수 있습니다.

```

var originValue = Point3D.Origin;
ref readonly var originReference = ref Point3D.Origin;

```

이전 코드의 첫 번째 할당에서는 `Origin` 상수의 복사본을 만들고 해당 복사본을 할당합니다. 두 번째 할당에서는 참조를 할당합니다. `readonly` 한정자는 변수 선언의 일부여야 합니다. 참조하는 항목에 대한 참조는 수정할 수 없습니다. 이를 수행하려고 시도하면 컴파일 시간 오류가 발생합니다.

`readonly` 한정자는 `originReference` 선언에 필요합니다.

컴파일러는 호출자가 참조를 수정할 수 없도록 합니다. 값을 직접 할당하려고 하면 컴파일 시간 오류가 발생합니다. 그러나 컴파일러는 멤버 메서드가 구조체의 상태를 수정하는지를 알 수 없습니다. 개체가 수정되지 않도록 하기 위해 컴파일러는 복사본을 만들고 해당 복사본을 사용하여 멤버 참조를 호출합니다. 모든 수정은 해당 방어 복사본에 대한 것입니다.

`in` 한정자를 `System.IntPtr.Size` 보다 큰 `readonly struct` 매개 변수에 적용

`in` 키워드는 기존 `ref` 및 `out` 키워드를 보완하여 참조로 인수를 전달합니다. `in` 키워드는 인수를 참조로 전달하도록 지정하지만 호출된 메서드는 값을 수정하지 않습니다.

이 추가는 설계 의도를 표현하기 위한 완벽한 어휘를 제공합니다. 메서드 서명에 다음 한정자 중 어떤 것도 지정하지 않으면 호출된 메서드에 전달될 때 값 형식이 복사됩니다. 이러한 각 한정자는 변수가 참조로 전달되도록 지정하여 복사를 방지합니다. 각 한정자는 각기 다른 의도를 표현합니다.

- `out` : 이 메서드는 이 매개 변수로 사용되는 인수의 값을 설정합니다.
- `ref` : 이 메서드는 이 매개 변수로 사용되는 인수의 값을 설정할 수 있습니다.
- `in` : 이 메서드는 이 매개 변수로 사용되는 인수의 값을 수정하지 않습니다.

참조로 인수를 전달하기 위해 `in` 한정자를 추가하고 불필요한 복사를 방지하기 위해 참조로 인수를 전달할 디자인 의도를 선언합니다. 해당 인수로 사용되는 개체를 수정할 의도가 없습니다.

이 방법은 종종 `IntPtr.Size`보다 큰 읽기 전용 값 형식에 대한 성능을 향상시킵니다. 단순 형식(`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool` 및 `enum` 형식)의 경우 모든 잠재적 성능 향상이 최소화됩니다. 사실 `IntPtr.Size`보다 작은 형식에 참조로 전달을 사용하면 성능이 저하될 수 있습니다.

다음 코드는 3D 공간에서 두 점 사이의 거리를 계산하는 메서드의 예를 보여 줍니다.

```
private static double CalculateDistance(in Point3D point1, in Point3D point2)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

인수는 각각 세 개의 `double`을 포함하는 두 개의 구조입니다. `double`은 8바이트이므로 각 인수는 24바이트입니다. `in` 한정자를 지정하여 머신 아키텍처에 따라 해당 인수에 4바이트 또는 8바이트 참조를 전달합니다. 크기의 차이는 작지만, 애플리케이션이 다양한 값을 사용하여 연속 루프에서 이 메서드를 호출하면 늘어납니다.

`in` 한정자는 `out` 및 `ref` 를 다른 방식으로도 보완합니다. `in`, `out` 또는 `ref` 가 있는 경우에만 다른 메서드의 오버로드는 만들 수 없습니다. 이러한 새 규칙은 항상 `out` 및 `ref` 매개 변수에 대해 정의된 같은 동작을 확장합니다. `out` 및 `ref` 한정자와 마찬가지로 `in` 한정자가 적용되므로 값 형식이 boxing되지 않습니다.

`in` 한정자는 메서드, 대리자, 람다, 로컬 함수, 인덱서, 연산자 매개 변수를 사용하는 모든 멤버에 적용될 수 있습니다.

`in` 매개 변수의 다른 기능은 `in` 매개 변수에 대한 인수에 리터럴 값 또는 상수를 사용하는 것입니다. 또한 `ref` 또는 `out` 매개 변수와 달리 호출 사이트에서 `in` 한정자를 적용할 필요가 없습니다. 다음 코드는 `CalculateDistance` 메서드를 호출하는 두 가지 예를 보여 줍니다. 첫 번째 예에서는 참조로 전달된 두 개의 로컬 변수를 사용합니다. 두 번째 예는 메서드 호출의 일부로 만들어진 임시 변수를 포함합니다.

```
var distance = CalculateDistance(pt1, pt2);
var fromOrigin = CalculateDistance(pt1, new Point3D());
```

컴파일러가 `in` 인수의 읽기 전용 특성을 강제 적용하는 여러 가지 방법이 있습니다. 먼저, 호출된 메서드는 `in` 매개 변수에 직접 할당할 수 없습니다. 값이 `struct` 형식일 때 `in` 매개 변수의 모든 필드에 직접 할당할 수 없습니다. 또한 `ref` 또는 `out` 한정자를 사용하여 모든 메서드에 `in` 매개 변수를 전달할 수 없습니다. 이러한 규칙은 필드가 `struct` 형식이고 매개 변수 또한 `struct` 형식인 경우 `in` 매개 변수의 모든 필드에 적용됩니다. 사실 이러한 규칙은 멤버 액세스의 모든 수준에서 형식이 `structs`인 경우 멤버 액세스의 여러 계층에 적용됩니다. 컴파일러는 `in` 인수로 전달된 `struct` 형식과 그 `struct` 멤버가 다른 메서드에 대한 인수로 사용될 경우 읽기 전용 변수가 되도록 합니다.

`in` 매개 변수를 사용하면 복사본 작성 시 발생할 수 있는 잠재적인 성능 비용을 방지할 수 있습니다. 어떠한 메서드 호출의 의미 체계도 변경되지 않습니다. 따라서 호출 사이트에서 `in` 한정자를 지정할 필요가 없습니다. 호출 사이트에서 `in` 한정자를 생략하면 다음과 같은 이유로 인수의 복사본을 만들 수 있음을 컴파일러에 알립니다.

- 암시적 변환은 있지만 인수 유형에서 매개 변수 유형으로의 ID 변환이 아닙니다.
- 인수는 식이지만 알려진 스토리지 변수는 없습니다.
- `in`의 유무의 따라 달라지는 오버로드가 있습니다. 이 경우에는 `by` 값 오버로드가 더 적합합니다.

이러한 규칙은 읽기 전용 참조 인수를 사용하도록 기존 코드를 업데이트할 때 유용합니다. 호출된 메서드 내에서 `by` 값 매개 변수를 사용하는 모든 인스턴스 메서드를 호출할 수 있습니다. 이러한 경우 `in` 매개 변수의 복사본이 만들어집니다. 컴파일러가 `in` 매개 변수에 대한 임시 변수를 만들 수 있으므로 사용자는 `in` 매개 변수에

대한 기본값을 지정할 수도 있습니다. 다음 코드에서는 원점(포인트 0,0)을 두 번째 점의 기본값으로 지정합니다.

```
private static double CalculateDistance2(in Point3D point1, in Point3D point2 = default)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

컴파일러가 읽기 전용 인수를 참조로 전달하도록 하려면 다음 코드에 나와 있는 것처럼 호출 사이트의 인수에 `in` 한정자를 지정합니다.

```
distance = CalculateDistance(in pt1, in pt2);
distance = CalculateDistance(in pt1, new Point3D());
distance = CalculateDistance(pt1, in Point3D.Origin);
```

이 동작을 통해 성능 향상이 가능한 대규모 코드 베이스에서 시간이 지남에 따라 `in` 매개 변수를 보다 쉽게 채택할 수 있습니다. 먼저 메서드 서명에 `in` 한정자를 추가합니다. 그런 다음, 호출 사이트에 `in` 한정자를 추가하고 `readonly struct` 형식을 생성하여 컴파일러가 더 많은 위치에서 `in` 매개 변수의 방어 복사본을 만들지 않도록 할 수 있습니다.

`in` 매개 변수 지정은 참조 형식 또는 숫자 값과 함께 사용될 수도 있습니다. 그러나 두 경우 모두의 이점은 있다고 하더라도 아주 적습니다.

## 변경 가능한 구조체를 `in` 인수로 사용하지 마세요.

앞에서 설명한 기술은 반환 참조에 의한 복사 및 참조로 값 전달을 방지하는 방법을 설명합니다. 이러한 기술은 인수 형식이 `readonly struct` 형식으로 선언되는 경우에 가장 적합합니다. 그렇지 않은 경우, 인수의 읽기 전용 특성을 적용하기 위해 많은 상황에서 컴파일러는 방어 복사본을 만들어야 합니다. 원점에서 3D 요소의 거리를 계산하는 다음과 같은 예제를 살펴보세요.

```
private static double CalculateDistance(in Point3D point1, in Point3D point2)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

`Point3D` 구조체는 읽기 전용 구조체가 아닙니다. 이 메서드의 본문에는 6개의 서로 다른 속성 액세스 호출이 있습니다. 첫 번째 검사에서 이러한 액세스가 안전하다고 생각했을 것입니다. 결국 `get` 접근자는 개체의 상태를 수정하면 안됩니다. 하지만 이를 적용하는 언어 규칙이 없습니다. 일반적인 관습일 뿐입니다. 모든 형식은 내부 상태를 수정한 `get` 접근자를 구현할 수 있습니다. 일부 언어 보장을 사용하지 않는 경우 컴파일러는 `readonly` 한정자를 사용하여 표시되지 않은 모든 멤버를 호출하기 전에 인수의 임시 복사본을 만들어야 합니다. 임시 스토리지가 스택에 만들어지고, 인수 값이 임시 스토리지에 복사되며, 값이 `this` 인수로 각 멤버 액세스에 대한 스택으로 복사됩니다. 다양한 상황에서 이러한 복사는 인수 형식이 `readonly struct`가 아니고 메서드가 `readonly`로 표시되지 않은 멤버를 호출하는 경우 값으로 전달이 읽기 전용 참조로 전달보다 더 빠를 정도로 성능을 저하시킵니다. 구조체 상태를 수정하지 않는 모든 메서드를 `readonly`로 표시할 경우 컴파일러는 구조체 상태가 수정되지 않고 방어형 복사본이 필요하지 않음을 안전하게 확인할 수 있습니다.

대신, 거리 계산에서 변경이 불가능한 구조체인 `ReadOnlyPoint3D`를 사용하는 경우에는 임시 개체가 필요하지

않습니다.

```
private static double CalculateDistance3(in ReadonlyPoint3D point1, in ReadonlyPoint3D point2 = default)
{
    double xDifference = point1.X - point2.X;
    double yDifference = point1.Y - point2.Y;
    double zDifference = point1.Z - point2.Z;

    return Math.Sqrt(xDifference * xDifference + yDifference * yDifference + zDifference * zDifference);
}
```

컴파일러가 `readonly struct`의 멤버를 호출할 때 더 효율적인 코드를 생성합니다. 수신기의 복사본 대신 `this` 참조는 항상 멤버 메서드에 대한 참조로 전달된 `in` 매개 변수입니다. 이 최적화는 `readonly struct`를 `in` 인수로 사용하는 경우 복사본을 저장합니다.

null 허용 값 형식은 `in` 인수로 전달할 수 없습니다. `Nullable<T>` 형식은 읽기 전용 구조체로 선언되지 않습니다. 즉 컴파일러가 매개 변수 선언에서 `in` 수정자를 사용하여 메서드에 전달된 모든 null 허용 값 형식 인수에 대해 방어용 복사본을 생성해야 합니다.

GitHub에 있는 [샘플 리포지토리](#)에서 `BenchmarkDotNet`을 사용하여 성능 차이를 설명하는 예제 프로그램을 확인할 수 있습니다. 값으로 및 참조로 변경할 수 있는 구조체를 전달하는 것과 값으로 및 참조로 변경할 수 없는 구조체를 전달하는 것을 비교합니다. 변경할 수 없는 구조체를 사용하고 참조로 전달하는 것이 가장 빠릅니다.

단일 스택 프레임의 블록 또는 메모리를 작업하기 위해 `ref struct` 형식을 사용합니다.

관련 언어 기능은 단일 스택 프레임에 제한되어야 하는 값 형식을 선언하는 기능입니다. 이 제한을 통해 컴파일러는 몇 가지 최적화를 수행할 수 있습니다. 이 기능의 기본 동기 부여는 `Span<T>` 및 관련 구조였습니다.

`Span<T>` 유형을 사용하는 새롭고 업데이트된 .NET API를 사용함으로써 이러한 향상된 기능으로부터 성능 향상을 달성할 수 있습니다.

`stackalloc`을 사용하여 만들어진 메모리로 작업할 때나 interop API에서 메모리를 사용할 때도 유사한 요구 사항이 있을 수 있습니다. 해당 요구 사항에 대한 사용자 고유의 `ref struct` 형식을 정의할 수 있습니다.

## `readonly ref struct` 형식

구조체를 `readonly ref`로 선언하면 `ref struct` 및 `readonly struct` 선언의 이점과 제한이 결합됩니다. 읽기 전용 범위에서 사용된 메모리는 단일 스택 프레임으로 제한되며 읽기 전용 범위에서 사용된 메모리는 수정할 수 없습니다.

## 결론

값 형식을 사용하면 할당 작업 수가 최소화됩니다.

- 값 형식에 대한 스토리지는 로컬 변수와 메서드 인수에 스택 할당됩니다.
- 다른 개체의 멤버인 값 형식에 대한 스토리지는 별도 할당이 아니라 해당 개체의 일부로 할당됩니다.
- 값 형식 반환 값에 대한 스토리지는 스택 할당됩니다.

그러한 동일한 상황에서 참조 형식과 대조를 보입니다.

- 참조 형식에 대한 스토리지는 로컬 변수 및 메서드 인수에 힙 할당됩니다. 참조는 스택에 저장됩니다.
- 다른 개체의 멤버인 참조 형식에 대한 스토리지는 별도로 힙에 할당됩니다. 포함하는 개체는 참조를 저장합니다.
- 참조 형식 반환 값에 대한 스토리지는 힙 할당됩니다. 해당 스토리지에 대한 참조는 스택에 저장됩니다.

할당을 최소화하는 작업에는 장단점이 있습니다. `struct`의 크기가 참조의 크기보다 큰 경우 더 많은 메모리를 복사할 수 있습니다. 참조는 일반적으로 64비트 또는 32비트이며 대상 머신 CPU에 따라 달라집니다.

이러한 장단점은 일반적으로 성능에 거의 영향을 주지 않습니다. 그러나 구조체 또는 컬렉션이 더 큰 경우 성능에 미치는 영향은 증가합니다. 연속 루프 및 과다 경로에서는 프로그램에 대한 영향이 더 커질 수 있습니다.

C# 언어에 대한 이러한 향상된 기능은 메모리 할당을 최소화하는 것이 필요한 성능을 달성하는데 중요한 요인이 되는 성능 중요 알고리즘을 위해 설계되었습니다. 작성하는 코드에서 이러한 기능을 자주 사용하지 않을 수 있습니다. 그러나 이러한 향상된 기능은 .NET 전반에서 채택되었습니다. 점점 더 많은 API에서 이러한 기능을 사용하므로 사용자의 애플리케이션 성능이 개선되는 것을 확인할 수 있습니다.

## 참조

- [ref 키워드](#)
- [Ref return 및 ref local](#)

# 식 트리

2020-11-02 • 6 minutes to read • [Edit Online](#)

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.

보다 풍부한 조작을 원하는 경우 [식 트리](#)를 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 작성할 수 있습니다. 보다 고급 시나리오에서는 실행 중인 알고리즘을 수정하고 C# 식을 다른 형태로 변환하여 다른 환경에서 실행할 수도 있습니다.

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.

이 자습서의 나머지 섹션에서는 식 트리의 정의를 살펴보고, 식 트리를 지원하는 프레임워크 클래스를 검사하며, 식 트리로 작업하는 방법을 보여 줍니다. 식 트리를 읽는 방법, 식 트리를 만드는 방법, 수정된 식 트리를 만드는 방법, 식 트리로 표시되는 코드를 실행하는 방법 등을 알아봅니다. 자습서를 읽고 나면 이러한 구조를 사용하여 풍부한 적응 알고리즘을 만들 수 있습니다.

## 1. 식 트리 설명

식 트리의 구조와 개념을 이해합니다.

## 2. 식 트리를 지원하는 프레임워크 형식

식 트리를 정의하고 조작하는 구조 및 클래스에 대해 알아봅니다.

## 3. 식 실행

람다 식으로 표시되는 식 트리를 대리자로 변환하고 결과 대리자를 실행하는 방법을 알아봅니다.

## 4. 식 해석

식 트리를 트래버스하고 검사하여 식 트리가 나타내는 코드를 이해하는 방법을 알아봅니다.

## 5. 식 작성

식 트리에 대한 노드를 생성하고 식 트리를 작성하는 방법을 알아봅니다.

## 6. 식 변환

식 트리의 수정된 복사본을 작성하거나 식 트리를 다른 형식으로 변환하는 방법을 알아봅니다.

## 7. 요약

식 트리에 대한 정보를 검토합니다.

# 식 트리 설명

2020-03-18 • 13 minutes to read • [Edit Online](#)

## 이전 -- 개요

식 트리는 코드를 정의하는 데이터 구조이며, 컴파일러에서 코드를 분석하고 컴파일된 출력을 생성하는 데 사용하는 것과 동일한 구조를 기반으로 합니다. 이 자습서를 읽다 보면 식 트리와 Roslyn API에서 [Analyzers and CodeFixes](#)(분석기 및 CodeFix)를 빌드하는 데 사용된 형식 사이에 상당히 많은 유사성이 있음을 확인할 수 있습니다. 분석기 및 CodeFix는 코드에 대해 정적 분석을 수행하고 개발자에게 잠재적 해결 방법을 제안할 수 있는 NuGet 패키지입니다. 개념이 유사하며 최종 결과는 의미 있는 방식으로 소스 코드를 검사할 수 있는 데이터 구조입니다. 그러나 식 트리는 Roslyn API와 완전히 다른 클래스 및 API 집합을 기반으로 합니다.

간단한 예제를 살펴보겠습니다. 코드 줄은 다음과 같습니다.

```
var sum = 1 + 2;
```

이 코드를 식 트리로 분석하려는 경우 트리에 여러 개의 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 `1 + 2`이며, 이진 식입니다. 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`)

다음 코드는 복잡해 보이지만 매우 강력합니다. 동일한 프로세스에 따라 훨씬 더 복잡한 식을 분해할 수 있습니다. 다음 식을 살펴보세요.

```
var finalAnswer = this.SecretSauceFunction(
    currentState.createInterimResult(), currentState.createSecondValue(1, 2),
    decisionServer.considerFinalOptions("hello")) +
    MoreSecretSauce('A', DateTime.Now, true);
```

위의 식은 할당을 사용하는 변수 선언이기도 합니다. 이 경우 할당의 오른쪽이 훨씬 더 복잡한 트리입니다. 이 식은 분해하지 않고 서로 다른 노드 부분을 살펴보겠습니다. 현재 개체를 수신기로 사용하는 메서드 호출, 명시적 `this` 수신기가 있는 호출과 그렇지 않은 호출이 있습니다. 다른 수신기 개체를 사용하는 메서드 호출이 있

고 다양한 형식의 상수 인수가 있습니다. 마지막으로 이진 더하기 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 복잡성에도 불구하고 위의 식은 첫 번째 샘플만큼 쉽게 탐색할 수 있는 트리 구조를 만듭니다. 식에서 자식 노드를 계속 트래버스하여 리프 노드를 찾을 수 있습니다. 부모 노드에는 자식에 대한 참조가 있으며 각 노드에는 노드의 종류를 설명하는 속성이 있습니다.

식 트리의 구조는 거의 일치합니다. 기본 사항을 익혔으면 식 트리로 표현될 때 가장 복잡한 코드도 이해할 수 있습니다. 간결한 데이터 구조는 C# 컴파일러에서 가장 복잡한 C# 프로그램을 분석하고 복잡한 해당 소스 코드에서 적절한 출력을 만드는 방법을 설명합니다.

식 트리의 구조에 익숙해지고 나면 신속하게 습득한 지식을 통해 훨씬 더 많은 고급 시나리오를 사용할 수 있습니다. 식 트리에는 정말 강력한 기능이 있습니다.

알고리즘을 변환하여 다른 환경에서 실행하는 것 외에 식 트리를 사용하면 코드를 실행하기 전에 검사하는 알고리즘을 더 쉽게 작성할 수 있습니다. 인수가 식인 메서드를 작성한 다음 코드를 실행하기 전에 해당 식을 검사할 수 있습니다. 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. 또한 식 트리를 실행 가능한 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 사용하기 보다는 `Task` 객체를 직접 조작해야 합니다. 또 다른 예로 루프 만들기가 있습니다. 일반적으로 `for`, `foreach`, `while` 또는 `do` 루프를 사용하여 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.

[다음 -- 식 트리를 지원하는 프레임워크 형식](#)

# 식 트리를 지원하는 프레임워크 형식

2020-11-02 • 8 minutes to read • [Edit Online](#)

## 이전 -- 식 트리 설명

.NET Core 프레임워크에는 식 트리에서 작동하는 클래스 목록이 많습니다. 전체 목록은 [System.Linq.Expressions](#)에서 확인할 수 있습니다. 전체 목록을 실행하는 대신 프레임워크 클래스가 어떻게 디자인되었는지 살펴보겠습니다.

언어 디자인에서 식이란 값을 계산하고 반환하는 코드의 본문입니다. 식은 매우 간단할 수 있습니다. 상수 식 1은 상수 값 1을 반환합니다. 식은 더 복잡할 수도 있습니다.  $(-B + \sqrt{B^2 - 4 * A * C}) / (2 * A)$  식은 2차 방정식의 근을 반환합니다(방정식에 솔루션이 있는 경우).

## 모두 System.Linq.Expression으로 시작

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 것이 가장 간단합니다. 그러나 더 많은 것을 알아야 하는 경우가 있습니다. 기본 식 클래스에는 `NodeType` 속성이 이러한 용도로 포함되어 있으며, 가능한 식 형식의 열거형인 `ExpressionType`을 반환합니다. 노드의 형식을 알고 있다면 해당 형식으로 캐스트할 수 있으며 식 노드의 형식을 알고 있으면 특정 작업을 수행할 수 있습니다. 특정 노드 유형을 검색한 다음 이러한 식의 특정 속성을 사용할 수 있습니다.

예를 들어 다음 코드는 변수 액세스 식에서 변수의 이름을 인쇄합니다. 노드 유형을 확인하고 변수 액세스 식에 캐스트한 다음 특정 식 형식의 속성을 확인하는 방법을 따릅니다.

```
Expression<Func<int, int>> addFive = (num) => num + 5;

if (addFive.NodeType == ExpressionType.Lambda)
{
    var lambdaExp = (LambdaExpression)addFive;

    var parameter = lambdaExp.Parameters.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 위로 식을 작성합니다. 예를 들어 다음 코드는 더하기 식을 작성합니다.

```
// 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. 다음은 모든 사항을 기억하는 대신 식 트리로 작업할 때 사용할 수 있는 기술입니다.

1. `ExpressionType` 열거형의 멤버를 확인하여 검색할 수 있는 노드를 결정합니다. 실제로 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.

세 영역을 각각 확인하면 더 자세히 알아볼 수 있습니다. 이러한 세 단계 중 한 단계로 시작하면 반드시 필요한 항목을 확인할 수 있습니다.

다음 -- 식 트리 실행

# 식 트리 실행

2020-03-18 • 16 minutes to read • [Edit Online](#)

## 이전 -- 식 트리를 지원하는 프레임워크 형식

식 트리는 일부 코드를 나타내는 데이터 구조입니다. 컴파일되고 실행 가능한 코드가 아닙니다. 식 트리로 표시되는 .NET 코드를 실행하려면 실행 가능한 IL 명령으로 변환해야 합니다.

## 람다 식을 함수로 변환

모든 LambdaExpression 또는 LambdaExpression에서 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. ConstantExpression 을 직접 실행하는 것의 의미를 생각해 보세요. 유용한 의미가 있나요? LambdaExpression 이거나 LambdaExpression에서 파생된 형식인 모든 식 트리는 IL로 변환할 수 있습니다. 식 형식 Expression<TDelegate> 는 .NET Core 라이브러리에서 유일하게 구체적인 예제입니다. 이 형식은 모든 대리자 형식에 매핑되는 식을 나타내는데 사용됩니다. 이 형식은 대리자 형식에 매핑되므로 .NET에서 식을 검사하고 람다 식의 시그니처와 일치하는 적절한 대리자에 대해 IL을 생성할 수 있습니다.

대부분의 경우 식과 해당 대리자 간에 간단한 매핑이 만들어집니다. 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.

LambdaExpression 형식에는 식 트리를 실행 코드로 변환하는 데 사용되는 Compile 및 CompileToMethod 멤버가 포함됩니다. Compile 메서드는 대리자를 만듭니다. CompileToMethod 메서드는 식 트리의 컴파일된 출력을 나타내는 IL로 MethodBuilder 개체를 업데이트합니다. CompileToMethod 는 .NET Core가 아니라 전체 데스크톱 프레임워크에서 사용할 수 있습니다.

필요에 따라 생성된 대리자 개체에 대한 기호 디버깅 정보를 수신하는 DebugInfoGenerator 를 제공할 수도 있습니다. 그러면 식 트리를 대리자 개체로 변환하고 생성된 대리자에 대한 전체 디버깅 정보를 받을 수 있습니다.

다음 코드를 사용하여 식을 대리자로 변환합니다.

```
Expression<Func<int>> add = () => 1 + 2;
var func = add.Compile(); // Create Delegate
var answer = func(); // Invoke Delegate
Console.WriteLine(answer);
```

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 반환합니다. 컴파일 시간 도구에서 인수 목록 또는 반환 형식을 확인할 수 있도록 하려면 올바른 대리자 형식으로 캐스팅해야 합니다.

## 실행 및 수명

LambdaExpression.Compile() 을 호출할 때 만든 대리자를 호출하여 코드를 실행합니다. 위의 add.Compile() 이 대리자를 반환하는 위치에서 이를 확인할 수 있습니다. 해당 대리자를 호출하고 func() 를 호출하면 코드가 실행됩니다.

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.

필요 없는 컴파일 호출을 방지하여 성능을 향상시키려면 보다 정교한 캐싱 메커니즘을 만들려고 해야 합니다. 임의의 식 트리 두 개를 비교하여 동일한 알고리즘을 나타내는지 확인하는 경우에도 실행하는데 시간이 오래 걸릴 수 있습니다. `LambdaExpression.Compile()`의 추가 호출을 방지하여 절약한 컴퓨팅 시간이 동일한 실행 코드에서 서로 다른 두 식 트리 결과를 확인하는 코드 실행 시간을 초과할 수 있습니다.

## 주의 사항

람다 식을 대리자로 컴파일하고 해당 대리자를 호출하는 것은 식 트리로 수행할 수 있는 가장 간단한 작업 중 하나입니다. 그러나 이 간단한 작업에서도 주의해야 할 사항이 있습니다.

람다 식은 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.

일반적으로 컴파일러는 이렇게 되도록 합니다. 그러나 식이 `IDisposable`을 구현하는 변수에 액세스하는 경우 코드는 식 트리에서 보유한 객체를 삭제할 수 있습니다.

예를 들어 다음 코드는 `int` 가 `IDisposable`을 구현하지 않기 때문에 제대로 작동합니다.

```
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 `IDisposable`을 구현하는 다음(다소 인위적임) 클래스를 살펴보세요.

```
public class Resource : IDisposable
{
    private bool isDisposed = false;
    public int Argument
    {
        get
        {
            if (!isDisposed)
                return 5;
            else throw new ObjectDisposedException("Resource");
        }
    }

    public void Dispose()
    {
        isDisposed = true;
    }
}
```

아래와 같이 식에서 사용하는 경우 `Resource.Argument` 속성에서 참조하는 코드를 실행하면 `ObjectDisposedException`이 표시됩니다.

```
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 많은 순열이 있으므로 이 문제를 방지하는 일반적인 지침을 제공하기가 어렵습니다. 식을 정의하는 경우 지역 변수에 액세스할 때 주의해야 하며 공용 API에서 반환할 수 있는 식 트리를 만드는 경우 현재 개체(`this`로 표시됨)의 상태에 액세스할 때도 주의해야 합니다.

식의 코드는 다른 어셈블리의 메서드나 속성을 참조할 수 있습니다. 해당 어셈블리는 식을 정의할 때, 식을 컴파일할 때 및 결과 대리자를 호출할 때 액세스할 수 있어야 합니다. 어셈블리가 없는 경우 `ReferencedAssemblyNotFoundException`이 발생하게 됩니다.

## 요약

람다 식을 나타내는 식 트리를 컴파일하면 실행할 수 있는 대리자를 만들 수 있습니다. 그러면 식 트리로 표시되는 코드를 실행하는 메커니즘이 제공됩니다.

식 트리는 생성되는 특정 구문에 대해 실행되는 코드를 나타냅니다. 코드를 컴파일하고 실행하는 환경이 식을 만드는 환경과 일치하는 경우 모든 작업이 예상대로 작동합니다. 그렇지 않을 경우 오류를 예측할 수 있으며 식 트리를 사용하여 코드를 처음 테스트할 때 `catch`됩니다.

[다음 -- 식 해석](#)

# 식 해석

2020-11-02 • 26 minutes to read • [Edit Online](#)

## 이전 -- 식 실행

이제 식 트리의 구조를 검사하는 몇 가지 코드를 작성해 보겠습니다. 식 트리의 모든 노드는 `Expression`에서 파생되는 클래스의 개체입니다.

해당 디자인은 식 트리의 모든 노드 방문을 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.

```
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}");
```

다음과 같이 출력됩니다.

```
This is 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. 해당 코드는 다음과 같습니다.

## 간단한 더하기 식 검사

이 섹션의 소개 부분에 있는 더하기 샘플로 시작해 보겠습니다.

```
Expression<Func<int>> sum = () => 1 + 2;
```

`var`을 사용하여 이 식 트리를 선언하지 않습니다. 왜냐하면 할당의 오른쪽이 암시적으로 형식화되어 있기 때문에 그렇게 할 수 없습니다.

루트 노드는 `LambdaExpression`입니다. `=>` 연산자의 오른쪽에서 흥미로운 코드를 가져오려면 `LambdaExpression`의 자식 중 하나를 찾아야 합니다. 이 섹션에서는 모든 식을 사용하여 찾습니다. 부모 노드는 `LambdaExpression`의 반환 형식을 찾는데 도움이 됩니다.

이 식의 각 노드를 검사하려면 많은 노드를 재귀적으로 방문해야 합니다. 다음은 간단한 첫 번째 구현입니다.

```

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($"{'\t'}Parameter 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($"{'\t'}Parameter Type: {left.Type.ToString()}, Name: {left.Name}");
Console.WriteLine($"The right side is a {additionBody.Right.NodeType} expression");
var right= (ParameterExpression)additionBody.Right;
Console.WriteLine($"{'\t'}Parameter Type: {right.Type.ToString()}, Name: {right.Name}");

```

이 샘플은 다음과 같이 출력됩니다.

```

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

```

위의 코드 샘플에서 많은 반복을 확인할 수 있습니다. 이러한 반복을 정리하고 보다 일반적인 용도의 식 노드 문자를 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.

```

// 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 => this.node.NodeType;
    public static Visitor CreateFromExpression(Expression node)
    {
        switch(node.NodeType)
        {
            case ExpressionType.Constant:
                return new ConstantVisitor((ConstantExpression)node);
            case ExpressionType.Lambda:
                return new LambdaVisitor((LambdaExpression)node);
            case ExpressionType.Parameter:
                return new ParameterVisitor((ParameterExpression)node);
        }
    }
}

```

```

        case ExpressionType.Add:
            return new BinaryVisitor((BinaryExpression)node);
        default:
            Console.Error.WriteLine($"Node not processed yet: {node.NodeType}");
            return default(Visitor);
    }
}

// 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($"{prefix}The return type is {node.ReturnType.ToString()}");
        Console.WriteLine($"{prefix}The expression has {node.Parameters.Count} argument(s). They are:");
        // Visit each parameter:
        foreach (var argumentExpression in node.Parameters)
        {
            var argumentVisitor = Visitor.CreateFromExpression(argumentExpression);
            argumentVisitor.Visit(prefix + "\t");
        }
        Console.WriteLine($"{prefix}The expression body is:");
        // Visit the body:
        var bodyVisitor = Visitor.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($"{prefix}This binary expression is a {NodeType} expression");
        var left = Visitor.CreateFromExpression(node.Left);
        Console.WriteLine($"{prefix}The Left argument is:");
        left.Visit(prefix + "\t");
        var right = Visitor.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.ToString()}, 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 방문할 수 있는 알고리즘의 기반입니다. 많은 허점이 있습니다. 즉, 만들어진 코드는 가능한 식 트리 노드 집합의 매우 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.

위에 표시된 더하기 식에서 이 방문자를 실행하면 다음과 같이 출력됩니다.

```

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.

## 많은 수준으로 더하기 식 검사

더 복잡한 예제를 시도해 보겠지만 노드 형식은 여전히 더하기로만 제한합니다.

```
Expression<Func<int>> sum = () => 1 + 2 + 3 + 4;
```

방문자 알고리즘에서 실행하기 전에 사고 연습을 통해 출력되는 사항을 파악합니다. `+` 연산자는 *이*진 연산자임을 기억하세요. 이 연산자에는 왼쪽과 오른쪽 피연산자를 나타내는 두 개의 자식이 있어야 합니다. 여러 가지 방법으로 올바른 트리를 생성할 수 있습니다.

```
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 결합성식을 나타내고, 두 번째는 왼쪽 결합형식을 나타냅니다. 이러한 두 형식의 장점은 형식이 임의 개수의 더하기 식으로 확장된다는 점입니다.

방문자를 통해 이 식을 실행하면 이 출력을 통해 간단한 더하기 식이 왼쪽 결합형임을 확인할 수 있습니다.

이 샘플을 실행하고 전체 식 트리를 보기 위해 소스 식 트리에서 한 가지를 변경해야 했습니다. 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.

```
Expression<Func<int, int>> sum = (a) => 1 + a + 3 + 4;
```

이 합계에 대한 방문자를 만들고 방문자를 실행하면 다음 출력이 표시됩니다.

```
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 코드가 나타내는 트리를 확인할 수도 있습니다. 다음은 컴파일러가 상수를 계산할 수 없도록 하는 추가 매개 변수가 있는 위의 sum3 식에 대한 예제입니다.

```
Expression<Func<int, int, int>> sum3 = (a, b) => (1 + a) + (3 + b);
```

방문자의 출력은 다음과 같습니다.

```

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 샘플에서 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 코드에 대해 작동하도록 만들어 보겠습니다.

```

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. 이 코드를 작성한 방법에서는 식에 람다 식을 할당하여 식 트리를 작성할 때의 두 가지 제한을 강조합니다. 첫째, 문 람다는 허용되지 않습니다. 즉, 루프, 블록, if/else 문 및 C#에서 일반적인 다른 제어 구조를 사용할 수 없습니다. 식 사용으로 제한됩니다. 둘째, 같은 식을 재귀적으로 호출할 수 없습니다. 이미 대리자인 경우에는 가능하지만 식 트리 형식으로는 호출할 수 없습니다. [식 트리 작성](#)에 대한 섹션에서는 이러한 제한 사항을 해결하는 기술을 알아봅니다.

이 식에서는 다음과 같은 형식의 노드가 모두 나타납니다.

1. 같음(이진 식)
2. 곱하기(이진 식)
3. 조건식(: 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 반복하면 각각의 잠재적 노드를 확인하게 됩니다. 그러면 다 끝났습니다. 결과는 다음과 같이 나타납니다.

```
public static Visitor CreateFromExpression(Expression node)
{
    switch(node.NodeType)
    {
        case ExpressionType.Constant:
            return new ConstantVisitor((ConstantExpression)node);
        case ExpressionType.Lambda:
            return new LambdaVisitor((LambdaExpression)node);
        case ExpressionType.Parameter:
            return new ParameterVisitor((ParameterExpression)node);
        case ExpressionType.Add:
        case ExpressionType.Equal:
        case ExpressionType.Multiply:
            return new BinaryVisitor((BinaryExpression)node);
        case ExpressionType.Conditional:
            return new ConditionalVisitor((ConditionalExpression)node);
        case ExpressionType.Call:
            return new MethodCallVisitor((MethodCallExpression)node);
        default:
            Console.Error.WriteLine($"Node not processed yet: {node.NodeType}");
            return default(Visitor);
    }
}
```

ConditionalVisitor 및 MethodCallVisitor는 해당 두 노드를 처리합니다.

```

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.

```

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 트리의 노드를 방문하고 액세스하는 핵심 작업에 집중하기 위해 필요할 수 있는 많은 작업에 주석을 달았습니다.

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. 이 예제가 실패하도록 하는 많은 식을 제공할 수도 있습니다. 전체 구현은 [ExpressionVisitor](#)라는 이름으로 .NET Standard에 포함되며 가능한 노드 형식을 모두 처리할 수 있습니다.

마지막으로 이 문서에서 사용된 라이브러리는 데모 및 학습용으로 작성되었으며, 최적화되지 않았습니다. 사용된 구조를 명확하게 하고 노드를 방문하여 포함된 항목을 분석하는데 사용된 기술을 강조하기 위해 작성되었습니다. 프로덕션 구현에서는 지금까지보다 성능에 더 많은 주의를 기울입니다.

이러한 제한 사항에도 불구하고 식 트리를 읽고 이해하는 알고리즘 작성은 완료해야 합니다.

[다음 -- 식 작성](#)

# 식 트리 작성

2020-11-02 • 13 minutes to read • [Edit Online](#)

## 이전 - 식 해석

지금까지 살펴본 모든 식 트리는 C# 컴파일러에서 만들어졌습니다. `Expression<Func<T>>` 또는 유사한 형식의 변수 형식에 할당된 람다 식을 만들기만 하면 됐습니다. 그러나 식 트리를 만드는 유일한 방법은 아닙니다. 대부분의 시나리오에서는 런타임에 메모리에서 식을 작성해야 함을 알 수 있습니다.

식 트리는 변경할 수 없다는 사실로 인해 식 트리 작성은 복잡합니다. 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

비교적 간단하게 다시 시작해 보겠습니다. 이러한 섹션 전체에서 사용한 더하기 식을 사용하겠습니다.

```
Expression<Func<int>> sum = () => 1 + 2;
```

해당 식 트리를 생성하려면 리프 노드를 생성해야 합니다. 리프 노드는 상수이므로 `Expression.Constant` 메서드를 사용하여 노드를 만들 수 있습니다.

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
```

다음으로 더하기 식을 작성해 보겠습니다.

```
var addition = Expression.Add(one, two);
```

더하기 식을 작성했으면 람다 식을 만들 수 있습니다.

```
var lambda = Expression.Lambda(addition);
```

이 식은 인수가 없기 때문에 매우 간단한 람다 식입니다. 이 섹션의 뒷부분에서는 인수를 매개 변수에 매핑하고 더 복잡한 식을 작성하는 방법을 살펴봅니다.

이 식처럼 간단한 식의 경우 모든 호출을 단일 문으로 결합할 수 있습니다.

```
var lambda = Expression.Lambda(
    Expression.Add(
        Expression.Constant(1, typeof(int)),
        Expression.Constant(2, typeof(int))
    )
);
```

## 트리 작성

메모리에서 식 트리를 작성할 때의 기본 사항입니다. 좀 더 복잡한 트리는 일반적으로 노드 유형과 트리의 노드

가 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.

```
Expression<Func<double, double, double>> distanceCalc =  
    (x, y) => Math.Sqrt(x * x + y * y);
```

`x` 및 `y`에 대한 매개 변수 식부터 만듭니다.

```
var xParameter = Expression.Parameter(typeof(double), "x");  
var yParameter = Expression.Parameter(typeof(double), "y");
```

곱하기와 더하기 식을 만들 때 이미 살펴본 패턴을 따릅니다.

```
var xSquared = Expression.Multiply(xParameter, xParameter);  
var ySquared = Expression.Multiply(yParameter, yParameter);  
var sum = Expression.Add(xSquared, ySquared);
```

다음으로 `Math.Sqrt`를 호출하기 위한 메서드 호출 식을 만들어야 합니다.

```
var sqrtMethod = typeof(Math).GetMethod("Sqrt", new[] { typeof(double) });  
var distance = Expression.Call(sqrtMethod, sum);
```

그런 다음 마지막으로 메서드 호출을 람다 식에 넣고 람다 식에 대한 인수를 정의해야 합니다.

```
var distanceLambda = Expression.Lambda(  
    distance,  
    xParameter,  
    yParameter);
```

더 복잡한 이 예제에서는 식 트리를 만드는데 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 `MethodInfo` 개체를 만들어야 합니다. .NET Core 플랫폼에서 사용할 수 있는 리플렉션 API의 하위 집합으로 제한해야 합니다. 이제 이러한 기술은 다른 식 트리로 확장됩니다.

## 코드 빌드에 대한 자세한 설명

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.

```

Func<int, int> factorialFunc = (n) =>
{
    var res = 1;
    while (n > 1)
    {
        res = res * n;
        n--;
    }
    return res;
};

```

위의 예제에서는 식 트리를 작성하지 않고 대리자만 작성했습니다. `Expression` 클래스를 사용하여 문 람다를 빌드할 수 없습니다. 다음은 동일한 기능을 빌드하는 데 필요한 코드입니다. `while` 루프를 빌드하는 API가 없기 때문에 복잡합니다. 대신 루프를 중단하려면 조건부 테스트를 포함하는 루프와 레이블 대상을 빌드해야 합니다.

```

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 방문하는 방문자 코드도 업데이트하고 이 샘플에서 만들어진 노드에 대한 정보를 기록했습니다. GitHub의 `dotnet/docs` 리포지토리에서 [샘플 코드를 보거나 다운로드](#)할 수 있습니다. 샘플을 빌드하고 실행하여 직접 실험합니다. 다운로드 지침은 [샘플 및 자습서](#)를 참조하세요.

## API 검사

식 트리 API는 .NET Core에서 탐색하기가 더 어렵지만 괜찮습니다. 용도는 런타임에 코드를 생성하는 코드를 작성하는 경우처럼 다소 복잡합니다. C# 언어에서 사용할 수 있는 모든 제어 구조를 지원하고 API의 노출 영역을 적절히 작게 유지하는 작업 간에 균형을 맞추려면 복잡할 수 밖에 없습니다. 이러한 균형은 많은 제어 구조가 C# 구문이 아니라 컴파일러가 더 높은 수준의 구조체에서 생성하는 기본 논리를 나타내는 구조체에 의해 표시됨을 의미합니다.

또한 `Expression` 클래스 메서드를 사용하여 직접 작성할 수 없는 C# 식도 있습니다. 일반적으로 C# 5 및 C# 6에서 추가된 최신 연산자 및 식이 있습니다. 예를 들어 `async` 식을 작성할 수 없으며 새로운 `?.` 연산자를 직접 만들 수 없습니다.

[다음 -- 식 변환](#)

# 식 트리 변환

2020-11-02 • 14 minutes to read • [Edit Online](#)

## 이전 -- 식 작성

이 최종 섹션에서는 식 트리의 각 노드를 방문하고 해당 식 트리의 수정된 복사본을 작성하는 방법을 알아봅니다. 이러한 기술은 두 가지 중요한 시나리오에서 사용됩니다. 첫 번째는 다른 환경으로 변환할 수 있도록 식 트리로 표현되는 알고리즘을 이해하려는 경우입니다. 두 번째는 생성된 알고리즘을 변경하려는 경우입니다. 이 경우는 로깅 추가, 메서드 호출 가로채기 및 추적 또는 다른 목적일 수 있습니다.

## 변환은 방문임

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 방문하고 몇 가지 대체 노드로 새 트리를 만들어 실제로 살펴보겠습니다. 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 대체하겠습니다.

여기에서 상수 노드를 찾은 다음에는 자식이 원래 상수 및 상수 `10`인 새 곱하기 노드를 만듭니다.

```
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 바꾸면 수정 사항을 포함하는 새 트리가 형성됩니다. 대체된 트리를 컴파일하고 실행하여 이를 확인할 수 있습니다.

```
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.

이 예제에서는 변경할 수 없는 식 트리의 중요도를 보여 줍니다. 위에서 만든 새 트리에는 새로 만든 노드와 기존 트리의 노드가 함께 포함됩니다. 기존 트리의 노드를 수정할 수 없기 때문에 안전합니다. 이렇게 하면 중요한

메모리 효율성이 발생할 수 있습니다. 트리 전체 또는 여러 식 트리에서 같은 노드를 사용할 수 있습니다. 노드는 수정할 수 없기 때문에 필요할 때마다 같은 노드를 다시 사용할 수 있습니다.

## 더하기 트래버스 및 실행

더하기 노드 트리를 이동하고 결과를 계산하는 두 번째 방문자를 작성하여 확인해 보겠습니다. 이렇게 하려면 지금까지 살펴본 방문자를 여러 번 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.

```
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 we 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 아주 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.

```

private static int Aggregate(Expression exp)
{
    if (exp.NodeType == ExpressionType.Constant)
    {
        var constantExp = (ConstantExpression)exp;
        Console.Error.WriteLine($"Found Constant: {constantExp.Value}");
        return (int)constantExp.Value;
    }
    else if (exp.NodeType == ExpressionType.Add)
    {
        var addExp = (BinaryExpression)exp;
        Console.Error.WriteLine("Found Addition Expression");
        Console.Error.WriteLine("Computing Left node");
        var leftOperand = Aggregate(addExp.Left);
        Console.Error.WriteLine($"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");
}

```

같은 식에서 이 메서드를 실행하면 다음과 같이 출력됩니다.

```

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 위의 코드도 함께 수행합니다. 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.

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
```

이 식을 검사한 출력은 다음과 같습니다.

```
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 동일하지만 트리 통과는 완전히 다릅니다. 먼저 발생하는 다른 작업으로 트리가 생성되었기 때문에 다른 순서로 노드를 이동합니다.

## 자세히 알아보기

이 샘플에서는 식 트리로 표시되는 알고리즘을 트래버스하고 해석하기 위해 작성하는 코드의 작은 하위 집합을 보여 줍니다. 식 트리를 다른 언어로 변환하는 일반적인 용도의 라이브러리를 작성하는데 필요한 모든 작업에 대한 자세한 내용은 Matt Warren의 [이 시리즈](#)를 참조하세요. 이 시리즈는 식 트리에서 찾을 수 있는 코드를 변환하는 방법에 대해 상세히 설명합니다.

식 트리의 진정한 기능을 확인하셨길 바랍니다. 코드 집합을 검사하고, 해당 코드를 원하는 대로 변경하고, 변경된 버전을 실행할 수 있습니다. 식 트리는 변경할 수 없기 때문에 기존 트리의 구성 요소를 사용하여 새 트리를 만들 수 있습니다. 이렇게 하면 수정된 식 트리를 만드는데 필요한 메모리 양이 최소화됩니다.

[다음 -- 요약](#)

# 식 트리 요약

2020-03-18 • 3 minutes to read • [Edit Online](#)

## 이전 -- 식 변환

이 시리즈에서는 [식 트리](#)를 사용하여 코드를 데이터로 해석하고 해당 코드에 기반한 새 기능을 구축하는 동적 프로그램을 만드는 방법을 살펴보았습니다.

식 트리를 검사하여 알고리즘의 의도를 파악할 수 있습니다. 해당 코드 검사 이상을 수행할 수 있습니다. 원래 코드의 수정된 버전을 나타내는 새로운 식 트리를 작성할 수 있습니다.

또한 식 트리를 사용하여 알고리즘을 확인하고 해당 알고리즘을 다른 언어나 환경으로 변환할 수 있습니다.

## 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 릴리스에서 추가된 많은 기능이 식 트리에 작성된 대로 정확하게 나타나지 않습니다. 대신 최신 기능은 해당되는 이전 구문으로 식 트리에 노출됩니다. 생각만큼 큰 제한 사항이 아닐 수 있습니다. 실제로는 새로운 언어 기능이 도입되어도 식 트리를 해석하는 코드가 동일하게 작동할 수 있습니다.

이러한 제한 사항에도 불구하고 식 트리를 사용하면 데이터 구조로 표시되는 코드를 해석하고 수정하는 동적 알고리즘을 만들 수 있습니다. 식 트리는 강력한 도구이며 기능을 수행하기 위해 Entity Framework와 같은 풍부한 라이브러리를 사용하는 .NET 에코시스템의 기능 중 하나입니다.

# 상호 운용성(C# 프로그래밍 가이드)

2020-11-02 • 3 minutes to read • [Edit Online](#)

상호 운용성은 비관리 코드에 대한 기존 투자를 보존하고 활용할 수 있도록 합니다. CLR(공용 언어 런타임)의 제어 하에서 실행되는 코드를 관리 코드라고 하고, CLR 외부에서 실행되는 코드를 비관리 코드라고 합니다. 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.

## 섹션 내용

### 상호 운용성 개요

C# 관리 코드와 비관리 코드 간에 상호 운용되도록 하는 방법을 설명합니다.

### C# 기능을 사용하여 Office interop 개체에 액세스하는 방법

Office 프로그래밍을 용이하게 하도록 Visual C#에 도입된 기능에 대해 설명합니다.

### COM interop 프로그래밍에서 인덱싱된 속성을 사용하는 방법

인덱싱된 속성을 사용하여 매개 변수가 있는 COM 속성에 액세스하는 방법을 설명합니다.

### 플랫폼 호출을 사용하여 WAV 파일을 재생하는 방법

플랫폼 호출 서비스를 사용하여 Windows 운영 체제에서 .wav 사운드 파일을 재생하는 방법을 설명합니다.

### 연습: Office 프로그래밍

Excel 통합 문서와 통합 문서에 대한 링크를 포함하는 Word 문서를 만드는 방법을 보여 줍니다.

### COM 클래스 예제

C# 클래스를 COM 개체로 노출하는 방법을 보여 줍니다.

## C# 언어 사양

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

## 참고 항목

- [Marshal.ReleaseComObject](#)
- [C# 프로그래밍 가이드](#)
- [비관리 코드와의 상호 운용](#)
- [연습: Office 프로그래밍](#)

# XML 주석을 사용하여 코드 문서화

2020-11-02 • 44 minutes to read • [Edit Online](#)

XML 문서 주석은 사용자 정의 형식이나 멤버의 정의 위에 추가되는 특수 주석입니다. 컴파일 시간에 XML 문서 파일을 생성하기 위해 컴파일러에서 처리될 수 있으므로 특별합니다. Visual Studio 및 기타 IDE가 IntelliSense를 사용하여 형식이나 멤버에 대한 빠른 정보를 표시할 수 있도록 컴파일러에서 생성된 XML 파일은 .NET 어셈블리와 함께 배포될 수 있습니다. 또한 DocFX 및 Sandcastle 같은 도구를 통해 XML 파일을 실행하여 API 참조 웹 사이트를 생성할 수 있습니다.

XML 문서 주석은 모든 다른 주석처럼 컴파일러에서 무시됩니다.

다음 중 하나를 수행하여 컴파일 시간에 XML 파일을 생성할 수 있습니다.

- 명령줄에서 .NET Core를 사용하여 애플리케이션을 개발할 경우 .csproj 프로젝트 파일의 `<PropertyGroup>` 섹션에 `GenerateDocumentationFile` 요소를 추가할 수 있습니다. `DocumentationFile` 요소를 사용하여 문서 파일의 경로를 직접 지정할 수도 있습니다. 다음 예제에서는 프로젝트 디렉터리에 어셈블리와 같은 루트 파일 이름을 가진 XML 파일을 생성합니다.

```
<GenerateDocumentationFile>true</GenerateDocumentationFile>
```

이 식은 다음 식과 같습니다.

```
<DocumentationFile>bin\$(Configuration)\$(TargetFramework)\$(AssemblyName).xml</DocumentationFile>
```

- Visual Studio를 사용하여 애플리케이션을 개발할 경우 프로젝트를 마우스 오른쪽 단추로 클릭하고 속성을 선택합니다. [속성] 대화 상자에서 빌드 탭을 선택하고 **XML 문서 파일**을 선택합니다. 컴파일러가 파일을 쓰는 위치를 변경할 수도 있습니다.
- 명령줄에서 .NET 애플리케이션을 컴파일할 경우 컴파일 시 `-doc` 컴파일러 옵션을 추가합니다.

XML 문서 주석에는 삼중 슬래시(`///`) 및 XML 형식의 주석 본문이 사용됩니다. 예를 들어:

```
/// <summary>
/// This class does something.
/// </summary>
public class SomeClass
{}
```

## 연습

새로운 개발자가 이해/사용하고 타사 개발자가 사용하기 쉬운 매우 기본적인 수학 라이브러리를 문서화하는 방법을 살펴보겠습니다.

다음은 간단한 수학 라이브러리에 대한 코드입니다.

```

/*
The main Math class
Contains all methods for performing basic math functions
*/
public class Math
{
    // Adds two integers and returns the result
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Adds two doubles and returns the result
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Subtracts an integer from another and returns the result
    public static int Subtract(int a, int b)
    {
        return a - b;
    }

    // Subtracts a double from another and returns the result
    public static double Subtract(double a, double b)
    {
        return a - b;
    }

    // Multiplies two integers and returns the result
    public static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Multiplies two doubles and returns the result
    public static double Multiply(double a, double b)
    {
        return a * b;
    }

    // Divides an integer by another and returns the result
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    // Divides a double by another and returns the result
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}

```

샘플 라이브러리는 `int` 및 `double` 데이터 형식에서 네 가지 주요 산술 연산인 `add`, `subtract`, `multiply` 및 `divide`를 지원합니다.

이제 라이브러리를 사용하지만 소스 코드에 대한 액세스 권한이 없는 타사 개발자를 위해 코드에서 API 참조 문서를 만들 수 있습니다. 앞에서 언급한 대로 XML 문서 태그를 사용하여 이 작업을 수행할 수 있습니다. 이제 C# 컴파일러가 지원하는 표준 XML 태그를 소개하겠습니다.

## <summary>

<summary> 태그는 형식 또는 멤버에 대한 간단한 정보를 추가합니다. Math 클래스 정의 및 첫 번째 Add 메서드에 태그를 추가하여 사용하는 방법을 설명합니다. 나머지 코드에 자유롭게 적용하세요.

```
/*
 * The main Math class
 * Contains all methods for performing basic math functions
 */
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    // Adds two integers and returns the result
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}
```

<summary> 태그는 중요하며, 태그 내용은 IntelliSense 또는 API 참조 문서에서 형식 또는 멤버 정보의 기본 소스 이므로 포함하는 것이 좋습니다.

## <remarks>

<remarks> 태그는 <summary> 태그가 제공하는 형식 또는 멤버에 대한 정보를 보완합니다. 이 예제에서는 클래스에 태그를 추가합니다.

```
/*
 * The main Math class
 * Contains all methods for performing basic math functions
 */
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
/// <remarks>
/// This class can add, subtract, multiply and divide.
/// </remarks>
public class Math
{
```

## <returns>

<returns> 태그는 메서드 선언의 반환 값을 설명합니다. 이전과 같이 다음 예제에서는 첫 번째 Add 메서드에

대한 `<returns>` 태그를 보여 줍니다. 다른 메서드에 대해 같은 작업을 수행할 수 있습니다.

```
// Adds two integers and returns the result
/// <summary>
/// Adds two integers and returns the result.
/// </summary>
/// <returns>
/// The sum of two integers.
/// </returns>
public static int Add(int a, int b)
{
    // If any parameter is equal to the max value of an integer
    // and the other is greater than zero
    if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
        throw new System.OverflowException();

    return a + b;
}
```

## <value>

`<value>` 태그는 속성에 사용한다는 점을 제외하고 `<returns>` 태그와 비슷합니다. `Math` 라이브러리에 `PI`라는 정적 속성이 있다고 가정할 경우 이 태그를 사용하는 방법은 다음과 같습니다.

```
/*
The main Math class
Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
/// <remarks>
/// This class can add, subtract, multiply and divide.
/// These operations can be performed on both integers and doubles
/// </remarks>
public class Math
{
    /// <value>Gets the value of PI.</value>
    public static double PI { get; }
}
```

## <example>

`<example>` 태그를 사용하여 XML 문서에 예제를 포함합니다. 이 과정에는 자식 `<code>` 태그 사용이 포함됩니다.

```

// Adds two integers and returns the result
/// <summary>
/// Adds two integers and returns the result.
/// </summary>
/// <returns>
/// The sum of two integers.
/// </returns>
/// <example>
/// <code>
/// int c = Math.Add(4, 5);
/// if (c > 10)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
public static int Add(int a, int b)
{
    // If any parameter is equal to the max value of an integer
    // and the other is greater than zero
    if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
        throw new System.OverflowException();

    return a + b;
}

```

`code` 태그는 더 긴 예제에 대한 줄 바꿈 및 들여쓰기를 보존합니다.

## <para>

`<para>` 태그를 사용하여 부모 태그 내에서 내용의 서식을 지정합니다. `<para>`는 대개 `<remarks>` 또는 `<returns>` 와 같은 태그 내부에서 텍스트를 단락으로 나누는데 사용됩니다. 클래스 정의에 대한 `<remarks>` 태그의 내용에 서식을 지정할 수 있습니다.

```

/*
The main Math class
Contains all methods for performing basic math functions
*/
/// <summary>
/// The main Math class.
/// Contains all methods for performing basic math functions.
/// </summary>
/// <remarks>
/// <para>This class can add, subtract, multiply and divide.</para>
/// <para>These operations can be performed on both integers and doubles.</para>
/// </remarks>
public class Math
{
}

```

## <C>

서식 지정 항목에서 텍스트 부분을 코드로 표시하는 데 `<c>` 태그를 사용합니다. `<code>` 태그와 비슷하지만 인라인입니다. 태그 내용의 일부로 빠른 코드 예제를 표시하려는 경우 유용합니다. `Math` 클래스에 대한 문서를 업데이트해 보겠습니다.

```

/*
The main Math class
Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
}

```

## <exception>

<exception> 태그를 사용하여 메서드가 특정 예외를 throw할 수 있음을 개발자에게 알립니다. Math 라이브러리를 살펴보면 특정 조건에 해당할 경우 두 Add 메서드가 모두 예외를 throw함을 알 수 있습니다. b 매개 변수가 0인 경우 정수 Divide 메서드도 throw하는지는 분명하지 않습니다. 이제 이 메서드에 예외 문서를 추가합니다.

```

/*
The main Math class
Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    public static int Add(int a, int b)
    {
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    /// <summary>
    /// Adds two doubles and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    public static double Add(double a, double b)
    {

```

```

        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    /// <summary>
    /// Divides an integer by another and returns the result.
    /// </summary>
    /// <returns>
    /// The division of two integers.
    /// </returns>
    /// <exception cref="System.DivideByZeroException">Thrown when a division by zero occurs.</exception>
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    /// <summary>
    /// Divides a double by another and returns the result.
    /// </summary>
    /// <returns>
    /// The division of two doubles.
    /// </returns>
    /// <exception cref="System.DivideByZeroException">Thrown when a division by zero occurs.</exception>
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}

```

**cref** 특성은 현재 컴파일 환경에서 사용할 수 있는 예외에 대한 참조를 나타냅니다. 이 특성은 프로젝트 또는 참조된 어셈블리에서 정의된 형식일 수 있습니다. 해당 값을 확인할 수 없는 경우 컴파일러에서 경고가 발생합니다.

## <see>

**<see>** 태그를 사용하여 다른 코드 요소에 대한 문서 페이지의 클릭 가능한 링크를 만들 수 있습니다. 다음 예제에서는 두 **Add** 메서드 간의 클릭 가능한 링크를 만듭니다.

```

/*
The main Math class
Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    /// See <see cref="Math.Add(double, double)"> to add doubles.
    public static int Add(int a, int b)
    {
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    /// <summary>
    /// Adds two doubles and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    /// See <see cref="Math.Add(int, int)"> to add integers.
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}

```

`cref` 는 현재 컴파일 환경에서 사용할 수 있는 형식 또는 멤버에 대한 참조를 나타내는 **required** 특성입니다.  
이 특성은 프로젝트 또는 참조된 어셈블리에서 정의된 형식일 수 있습니다.

## <seealso>

<seealso> 태그는 <see> 태그와 같은 방법으로 사용합니다. 유일한 차이점은 내용이 일반적으로 "참고 항목" 섹션에 배치된다는 것입니다. 이제 정수 `Add` 메서드에 대해 `seealso` 태그를 추가하여 클래스에서 정수 매개 변수를 허용하는 다른 메서드를 참조합니다.

```

/*
The main Math class
Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than 0.</exception>
    /// See <see cref="Math.Add(double, double)"> to add doubles.
    /// <seealso cref="Math.Subtract(int, int)">
    /// <seealso cref="Math.Multiply(int, int)">
    /// <seealso cref="Math.Divide(int, int)">
    public static int Add(int a, int b)
    {
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}

```

`cref`는 현재 컴파일 환경에서 사용할 수 있는 형식 또는 멤버에 대한 참조를 나타냅니다. 이 특성은 프로젝트 또는 참조된 어셈블리에서 정의된 형식일 수 있습니다.

## <param>

<param> 태그를 사용하여 메서드의 매개 변수를 설명합니다. 다음은 double `Add` 메서드에 대한 예제입니다.  
태그가 설명하는 매개 변수는 **required** `name` 특성에서 지정됩니다.

```

/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two doubles and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    /// See <see href="Math.Add(int, int)"> to add integers.
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}

```

## <typeparam>

<typeparam> 태그는 <param> 태그처럼 사용하지만 제네릭 매개 변수를 설명하는 제네릭 형식 또는 메서드 선언에 사용합니다. Math 클래스에 빠른 제네릭 메서드를 추가하여 한 수량이 다른 수량보다 큰지 확인합니다.

```

/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Checks if an IComparable is greater than another.
    /// </summary>
    /// <typeparam name="T">A type that inherits from the IComparable interface.</typeparam>
    public static bool GreaterThan<T>(T a, T b) where T : IComparable
    {
        return a.CompareTo(b) > 0;
    }
}

```

## <paramref>

<summary> 태그와 같은 태그에서 메서드가 수행하는 작업을 설명하는 동안 매개 변수에 대한 참조를 만들려는 경우 <paramref> 태그를 사용하는 것이 좋습니다. double 기반 Add 메서드의 요약을 업데이트해 보겠습니다.

<param> 태그처럼 매개 변수 이름은 required name 특성에서 지정됩니다.

```

/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Adds two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two doubles.
    /// </returns>
    /// <exception cref="System.OverflowException">Thrown when one parameter is max
    /// and the other is greater than zero.</exception>
    /// See <see cref="Math.Add(int, int)"> to add integers.
    /// <param name="a">A double precision number.</param>
    /// <param name="b">A double precision number.</param>
    public static double Add(double a, double b)
    {
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }
}

```

## <typeparamref>

<typeparamref> 태그는 <paramref> 태그처럼 사용하지만 제네릭 매개 변수를 설명하는 제네릭 형식 또는 메서드 선언에 사용합니다. 이전에 만든 것과 같은 제네릭 메서드를 사용할 수 있습니다.

```

/*
    The main Math class
    Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// </summary>
public class Math
{
    /// <summary>
    /// Checks if an IComparable <typeparamref name="T"/> is greater than another.
    /// </summary>
    /// <typeparam name="T">A type that inherits from the IComparable interface.</typeparam>
    public static bool GreaterThan<T>(T a, T b) where T : IComparable
    {
        return a.CompareTo(b) > 0;
    }
}

```

## <list>

<list> 태그를 사용하여 문서 정보의 서식을 순서가 지정된 목록, 순서가 지정되지 않은 목록 또는 표로 지정합니다. **Math** 라이브러리에서 지원하는 모든 수학 연산의 순서가 지정되지 않은 목록을 만듭니다.

```

/*
   The main Math class
   Contains all methods for performing basic math functions
*/
/// <summary>
/// The main <c>Math</c> class.
/// Contains all methods for performing basic math functions.
/// <list type="bullet">
/// <item>
/// <term>Add</term>
/// <description>Addition Operation</description>
/// </item>
/// <item>
/// <term>Subtract</term>
/// <description>Subtraction Operation</description>
/// </item>
/// <item>
/// <term>Multiply</term>
/// <description>Multiplication Operation</description>
/// </item>
/// <item>
/// <term>Divide</term>
/// <description>Division Operation</description>
/// </item>
/// </list>
/// </summary>
public class Math
{
}

```

`type` 특성을 각각 `number` 또는 `table`로 변경하여 순서가 지정된 목록 또는 표를 만들 수 있습니다.

## <inheritdoc>

`<inheritdoc>` 태그를 사용하여 기본 클래스, 인터페이스 및 유사한 메서드에서 XML 주석을 상속할 수 있습니다. 이렇게 하면 중복 XML 주석을 복사하여 붙여 넣을 필요가 없으며 XML 주석이 자동으로 동기화됩니다.

```

/*
   The IMath interface
   The main Math class
   Contains all methods for performing basic math functions
*/
/// <summary>
/// This is the IMath interface.
/// </summary>
public interface IMath
{
}

/// <inheritdoc/>
public class Math : IMath
{
}

```

## 모든 요소 결합

이 자습서에 따라 필요한 경우 코드에 태그를 적용했습니다. 이제 코드는 다음과 같이 표시되어야 합니다.

```

/*
   The main Math class
   Contains all methods for performing basic math functions
*/
/// <summary>

```

```
/// The main <c>Math</c> class.  
/// Contains all methods for performing basic math functions.  
/// <list type="bullet">  
/// <item>  
/// <term>Add</term>  
/// <description>Addition Operation</description>  
/// </item>  
/// <item>  
/// <term>Subtract</term>  
/// <description>Subtraction Operation</description>  
/// </item>  
/// <item>  
/// <term>Multiply</term>  
/// <description>Multiplication Operation</description>  
/// </item>  
/// <item>  
/// <term>Divide</term>  
/// <description>Division Operation</description>  
/// </item>  
/// </list>  
/// </summary>  
/// <remarks>  
/// <para>This class can add, subtract, multiply and divide.</para>  
/// <para>These operations can be performed on both integers and doubles.</para>  
/// </remarks>  
public class Math  
{  
    // Adds two integers and returns the result  
    /// <summary>  
    /// Adds two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.  
    /// </summary>  
    /// <returns>  
    /// The sum of two integers.  
    /// </returns>  
    /// <example>  
    /// <code>  
    /// int c = Math.Add(4, 5);  
    /// if (c > 10)  
    /// {  
    ///     Console.WriteLine(c);  
    /// }  
    /// </code>  
    /// </example>  
    /// <exception cref="System.OverflowException">Thrown when one parameter is max  
    /// and the other is greater than 0.</exception>  
    /// See <see href="Math.Add(double, double)"> to add doubles.  
    /// <seealso href="Math.Subtract(int, int)">  
    /// <seealso href="Math.Multiply(int, int)">  
    /// <seealso href="Math.Divide(int, int)">  
    /// <param name="a">An integer.</param>  
    /// <param name="b">An integer.</param>  
    public static int Add(int a, int b)  
    {  
        // If any parameter is equal to the max value of an integer  
        // and the other is greater than zero  
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))  
            throw new System.OverflowException();  
  
        return a + b;  
    }  
  
    // Adds two doubles and returns the result  
    /// <summary>  
    /// Adds two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.  
    /// </summary>  
    /// <returns>  
    /// The sum of two doubles.  
    /// </returns>  
    /// <example>
```

```
/// <code>
/// double c = Math.Add(4.5, 5.4);
/// if (c > 10)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// <exception cref="System.OverflowException">Thrown when one parameter is max
/// and the other is greater than 0.</exception>
/// See <see cref="Math.Add(int, int)" /> to add integers.
/// <seealso cref="Math.Subtract(double, double)" />
/// <seealso cref="Math.Multiply(double, double)" />
/// <seealso cref="Math.Divide(double, double)" />
/// <param name="a">A double precision number.</param>
/// <param name="b">A double precision number.</param>
public static double Add(double a, double b)
{
    // If any parameter is equal to the max value of an integer
    // and the other is greater than zero
    if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
        throw new System.OverflowException();

    return a + b;
}

// Subtracts an integer from another and returns the result
/// <summary>
/// Subtracts <paramref name="b"/> from <paramref name="a"/> and returns the result.
/// </summary>
/// <returns>
/// The difference between two integers.
/// </returns>
/// <example>
/// <code>
/// int c = Math.Subtract(4, 5);
/// if (c > 1)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// See <see cref="Math.Subtract(double, double)" /> to subtract doubles.
/// <seealso cref="Math.Add(int, int)" />
/// <seealso cref="Math.Multiply(int, int)" />
/// <seealso cref="Math.Divide(int, int)" />
/// <param name="a">An integer.</param>
/// <param name="b">An integer.</param>
public static int Subtract(int a, int b)
{
    return a - b;
}

// Subtracts a double from another and returns the result
/// <summary>
/// Subtracts a double <paramref name="b"/> from another double <paramref name="a"/> and returns the
/// result.
/// </summary>
/// <returns>
/// The difference between two doubles.
/// </returns>
/// <example>
/// <code>
/// double c = Math.Subtract(4.5, 5.4);
/// if (c > 1)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
```

```
/// </example>
/// See <see cref="Math.Subtract(int, int)"> to subtract integers.
/// <seealso cref="Math.Add(double, double)">
/// <seealso cref="Math.Multiply(double, double)">
/// <seealso cref="Math.Divide(double, double)">
/// <param name="a">A double precision number.</param>
/// <param name="b">A double precision number.</param>
public static double Subtract(double a, double b)
{
    return a - b;
}

// Multiplies two integers and returns the result
/// <summary>
/// Multiplies two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
/// </summary>
/// <returns>
/// The product of two integers.
/// </returns>
/// <example>
/// <code>
/// int c = Math.Multiply(4, 5);
/// if (c > 100)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// See <see cref="Math.Multiply(double, double)"> to multiply doubles.
/// <seealso cref="Math.Add(int, int)">
/// <seealso cref="Math.Subtract(int, int)">
/// <seealso cref="Math.Divide(int, int)">
/// <param name="a">An integer.</param>
/// <param name="b">An integer.</param>
public static int Multiply(int a, int b)
{
    return a * b;
}

// Multiplies two doubles and returns the result
/// <summary>
/// Multiplies two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
/// </summary>
/// <returns>
/// The product of two doubles.
/// </returns>
/// <example>
/// <code>
/// double c = Math.Multiply(4.5, 5.4);
/// if (c > 100.0)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// See <see cref="Math.Multiply(int, int)"> to multiply integers.
/// <seealso cref="Math.Add(double, double)">
/// <seealso cref="Math.Subtract(double, double)">
/// <seealso cref="Math.Divide(double, double)">
/// <param name="a">A double precision number.</param>
/// <param name="b">A double precision number.</param>
public static double Multiply(double a, double b)
{
    return a * b;
}

// Divides an integer by another and returns the result
/// <summary>
/// Divides an integer <paramref name="a"/> by another integer <paramref name="b"/> and returns the

```

```

    /// Divides an integer <paramref name="a"/> by another integer <paramref name="b"/> and returns the
result.
    /// </summary>
    /// <returns>
    /// The quotient of two integers.
    /// </returns>
    /// <example>
    /// <code>
    /// int c = Math.Divide(4, 5);
    /// if (c > 1)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
    /// See <see cref="Math.Divide(double, double)"> to divide doubles.
    /// <seealso cref="Math.Add(int, int)">
    /// <seealso cref="Math.Subtract(int, int)">
    /// <seealso cref="Math.Multiply(int, int)">
    /// <param name="a">An integer dividend.</param>
    /// <param name="b">An integer divisor.</param>
public static int Divide(int a, int b)
{
    return a / b;
}

// Divides a double by another and returns the result
/// <summary>
/// Divides a double <paramref name="a"/> by another double <paramref name="b"/> and returns the result.
/// </summary>
/// <returns>
/// The quotient of two doubles.
/// </returns>
/// <example>
/// <code>
/// double c = Math.Divide(4.5, 5.4);
/// if (c > 1.0)
/// {
///     Console.WriteLine(c);
/// }
/// </code>
/// </example>
/// <exception cref="System.DivideByZeroException">Thrown when <paramref name="b"/> is equal to 0.
</exception>
    /// See <see cref="Math.Divide(int, int)"> to divide integers.
    /// <seealso cref="Math.Add(double, double)">
    /// <seealso cref="Math.Subtract(double, double)">
    /// <seealso cref="Math.Multiply(double, double)">
    /// <param name="a">A double precision dividend.</param>
    /// <param name="b">A double precision divisor.</param>
public static double Divide(double a, double b)
{
    return a / b;
}
}

```

코드에서 클릭 가능한 상호 참조가 포함된 자세한 문서 웹 사이트를 생성할 수 있지만 코드를 읽기 어렵게 되는 다른 문제에 직면할 수 있습니다. 자세히 살펴볼 정보가 너무 많으므로 이 코드에 참여하려는 개발자에게는 큰 문제일 수 있습니다. 다행히도 이 문제를 처리할 수 있는 XML 태그가 있습니다.

## <include>

소스 코드 파일에 직접 문서 주석을 배치하는 것과 달리 <include> 태그를 사용하면 소스 코드에서 형식과 멤버를 설명하는 주석을 개별 XML 파일에서 참조할 수 있습니다.

이제 모든 XML 태그를 `docs.xml`이라는 개별 XML 파일로 이동합니다. 원하는 대로 파일 이름을 지정합니다.

```
<docs>
  <members name="math">
    <Math>
      <summary>
        The main <c>Math</c> class.
        Contains all methods for performing basic math functions.
      </summary>
      <remarks>
        <para>This class can add, subtract, multiply and divide.</para>
        <para>These operations can be performed on both integers and doubles.</para>
      </remarks>
    </Math>
    <AddInt>
      <summary>
        Adds two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
      </summary>
      <returns>
        The sum of two integers.
      </returns>
      <example>
        <code>
          int c = Math.Add(4, 5);
          if (c > 10)
          {
            Console.WriteLine(c);
          }
        </code>
      </example>
      <exception cref="System.OverflowException">Thrown when one parameter is max
        and the other is greater than 0.</exception>
      See <see cref="Math.Add(double, double)"> to add doubles.
      <seealso cref="Math.Subtract(int, int)">
      <seealso cref="Math.Multiply(int, int)">
      <seealso cref="Math.Divide(int, int)">
      <param name="a">An integer.</param>
      <param name="b">An integer.</param>
    </AddInt>
    <AddDouble>
      <summary>
        Adds two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
      </summary>
      <returns>
        The sum of two doubles.
      </returns>
      <example>
        <code>
          double c = Math.Add(4.5, 5.4);
          if (c > 10)
          {
            Console.WriteLine(c);
          }
        </code>
      </example>
      <exception cref="System.OverflowException">Thrown when one parameter is max
        and the other is greater than 0.</exception>
      See <see cref="Math.Add(int, int)"> to add integers.
      <seealso cref="Math.Subtract(double, double)">
      <seealso cref="Math.Multiply(double, double)">
      <seealso cref="Math.Divide(double, double)">
      <param name="a">A double precision number.</param>
      <param name="b">A double precision number.</param>
    </AddDouble>
    <SubtractInt>
      <summary>
        Subtracts <paramref name="b"/> from <paramref name="a"/> and returns the result.
      </summary>
    </SubtractInt>
  </members>
</docs>
```

```
., ., ., .
<returns>
The difference between two integers.
</returns>
<example>
<code>
int c = Math.Subtract(4, 5);
if (c > 1)
{
    Console.WriteLine(c);
}
</code>
</example>
See <see cref="Math.Subtract(double, double)"/> to subtract doubles.
<seealso cref="Math.Add(int, int)"/>
<seealso cref="Math.Multiply(int, int)"/>
<seealso cref="Math.Divide(int, int)"/>
<param name="a">An integer.</param>
<param name="b">An integer.</param>
</SubtractInt>
<SubtractDouble>
<summary>
Subtracts a double <paramref name="b"/> from another double <paramref name="a"/> and returns the result.
</summary>
<returns>
The difference between two doubles.
</returns>
<example>
<code>
double c = Math.Subtract(4.5, 5.4);
if (c > 1)
{
    Console.WriteLine(c);
}
</code>
</example>
See <see cref="Math.Subtract(int, int)"/> to subtract integers.
<seealso cref="Math.Add(double, double)"/>
<seealso cref="Math.Multiply(double, double)"/>
<seealso cref="Math.Divide(double, double)"/>
<param name="a">A double precision number.</param>
<param name="b">A double precision number.</param>
</SubtractDouble>
<MultiplyInt>
<summary>
Multiplies two integers <paramref name="a"/> and <paramref name="b"/> and returns the result.
</summary>
<returns>
The product of two integers.
</returns>
<example>
<code>
int c = Math.Multiply(4, 5);
if (c > 100)
{
    Console.WriteLine(c);
}
</code>
</example>
See <see cref="Math.Multiply(double, double)"/> to multiply doubles.
<seealso cref="Math.Add(int, int)"/>
<seealso cref="Math.Subtract(int, int)"/>
<seealso cref="Math.Divide(int, int)"/>
<param name="a">An integer.</param>
<param name="b">An integer.</param>
</MultiplyInt>
<MultiplyDouble>
<summary>
Multiplies two doubles <paramref name="a"/> and <paramref name="b"/> and returns the result.
</summary>
```

**Multiples** two doubles `\param{a}` and `\param{b}` and returns the result.

**Summary**

**Returns**

The product of two doubles.

**Code**

```
double c = Math.Multiply(4.5, 5.4);
if (c > 100.0)
{
    Console.WriteLine(c);
}
```

**Example**

See `<see cref="Math.Multiply(int, int)">` to multiply integers.

**SeeAlso**

- `<seealso cref="Math.Add(double, double)">`
- `<seealso cref="Math.Subtract(double, double)">`
- `<seealso cref="Math.Divide(double, double)">`

**Parameters**

- `<param name="a">`A double precision number.
- `<param name="b">`A double precision number.

**MultiplyDouble**

**DivideInt**

**Summary**

Divides an integer `<paramref name="a">` by another integer `<paramref name="b">` and returns the result.

**Returns**

The quotient of two integers.

**Code**

```
int c = Math.Divide(4, 5);
if (c > 1)
{
    Console.WriteLine(c);
}
```

**Example**

**Exception**

`<exception cref="System.DivideByZeroException">`Thrown when `<paramref name="b">` is equal to 0.

**DivideDouble**

**Summary**

Divides a double `<paramref name="a">` by another double `<paramref name="b">` and returns the result.

**Returns**

The quotient of two doubles.

**Code**

```
double c = Math.Divide(4.5, 5.4);
if (c > 1.0)
{
    Console.WriteLine(c);
}
```

**Example**

**Exception**

`<exception cref="System.DivideByZeroException">`Thrown when `<paramref name="b">` is equal to 0.

**DivideInt**

**Summary**

See `<see cref="Math.Divide(int, int)">` to divide integers.

**SeeAlso**

- `<seealso cref="Math.Add(double, double)">`
- `<seealso cref="Math.Subtract(double, double)">`

```

<seealso cref="Math.Multiply(double, double)"/>
<param name="a">A double precision dividend.</param>
<param name="b">A double precision divisor.</param>
</DivideDouble>
</members>
</docs>

```

위 XML에서 각 멤버의 문서 주석은 멤버가 수행하는 작업과 같은 이름이 지정된 태그 내부에 직접 나타납니다. 자신만의 전략을 선택할 수 있습니다. 이제 XML 주석이 개별 파일에 있으므로 `<include>` 태그를 사용하여 코드를 더 읽기 좋게 만드는 방법을 살펴봅니다.

```

/*
The main Math class
Contains all methods for performing basic math functions
*/
/// <include file='docs.xml' path='docs/members[@name="math"]/Math/*'>
public class Math
{
    // Adds two integers and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/AddInt/*'>
    public static int Add(int a, int b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Adds two doubles and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/AddDouble/*'>
    public static double Add(double a, double b)
    {
        // If any parameter is equal to the max value of an integer
        // and the other is greater than zero
        if ((a == double.MaxValue && b > 0) || (b == double.MaxValue && a > 0))
            throw new System.OverflowException();

        return a + b;
    }

    // Subtracts an integer from another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/SubtractInt/*'>
    public static int Subtract(int a, int b)
    {
        return a - b;
    }

    // Subtracts a double from another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/SubtractDouble/*'>
    public static double Subtract(double a, double b)
    {
        return a - b;
    }

    // Multiplies two integers and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/MultiplyInt/*'>
    public static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Multiplies two doubles and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/MultiplyDouble/*'>
    public static double Multiply(double a, double b)
    {

```

```

        return a * b;
    }

    // Divides an integer by another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/DivideInt/*'>
    public static int Divide(int a, int b)
    {
        return a / b;
    }

    // Divides a double by another and returns the result
    /// <include file='docs.xml' path='docs/members[@name="math"]/DivideDouble/*'>
    public static double Divide(double a, double b)
    {
        return a / b;
    }
}

```

해냈습니다. 코드를 다시 읽을 수 있고 문서 정보가 손실되지 않았습니다.

`file` 특성은 문서가 포함된 XML 파일의 이름을 나타냅니다.

`path` 특성은 지정된 `file`에 있는 `tag name`에 대한 XPath 쿼리를 나타냅니다.

`name` 특성은 주석 앞에 오는 태그의 이름 지정자를 나타냅니다.

`name` 대신 사용할 수 있는 `id` 특성은 주석 앞에 오는 태그의 ID를 나타냅니다.

### 사용자 정의 태그

위에서 설명한 모든 태그는 C# 컴파일러에서 인식되는 태그를 나타냅니다. 그러나 사용자가 자유롭게 태그를 정의할 수 있습니다. Sandcastle 등의 도구는 `<event>`, `<note>` 같은 추가 태그를 지원하고 [네임스페이스 문서화](#)도 지원합니다. 사용자 지정 또는 사내 문서 생성 도구는 표준 태그와 함께 사용할 수도 있고 HTML에서 PDF 까지 다양한 출력 형식을 지원할 수 있습니다.

## 권장 사항

여러 가지 이유로 코드를 문서화하는 것이 좋습니다. 이어서 일부 모범 사례, 일반적인 사용 사례 시나리오 및 C# 코드에서 XML 문서 태그를 사용할 때 알아야 하는 내용을 살펴봅니다.

- 일관성을 보장하기 위해 모든 공개 형식과 해당 멤버를 문서화해야 합니다. 문서화해야 한다면 모두 문서화하세요.
- Private 멤버는 XML 주석을 사용하여 문서화할 수도 있습니다. 그러나 이렇게 하면 라이브러리의 내부(잠재적으로 기밀) 작업이 표시됩니다.
- IntelliSense에는 `<summary>` 태그의 내용이 필요하므로 최소한 형식과 해당 멤버에 이 태그가 있어야 합니다.
- 문서 텍스트는 마침표로 끝나는 전체 문장을 사용하여 기록되어야 합니다.
- Partial 클래스는 완전히 지원되고 문서 정보는 해당 형식의 단일 항목에 연결됩니다.
- 컴파일러에서 `<exception>`, `<include>`, `<param>`, `<see>`, `<seealso>` 및 `<typeparam>` 태그의 구문을 확인합니다.
- 컴파일러는 파일 경로와 코드의 다른 부분에 대한 참조가 포함된 매개 변수의 유효성을 검사합니다.

## 참고 항목

- [XML 문서 주석\(C# 프로그래밍 가이드\)](#)
- [문서 주석에 대한 권장 태그\(C# 프로그래밍 가이드\)](#)

# C#으로 버전 관리

2020-04-27 • 16 minutes to read • [Edit Online](#)

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 버전 등과 같은 기타 시나리오를 지정할 수도 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 이전 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 기본값을 변경하는 경우, 해당 인수를 제공하지 않는 모든 코드를 업데이트해야 합니다.

### NOTE

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` 파일을 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.

```
<dependentAssembly>
  <assemblyIdentity name="ReferencedLibrary" publicKeyToken="32ab4ba45e0a69a1" culture="en-us" />
  <bindingRedirect oldVersion="1.0.0" newVersion="1.0.1" />
</dependentAssembly>
```

### NOTE

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.

```

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

```

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.

```
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

```
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#)

2020-11-02 • 11 minutes to read • [Edit Online](#)

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 이니셜라이저**를 사용하여 개체를 초기화합니다.
- **메서드에 구조체를 전달하는 것과 클래스를 전달하는 것의 차이점에 대해 알아보세요.**
- **연산자 오버로드**를 사용합니다.
- **사용자 지정 확장 메서드**를 구현하고 호출합니다.
- C# 프로그래머도 Visual Basic에서 `My` 네임스페이스를 사용하려 할 수 있습니다.
- **확장 메서드**를 사용하여 `enum` 형식에 대해 새 메서드를 만듭니다.

### 클래스 및 구조체 멤버

구조체 및 클래스를 만들어 프로그램을 구현합니다. 이러한 기술은 클래스 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 작업

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.Split`를 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 `DateTime`로 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 `catch` 를 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.
- 쿼리에서 람다 식을 사용합니다.
- 쿼리 식에서 `var` 를 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.WhenAll` 를 사용하여 비동기 성능을 개선합니다.
- `async` 및 `await` 를 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.

# C#에서 String.Split을 사용하여 문자열을 분리하는 방법

2021-02-18 • 5 minutes to read • [Edit Online](#)

`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.

## NOTE

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.

```
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. 연속된 구분 문자는 반환된 배열의 값으로 빈 문자열을 생성합니다. 공백 문자를 구분 기호로 사용하는 다음 예제에서 빈 문자열을 만드는 방법을 확인할 수 있습니다.

```
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.

```

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.

```

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.

```

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);
}

```

## 참조

- [문자열에서 요소 추출](#)
- [C# 프로그래밍 가이드](#)
- [문자열](#)
- [.NET 정규식](#)

# 여러 문자열 연결 방법(C# 가이드)

2020-11-02 • 8 minutes to read • [Edit Online](#)

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.

## NOTE

이 문서의 C# 예제는 [Try.NET](#) 인라인 코드 러너 및 놀이터에서 실행됩니다. 대화형 창에서 예제를 실행하려면 실행 버튼을 선택합니다. 코드를 실행하면 실행을 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 예제에서는 소스 코드의 가독성을 향상하기 위해 긴 문자열 리터럴을 작은 문자열로 분할하기 위해 연결을 사용합니다. 분할된 문자열은 컴파일 시간에 단일 문자열로 연결됩니다. 이 경우 처리되는 문자열 수가 런타임 성능에 미치는 영향은 없습니다.

```
// 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);
```

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 만듭니다. 하나의 문에서 여러 `+` 연산자를 사용해도 문자열 콘텐츠는 한 번만 복사됩니다. 다음 코드는 `+` 및 `+=` 연산자를 사용하여 문자열을 연결하는 예제를 보여줍니다.

```
string userName = "<Type your name here>";  
string dateString = DateTime.Today.ToString("dd/MM/yyyy");  
  
// 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);
```

일부 식에서는 다음 코드와 같이 문자열 보간을 사용하여 문자열을 연결하는 것이 더 쉽습니다.

```

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);

```

#### NOTE

문자열 연결 연산에서 C# 컴파일러는 null 문자열을 빈 문자열과 동일하게 처리합니다.

문자열을 연결하는 다른 메서드는 [String.Format](#)입니다. 이 메서드는 작은 수의 구성 요소 문자열에서 문자열을 빌드할 때 잘 작동합니다.

다른 경우는 결합하는 소스 문자열의 개수를 모르는 루프에서 문자열을 결합할 수 있으며 소스 문자열의 실제 개수는 알 수 있습니다. [StringBuilder](#) 클래스는 이러한 시나리오를 위해 설계되었습니다. 다음 코드는 [StringBuilder](#) 클래스의 [Append](#) 메서드를 사용하여 문자열을 연결합니다.

```

// 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.

컬렉션의 문자열을 조인하는 또 다른 옵션은 [String.Concat](#) 메서드를 사용하는 것입니다. 소스 문자열을 구분 기호로 구분해야 하는 경우 [String.Join](#) 메서드를 사용합니다. 다음 코드는 두 메서드 모두를 사용하여 단어 배열을 결합합니다.

```

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);

```

마지막으로 [LINQ](#)와 [Enumerable.Aggregate](#) 메서드를 사용하여 컬렉션의 문자열을 조인할 수 있습니다. 이 메서드는 람다식을 사용하여 소스 문자열을 결합합니다. 람다식은 각 문자열을 기존의 누적 문자열에 추가하는 작업을 수행합니다. 다음 예에서는 배열의 각 단어 사이에 공백을 추가하여 단어 배열을 결합합니다.

```

string[] words = { "The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog." };

var phrase = words.Aggregate((partialPhrase, word) => $"{partialPhrase} {word}");
System.Console.WriteLine(phrase);

```

## 참조

- [String](#)

- [StringBuilder](#)
- [C# 프로그래밍 가이드](#)
- [문자열](#)

# 문자열 검색 방법

2020-11-02 • 10 minutes to read • [Edit Online](#)

문자열에서 텍스트를 검색하는 두 가지 기본 전략을 사용할 수 있습니다. [String](#) 클래스의 메서드는 특정 텍스트를 검색합니다. 정규식은 텍스트에서 패턴을 검색합니다.

## NOTE

이 문서의 C# 예제는 [Try.NET](#) 인라인 코드 러너 및 놀이터에서 실행됩니다. 대화형 창에서 예제를 실행하려면 실행 버튼을 선택합니다. 코드를 실행하면 실행을 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.

```
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.

```

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. 검색 할 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 검색 할 텍스트를 설명합니다. 다음 표에서는 검색 패턴의 각 요소에 대해 설명합니다. (아래 표에서는 C# 문자열에서 `\`로 이스케이프되어야 하는 단일 `\`를 사용합니다.)

무늬	의미
<code>the</code>	텍스트 "the" 일치
<code>(eir)?</code>	"eir"과 0 또는 1개 항목 일치
<code>\s</code>	공백 문자 찾기

```

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();
    }
}

```

**TIP**

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.

무늬	의미
<code>^</code>	문자열의 시작 부분 일치
<code>\d{3}</code>	정확히 3자리 문자 일치
<code>-</code>	'-' 문자 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

```

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.

## 참조

- [C# 프로그래밍 가이드](#)
- [문자열](#)
- [LINQ 및 문자열](#)
- [System.Text.RegularExpressions.Regex](#)
- [.NET 정규식](#)
- [정규식 언어 - 빠른 참조](#)
- [.NET에서 문자열 사용에 대한 모범 사례](#)

# C#에서 문자열 내용을 수정하는 방법

2020-11-02 • 13 minutes to read • [Edit Online](#)

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.

## NOTE

이 문서의 C# 예제는 [Try.NET](#) 인라인 코드 러너 및 놀이터에서 실행됩니다. 대화형 창에서 예제를 실행하려면 실행 버튼을 선택합니다. 코드를 실행하면 실행을 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.

```
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 속성을 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.

```
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.

```
// 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.

```
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 함수입니다. `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.

```

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](#) 개체의 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. [IndexOf](#) 메서드를 사용하여 단어 "fox"의 시작 인덱스를 찾습니다. 다음 세 개의 문자는 서로 다른 단어로 바뀝니다. 마지막으로 새 문자열은 업데이트된 문자 배열에서 생성됩니다.

```

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 있으므로 이전 예제에서는 모두 임시 문자열 또는 문자 배열을 만듭니다. 고성능 시나리오에서는 이 힘 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.

```
// 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

2020-11-02 • 25 minutes to read • [Edit Online](#)

문자열을 비교하여 다음 두 가지 질문 중 하나를 해결할 수 있습니다. "이 두 문자열이 같나요?" 또는 "정렬할 때 이 문자열을 어떤 순서로 정렬해야 하는가?"

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.

## NOTE

이 문서의 C# 예제는 Try.NET 인라인 코드 러너 및 놀이터에서 실행됩니다. 대화형 창에서 예제를 실행하려면 실행 버튼을 선택합니다. 코드를 실행하면 실행을 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 모두를 검색하는 것이 더 쉽습니다. 다른 비교로 문자열이 같은지 확인할 수 있습니다. 이러한 동일성 검사는 동등 여부와 유사하지만 대/소문자 차이 같은 일부 차이점은 무시할 수 있습니다.

## 기본 서수 비교

기본적으로 가장 일반적인 작업:

- [String.CompareTo](#)
- [String.Equals](#)
- [String.Equality](#) 및 [String.Inequality](#), 즉 같은 연산자 `==` 및 `!=`는 각각

대/소문자 구분 서수 비교를 수행하고 필요한 경우 현재 문화권을 사용합니다. 다음은 해당 예입니다.

```
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 같은 테스트는 [String.CompareTo](#) 및 [Compare\(String, String\)](#) 메서드를 사용한 문자열 비교와 다릅니다. 동일성에 대한 테스트가 대/소문자 구분 서수 비교를 수행하는 동안 비교 메서드는 현재 문화권을 사용하여 대/소문자를 구분하고 문화권 구분 비교를 수행합니다. 기본 비교 메서드는 종

종 다양한 유형의 비교를 수행하기 때문에 수행할 비교 형식을 명시적으로 지정하는 오버로드를 호출하여 코드의 의도를 항상 명확히 하는 것이 좋습니다.

## 대/소문자를 구분하지 않는 서수 비교

`String.Equals(String, StringComparison)` 메서드를 사용하면 `StringComparison.OrdinalIgnoreCase`의 `StringComparison` 값을 지정하여 대/소문자를 구분하지 않는 서수 비교. `StringComparison` 인수에 대해 `StringComparison.OrdinalIgnoreCase` 값을 지정하는 경우 대/소문자를 구분하지 않는 서수 비교를 수행하는 정적 `String.Compare(String, String, StringComparison)` 메서드도 있습니다. 이는 다음 코드에서 확인할 수 있습니다.

```
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, 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");
```

대/소문자를 구분하지 않는 서수 비교를 수행하는 경우 이러한 메서드는 [고정 문화권](#)의 대/소문자 규칙을 사용합니다.

## 언어 비교

또한 현재 문화권에 대한 언어 규칙을 사용하여 문자열을 정렬할 수 있습니다. 이를 종종 "단어 정렬 순서"라고 합니다. 언어 비교를 수행할 때 일부 영숫자가 아닌 유니코드 문자에 특별한 가중치가 할당될 수 있습니다. 예를 들어, 하이픈 “-”은 작은 가중치가 할당될 수 있으므로 “co-op” 및 “coop”는 정렬 순서에 나란히 표시됩니다. 또한 일부 유니코드 문자는 `Char` 인스턴스의 시퀀스와 동일할 수 있습니다. 다음 예에서는 “ss” 및 ‘ß’를 사용하여 독일어로 독일어에서는 “ss”(U+0073 U+0073)가 한 문자열에 있고 ‘ß’(U+00DF)가 다른 문자열에 있습니다. 언어적으로(Windows의 경우) “ss”는 “en-US” 및 “de-DE” 문화권에서 독일어 Esszet: ‘ß’ 문자와 같습니다.

```

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");
}

```

이 샘플에서는 운영 체제별 언어 비교 특성을 보여줍니다. 대화형 창 호스트는 Linux 호스트입니다. 언어 및 서수 비교는 동일한 결과를 생성합니다. 같은 샘플을 Windows 호스트에서 실행한 경우 다음과 같은 출력이 표시됩니다.

```

<coop> is less than <co-op> using invariant culture
<coop> is greater than <co-op> using ordinal comparison
<coop> is less than <cop> using invariant culture
<coop> is less than <cop> using ordinal comparison
<co-op> is less than <cop> using invariant culture
<co-op> is less than <cop> using ordinal comparison

```

Windows에서 언어 비교를 서수 비교로 변경하면 "cop", "coop" 및 "co-op"의 정렬 순서가 변경됩니다. 두 개의 독일어 문장도 서로 다른 비교 형식을 사용하여 다르게 비교됩니다.

## 특정 문화권을 사용한 비교

이 샘플은 en-US 및 de-DE 문화권에 대한 [CultureInfo](#) 개체를 저장합니다. 비교는 문화권별 비교를 보장하기 위해 [CultureInfo](#) 개체를 사용하여 수행됩니다.

사용된 문화권은 언어 비교에 영향을 줍니다. 다음 예에서는 "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 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. 또한 이 샘플 코드를 Windows 머신에서 로컬로 시도하면 다음과 같은 결과가 나타납니다.

```

<coop> is less than <co-op> using en-US culture
<coop> is greater than <co-op> using ordinal comparison
<coop> is less than <cop> using en-US culture
<coop> is less than <cop> using ordinal comparison
<co-op> is less than <cop> using en-US culture
<co-op> is less than <cop> using ordinal comparison

```

언어 비교는 현재 문화권에 따라 달라지며 OS에 종속됩니다. 문자열 비교 작업을 할 때 이 점을 고려하세요.

## 배열의 언어적 정렬 및 문자열 검색

다음 예에서는 현재 문화권에 따라 언어 비교를 사용하여 배열에서 문자열을 정렬하고 검색하는 방법을 보여줍니다.

니다. [System.StringComparer](#) 매개 변수를 사용하는 정적 [Array](#) 메서드를 사용합니다.

이 예에서는 현재 문화권을 사용하여 문자열 배열을 정렬하는 방법을 보여줍니다.

```
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 문자열이 발견된 위치에 대한 정보를 표시합니다. 문자열을 찾을 수 없는 경우 반환된 값은 발견된 그 위치를 나타냅니다.

```

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.WriteLine($"{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.

```
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\nSorted 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.

```

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.Write($"{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.

## 참조 동일성과 문자열 인터닝

샘플 중에 [ReferenceEquals](#)를 사용한 것은 없습니다. 이 메서드는 두 문자열이 동일한 개체인지 여부를 결정하므로 문자열 비교의 결과가 일관되지 않을 수 있습니다. 다음 예에서는 C#의 문자열 인터닝 기능을 보여줍니다. 프로그램이 두 개 이상의 동일 문자열 변수를 선언할 경우 컴파일러는 변수를 모두 같은 위치에 저장합니다. [ReferenceEquals](#) 메서드를 호출하여 두 문자열이 실제로 메모리에서 같은 개체를 참조하는지 확인할 수 있습니다. 인터닝을 방지하려면 [String.Copy](#) 메서드를 사용합니다. 복사본이 생성된 후 두 개의 문자열은 동일한 값을 가지더라도 스토리지 위치가 다릅니다. `a` 및 `b`가 인터닝되었음을, 즉 동일한 스토리지를 공유한다는 것을 보여주는 다음 샘플을 실행합니다. 문자열 `a`와 `c`는 그렇지 않습니다.

```
string a = "The computer ate my source code.";
string b = "The computer ate my source code.";

if (String.ReferenceEquals(a, b))
    Console.WriteLine("a and b are interned.");
else
    Console.WriteLine("a and b are not interned.");

string c = String.Copy(a);

if (String.ReferenceEquals(a, c))
    Console.WriteLine("a and c are interned.");
else
    Console.WriteLine("a and c are not interned.");
```

#### NOTE

문자열의 동일성을 테스트할 때, 어떤 유형의 비교를 수행할지 명시적으로 지정하는 메서드를 사용해야 합니다. 코드를 훨씬 더 쉽게 유지 관리하고 읽을 수 있습니다. [StringComparison](#) 열거형 매개 변수를 사용하는 [System.String](#) 및 [System.Array](#) 클래스의 메서드 오버로드를 사용합니다. 수행할 비교 형식을 지정합니다. 동일성을 테스트할 때 `==` 및 `!=` 연산자를 사용하지 않습니다. [String.CompareTo](#) 인스턴스 메서드는 항상 서수 대/소문자 구분 비교를 수행합니다. 주로 알파벳순으로 문자열을 정렬하는 데 적합합니다.

[String.Intern](#) 메서드를 호출하여 문자열을 인터트하거나 기존의 인턴된 문자열에 대한 참조를 검색할 수 있습니다. 문자열이 인턴트되었는지 확인하려면 [String.IsInterned](#) 메서드를 호출합니다.

## 참조

- [System.Globalization.CultureInfo](#)
- [System.StringComparer](#)
- [문자열](#)
- [문자열 비교](#)
- [애플리케이션 전역화 및 지역화](#)

# 패턴 일치와 is 및 as 연산자를 사용하여 안전하게 캐스트하는 방법

2020-11-02 • 8 minutes to read • [Edit Online](#)

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.

```

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 { }

class Program
{
    static void Main(string[] args)
    {
        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);
        // Output:
        // I am an animal.
        // SuperNova is not a Mammal
    }

    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");
        }
    }
}

```

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 범위에 만 있습니다. 나중에 같은 방법으로 `m`에 액세스할 수 없습니다. 앞의 예제에서는 `as 연산자`를 사용하여 개체

를 지정된 형식으로 변환하는 방법도 보여 줍니다.

다음 예제에 표시된 대로 [null 허용 값 형식](#)에 값이 있는 경우 테스트할 때도 동일한 구문을 사용할 수 있습니다.

```
class Program
{
    static void Main(string[] args)
    {
        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(System.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(System.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. 앞의 예제에서 `is` 패턴은 nullable 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.

```
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 { }

class Program
{
    static void Main(string[] args)
    {
        // Use the is operator to verify the type.
        // before performing a cast.
        Giraffe g = new Giraffe();
        UseIsOperator(g);

        // Use the as operator and test for null
        // before referencing the variable.
        UseAsOperator(g);

        // Use the as operator to test
        // an incompatible type.
        SuperNova sn = new SuperNova();
        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();
        }
    }
}
```

```
    {
        m.Eat();
    }
}

static void UseAsOperator(object o)
{
    Mammal m = o as Mammal;
    if (m != 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 != null)
    {
        Console.WriteLine(j);
    }
    else
    {
        Console.WriteLine("Could not convert " + val.ToString());
    }
}
```

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.

# .NET Compiler Platform SDK

2021-02-18 • 20 minutes to read • [Edit Online](#)

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 것입니다. 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 수정 사항을 빌드할 수 있습니다. 분석기는 구문(코드의 구조) 및 의미 체계를 이해하고 수정되어야 하는 습관을 검색합니다. 코드 수정 사항은 분석기 또는 컴파일러 진단에서 발견한 코딩 실수를 해결하기 위한 한 가지 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 사항 외에도 .NET Compiler Platform SDK를 사용하여 코드 리팩터링을 빌드할 수도 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 작업에서 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 띠는 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 공

유할 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 불게 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.

#### TIP

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, 나머지는 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.

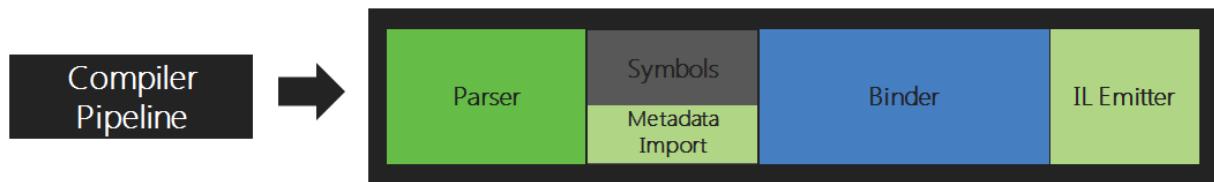
# .NET Compiler Platform SDK 모델 이해

2021-02-18 • 10 minutes to read • [Edit Online](#)

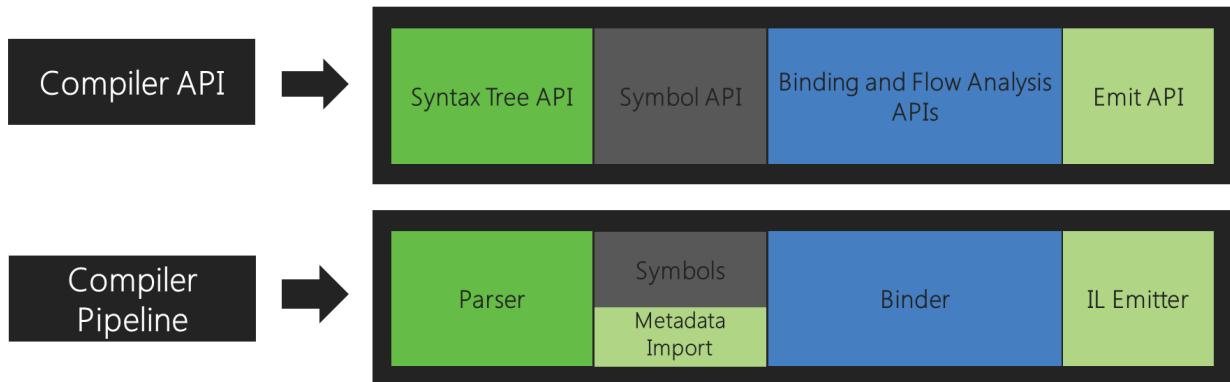
컴파일러는 종종 사람이 코드를 읽고 이해하는 방식과 다른 구조적 규칙을 따라 작성하는 코드를 처리합니다. 컴파일러에서 사용되는 모델에 대한 기본적인 이해는 Roslyn 기반 도구를 빌드할 때 사용하는 API를 이해하는데 반드시 필요합니다.

## 컴파일러 파이프라인 기능 영역

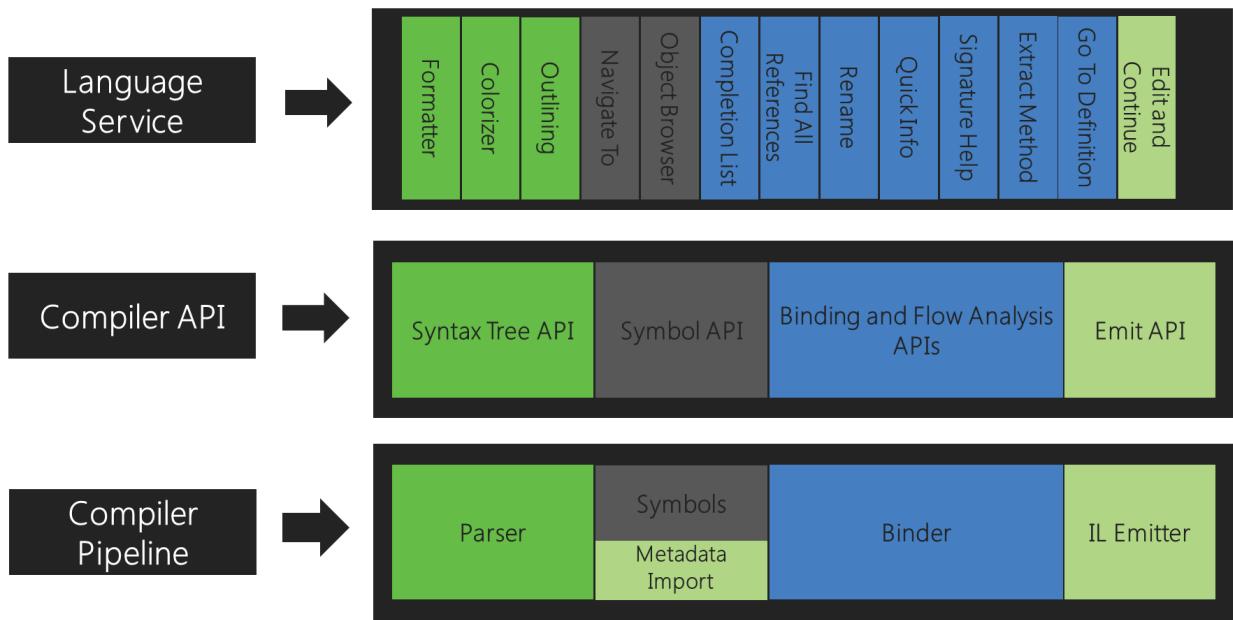
.NET Compiler Platform SDK는 기존의 컴파일러 파이프라인을 미러링하는 API 계층을 제공하여 소비자로서 사용자에게 C# 및 Visual Basic 컴파일러의 코드 분석을 노출합니다.



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 액세스할 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 이동은 의미 체계 모델을 사용하고, 편집하며 계속하기는 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 정의 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 리

팩터링 할 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

2021-02-18 • 24 minutes to read • [Edit Online](#)

'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 빌드할 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 액세스할 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 항

목을 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 `null`을 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 포함하는 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:

```
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 VB에는 문법에서 가능한 모든 노드, 토큰 및 기타 정보 요소를 나열하는 단일 `SyntaxKind` 열거형(각각 `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.

### TIP

`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.IsMatched` 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

2021-02-18 • 10 minutes to read • [Edit Online](#)

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 참조되는 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 컴파일하는데 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 찾는데 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 ISymbol에서 파생된 별도 인터페이스로 표시됩니다. 이러한 속성의 상당수는 다른 기호를 직접 참조합니다. 예를 들어 IMethoSymbol.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 IMethoSymbol로 표시

됩니다.

기호는 [System.Reflection API](#)로 표시된 CLR 형식 시스템과 개념상 비슷하지만 형식 이상을 모델링 하므로 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

2021-02-18 • 8 minutes to read • [Edit Online](#)

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

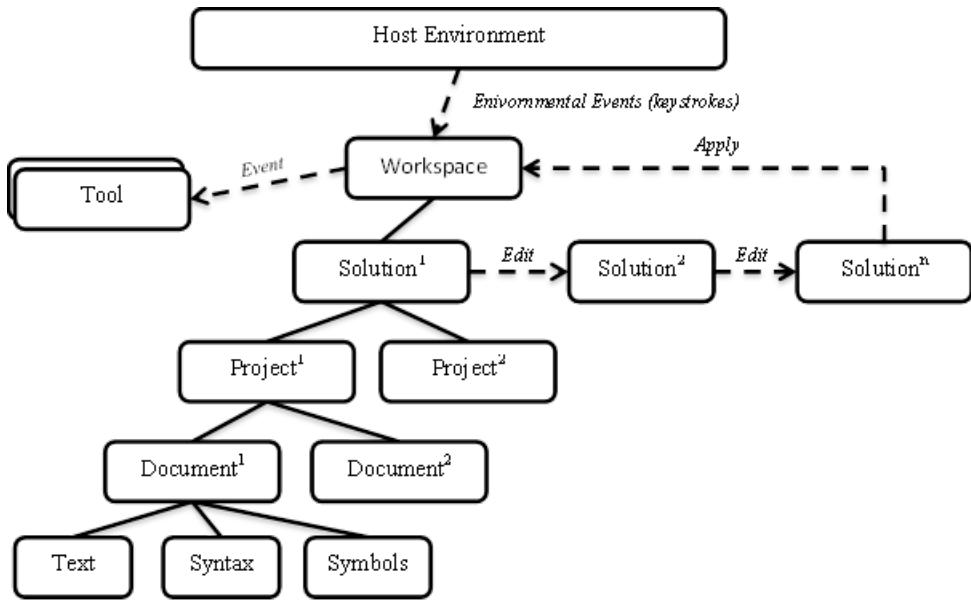
작업 영역은 키를 누를 때마다 변경될 수 있지만 격리 상태에서 솔루션의 모델을 사용하여 작업할 수 있습니다.

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

2020-11-02 • 27 minutes to read • [Edit Online](#)

이 아티클에서는 .NET Compiler Platform("Roslyn") SDK의 일부로 제공되는 구문 시각화 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 관리자**를 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 SDK**에 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 관리자**를 실행합니다.
2. **수정**을 선택합니다.
3. **개별 구성 요소** 탭을 선택합니다.
4. **.NET Compiler Platform SDK**에 대한 확인란을 선택합니다. 컴파일러, 빌드 도구 및 런타임 섹션의 위쪽에서 찾을 수 있습니다.

필요에 따라, 시각화 도우미에서 **DGML 편집기**에 그래프도 표시할 수 있습니다.

1. **DGML 편집기** 확인란을 선택합니다. 코드 도구 섹션에서 찾을 수 있습니다.

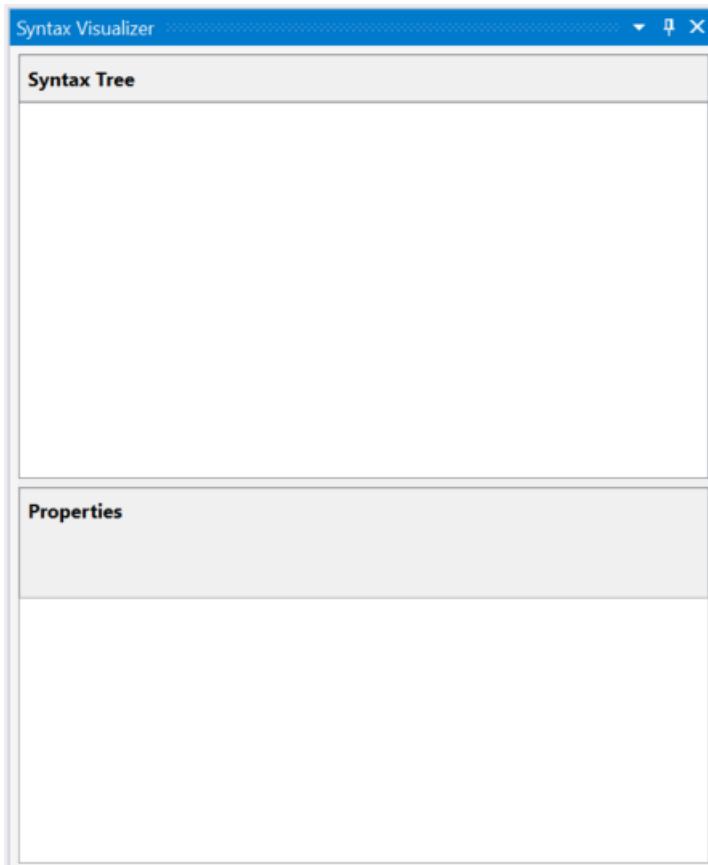
**개요** 아티클을 읽고 .NET Compiler Platform SDK에서 사용된 개념을 숙지하세요. 구문 트리, 노드, 토큰 및 퀴즈에 대한 소개를 제공합니다.

## 구문 시각화 도우미

**Syntax Visualizer**를 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.

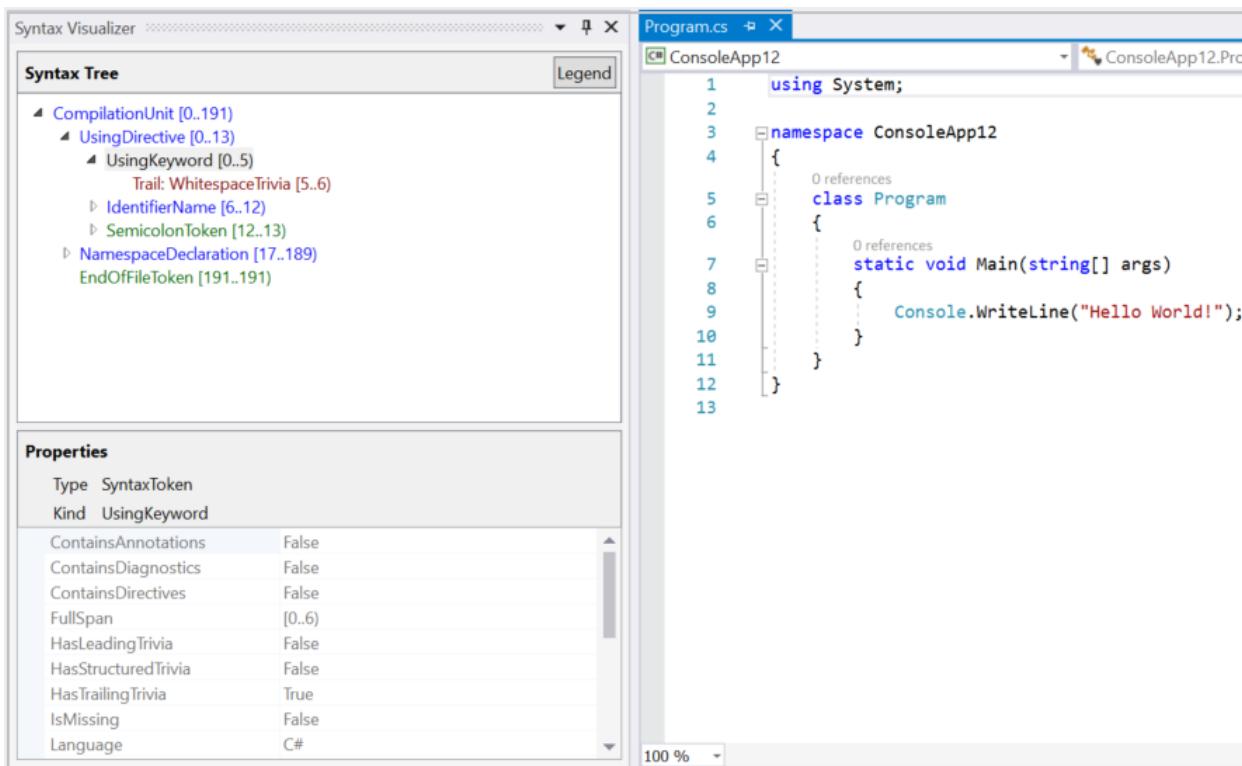
이 명령은 구문 시각화 도우미를 부동 도구 창으로 엽니다. 코드 편집기 창이 열려 있지 않은 경우 다음 그림과 같이 디스플레이가 비어 있습니다.



이 도구 창을 Visual Studio 내에서 원쪽과 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#](#)
- [Visual Basic](#)



위 이미지에서와 같이 시각화 도우미 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 도우미는 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 큐즈에 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.

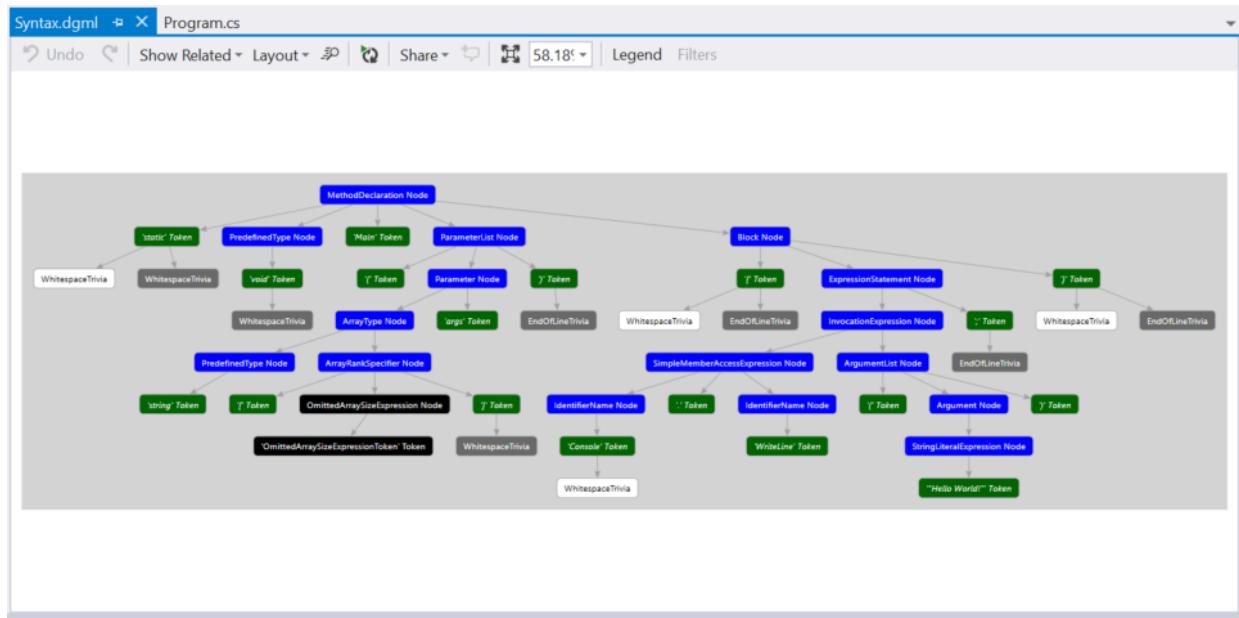
## 구문 그래프

트리에서 원하는 항목을 마우스 오른쪽 단추로 클릭하고 직접 구문 그래프 보기 를 클릭합니다.

- C#

- Visual Basic

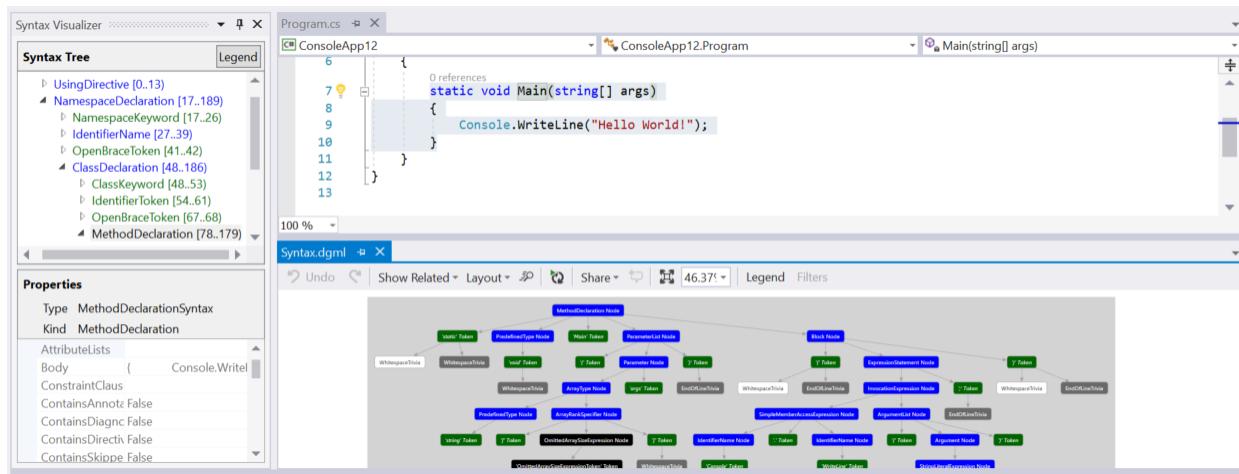
시각화 도우미는 선택한 항목을 루트로 하는 하위 트리의 그래프 표현을 표시합니다. C# 예제에서 `Main()` 메서드에 해당하는 **MethodDeclaration** 노드에 대해 이 단계를 시도합니다. 시각화 도우미는 다음과 같이 보이는 구문 그래프를 표시합니다.



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 그래프는 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.



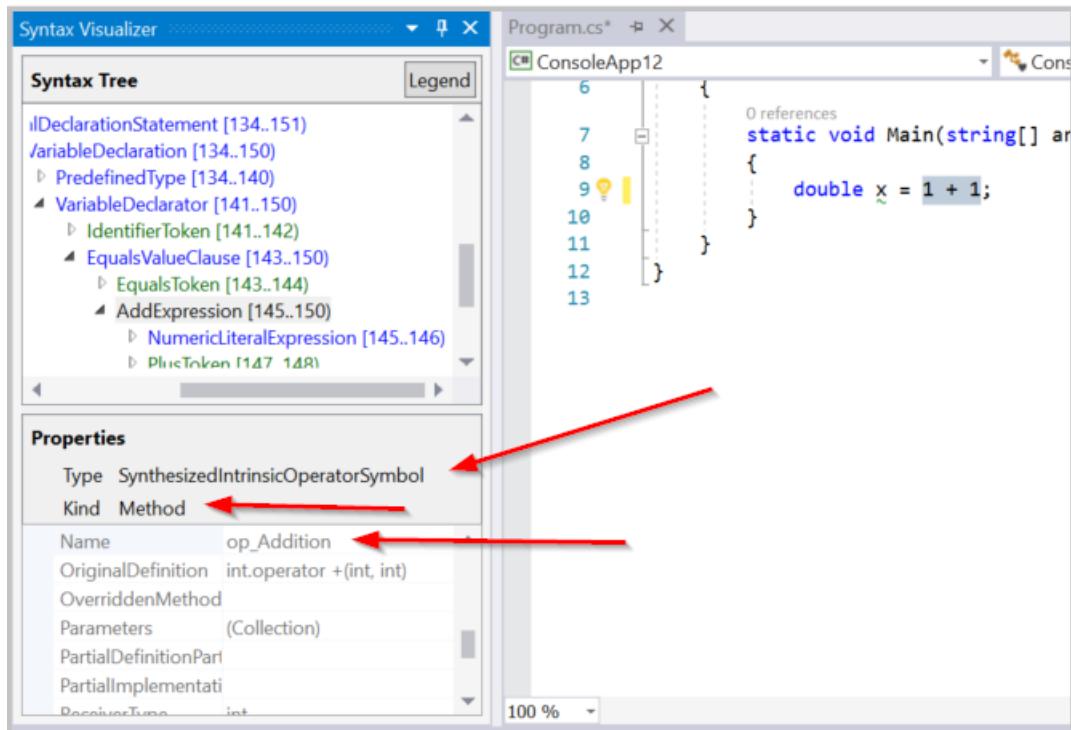
또 다른 음션은 듀얼 모니터 환경에서 두 번째 모니터에 구문 그래프 창을 배치하는 것입니다.

## 의미 체계 검사

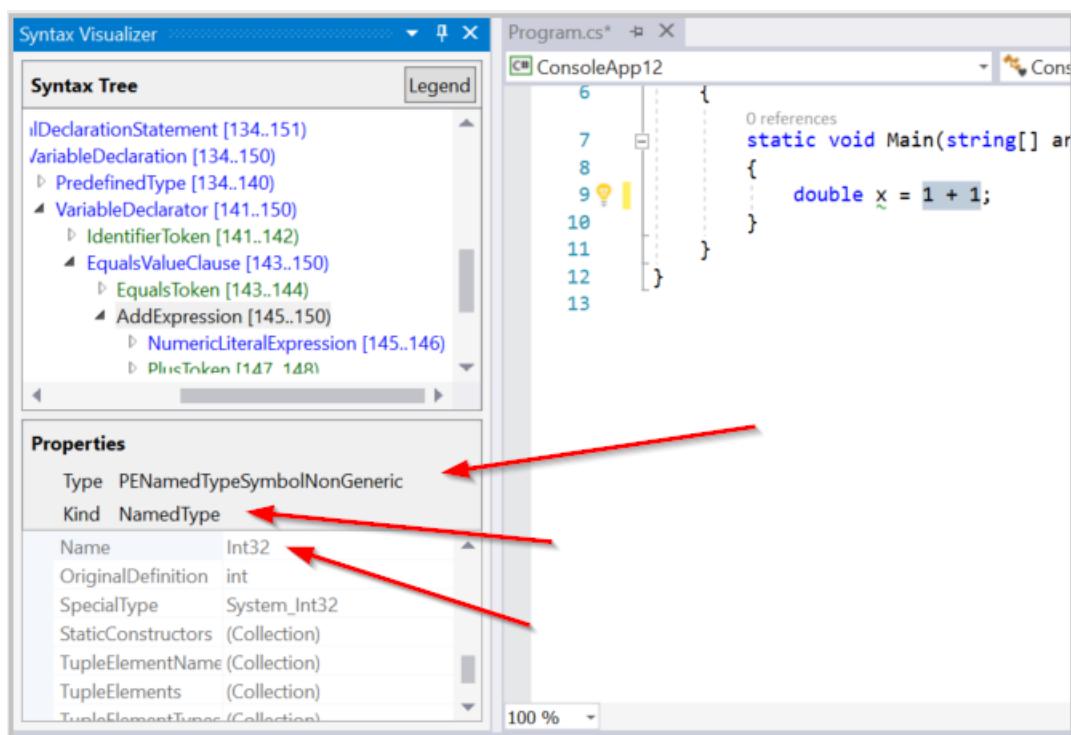
구문 시각화 도우미는 기호 및 의미 체계 정보에 대한 기본 검사를 수행합니다. C# 예제에서는 Main() 안에 `double x = 1 + 1;` 을 입력합니다. 그런 다음, 코드 편집기 창에서 식 `1 + 1`을 선택합니다. 시각화 도우미는 시각화 도우미에서 **AddExpression** 노드를 강조 표시합니다. 이 **AddExpression**을 마우스 오른쪽 단추로 클릭하고 **기호 보기(있는 경우)**를 클릭합니다. 대부분의 메뉴 항목에는 "해당되는 경우" 한정자가 있습니다. 구문 시각화 도우미는 모든 노드에 대해 나타나지 않을 수 있는 속성을 포함한 노드의 속성을 검사합니다.

식각화 도우미의 솔션 그리드는 다음 그림과 같이 업데이트됩니다. 식의 기호는 **Kind = Method**입니다.

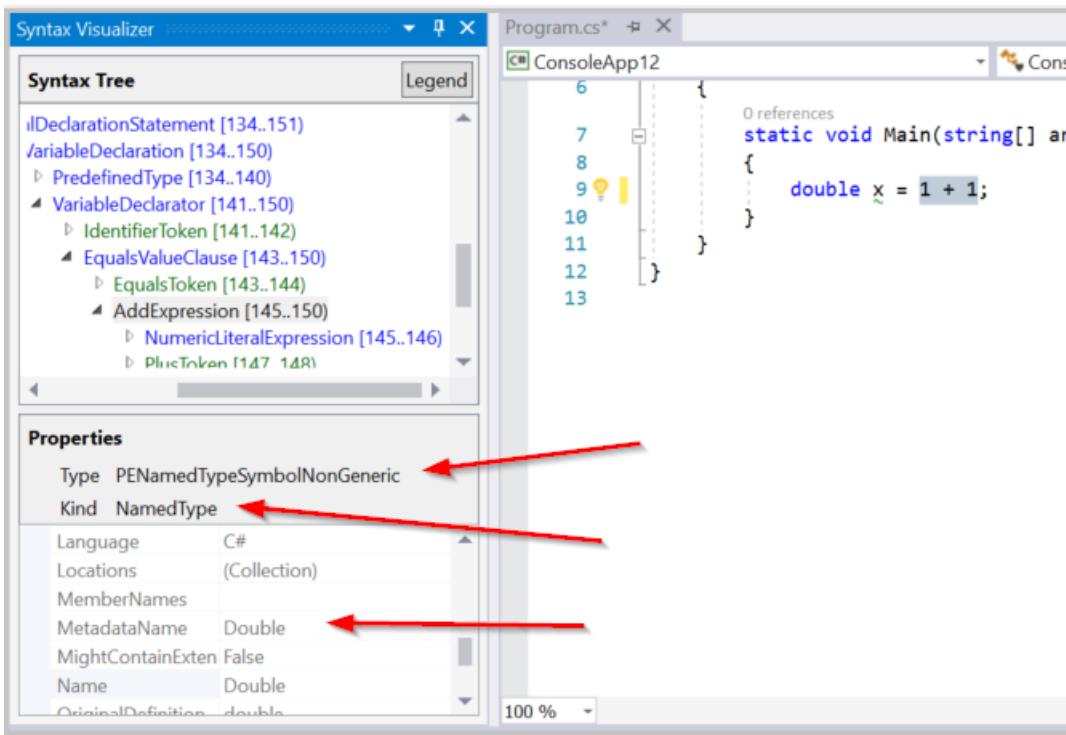
SynthesizedIntrinsicOperatorSymbol입니다.



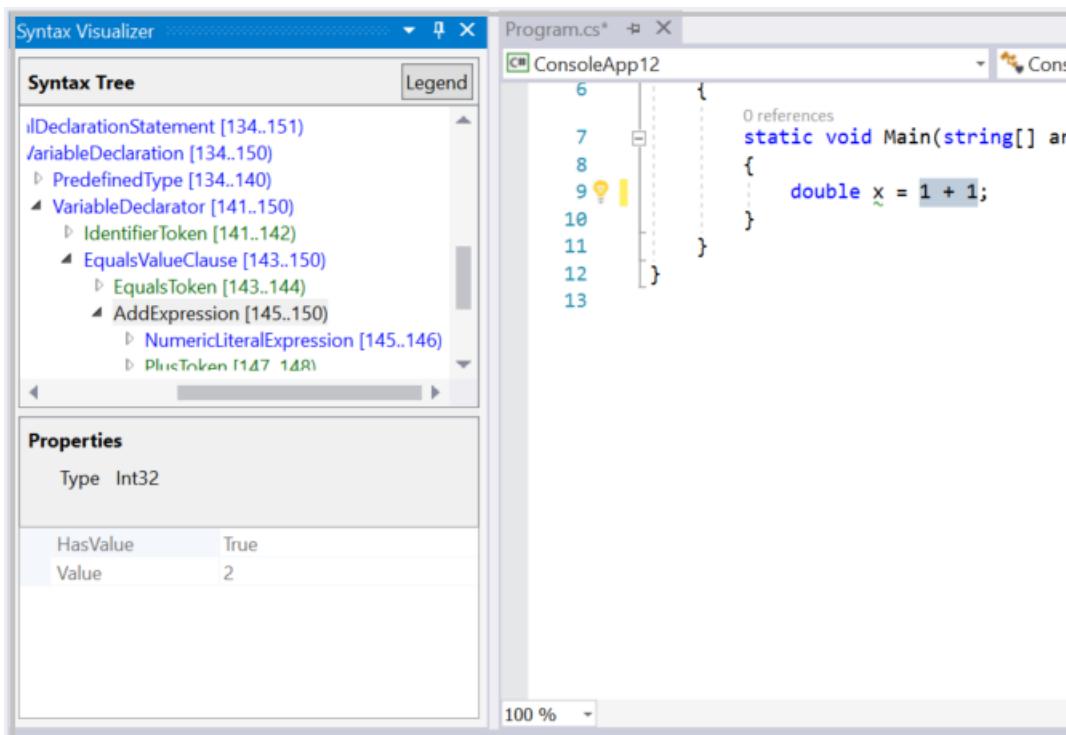
동일한 AddExpression 노드에 대해 TypeSymbol 보기(있는 경우)를 시도합니다. 시각화 도우미의 속성 그리드가 다음 그림과 같이 업데이트되어 선택한 식의 형식이 Int32 임을 나타냅니다.



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 Double로 변환되어야 하는 컨텍스트에서 발생하므로 이 노드에는 변환된 형식 기호 정보가 포함됩니다. 이 변환은 대입 연산자의 왼쪽에 있는 변수 x에 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.

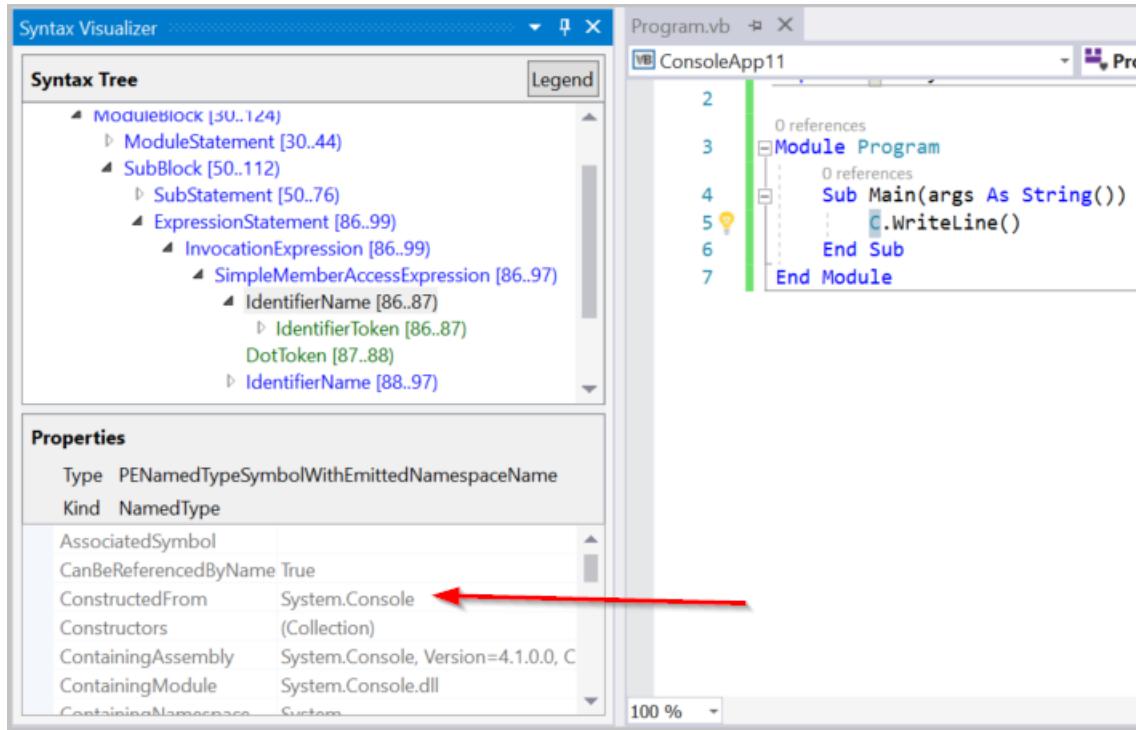
```

Imports C = System.Console

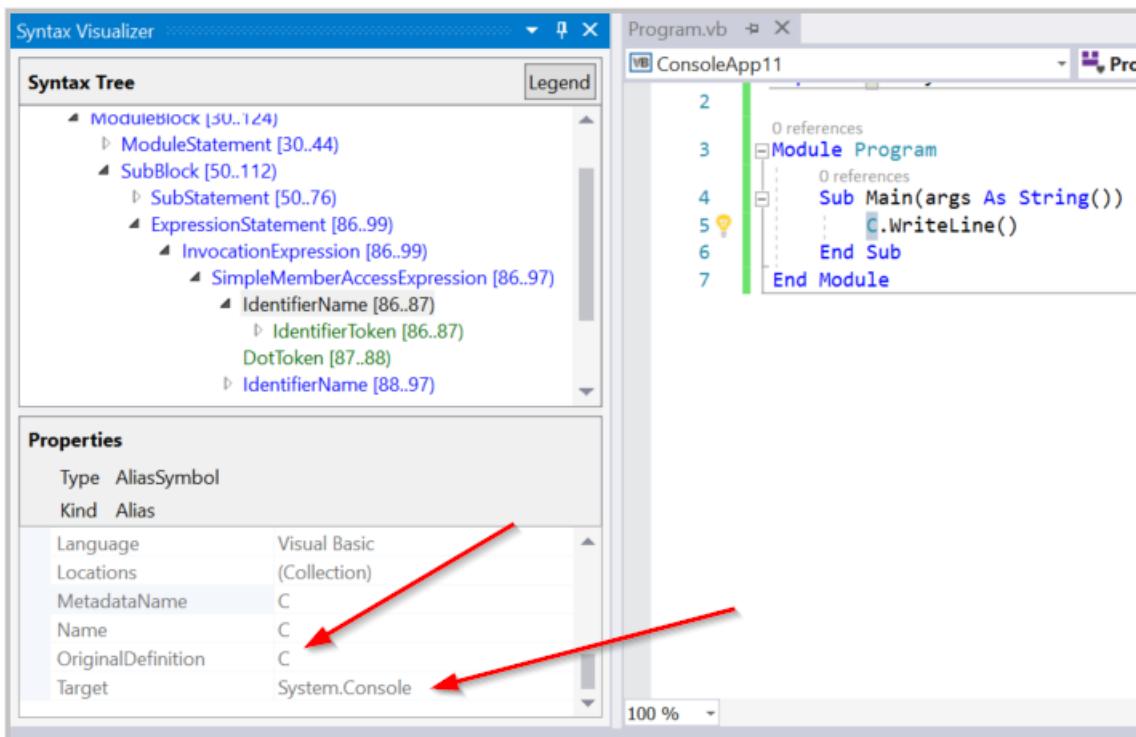
Module Program
    Sub Main(args As String())
        C.WriteLine()
    End Sub
End Module

```

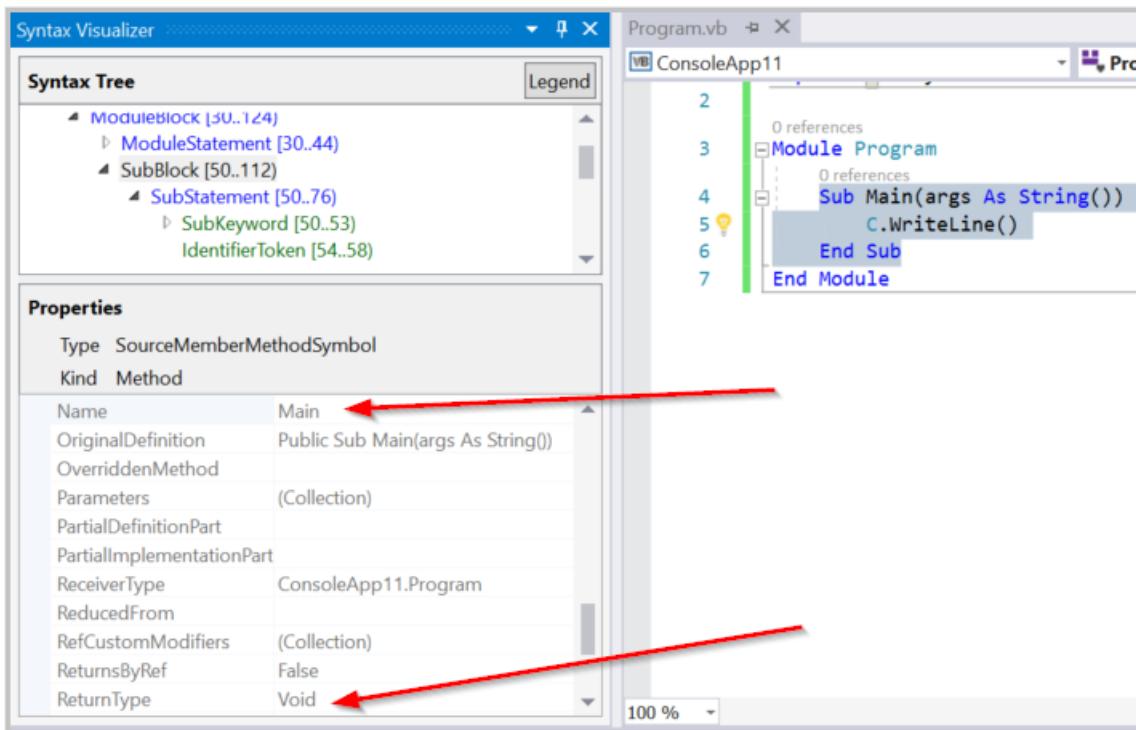
이 코드에서는 파일 위에 있는 `System.Console` 형식에 매핑되는 `c`라는 별칭을 소개하고 `Main()` 내부에서 이 별칭을 사용합니다. `Main()` 메서드 내에서 이 별칭(`C.WriteLine()`의 `C`)의 사용을 선택합니다. 시각화 도우미는 시각화 도우미에서 해당 **IdentifierName** 노드를 선택합니다. 이 노드를 마우스 오른쪽 단추로 클릭하고 **기호 보기(있는 경우)**를 클릭합니다. 속성 그리드는 다음 그림과 같이 이 식별자가 형식 `System.Console`에 바인딩되어 있음을 나타냅니다.



동일한 **IdentifierName** 노드에 대해 **AliasSymbol** 보기(있는 경우)를 시도합니다. 속성 그리드는 식별자가 `System.Console` 대상에 바인딩된 이름이 `c`인 별칭임을 나타냅니다. 즉, 속성 그리드는 식별자 `c`에 해당하는 **AliasSymbol**에 대한 정보를 제공합니다.



선언된 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 **SubBlock**의 **MethodSymbol** 이름이 **Main**이고 반환 형식이 **Void**임을 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 흥미있는 의미 체계 정보가 있는 것은 아닙니다. 노드에 흥미로운 의미 체계 정보가 없을 경우 \* **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 업데이트합니다. 사용하지 않을 때 혼란을 가져올 수 있습니다.

# 구문 분석 시작

2020-11-02 • 36 minutes to read • [Edit Online](#)

이 자습서에서는 구문 API를 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.

```
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 단위를 나타냅니다. 해당 원본 파일의 처음 세 줄은 **using** 지시문입니다. 나머지 원본은 네임스페이스 선언에 포함됩니다. 네임스페이스 선언에는 자식 **클래스** 선언이 포함됩니다. 클래스 선언에는 하나의 메서드 선언이 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 수준인 문자열 "Hello World!"까지 계속되고 인수의 하위 항목인 문자열 리터럴 토큰입니다. 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 들어쓰기를 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 SDK**를 설치해야 합니다.

## 설치 지침 - Visual Studio 설치 관리자

Visual Studio 설치 관리자에서 **.NET Compiler Platform SDK**를 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 관리자**를 실행합니다.
2. **수정**을 선택합니다.
3. **Visual Studio 확장 개발 워크로드**를 확인합니다.
4. **요약 트리**에서 **Visual Studio 확장 개발 노드**를 엽니다.
5. **.NET Compiler Platform SDK**에 대한 확인란을 선택합니다. 선택적 구성 요소 아래에서 마지막에 찾을 수 있습니다.

필요에 따라, 시각화 도우미에서 **DGML 편집기**에 그래프도 표시할 수 있습니다.

1. **요약 트리**에서 **개별 구성 요소 노드**를 엽니다.
2. **DGML 편집기 확인란**을 선택합니다.

#### **Visual Studio 설치 관리자를 사용한 설치 - 개별 구성 요소 탭**

1. **Visual Studio 설치 관리자**를 실행합니다.
2. **수정**을 선택합니다.
3. **개별 구성 요소 탭**을 선택합니다.
4. **.NET Compiler Platform SDK**에 대한 확인란을 선택합니다. 컴파일러, 빌드 도구 및 런타임 섹션의 위쪽에서 찾을 수 있습니다.

필요에 따라, 시각화 도우미에서 **DGML 편집기**에 그래프도 표시할 수 있습니다.

1. **DGML 편집기 확인란**을 선택합니다. 코드 도구 섹션에서 찾을 수 있습니다.

## 구문 트리 이해

C# 코드 구조의 분석에 구문 API를 사용합니다. 구문 API는 구문 트리를 분석하고 생성하기 위한 파서, 구문 트리 및 유ти리티를 노출합니다. 그렇게 특정 구문 요소에 대한 코드를 검색하거나 프로그램에 대한 코드를 읽을 수 있습니다.

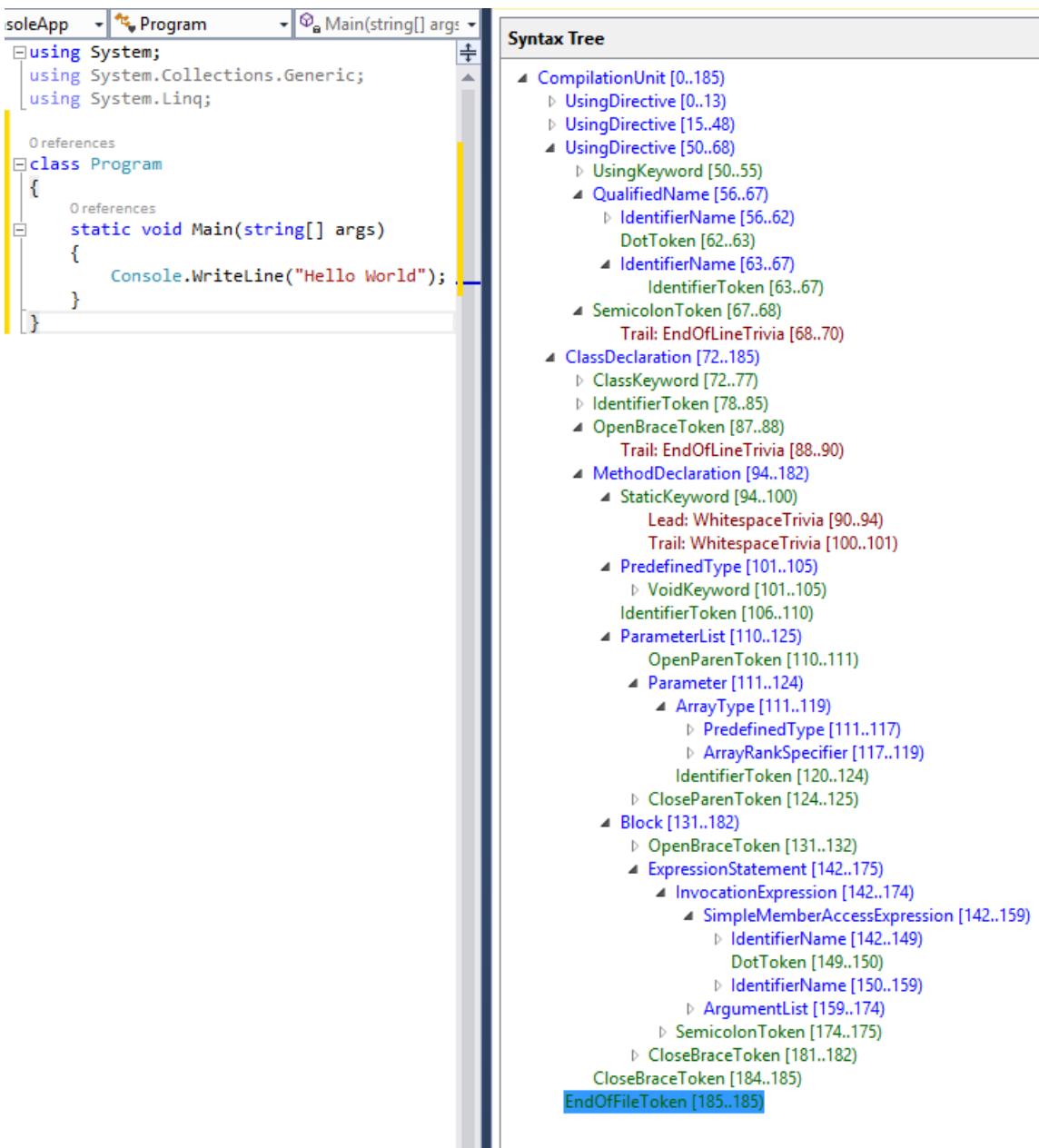
구문 트리는 C# 및 Visual Basic 프로그램을 이해하기 위해 C# 및 Visual Basic 컴파일러에서 사용하는 데이터 구조입니다. 구문 트리는 프로젝트가 빌드되거나 개발자가 F5 키를 누를 때 실행되는 동일한 파서에서 생성됩니다. 구문 트리는 언어를 완전히 제공합니다. 코드 파일의 모든 정보가 트리에 표시됩니다. 텍스트에 대한 구문 트리를 작성하면 구문 분석된 정확한 원본 텍스트를 재현합니다. 또한 구문 트리는 변경할 수 없습니다. 만들어진 구문 트리는 변경할 수 없습니다. 트리의 소비자는 잠금 또는 기타 동시성 조치 없이 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 도우미를 선택합니다. 예를 들어 구문 시각화 도우미를 사용하여 검사된 위의 C# 원본 파일은 다음 그림처럼 표시됩니다.

**SyntaxNode:** 파랑 | **SyntaxToken:** 초록색 | **SyntaxTrivia:** 빨강



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.

## NOTE

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.

앞에 표시된 기본 "Hello World!" 프로그램을 분석하려고 합니다. Hello World 프로그램의 텍스트를 Program 클래스의 상수로 추가합니다.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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.

```
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.ToString());

var argsParameter = mainDeclaration.ParameterList.Parameters[0];
```

프로그램을 실행하여 이 프로그램에 대해 알게 된 모든 정보를 확인합니다.

```
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.

```
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.

```

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.

```

SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();

```

다음으로 새 클래스를 만듭니다. Visual Studio에서 프로젝트 > 새 항목 추가를 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.

```

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.

```

public ICollection<UsingDirectiveSyntax> Usings { get; } = new List<UsingDirectiveSyntax>();

```

[CSharpSyntaxWalker](#) 기본 클래스는 구문 트리에서 각 노드를 방문하는 논리를 구현합니다. 파생된 클래스는 관

심이 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.

```
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. 호출할 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.

```
var collector = new UsingCollector();
collector.Visit(root);
foreach (var directive in collector.Usings)
{
    WriteLine(directive.Name);
}
```

프로그램을 컴파일하고 실행합니다. 다음과 같은 내용이 출력됩니다.

```
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 API를 사용하여 C# 소스 코드에서 특정 종류의 C# 문 및 선언을 찾았습니다.

# 의미 체계 분석 시작

2020-05-20 • 23 minutes to read • [Edit Online](#)

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 체계\_에 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. 수정을 선택합니다.
3. **Visual Studio** 확장 개발 워크로드를 확인합니다.
4. 요약 트리에서 **Visual Studio** 확장 개발 노드를 엽니다.
5. **.NET Compiler Platform SDK**에 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. 수정을 선택합니다.
3. **개별 구성 요소** 탭을 선택합니다.
4. **.NET Compiler Platform SDK**에 대한 확인란을 선택합니다. 컴파일러, 빌드 도구 및 런타임 섹션의 위쪽에서 찾을 수 있습니다.

필요에 따라, 시각화 도우미에서 **DGML 편집기**에 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 \_구조\_를 볼 수 있습니다. 그러나 프로그램의 의미 체계 또는 \_의미\_에 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 컴

파일하는 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 바인딩이라고 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.

### NOTE

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.

앞에 표시된 기본 "Hello World!" 프로그램을 분석하려고 합니다. Hello World 프로그램의 텍스트를 [Program](#) 클래스의 상수로 추가합니다.

```
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.

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);

CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

다음으로 이미 만든 트리에서 [CSharpCompilation](#)을 빌드합니다. "Hello World" 샘플은 [String](#) 및 [Console](#) 형식으로 사용합니다. 컴파일에서 두 가지 해당 형식을 선언하는 어셈블리를 참조해야 합니다. 다음 줄을 [Main](#) 메서드에 추가하여 적절한 어셈블리에 대한 참조를 비롯한 구문 트리의 컴파일을 만듭니다.

```
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 [IntelliSense](#)에서 가져오는 경우 의미 체계 모델을 고려할 수 있습니다.

[SemanticModel](#)은 "이 위치에서 범위에 있는 이름은 무엇입니까?", "이 메서드에서 어떤 멤버에 액세스할 수 있습니까?", "이 텍스트의 블록에서 사용되는 변수는 무엇입니까?" 및 "이 이름/식은 무엇을 참조합니까?"와 같은 질문에 대답할 수 있습니다. 의미 체계 모델을 만드는 이 문을 추가합니다.

```
SemanticModel model = compilation.GetSemanticModel(tree);
```

## 이름 바인딩

[Compilation](#)은 [SyntaxTree](#)에서 [SemanticModel](#)을 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.

```
// 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](#)를 검색하는 방법을 보여 줍니다. 또한 위의 코드에서는 구문 모델을 사용하여 코드의 구조를 찾는 것을 설명합니다. 의미 체계 모델을 사용하여 해당 의미를 이해합니다. 구문 모델은 [using](#) 문에서 [System](#) 문자열을 찾습니다. 의미 체계 모델에는 [System](#) 네임스페이스에서 정의된 형식에 대한 모든 정보가 있습니다.

[SymbolInfo](#) 개체에서 [SymbolInfo.Symbol](#) 속성을 사용하여 [Microsoft.CodeAnalysis.ISymbol](#)를 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.

```
var systemSymbol = (INamespaceSymbol)nameInfo.Symbol;
foreach (INamespaceSymbol ns in systemSymbol.GetNamespaceMembers())
{
    Console.WriteLine(ns);
}
```

프로그램을 실행하고 다음과 같은 출력이 표시됩니다.

```
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 . . .
```

#### NOTE

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 문자열 리터럴에 대한 바인딩에 액세스하겠습니다.

"Hello World" 프로그램에는 콘솔에 표시된 `Microsoft.CodeAnalysis.CSharp.Syntax.LiteralExpressionSyntax`, "Hello, World!" 문자열이 있습니다.

프로그램에 단일 리터럴 문자열을 배치하여 "Hello, World!" 문자열을 찾습니다. 그런 다음, 구문 노드를 찾으면 의미 체계 모델에서 노드의 형식 정보를 가져옵니다. `Main` 메서드에 다음 코드를 추가합니다.

```
// 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.

```
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.

```
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.

```
var methods = allMembers.OfType<IMethodSymbol>();
```

다음으로 공용이며 `string`을 반환하는 해당 메서드만을 반환하는 다른 필터를 추가합니다.

```
var publicStringReturningMethods = methods
    .Where(m => m.ReturnType.Equals(stringTypeSymbol) &&
    m.DeclaredAccessibility == Accessibility.Public);
```

이름 속성만을 선택하고, 오버로드를 제거하여 고유 이름만을 선택합니다.

```
var distinctMethods = publicStringReturningMethods.Select(m => m.Name).Distinct();
```

LINQ 쿼리 구문을 사용하여 전체 쿼리를 빌드한 다음, 콘솔에서 모든 메서드 이름을 표시할 수 있습니다.

```
foreach (string name in (from method in stringTypeSymbol
                           .GetMembers().OfType<IMethodSymbol>()
                           where method.ReturnType.Equals(stringTypeSymbol) &&
                           method.DeclaredAccessibility == Accessibility.Public
                           select method.Name).Distinct())
{
    Console.WriteLine(name);
}
```

프로그램을 빌드하고 실행합니다. 다음과 같은 내용이 출력됩니다.

```
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

2020-11-02 • 35 minutes to read • [Edit Online](#)

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. **수정**을 선택합니다.
3. **Visual Studio** 확장 개발 워크로드를 확인합니다.
4. 요약 트리에서 **Visual Studio 확장 개발** 노드를 엽니다.
5. **.NET Compiler Platform SDK**에 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 SDK**에 대한 확인란을 선택합니다. 컴파일러, 빌드 도구 및 런타임 섹션의 위쪽에서 찾을 수 있습니다.

필요에 따라, 시각화 도우미에서 **DGML 편집기**에 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 정보에 대해 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.

```
using static Microsoft.CodeAnalysis.CSharp.SyntaxFactory;
using static System.Console;
```

`name syntax nodes`를 만들어 `using System.Collections.Generic;` 문을 나타내는 트리를 빌드합니다.

`NameSyntax`은 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 품 `<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.cs`에서 `Main` 메서드에 다음 코드를 추가합니다.

```
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. `QualifiedNameSyntax`를 빌드할 때 `name` 변수인 `NameSyntax`를 재사용할 수 있습니다. 샘플을 빌드할 때 형식 유추를 사용하지 마세요. 이 프로젝트에서 해당 단계를 자동화합니다.

이름을 만들었습니다. 이제 `QualifiedNameSyntax`를 빌드하여 더 많은 노드를 트리로 빌드해 보겠습니다. 새 트

리는 `name`을 이름의 왼쪽으로 사용하고 `Collections` 네임스페이스의 새 [IdentifierNameSyntax](#)를 [QualifiedNameSyntax](#)의 오른쪽으로 사용합니다. 다음 코드를 `program.cs`에 추가합니다.

```
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.

```
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 만드는데는 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 API는 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.

```
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!");
        }
    }
}";
```

### NOTE

예제 코드는 `System.Collections.Generic` 네임스페이스가 아닌 `System.Collections` 네임스페이스를 사용합니다.

다음으로 `Main` 메서드의 맨 아래에 다음 코드를 추가하여 텍스트를 구문 분석하고 트리를 만듭니다.

```
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.

```
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.

```
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.

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Syntax;
```

다음으로 [TypeInferenceRewriter](#) 클래스가 [CSharpSyntaxRewriter](#) 클래스를 확장하도록 합니다.

```
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.

```
private readonly SemanticModel SemanticModel;

public TypeInferenceRewriter(SemanticModel semanticModel) => SemanticModel = semanticModel;
```

[VisitLocalDeclarationStatement](#)([LocalDeclarationStatementSyntax](#)) 메서드를 재정의합니다.

```
public override SyntaxNode VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax node)
{
}
```

#### NOTE

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 [VisitLocalDeclarationStatement](#)([LocalDeclarationStatementSyntax](#)) 메서드는 [LocalDeclarationStatementSyntax](#) 파생 형식 대신 [SyntaxNode](#)를 반환합니다. 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.

```
Type variable = expression;
```

혼자서 살펴보려면 다음 형식의 변수 선언에 대한 완료된 샘플을 확장해 보세요.

```
// 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.

```
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.

```
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.

```
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.

```
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.

```
Compilation test = CreateTestCompilation();
```

잠시 후에 `CreateTestCompilation` 메서드가 없음을 보고하는 오류 물결선이 표시됩니다. **Ctrl+마침표**를 눌러 전구를 열고 **Enter** 키를 눌러 메서드 스텝 생성 명령을 호출합니다. 이 명령은 `Program` 클래스에서 `CreateTestCompilation` 메서드에 대한 메서드 스텝을 생성합니다. 나중에 돌아와서 이 메서드를 입력합니다.

The screenshot shows a Visual Studio code editor with two tabs: 'Program.cs' and 'TypeInferenceRewriter.cs'. The 'Program.cs' tab is active, displaying the following C# code:

```

6  namespace TransformationCS
7  {
8      class Program
9      {
10         static void Main(string[] args)
11         {
12             Compilation test = CreateTestCompilation();
13         }
14     }
15 }

```

A red squiggly underline appears under the call to 'CreateTestCompilation()'. A tooltip for 'Generate method 'Program.CreateTestCompilation'' is shown, along with an error message: 'CS0103 The name 'CreateTestCompilation' does not exist in the current context'. Below the tooltip, a preview of the generated code is shown:

```

...
private static Compilation CreateTestCompilation()
{
    throw new NotImplementedException();
}
...

```

**Preview changes**

다음 코드를 작성하여 테스트 `Compilation`에서 각 `SyntaxTree`를 반복합니다. 각 트리에 대해 해당 트리의 `SemanticModel`을 사용하여 새 `TypeInferenceRewriter`를 초기화합니다.

```

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.

```

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.

```

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 mscorelib =
    MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
MetadataReference codeAnalysis =
    MetadataReference.CreateFromFile(typeof(SyntaxTree).Assembly.Location);
MetadataReference csharpCodeAnalysis =
    MetadataReference.CreateFromFile(typeof(CSharpSyntaxTree).Assembly.Location);

MetadataReference[] references = { mscorelib, codeAnalysis, csharpCodeAnalysis };

return CSharpCompilation.Create("TransformationCS",
    sourceTrees,
    references,
    new CSharpCompilationOptions(OutputKind.ConsoleApplication));

```

행운을 빌고 프로젝트를 실행합니다. Visual Studio에서 디버그 > 디버깅 시작을 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 API를 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

2020-11-02 • 65 minutes to read • [Edit Online](#)

.NET Compiler Platform SDK는 C# 또는 Visual Basic 코드를 대상으로 하는 사용자 지정 경고를 만드는데 필요한 도구를 제공합니다. 분석기에는 규칙 위반을 인식하는 코드가 포함됩니다. 코드 수정 사항에는 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 사항을 만드는 방법을 살펴봅니다. 분석기는 소스 코드 분석을 수행하고 사용자에게 문제를 보고하는 방법입니다. 필요한 경우 분석기는 사용자의 소스 코드에 대한 수정 사항을 나타내는 코드 수정 사항도 제공할 수 있습니다. 이 자습서에서는 `const` 한정자를 사용하여 선언할 수 있는 지역 변수 선언을 찾는 분석기를 만듭니다. 함께 제공되는 코드 수정 사항은 해당 선언을 수정하여 `const` 한정자를 추가합니다.

## 사전 요구 사항

- Visual Studio 2019 버전 16.7 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.

- Visual Studio 설치 관리자를 실행합니다.
- 수정을 선택합니다.
- Visual Studio 확장 개발 워크로드를 확인합니다.
- 요약 트리에서 Visual Studio 확장 개발 노드를 엽니다.
- .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.

- 요약 트리에서 개별 구성 요소 노드를 엽니다.
- DGML 편집기 확인란을 선택합니다.

### Visual Studio 설치 관리자를 사용한 설치 - 개별 구성 요소 탭

- Visual Studio 설치 관리자를 실행합니다.
- 수정을 선택합니다.
- 개별 구성 요소 탭을 선택합니다.
- .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.

## 분석기 템플릿 살펴보기

분석기는 지역 상수로 변환할 수 있는 모든 지역 변수 선언을 사용자에게 보고합니다. 예를 들어, 다음 코드를 고려하세요.

```
int x = 0;  
Console.WriteLine(x);
```

위의 코드에서 `x` 는 상수 값이 할당되고 수정되지 않습니다. `const` 키워드를 사용하여 선언할 수 있습니다.

```
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. 첫 번째 단계는 새로운 C# 코드 수정 사항이 포함된 분석기 프로젝트를 만드는 것입니다.

- Visual Studio에서 파일 > 새로 만들기 > 프로젝트... 를 선택하여 [새 프로젝트] 대화 상자를 표시합니다.
- Visual C# > 확장성에서 코드 수정 사항이 포함된 분석기(.NET Standard) 를 선택합니다.
- 프로젝트 이름을 "MakeConst"로 지정하고 [확인]을 클릭합니다.

코드 수정 사항 템플릿이 포함된 분석기는 세 개의 프로젝트를 만듭니다. 하나에는 분석기 및 코드 수정 사항이 포함되고, 두 번째는 단위 테스트 프로젝트이고, 세 번째는 VSIX 프로젝트입니다. 기본 시작 프로젝트는 VSIX 프로젝트입니다. F5 키를 눌러 VSIX 프로젝트를 시작합니다. 그러면 새 분석기를 로드한 Visual Studio의 두 번째 인스턴스가 시작됩니다.

### TIP

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.

방금 시작한 두 번째 Visual Studio 인스턴스에서 새 C# 콘솔 애플리케이션 프로젝트를 만듭니다(.NET Core 또는 .NET Framework 프로젝트가 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.

#### TIP

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.

## 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 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 노드를 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.

- `AnalyzerTitle` 을 "Variable can be made constant"(변수를 상수로 설정할 수 있음)로 변경합니다.
- `AnalyzerMessageFormat` 을 "Can be made constant"(상수로 설정할 수 있음)로 변경합니다.
- `AnalyzerDescription` 을 "Make Constant"(상수 만들기)로 변경합니다.

또한 액세스 한정자 드롭다운을 `public` 으로 변경합니다. 이렇게 하면 단위 테스트에서 이러한 상수를 더 쉽게 사용할 수 있습니다. 작업을 마치면 리소스 편집기가 다음 그림과 같이 표시됩니다.

Name	Value	Comment
AnalyzerTitle	Variable can be made constant.	The title of the diagnostic.
AnalyzerMessageFormat	can be made constant	The format-able message the diagnostic displays.
▶ AnalyzerDescription	Make Constant.	An optional longer localizable description 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.

```
context.RegisterSymbolAction>AnalyzeSymbol, SymbolKind.NamedType);
```

해당 줄을 다음 줄로 바꿉니다.

```
context.ConfigureGeneratedCodeAnalysis(GeneratedCodeAnalysisFlags.Analyze |  
GeneratedCodeAnalysisFlags.None);  
context.EnableConcurrentExecution();  
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.

```
private void AnalyzeNode(SyntaxNodeAnalysisContext context)  
{  
}
```

다음 코드에 표시된 대로 `MakeConstAnalyzer.cs` 에서 `Category` 를 "Usage"(사용법)로 변경합니다.

```
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.

```
int x = 0;  
Console.WriteLine(x);
```

첫 번째 단계는 로컬 선언을 찾는 것입니다. `MakeConstAnalyzer.cs` 에서 `AnalyzeNode` 에 다음 코드를 추가합니다.

```
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.

```
// 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 모델](#)을 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.

```
// Perform data flow analysis on the local declaration.  
var 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.  
var variable = localDeclaration.Declaration.Variables.Single();  
var variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable);  
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.

```
context.ReportDiagnostic(Diagnostic.Create(Rule, context.Node.GetLocation()));
```

F5 키를 눌러 분석기를 실행하여 진행 상황을 확인할 수 있습니다. 이전에 만든 콘솔 애플리케이션을 로드한 후 다음 테스트 코드를 추가할 수 있습니다.

```
int x = 0;  
Console.WriteLine(x);
```

전구가 표시되고 분석기가 진단을 보고해야 합니다. 그러나 전구는 템플릿에서 생성된 코드 수정 사항을 계속 사용하고 대문자로 설정될 수 있음을 알려 줍니다. 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.

```
const int x = 0;  
Console.WriteLine(x);
```

사용자가 편집기에서 전구 UI를 선택하면 Visual Studio가 코드를 변경합니다.

템플릿에서 추가된 `MakeConstCodeFixProvider.cs` 파일을 엽니다. 이 코드 수정 사항은 진단 분석기에서 생성된 진단 ID에 연결되어 있지만 아직 적합한 코드 변환을 구현하지 않습니다. 먼저 일부 템플릿 코드를 제거해야 합니다. 제목 문자열을 "Make constant"(상수 만들기)로 변경합니다.

```
private const string title = "Make constant";
```

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.

```
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` 키워드를 추가함으로써 생성되는 새 문서를 만듭니다.

```
// Register a code action that will invoke the fix.
context.RegisterCodeFix(
    CodeAction.Create(
        title: title,
        createChangedDocument: c => MakeConstAsync(context.Document, declaration, c),
        equivalenceKey: title),
    diagnostic);
```

방금 추가한 기호 `MakeConstAsync`에 대한 코드에 빨간색 물결선이 표시됩니다. 다음 코드와 같이 `MakeConstAsync`에 대한 선언을 추가합니다.

```
private 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 선형 trivia를 제거하고 이를 `const` 토큰에 연결해야 합니다. `MakeConstAsync` 메서드에 다음 코드를 추가합니다.

```
// Remove the leading trivia from the local declaration.
var firstToken = localDeclaration.GetFirstToken();
var leadingTrivia = firstToken.LeadingTrivia;
var trimmedLocal = localDeclaration.ReplaceToken(
    firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));

// Create a const token with the leading trivia.
var constToken = SyntaxFactory.Token(leadingTrivia, SyntaxKind.ConstKeyword,
    SyntaxFactory.TriviaList(SyntaxFactory.ElasticMarker));
```

그런 다음, 다음 코드를 사용하여 `const` 토큰을 선언에 추가합니다.

```
// Insert the const token into the modifiers list, creating a new modifiers list.  
var newModifiers = trimmedLocal.Modifiers.Insert(0, constToken);  
// Produce the new local declaration.  
var 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.

```
// Add an annotation to format the new local declaration.  
var formattedLocal = newLocal.WithAdditionalAnnotations(Formatter.Annotation);
```

이 코드에는 새 네임스페이스가 필요합니다. 다음 `using` 지시문을 파일의 맨 위에 추가합니다.

```
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.

```
// Replace the old local declaration with the new local declaration.  
var oldRoot = await document.GetSyntaxRootAsync(cancellationToken);  
var 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 추가하면 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.

## 데이터 기반 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.

분석기의 거의 모든 테스트에 대한 코드는 이러한 두 패턴 중 하나를 따릅니다. 첫 번째 단계에서는 이러한 테스트를 데이터 기반 테스트로 재작업할 수 있습니다. 그런 다음, 다른 테스트 입력을 나타내기 위한 새 문자열 상수를 추가하여 새 테스트를 쉽게 만들 수 있습니다.

효율성을 위해 첫 번째 단계는 두 테스트를 데이터 기반 테스트로 리팩터링하는 것입니다. 그런 다음, 각각 새 테스트에 대한 두 개의 문자열 상수를 정의하면 됩니다. 리팩터링하는 동안 두 메서드의 이름을 더 나은 이름으로 바꿉니다. **TestMethod1**을 진단이 실행되지 않는지 확인하는 이 테스트로 바꿉니다.

```
[DataTestMethod]
[DataRow("")]
public void WhenTestCodeIsValidNoDiagnosticIsTriggered(string testCode)
{
    VerifyCSharpDiagnostic(testCode);
}
```

진단이 경고를 트리거하지 않도록 해야 하는 코드 조각을 정의하여 이 테스트에 대한 새 데이터 행을 만들 수 있습니다. **VerifyCSharpDiagnostic**의 이 오버로드는 소스 코드 조각에 대해 트리거된 진단이 없는 경우에 성공합니다.

다음으로, **TestMethod2**를 진단이 실행되고 소스 코드 조각에 대한 코드 수정 사항이 적용되는지 확인하는 이 테스트로 바꿉니다.

```
[DataTestMethod]
[DataRow(LocalIntCouldBeConstant, LocalIntCouldBeConstantFixed, 10, 13)]
public void WhenDiagnosticIsRaisedFixUpdatesCode(
    string test,
    string fixTest,
    int line,
    int column)
{
    var expected = new DiagnosticResult
    {
        Id = MakeConstAnalyzer.DiagnosticId,
        Message = new LocalizableResourceString(nameof(MakeConst.Resources.AnalyzerMessageFormat),
            MakeConst.Resources.ResourceManager, typeof(MakeConst.Resources)).ToString(),
        Severity = DiagnosticSeverity.Warning,
        Locations =
        new[] {
            new DiagnosticResultLocation("Test0.cs", line, column)
        }
    };

    VerifyCSharpDiagnostic(test, expected);

    VerifyCSharpFix(test, fixTest);
}
```

이전 코드에서도 예상 진단 결과를 빌드하는 두 개의 변경 내용을 코드에 적용했습니다. 이전 코드는 **MakeConst** 분석기에 등록된 공용 상수를 사용합니다. 또한 입력 및 수정된 소스에 두 개의 문자열 상수를 사용합니다. **UnitTest** 클래스에 다음 문자열 상수를 추가합니다.

```

private const string LocalIntCouldBeConstant = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            Console.WriteLine(i);
        }
    }
}"';

private const string LocalIntCouldBeConstantFixed = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            const int i = 0;
            Console.WriteLine(i);
        }
    }
}"';

```

이러한 두 테스트를 실행하여 성공하는지 확인합니다. Visual Studio에서 테스트 > Windows > 테스트 탐색기 를 선택하여 테스트 탐색기 를 엽니다. 그런 다음, 모두 실행 링크를 선택합니다.

## 유용한 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. 다음 문자열 상수를 테스트에 추가하여 해당 사례를 나타냅니다.

```

private const string VariableAssigned = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;
            Console.WriteLine(i++);
        }
    }
}"';

```

그런 다음, 아래 코드 조각에 표시된 대로 이 테스트의 데이터 행을 추가합니다.

```
[DataTestMethod]
[DataRow("")],
[DataRow(VariateAssigned)]
public void WhenTestCodeIsValidNoDiagnosticIsTriggered(string testCode)
```

이 테스트도 성공합니다. 다음으로, 아직 처리하지 않은 조건에 대한 상수를 추가합니다.

- 이미 상수이므로 이미 `const` 인 선언:

```
private const string AlreadyConst = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            const int i = 0;
            Console.WriteLine(i);
        }
    }
}";
```

- 사용할 값이 없으므로 이니셜라이저가 없는 선언:

```
private const string NoInitializer = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            int i;
            i = 0;
            Console.WriteLine(i);
        }
    }
}";
```

- 컴파일 시간 상수일 수 없으므로 이니셜라이저가 상수가 아닌 선언:

```
private const string InitializerNotConstant = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            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 고

려합니다.

```
private const string MultipleInitializers = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0, j = DateTime.Now.DayOfYear;
            Console.WriteLine(i, j);
        }
    }
}";
```

i 변수는 상수로 설정될 수 있지만, j 변수는 상수로 설정될 수 없습니다. 따라서 이 문은 const 선언으로 설정될 수 없습니다. 다음 모든 테스트에 대한 DataRow 선언을 추가합니다.

```
[DataTestMethod]
[DataRow(""),
 DataRow(VariableAssigned),
 DataRow(AlreadyConst),
 DataRow(NoInitializer),
 DataRow(InitializerNotConstant),
 DataRow(MultipleInitializers)]
public void WhenTestCodeIsValidNoDiagnosticIsTriggered(string testCode)
```

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. AnalyzeNode에 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

```
// Perform data flow analysis on the local declaration.
var 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.
var variable = localDeclaration.Declaration.Variables.Single();
var variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

다음 코드 조각으로 바꿉니다.

```

// Ensure that all variables in the local declaration have initializers that
// are assigned with constant values.
foreach (var variable in localDeclaration.Declaration.Variables)
{
    var initializer = variable.Initializer;
    if (initializer == null)
    {
        return;
    }

    var constantValue = context.SemanticModel.GetConstantValue(initializer.Value);
    if (!constantValue.HasValue)
    {
        return;
    }
}

// Perform data flow analysis on the local declaration.
var dataFlowAnalysis = context.SemanticModel.AnalyzeDataFlow(localDeclaration);

foreach (var 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.
    var variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable);
    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"`와 같은 잘못된 선언을 로컬 상수로 변환합니다. 해당 조건에 대한 소스 문자열 상수를 추가합니다.

```

private const string DeclarationIsInvalid = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = ""abc"";;
        }
    }
}";

```

또한 참조 형식이 제대로 처리되지 않습니다. 참조 형식에 허용되는 유일한 상수 값은 문자열 리터럴을 허용하는 `System.String`의 이 경우를 제외하고 `null`입니다. 즉, `const string s = "abc"`는 적합하지만 `const object s = "abc"`는 적합하지 않습니다. 이 코드 조작은 해당 조건을 확인합니다.

```

private const string ReferenceTypeIsntString = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            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.

```

private const string ConstantIsString = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            string s = ""abc"";
        }
    }
}";

private const string ConstantIsStringFixed = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            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.

```

private const string DeclarationUsesVar = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            var item = 4;
        }
    }
}";

private const string DeclarationUsesVarFixedHasType = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            const int item = 4;
        }
    }
}";

private const string StringDeclarationUsesVar = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            var item = ""abc"";;
        }
    }
}";

private const string StringDeclarationUsesVarFixedHasType = @"
using System;

namespace MakeConstTest
{
    class Program
    {
        static void Main(string[] args)
        {
            const string item = ""abc"";;
        }
    }
}";

```

이러한 변경은 두 테스트에 대한 데이터 행 선언을 업데이트합니다. 다음 코드는 모든 데이터 행 특성을 사용하여 이러한 테스트를 보여 줍니다.

```

//No diagnostics expected to show up
[TestMethod]
[DataRow(""),
 DataRow(VariableAssigned),
 DataRow(AlreadyConst),
 DataRow(NoInitializer),
 DataRow(InitializerNotConstant),
 DataRow(MultipleInitializers),
 DataRow(DeclarationIsInvalid),
 DataRow(ReferenceTypeIsntString)]
public void WhenTestCodeIsValidNoDiagnosticIsTriggered(string testCode)
{
    VerifyCSharpDiagnostic(testCode);
}

[TestMethod]
[DataRow(LocalIntCouldBeConstant, LocalIntCouldBeConstantFixed, 10, 13),
 DataRow(ConstantIsString, ConstantIsStringFixed, 10, 13),
 DataRow(DeclarationUsesVar, DeclarationUsesVarFixedHasType, 10, 13),
 DataRow(StringDeclarationUsesVar, StringDeclarationUsesVarFixedHasType, 10, 13)]
public void WhenDiagnosticIsRaisedFixUpdatesCode(
    string test,
    string fixTest,
    int line,
    int column)
{
    var expected = new DiagnosticResult
    {
        Id = MakeConstAnalyzer.DiagnosticId,
        Message = new LocalizableResourceString(nameof(MakeConst.Resources.AnalyzerMessageFormat),
MakeConst.Resources.ResourceManager, typeof(MakeConst.Resources)).ToString(),
        Severity = DiagnosticSeverity.Warning,
        Locations =
        new[] {
            new DiagnosticResultLocation("Test0.cs", line, column)
        }
    };

    VerifyCSharpDiagnostic(test, expected);

    VerifyCSharpFix(test, fixTest);
}

```

다행히도 위의 버그는 모두 방금 알아본 동일한 기술을 사용하여 해결할 수 있습니다.

첫 번째 버그를 수정하려면 먼저 **DiagnosticAnalyzer.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.

```

var variableTypeName = localDeclaration.Declaration.Type;
var variableType = context.SemanticModel.GetTypeInfo(variableTypeName).ConvertedType;

```

그런 다음, `foreach` 루프 내부에서 각 이니셜라이저를 검사하여 변수 형식으로 변환할 수 있는지 확인합니다. 이니셜라이저가 상수인지 확인한 후 다음 검사를 추가합니다.

```

// Ensure that the initializer value can be converted to the type of the
// local declaration without a user-defined conversion.
var 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.

```
// 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. `CodeFixProvider.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.

```

// If the type of the declaration is 'var', create a new type name
// for the inferred type.
var variableDeclaration = localDeclaration.Declaration;
var variableTypeName = variableDeclaration.Type;
if (variableTypeName.IsVar)
{
    var semanticModel = await document.GetSemanticModelAsync(cancellationToken);

    // Special case: Ensure that 'var' isn't actually an alias to another type
    // (e.g. using var = System.String).
    var aliasInfo = semanticModel.GetAliasInfo(variableTypeName);
    if (aliasInfo == null)
    {
        // Retrieve the type inferred for var.
        var type = semanticModel.GetTypeInfo(variableTypeName).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.
            var typeName = SyntaxFactory.ParseTypeName(type.ToString())
                .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
                .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

            // Add an annotation to simplify the type name.
            var simplifiedTypeName = typeName.WithAdditionalAnnotations(Simplifier.Annotation);

            // Replace the type in the variable declaration.
            variableDeclaration = variableDeclarationWithType(simplifiedTypeName);
        }
    }
}

// Produce the new local declaration.
var newLocal = trimmedLocal.WithModifiers(newModifiers)
    .WithDeclaration(variableDeclaration);

```

Simplifier 형식을 사용하려면 `using` 지시문을 하나 추가해야 합니다.

```
using Microsoft.CodeAnalysis.Simplification;
```

테스트를 실행하면 모두 성공합니다. 완료된 분석기를 직접 실행할 수 있습니다. Ctrl+F5를 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.

```
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`을 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](#)

# C# 프로그래밍 가이드

2020-11-02 • 2 minutes to read • [Edit Online](#)

이 섹션에서는 핵심 C# 언어 기능과 .NET을 통해 C#에서 액세스할 수 있는 기능에 대한 자세한 정보를 제공합니다.

이 섹션 대부분에서는 사용자가 C# 및 일반적인 프로그래밍 개념에 대해 이미 알고 있다고 가정합니다. 프로그래밍 또는 C# 초급자는 프로그래밍에 대한 사전 지식이 필요하지 않은 [C# 자습서 소개](#) 또는 [.NET In-Browser 자습서](#)를 참조할 수 있습니다.

특정 키워드, 연산자 및 전처리기 지시문에 대한 자세한 내용은 [C# 참조](#)를 참조하세요. C# 언어 사양에 대한 자세한 내용은 [C# 언어 사양](#)을 참조하세요.

## 프로그램 섹션

[C# 프로그램 내용](#)

[Main\(\)과 명령줄 인수](#)

## 언어 섹션

[문, 식, 연산자](#)

[유형](#)

[클래스 및 구조체](#)

[인터페이스](#)

[대리자](#)

[배열](#)

[문자열](#)

[속성](#)

[인덱서](#)

[이벤트](#)

[제네릭](#)

[반복기](#)

[LINQ 쿼리 식](#)

[네임스페이스](#)

[안전하지 않은 코드 및 포인터](#)

[XML 문서 주석](#)

## 플랫폼 섹션

[애플리케이션 도메인](#)

[.NET 어셈블리](#)

[특성](#)

[컬렉션](#)

[예외 및 예외 처리](#)

[파일 시스템 및 레지스트리\(C# 프로그래밍 가이드\)](#)

[상호 운용성](#)

리플렉션

## 참조

- C# 참조

# C# 프로그램 내부

2021-02-18 • 2 minutes to read • [Edit Online](#)

이 단원에서는 C# 프로그램의 일반적인 구조를 설명하고 표준 "Hello, World!" 예가 포함됩니다.

## 단원 내용

- [C# 프로그램의 일반적인 구조](#)
- [식별자 이름](#)
- [C# 코딩 규칙](#)

## 관련 단원

- [C# 시작](#)
- [C# 프로그래밍 가이드](#)
- [C# 참조](#)
- [샘플 및 자습서](#)

## C# 언어 사양

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

## 참조

- [C# 프로그래밍 가이드](#)

# C# 프로그램의 일반적인 구조(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

C# 프로그램은 하나 이상의 파일로 구성됩니다. 각 파일에는 0개 이상의 네임스페이스가 포함될 수 있습니다. 네임스페이스에는 다른 네임스페이스 외에 클래스, 구조체, 인터페이스, 열거형 및 대리자 같은 형식이 포함될 수 있습니다. 다음은 이러한 모든 요소를 포함하는 C# 프로그램의 기본 구조입니다.

```
// 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 YourMainClass
    {
        static void Main(string[] args)
        {
            //Your program starts here...
        }
    }
}
```

## 관련 단원

추가 정보

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

- 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.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [C# 프로그램 내용](#)
- [C# 참조](#)

# 식별자 이름

2020-03-18 • 4 minutes to read • [Edit Online](#)

식별자는 형식(클래스, 인터페이스, 구조체, 대리자 또는 열거형), 멤버, 변수 또는 네임스페이스에 할당하는 이름입니다. 유효한 식별자는 다음 규칙을 따라야 합니다.

- 식별자는 문자 또는 `_`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. `@`는 식별자 이름의 일부가 아닙니다. 예를 들어 `@if`는 `if`라는 식별자를 선언합니다. 이러한 [verbatim](#) 식별자는 주로 다른 언어로 선언된 식별자와의 상호 운용성을 위한 것입니다.

유효한 식별자의 전체 정의는 [C# 언어 사양의 식별자 항목](#)을 참조하세요.

## 명명 규칙

규칙 외에도 .NET API 전체에서 사용되는 여러 식별자 [명명 규칙](#)이 있습니다. 규칙에 따라 C# 프로그램은 형식 이름, 네임스페이스 및 모든 공용 멤버에 `PascalCase`를 사용합니다. 또한, 다음 규칙이 일반적입니다.

- 인터페이스 이름은 대문자 `I`로 시작합니다.
- 특성 유형은 `Attribute` 단어로 끝납니다.
- 열거형 형식은 플래그가 아닌 경우에는 단수 명사를 사용하고 플래그인 경우에는 복수 명사를 사용합니다.
- 식별자에는 두 개의 연속 `_` 문자가 없어야 합니다. 이러한 이름은 컴파일러 생성 식별자용으로 예약되어 있습니다.

## C# 언어 사양

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

## 참조

- [C# 프로그래밍 가이드](#)
- [C# 프로그램 내부](#)
- [C# 참조](#)
- [클래스](#)
- [구조체 형식](#)
- [네임스페이스](#)
- [인터페이스](#)
- [대리자](#)

# C# 코딩 규칙(C# 프로그래밍 가이드)

2020-11-02 • 19 minutes to read • [Edit Online](#)

코딩 규칙은 다음과 같은 용도로 사용됩니다.

- 코드를 확인하는 사용자들이 레이아웃이 아닌 내용에 집중할 수 있도록 일관성 있게 표시되는 코드를 만들립니다.
- 코드를 확인하는 사용자들이 이전 경험을 토대로 한 가정을 통해 코드를 보다 빠르게 이해할 수 있도록 합니다.
- 코드를 보다 쉽게 복사, 변경 및 유지 관리할 수 있도록 합니다.
- C# 모범 사례를 제시합니다.

이 문서의 지침은 Microsoft에서 샘플과 설명서를 개발하는 데 사용됩니다.

## 명명 규칙

- [using 지시문](#)이 포함되지 않는 간단한 예제에서 네임스페이스 한정자를 사용합니다. 프로젝트에서 네임스페이스를 기본적으로 가져오는 경우에는 해당 네임스페이스의 이름을 정규화하지 않아도 됩니다. 정규화된 이름은 한 줄에 표시하기가 너무 길면 다음 예제에 나와 있는 것처럼 점(.)으로 분할할 수 있습니다.

```
var currentPerformanceCounterCategory = new System.Diagnostics.  
    PerformanceCounterCategory();
```

- 다른 지침에 맞도록 조정하기 위해 Visual Studio 디자이너 도구를 사용하여 만든 개체 이름을 변경할 필요는 없습니다.

## 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 사용합니다. 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 하나(공백 4개)만큼 들여씁니다.
- 메서드 정의와 속성 정의 간에는 빈 줄을 하나 이상 추가합니다.
- 다음 코드에 나와 있는 것처럼 괄호를 사용하여 식의 절을 명확하게 구분합니다.

```
if ((val1 > val2) && (val1 > val3))  
{  
    // Take appropriate action.  
}
```

주석 규칙

- 코드 줄의 끝이 아닌 별도의 줄에 주석을 배치합니다.
  - 주석 텍스트는 대문자로 시작합니다.
  - 주석 텍스트 끝에는 마침표를 붙입니다.
  - 다음 코드에 나와 있는 것처럼 주석 구분 기호(//)와 주석 텍스트 사이에 공백을 하나 삽입합니다.

```
// The following declaration creates a query. It does not run  
// the query.
```

- 서식이 지정된 별표 블록으로 주석을 묶지 않습니다.

## 언어 지침

다음 섹션에서는 C# 팀이 코드 예제와 샘플을 준비할 때 따르는 방식에 대해 설명합니다.

## 문자열 데이터 형식

- 다음 코드에 나와 있는 것처럼 [문자열 보간](#)을 사용하여 짧은 문자열을 연결합니다.

```
string displayName = $"{nameList[n].LastName}, {nameList[n].FirstName}";
```

- 특히 많은 양의 텍스트를 사용할 때 문자열을 루프에 추가하려면 `StringBuilder` 객체를 사용합니다.

### 암시적으로 형식화한 지역 변수

- 할당 오른쪽에서 변수 형식이 명확하거나 정확한 형식이 중요하지 않으면 지역 변수에 대해 **암시적 형식**을 사용합니다

```
// When the type of a variable is clear from the context, use var  
// in the declaration.  
var var1 = "This is clearly a string.";  
var var2 = 27;
```

- 할당 오른쪽에서 변수 형식이 명확하지 않으면 `var`를 사용하지 않습니다.

```
// When the type of a variable is not clear from the context, use an
// explicit type. You generally don't assume the type clear from a method name.
// A variable type is considered clear if it's a new operator or an explicit cast.
int var3 = Convert.ToInt32(Console.ReadLine());
int var4 = ExampleClass.ResultSoFar();
```

- 변수 이름을 사용하여 변수 형식을 지정하지 않습니다. 이렇게 하면 형식이 올바르게 지정되지 않을 수 있습니다.

```
// Naming the following variable inputInt is misleading.  
// It is a string.  
var inputInt = Console.ReadLine();  
Console.WriteLine(inputInt);
```

- **dynamic** 대신 `var` 를 사용하지 않습니다.
  - **for** 루프의 루프 변수 형식을 결정하려면 암시적 형식을 사용합니다.

다음 예제에서는 `for` 문에서 암시적 형식을 사용합니다.

- `foreach` 루프의 루프 변수 형식을 결정하기 위해 암시적 형식을 사용하지 마세요.

다음 예제에서는 `foreach` 문에서 명시적 형식을 사용합니다.

```
foreach (char ch in laugh)
{
    if (ch == 'h')
        Console.Write("H");
    else
        Console.Write(ch);
}
Console.WriteLine();
```

## NOTE

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.

## 부호 없는 데이터 형식

일반적으로는 부호 없는 형식 대신 `int` 를 사용합니다. `int` 는 C# 전체에서 일반적으로 사용되며, `int` 를 사용하는 경우 다른 라이브러리와 보다 쉽게 상호 작용할 수 있습니다.

백  
열

선언 줄에서 배열을 초기화할 때는 간결한 구문을 사용합니다.

```
// Preferred syntax. Note that you cannot use var here instead of string[].
string[] vowels1 = { "a", "e", "i", "o", "u" };

// If you use explicit instantiation, you can use var.
var vowels2 = new string[] { "a", "e", "i", "o", "u" };

// If you specify an array size, you must initialize the elements one at a time.
var vowels3 = new string[5];
vowels3[0] = "a";
vowels3[1] = "e";
// And so on.
```

대리자

대리자 형식의 인스턴스를 만들려면 간결한 구문을 사용합니다.

```
// First, in class Program, define the delegate type and a method that
// has a matching signature.

// Define the type.
public delegate void Del(string message);

// Define a method that has a matching signature.
public static void DelMethod(string str)
{
    Console.WriteLine("DelMethod argument: {0}", str);
}
```

```
// In the Main method, create an instance of Del.

// Preferred: Create an instance of Del by using condensed syntax.
Del exampleDel2 = DelMethod;

// The following declaration uses the full syntax.
Del exampleDel1 = new Del(DelMethod);
```

#### 예외 처리의 **try-catch** 및 **using** 문

- 대부분의 예외 처리에서는 **try-catch** 문을 사용합니다.

```
static string GetValueFromArray(string[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (System.IndexOutOfRangeException ex)
    {
        Console.WriteLine("Index is out of range: {0}", index);
        throw;
    }
}
```

- C# **using** 문을 사용하면 코드를 간소화할 수 있습니다. **finally** 블록의 코드가 **Dispose** 메서드 호출뿐인 **try-finally** 문이 있는 경우에는 **using** 문을 대신 사용합니다.

```
// This try-finally statement only calls Dispose in the finally block.
Font font1 = new Font("Arial", 10.0f);
try
{
    byte charset = font1.GdiCharSet;
}
finally
{
    if (font1 != null)
    {
        ((IDisposable)font1).Dispose();
    }
}

// You can do the same thing with a using statement.
using (Font font2 = new Font("Arial", 10.0f))
{
    byte charset = font2.GdiCharSet;
}
```

## && 및 || 연산자

예외를 방지하고 불필요한 비교를 건너뛰어 성능을 개선하려면 비교를 수행할 때 다음 예제에 나와 있는 것처럼 & 대신 &&를 사용하고 | 대신 ||를 사용합니다.

```
Console.Write("Enter a dividend: ");
var dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");
var divisor = Convert.ToInt32(Console.ReadLine());

// If the divisor is 0, the second clause in the following condition
// causes a run-time error. The && operator short circuits when the
// first expression is false. That is, it does not evaluate the
// second expression. The & operator evaluates both, and causes
// a run-time error when divisor is 0.
if ((divisor != 0) && (dividend / divisor > 0))
{
    Console.WriteLine("Quotient: {0}", dividend / divisor);
}
else
{
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

## New 연산자

- 다음 선언에 나와 있는 것처럼 암시적 형식이 포함된 간결한 형태의 개체 인스턴스화를 사용합니다.

```
var instance1 = new ExampleClass();
```

위의 줄은 다음 선언과 동일합니다.

```
ExampleClass instance2 = new ExampleClass();
```

- 개체를 간편하게 만들려면 개체 이니셜라이저를 사용합니다.

```
// Object initializer.
var instance3 = new ExampleClass { Name = "Desktop", ID = 37414,
    Location = "Redmond", Age = 2.3 };

// Default constructor and assignment statements.
var instance4 = new ExampleClass();
instance4.Name = "Desktop";
instance4.ID = 37414;
instance4.Location = "Redmond";
instance4.Age = 2.3;
```

## 이벤트 처리

나중에 제거할 필요가 없는 이벤트 처리기를 정의하는 경우 람다 식을 사용합니다.

```

public Form2()
{
    // You can use a lambda expression to define an event handler.
    this.Click += (s, e) =>
    {
        MessageBox.Show(
            ((MouseEventArgs)e).Location.ToString());
    };
}

```

```

// Using a lambda expression shortens the following traditional definition.
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 정규화해서는 안 됩니다. 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.

```

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).

```

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.

```

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.

```
var seattleCustomers = from customer in customers
    where customer.City == "Seattle"
    select customer.Name;
```

- 위의 예제에 나와 있는 것처럼 `from` 절 아래의 쿼리 절을 정렬합니다.
- 뒷부분의 쿼리 절이 필터링을 통해 범위가 좁아진 데이터 집합에 대해 작동하도록 다른 쿼리 절 앞에 `where` 절을 사용합니다.

```
var seattleCustomers2 = from customer in customers
    where customer.City == "Seattle"
    orderby customer.Name
    select customer;
```

- 내부 컬렉션에 액세스하려면 `join` 절 대신 여러 `from` 절을 사용합니다. 예를 들어 `Student` 개체 컬렉션이 각각 테스트 점수 컬렉션을 포함하는 경우 다음 쿼리를 실행하면 90점보다 높은 각 점수와 해당 점수를 받은 학생의 성이 반환됩니다.

```
// Use a compound from to access the inner sequence within each element.
var scoreQuery = from student in students
    from score in student.Scores
    where score > 90
    select new { Last = student.LastName, score };
```

## 보안

[보안 코딩 지침](#)의 지침을 따르세요.

## 참조

- [Visual Basic 코딩 규칙](#)
- [보안 코딩 지침](#)

# Main()과 명령줄 인수(C# 프로그래밍 가이드)

2020-11-02 • 5 minutes to read • [Edit Online](#)

`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 `-main` 컴파일러 옵션으로 프로그램을 컴파일하여 진입점으로 사용할 `Main` 메서드를 지정해야 합니다. 자세한 내용은 [-main\(C# 컴파일러 옵션\)](#)을 참조하세요.

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

## 개요

- `Main` 메서드는 실행 가능한 프로그램의 진입점으로, 프로그램의 제어가 시작되고 끝나는 위치합니다.
- `Main`은 클래스 또는 구조체 내부에 선언됩니다. `Main`은 정적이어야 하며 공용일 필요가 없습니다. (앞의 예제에서는 기본 액세스 권한 개인을 받습니다.) 바깥쪽 클래스 또는 구조체는 정적일 필요가 없습니다.
- `Main`에는 `void` 또는 `int`를 가지고 있거나 C# 7.1로 시작하거나 `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 구체적으로 `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.

다음은 유효한 `Main` 시그니처 목록입니다.

```
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 예제에서는 모두 공용 접근자 한정자를 사용합니다. 일반적으로 그렇게 하지만 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.

## C# 언어 사양

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

## 참조

- [csc.exe를 사용한 명령줄 빌드](#)
- [C# 프로그래밍 가이드](#)
- [메서드](#)
- [C# 프로그램 내부](#)

# 명령줄 인수(C# 프로그래밍 가이드)

2020-11-02 • 7 minutes to read • [Edit Online](#)

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

```
static int Main(string[] args)
```

```
static void Main(string[] args)
```

## NOTE

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

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

```
if (args.Length == 0)
{
    System.Console.WriteLine("Please enter a numeric argument.");
    return 1;
}
```

## TIP

`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.

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

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

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

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

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

자세한 내용은 `Parse` 및 `Convert`을 참조하세요.

## 예제

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 이름 *Factorial.cs*를 사용하여 파일을 텍스트 파일로 저장합니다.

```

// Add a using directive for System if the directive isn't already present.

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 방금 만든 파일이 포함된 폴더로 이동합니다.
3. 다음 명령을 입력하여 애플리케이션을 컴파일합니다.

```
csc Factorial.cs
```

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

4. 다음 명령을 입력하여 3의 계승을 계산합니다.

```
Factorial 3
```

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

#### NOTE

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

## 참조

- [System.Environment](#)
- [C# 프로그래밍 가이드](#)
- [Main\(\)과 명령줄 인수](#)
- [명령줄 인수를 표시하는 방법](#)
- [Main\(\) 반환 값](#)
- [클래스](#)

# 명령줄 인수를 표시하는 방법(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

명령줄에서 실행 파일에 제공된 인수는 `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"

## NOTE

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.

```
class CommandLine
{
    static void Main(string[] args)
    {
        // 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

- [C# 프로그래밍 가이드](#)
- [csc.exe를 사용한 명령줄 빌드](#)
- [Main\(\)과 명령줄 인수](#)
- [Main\(\) 반환 값](#)

# Main() 반환 값(C# 프로그래밍 가이드)

2020-11-02 • 7 minutes to read • [Edit Online](#)

Main 메서드는 void를 반환할 수 있습니다.

```
static void Main()
{
    //...
}
```

int를 반환할 수도 있습니다.

```
static int Main()
{
    //...
    return 0;
}
```

Main의 반환 값을 사용하지 않는 경우 void를 반환하면 코드가 다소 단순해집니다. 그러나 정수를 반환하면 프로그램이 실행 파일을 호출하는 다른 프로그램 또는 스크립트에 상태 정보를 전달할 수 있습니다. Main의 반환 값은 프로세스에 대한 종료 코드로 처리됩니다. void가 Main에서 반환되는 경우 종료 코드는 암시적으로 0이 됩니다. 다음 예제에서는 Main의 반환 값을 어떻게 액세스할 수 있는지를 보여 줍니다.

## 예제

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

program.cs에서 Main 메서드를 다음과 같이 수정합니다.

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

Windows에서 프로그램을 실행하는 경우 Main 함수에서 반환된 값은 환경 변수에 저장됩니다. 이 환경 변수는 배치 파일에서 ERRORLEVEL을 사용하거나 PowerShell에서 \$LastExitCode를 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.ps1로 저장합니다. PowerShell 프롬프트에 test.ps1을 입력하여 PowerShell 스크립트를 실행합니다.

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

```
dotnet run
```

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

## 샘플 출력

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

## 비동기 Main 반환 값

비동기 Main 반환 값은 `Main`에서 비동기 메서드를 호출하는 데 필요한 상용구 코드를 컴파일러에서 생성하는 코드로 이동합니다. 기존에는 이 구문을 작성하여 비동기 코드를 호출하고 비동기 작업이 완료될 때까지 프로그램이 실행되는지 확인해야 했습니다.

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

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

이제는 이 구문을 다음으로 바꿀 수 있습니다.

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

새 구문의 장점은 컴파일러에서 항상 올바른 코드를 생성한다는 것입니다.

## 컴파일러 생성 복사

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.

#### NOTE

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

## 참조

- [C# 프로그래밍 가이드](#)
- [C# 참조](#)
- [Main\(\)과 명령줄 인수](#)
- [명령줄 인수를 표시하는 방법](#)

# 프로그래밍 개념(C#)

2020-11-02 • 4 minutes to read • [Edit Online](#)

이 섹션에서는 C# 언어의 프로그래밍 개념을 설명합니다.

## 섹션 내용

제목	설명
<a href="#">.NET 어셈블리</a>	.NET 어셈블리를 만들고 사용하는 방법을 설명합니다.
<a href="#">async 및 await를 사용한 비동기 프로그래밍(C#)</a>	C#에서 <code>async</code> 및 <code>await</code> 키워드를 사용하여 비동기 솔루션을 작성하는 방법을 설명합니다. 연습이 포함되어 있습니다.
<a href="#">특성(C#)</a>	특성을 사용하여 형식, 필드, 메서드 및 속성 등의 요소를 프로그래밍하는 방법에 대한 추가 정보를 제공하는 방법을 설명합니다.
<a href="#">컬렉션(C#)</a>	.NET에서 제공하는 컬렉션의 형식 중 일부를 설명합니다. 간단한 컬렉션 및 키/값 쌍의 컬렉션을 사용하는 방법을 보여 줍니다.
<a href="#">공변성(Covariance) 및 반공변성(Contravariance)(C#)</a>	인터페이스 및 대리자에서 제네릭 형식 매개 변수의 암시적 변환을 사용하도록 설정하는 방법을 보여 줍니다.
<a href="#">식 트리(C#)</a>	식 트리를 사용하여 실행 코드의 동적 수정을 허용하는 방법을 설명합니다.
<a href="#">반복기(C#)</a>	컬렉션을 단계별로 실행하면서 한 번에 하나씩 요소를 반환하는 데 사용되는 반복기에 대해 설명합니다.
<a href="#">LINQ(Language-Integrated Query)(C#)</a>	C#의 언어 구문의 강력한 쿼리 기능과 관계형 데이터베이스, XML 문서, 데이터 세트 및 메모리 내 컬렉션을 쿼리하기 위한 모델에 대해 설명합니다.
<a href="#">리플렉션(C#)</a>	리플렉션을 사용하면 동적으로 형식 인스턴스를 만들거나, 형식을 기준 개체에 바인딩하거나, 기준 개체에서 형식을 가져와 해당 메서드를 호출하거나, 필드 및 속성에 액세스하는 방법을 설명합니다.
<a href="#">Serialization(C#)</a>	이진, XML 및 SOAP serialization의 주요 개념에 대해 설명합니다.

## 관련 단원

<a href="#">성능 팁</a>	애플리케이션의 성능을 향상시키는 데 도움이 되는 여러 가지 기본 규칙에 대해 설명합니다.
----------------------	---

# async 및 await를 사용한 비동기 프로그래밍

2020-11-02 • 32 minutes to read • [Edit Online](#)

TAP(Task 비동기 프로그래밍) 모델은 비동기 코드에 대한 추상화를 제공합니다. 항상 그렇듯이 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 데운 다음, 계란 프라이 두 개를 만듭니다.
3. 베이컨 세 조각을 튀깁니다.
4. 빵 두 조각을 굽습니다.
5. 토스트에 버터와 잼을 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 별별로 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.

별별 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.

```
using System;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    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");

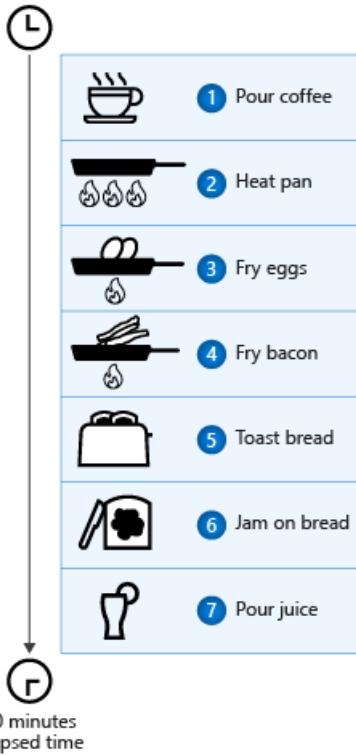
        return new Bacon();
    }

    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 식사에는 대략 30분이 걸렸는데, 총계는 개별 작업의 합계이기 때문입니다.

#### NOTE

`Coffee`, `Egg`, `Bacon`, `Toast` 및 `Juice` 클래스는 비어 있습니다. 해당 클래스는 데모 목적의 마커 클래스일 뿐이며 속성을 포함하지 않고 다른 용도로 사용할 수 없습니다.

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 죽은 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 것입니다. 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 않

을 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.

```
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!");
}
```

#### IMPORTANT

총 경과 시간은 초기 동기 버전과 거의 같습니다. 이 코드에서는 아직 비동기 프로그래밍의 몇 가지 주요 기능을 활용하지 않았습니다.

#### TIP

`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.

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 준비하는 방법과 더 비슷한 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 처

리한 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.

```
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!");
```

다음으로, 아침 식사를 제공하기 전에 베이컨과 달걀에 대한 `await` 문을 메서드 끝으로 이동할 수 있습니다.

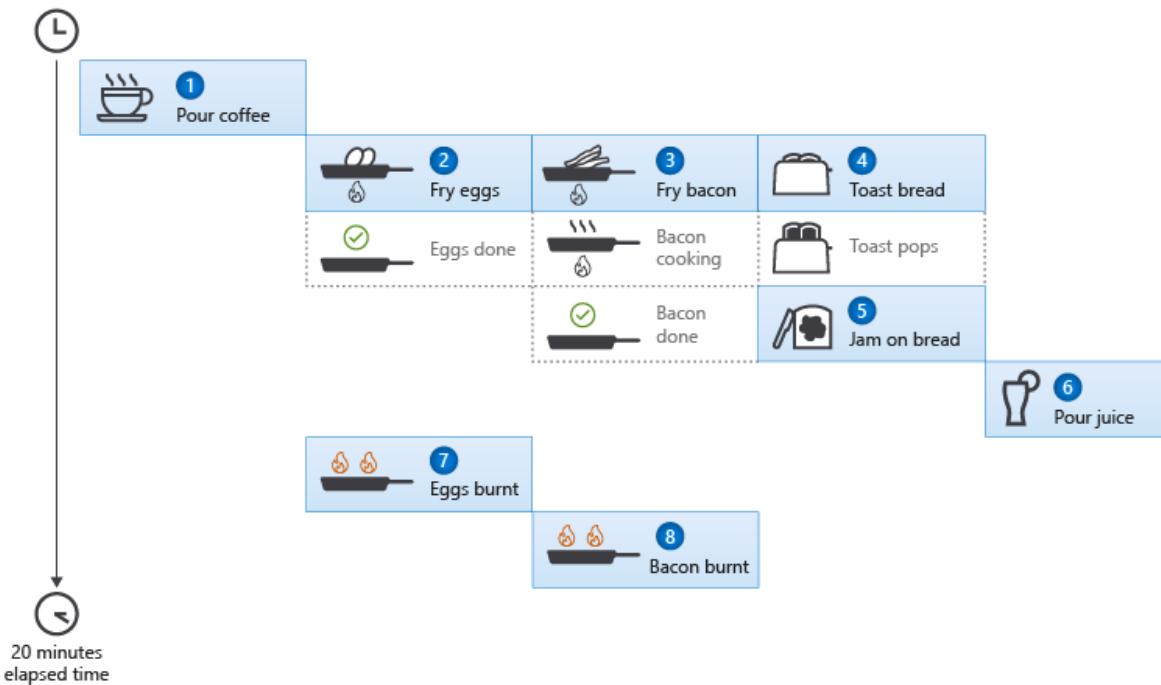
```
Coffee cup = PourCoffee();
Console.WriteLine("coffee is ready");

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 작업을 동시에 실행할 수 있었기 때문입니다.

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.

### IMPORTANT

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.

```
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.

```

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.

## 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.

```

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.

```

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");
    }
    breakfastTasks.Remove(finishedTask);
}

```

변경 내용을 모두 적용한 후 코드의 최종 버전은 다음과 같습니다.

```

using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    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");
                }
                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");

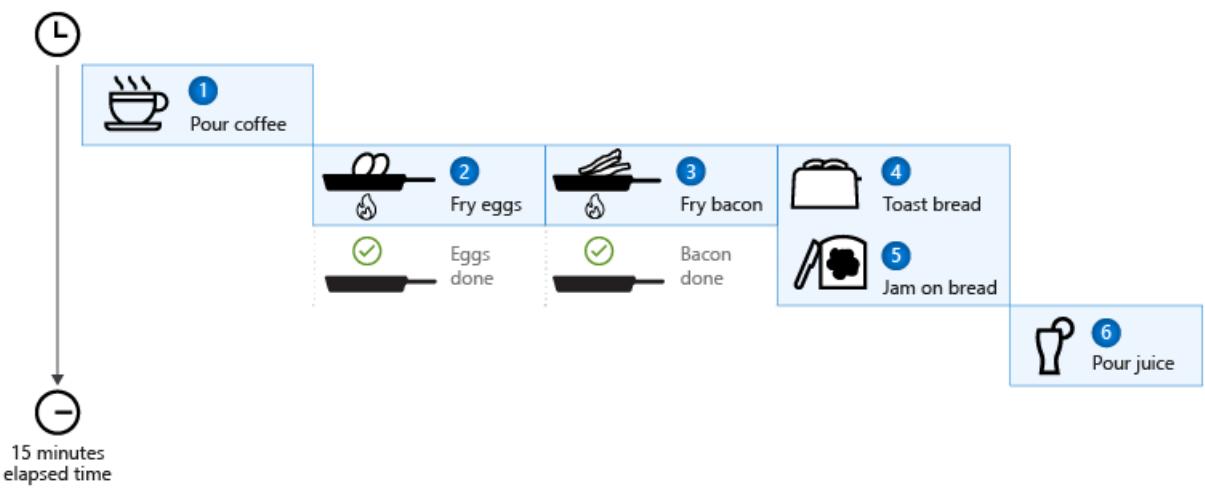
        return new Bacon();
    }

    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 15분이 걸렸는데, 일부 작업을 동시에 실행할 수 있었고 코드가 여러 작업을 한 번에 모니터링하고 필요한 경우에만 작업을 수행할 수 있었기 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

[작업 비동기 프로그래밍 모델에 대해 알아보기](#)

# 작업 비동기 프로그래밍 모델

2020-11-02 • 40 minutes to read • [Edit Online](#)

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# 5](#)에는 .NET Framework 4.5 이상, .NET Core 및 Windows 런타임의 비동기 지원을 활용하는 간단한 비동기 프로그래밍 접근 방법이 도입되었습니다. 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 가능성 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	<a href="#">HttpClient</a>	<a href="#">Windows.Web.Http.HttpClient</a> <a href="#">SyndicationClient</a>
파일 작업	<a href="#">JsonSerializer</a> <a href="#">StreamReader</a> <a href="#">StreamWriter</a> <a href="#">XmlReader</a> <a href="#">XmlWriter</a>	<a href="#">StorageFile</a>
이미지 작업		<a href="#">MediaCapture</a> <a href="#">BitmapEncoder</a> <a href="#">BitmapDecoder</a>
WCF 프로그래밍	<a href="#">동기 및 비동기 작업</a>	

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.

```
public async Task<int> GetUrlContentLengthAsync()
{
    var client = new HttpClient();

    Task<string> getStringTask =
        client.GetStringAsync("https://docs.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 시작해 봅시다. 여기에는 `async` 한정자가 포함됩니다. 반환 형식은 `Task<int>`입니다(추가 옵션은 "반환 형식" 섹션 참조). 메서드 이름은 `GetStringAsync`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.

```
string contents = await client.GetStringAsync("https://docs.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(C# 7.0부터).
- 자세한 내용은 [반환 형식 및 매개 변수](#) 섹션을 참조하세요.

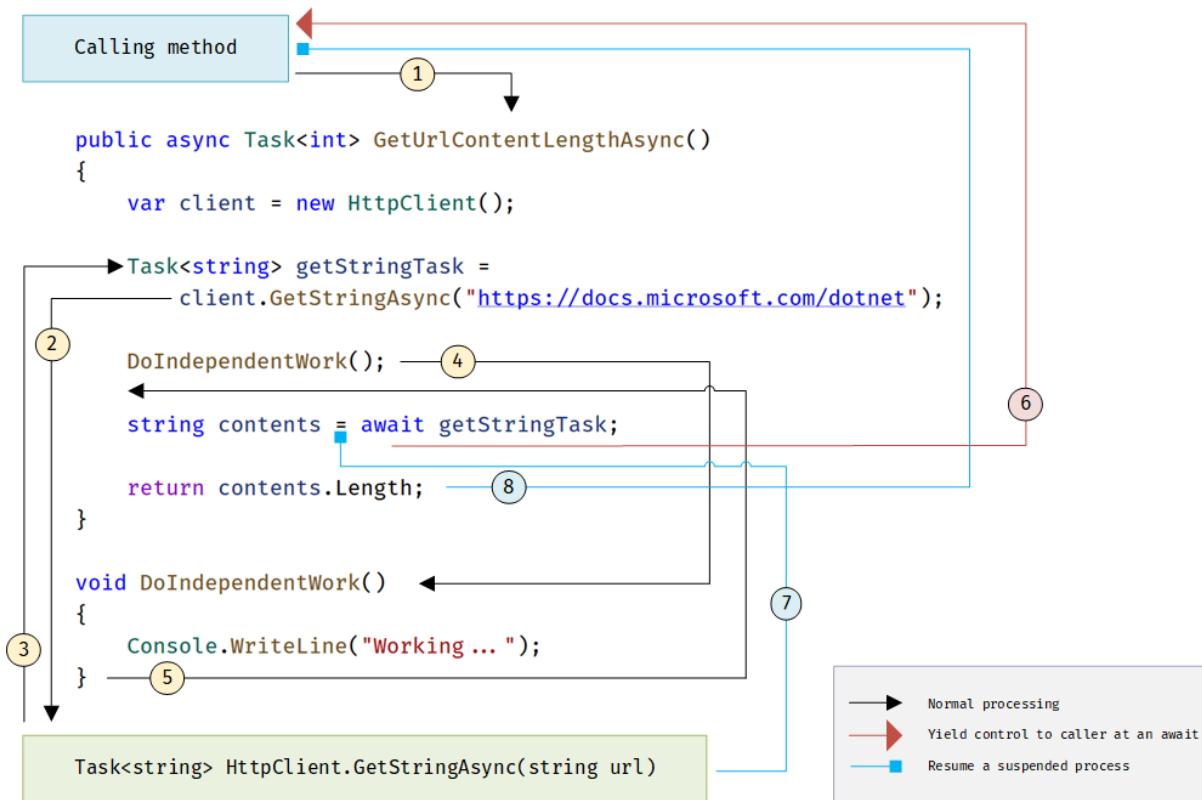
- 메서드는 일반적으로 비동기 작업이 완료될 때까지 메서드가 계속될 수 없는 지점을 표시하는 하나 이상의 `await` 표현을 포함하고 있습니다. 한편, 메서드가 일시 중단되고 컨트롤이 메서드의 호출자로 반환됩니다. 이 항목의 다음 단원은 일시 중단 지점에서 발생하는 상황을 보여줍니다.

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.

1. 호출하는 메서드는 `GetUrlContentLengthAsync` 비동기 메서드를 호출하고 기다립니다.
2. `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 종단하고 `GetUrlContentLengthAsync` 를 호출한 메서드에 제어 권한을 양도합니다. `GetUrlContentLengthAsync` 는 `Task<int>` 를 호출자에게 반환합니다. 작업은 다운로드한 문자열의 길이인 정수 결과를 만든다는 약속을 나타냅니다.

#### NOTE

`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 종단한 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 작업에 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. 비동기 메서드는 작업이 완료될 때 반환되지만(5단계) 비동기 메서드는 작업이 일시 종단될 때 반환됩니다. 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 버전

을 사용하는 경우 [비동기 프로그래밍\(Windows 스토어 앱\)](#) 및 [빠른 시작: C# 또는 Visual Basic에서 비동기 API 호출](#)을 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 때문에 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.

C# 7.0부터는 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.

```
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 정의하는데 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. 비동기 메서드에서 예외를 처리하는 방법에 대한 자세한 내용은 [try-catch](#)를 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 시작하지만 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 항목 및 샘플(Visual Studio)

제목	설명	예제
<a href="#">async 및 await를 사용하여 병렬로 여러 웹 요청을 만드는 방법(C#)</a>	동시에 여러 작업을 시작하는 방법을 보여줍니다.	<a href="#">비동기 샘플: 병렬로 여러 웹 요청 만들기</a>
<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.  - <a href="#">작업 목록 취소(C#)</a> - <a href="#">일정 기간 이후 작업 취소(C#)</a> - <a href="#">완료되면 비동기 작업 처리(C#)</a>	
<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.	
<a href="#">비동기 Channel 9 비디오</a>	비동기 프로그래밍에 대한 다양한 비디오로 연결되는 링크를 제공합니다.	

## 참조

- [async](#)
- [await](#)
- [비동기 프로그래밍](#)
- [비동기 개요](#)

# 비동기 반환 형식(C#)

2020-11-02 • 19 minutes to read • [Edit Online](#)

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
- C# 7.0부터 액세스 가능한 `GetAwaiter` 메서드가 있는 모든 형식. `GetAwaiter` 메서드에서 반환된 개체는 `System.Runtime.CompilerServices.ICriticalNotifyCompletion` 인터페이스를 구현해야 합니다.
- C# 8.0부터 '비동기 스트림'을 반환하는 비동기 메서드의 경우 `IAsyncEnumerable<T>`.

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>` - 진행률을 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.

```

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. 문 및 식이라는 용어를 분명하게 설명하려면 아래 표를 참조하세요.

AWAIT 종류	예제	TYPE
인수를 제거합니다.	<code>await SomeTaskMethodAsync()</code>	Task
식	<code>T result = await SomeTaskMethodAsync&lt;T&gt;();</code>	Task<TResult>

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.

```

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. 따라서 메서드 선언은 `Task<int>` 의 반환 형식을 지정해야 합니다. `FromResult` 비동기 메서드는 `DayOfWeek`를 반환하는 작업의 자리 표시자입니다.

```

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.

### IMPORTANT

`Result` 속성은 차단 속성입니다. 해당 작업이 완료되기 전에 액세스하려고 하면, 작업이 완료되고 값을 사용할 수 있을 때 까지 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.

```

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`를 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 진행

해야 합니다. 호출자는 비동기 메서드가 생성하는 모든 값 또는 예외와 독립되어 있어야 합니다.

Void를 반환하는 비동기 메서드의 호출자는 메서드에서 throw되는 예외를 catch할 수 없으므로 처리되지 않은 예외를 사용하면 애플리케이션이 실패할 수 있습니다. Task 또는 Task<TResult>를 반환하는 메서드가 예외를 throw하는 경우 이 예외는 반환된 작업에 저장됩니다. 작업이 대기하는 경우 예외가 다시 throw됩니다. 따라서 예외를 생성할 수 있는 모든 비동기 메서드에 Task 또는 Task<TResult>의 반환 형식이 있고 메서드 호출이 대기 상태인지 확인해야 합니다.

비동기 메서드에서 예외를 catch하는 방법에 관한 자세한 내용은 try-catch 문서의 [비동기 메서드의 예외](#) 섹션을 참조하세요.

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.

```
using System;
using System.Threading.Tasks;

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>

C# 7.0부터 비동기 메서드는 액세스 가능한 `GetAwaiter` 메서드가 있는 모든 형식을 반환할 수 있습니다.

`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. `System.Threading.Tasks.ValueTask<TResult>` 형식을 사용하려면 `System.Threading.Tasks.Extensions` NuGet 패키지를 프로젝트에 추가해야 합니다. 다음 예제에서는 `ValueTask<TResult>` 구조체를 사용하여 두 주사위 굴리기 값을 검색합니다.

```

using System;
using System.Threading.Tasks;

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

```

## IAsyncEnumerable<T>을 사용하는 비동기 스트림

C# 8.0부터 비동기 메서드는 [IAsyncEnumerable<T>](#)을 통해 표시되는 '비동기 스트림'을 반환할 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.

```

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](#)

# 작업 목록 취소(C#)

2021-02-18 • 9 minutes to read • [Edit Online](#)

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 구성 요소

이 자습서를 사용하려면 다음이 필요합니다.

- [.NET 5.0 이상 SDK](#)
- IDE(통합 개발 환경)
  - [Visual Studio](#), [Visual Studio Code](#) 또는 Mac용 [Visual Studio](#) 권장

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.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
```

## 필드 추가하기

`Program` 클래스 정의에서 다음 세 필드를 추가합니다.

```

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://docs.microsoft.com",
    "https://docs.microsoft.com/aspnet/core",
    "https://docs.microsoft.com/azure",
    "https://docs.microsoft.com/azure/devops",
    "https://docs.microsoft.com/dotnet",
    "https://docs.microsoft.com/dynamics365",
    "https://docs.microsoft.com/education",
    "https://docs.microsoft.com/enterprise-mobility-security",
    "https://docs.microsoft.com/gaming",
    "https://docs.microsoft.com/graph",
    "https://docs.microsoft.com/microsoft-365",
    "https://docs.microsoft.com/office",
    "https://docs.microsoft.com/powershell",
    "https://docs.microsoft.com/sql",
    "https://docs.microsoft.com/surface",
    "https://docs.microsoft.com/system-center",
    "https://docs.microsoft.com/visualstudio",
    "https://docs.microsoft.com/windows",
    "https://docs.microsoft.com/xamarin"
};

```

`CancellationTokenSource`는 요청된 취소를 `CancellationToken`에 신호로 보내는 데 사용됩니다. `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.

```

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();

    await Task.WhenAny(new[] { cancelTask, sumPageSizesTask });

    Console.WriteLine("Application ending.");
}

```

업데이트된 `Main` 메서드는 이제 `Async main`으로 간주되어 실행 파일에 대한 비동기 진입점을 허용합니다. 콘

솔에 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.

## 비동기 합계 페이지 크기 메서드 만들기

`SumPageSizesAsync` 메서드 아래에 `Main` 메서드를 추가합니다.

```
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`은 정수입니다.

```
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.

```
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

```

Application started.
Press the ENTER key to cancel...

https://docs.microsoft.com                                         37,357
https://docs.microsoft.com/aspnet/core                           85,589
https://docs.microsoft.com/azure                                398,939
https://docs.microsoft.com/azure/devops                         73,663
https://docs.microsoft.com/dotnet                             67,452
https://docs.microsoft.com/dynamics365                        48,582
https://docs.microsoft.com/education                          22,924

ENTER key pressed: cancelling downloads.

Application ending.

```

## 전체 예제

다음 코드는 예제에 관한 *Program.cs* 파일의 전체 텍스트입니다.

```

using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;

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://docs.microsoft.com",
        "https://docs.microsoft.com/aspnet/core",
        "https://docs.microsoft.com/azure",
        "https://docs.microsoft.com/azure/devops",
        "https://docs.microsoft.com/dotnet",
        "https://docs.microsoft.com/dynamics365",
        "https://docs.microsoft.com/education",
        "https://docs.microsoft.com/enterprise-mobility-security",
        "https://docs.microsoft.com/gaming",
        "https://docs.microsoft.com/graph",
        "https://docs.microsoft.com/microsoft-365",
        "https://docs.microsoft.com/office",
        "https://docs.microsoft.com/powershell",
        "https://docs.microsoft.com/sql",
        "https://docs.microsoft.com/surface",
        "https://docs.microsoft.com/system-center",
        "https://docs.microsoft.com/visualstudio",
        "https://docs.microsoft.com/windows",
        "https://docs.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();

await Task.WhenAny(new[] { cancelTask, sumPageSizesTask });

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\(C#\)](#)

# 일정 기간 이후 비동기 작업 취소(C#)

2021-02-18 • 4 minutes to read • [Edit Online](#)

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 구성 요소

이 자습서를 사용하려면 다음이 필요합니다.

- [작업 목록 취소\(C#\)](#) 자습서에서 애플리케이션을 만들었어야 함
- [.NET 5.0 이상 SDK](#)
- IDE(통합 개발 환경)
  - [Visual Studio](#), [Visual Studio Code](#) 또는 Mac용 [Visual Studio](#) 권장

## 애플리케이션 진입점 업데이트

기존 `Main` 메서드를 다음으로 바꿉니다.

```
static async Task Main()
{
    Console.WriteLine("Application started.");

    try
    {
        s_cts.CancelAfter(3500);

        await SumPageSizesAsync();
    }
    catch (TaskCanceledException)
    {
        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 애플리케

이션이 종료됩니다. 그러나 모든 URL이 처리되기 전에 예약된 취소가 트리거되면 `TaskCanceledException`이 throw됩니다.

#### 예제 애플리케이션 출력

```
Application started.

https://docs.microsoft.com 37,357
https://docs.microsoft.com/aspnet/core 85,589
https://docs.microsoft.com/azure 398,939
https://docs.microsoft.com/azure/devops 73,663

Tasks cancelled: timed out.

Application ending.
```

## 전체 예제

다음 코드는 예제에 관한 `Program.cs` 파일의 전체 텍스트입니다.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;

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://docs.microsoft.com",
        "https://docs.microsoft.com/aspnet/core",
        "https://docs.microsoft.com/azure",
        "https://docs.microsoft.com/azure/devops",
        "https://docs.microsoft.com/dotnet",
        "https://docs.microsoft.com/dynamics365",
        "https://docs.microsoft.com/education",
        "https://docs.microsoft.com/enterprise-mobility-security",
        "https://docs.microsoft.com/gaming",
        "https://docs.microsoft.com/graph",
        "https://docs.microsoft.com/microsoft-365",
        "https://docs.microsoft.com/office",
        "https://docs.microsoft.com/powershell",
        "https://docs.microsoft.com/sql",
        "https://docs.microsoft.com/surface",
        "https://docs.microsoft.com/system-center",
        "https://docs.microsoft.com/visualstudio",
        "https://docs.microsoft.com/windows",
        "https://docs.microsoft.com/xamarin"
    };

    static async Task Main()
    {
        Console.WriteLine("Application started.");

        try
        {
```

```

        s_cts.CancelAfter(3500);

        await SumPageSizesAsync();
    }
    catch (TaskCanceledException)
    {
        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\(C#\)](#)

# 완료되면 비동기 작업 처리(C#)

2021-02-18 • 9 minutes to read • [Edit Online](#)

[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.

## 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.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;
```

## 필드 추가하기

`Program` 클래스 정의에 다음 두 개 필드를 추가합니다.

```
static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://docs.microsoft.com",
    "https://docs.microsoft.com/aspnet/core",
    "https://docs.microsoft.com/azure",
    "https://docs.microsoft.com/azure/devops",
    "https://docs.microsoft.com/dotnet",
    "https://docs.microsoft.com/dynamics365",
    "https://docs.microsoft.com/education",
    "https://docs.microsoft.com/enterprise-mobility-security",
    "https://docs.microsoft.com/gaming",
    "https://docs.microsoft.com/graph",
    "https://docs.microsoft.com/microsoft-365",
    "https://docs.microsoft.com/office",
    "https://docs.microsoft.com/powershell",
    "https://docs.microsoft.com/sql",
    "https://docs.microsoft.com/surface",
    "https://docs.microsoft.com/system-center",
    "https://docs.microsoft.com/visualstudio",
    "https://docs.microsoft.com/windows",
    "https://docs.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.

```
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.

```

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");
}

```

`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`은 정수입니다.

```

IEnumerable<Task<int>> downloadTasksQuery =
    from url in s_urlList
    select ProcessUrlAsync(url, s_client);

```

LINQ를 통한 [지연된 실행](#)으로 인해 `Enumerable.ToList`을 호출하여 각 작업을 시작합니다.

```
List<Task<int>> downloadTasks = downloadTasksQuery.ToList();
```

`while` 루프는 컬렉션의 각 작업에서 다음 단계를 수행합니다.

- 다운로드를 완료한 컬렉션에서 첫 번째 작업을 식별하기 위해 `WhenAny` 호출을 기다립니다.

```
Task<int> finishedTask = await Task.WhenAny(downloadTasks);
```

- 컬렉션에서 해당 작업을 제거합니다.

```
downloadTasks.Remove(finishedTask);
```

- `ProcessUrlAsync` 호출에서 반환된 `finishedTask`을 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.

```
total += await finishedTask;
```

## 프로세스 메서드 추가

`SumPageSizesAsync` 메서드 아래에 다음 `ProcessUrlAsync` 메서드를 추가합니다.

```
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.

### Caution

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.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;

namespace ProcessTasksAsTheyFinish
{
    class Program
    {
        static readonly HttpClient s_client = new HttpClient
        {
            MaxResponseContentBufferSize = 1_000_000
        };

        static readonly IEnumerable<string> s_urlList = new string[]
        {
            "https://docs.microsoft.com",
            "https://docs.microsoft.com/aspnet/core",
            "https://docs.microsoft.com/azure",
            "https://docs.microsoft.com/azure/devops",
            "https://docs.microsoft.com/dotnet",
            "https://docs.microsoft.com/dynamics365",
            "https://docs.microsoft.com/education",
            "https://docs.microsoft.com/enterprise-mobility-security",
            "https://docs.microsoft.com/gaming",
            "https://docs.microsoft.com/graph",
            "https://docs.microsoft.com/microsoft-365",
            "https://docs.microsoft.com/office",
            "https://docs.microsoft.com/powershell",
            "https://docs.microsoft.com/sql",
            "https://docs.microsoft.com/surface",
            "https://docs.microsoft.com/system-center",
            "https://docs.microsoft.com/visualstudio",
            "https://docs.microsoft.com/windows",
            "https://docs.microsoft.com/xamarin"
        };
    }
}
```

```

};

static Task Main() => SumPageSizesAsync();

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");
}

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://docs.microsoft.com/windows          25,513
// https://docs.microsoft.com/gaming           30,705
// https://docs.microsoft.com/dotnet            69,626
// https://docs.microsoft.com/dynamics365       50,756
// https://docs.microsoft.com/surface           35,519
// https://docs.microsoft.com                  39,531
// https://docs.microsoft.com/azure/devops      75,837
// https://docs.microsoft.com/xamarin           60,284
// https://docs.microsoft.com/system-center     43,444
// https://docs.microsoft.com/enterprise-mobility-security 28,946
// https://docs.microsoft.com/microsoft-365       43,278
// https://docs.microsoft.com/visualstudio       31,414
// https://docs.microsoft.com/office             42,292
// https://docs.microsoft.com/azure              401,113
// https://docs.microsoft.com/graph              46,831
// https://docs.microsoft.com/education          25,098
// https://docs.microsoft.com/powershell         58,173
// https://docs.microsoft.com/aspnet/core        87,763
// https://docs.microsoft.com/sql                 53,362

// Total bytes returned: 1,249,485
// Elapsed time: 00:00:02.7068725

```

## 참고 항목

- [WhenAny](#)
- [async 및 await를 사용한 비동기 프로그래밍\(C#\)](#)

# 비동기 파일 액세스(C#)

2020-11-02 • 12 minutes to read • [Edit Online](#)

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 1,000개의 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

```
public async Task SimpleWriteAsync()
{
    string filePath = "simple.txt";
    string text = $"Hello World";

    await File.WriteAllTextAsync(filePath, text);
}
```

### 한정된 제어 예

```

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 촉약된 것입니다.

```

Task theTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
await theTask;

```

첫 번째 문은 작업을 반환하여 파일 처리가 시작되도록 합니다. `await`가 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

```

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 `await`의 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.

```

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

```

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 끌어 두지 않고 사용자 인터페이스 스레드를 끌어 두지 않는다는 것입니다.

```
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);

            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 `CancellationToken`을 지정할 수 있습니다. 자세한 내용은 [관리형 스레드의 취소](#)를 참조하세요.

## 참고 항목

- [async 및 await를 사용한 비동기 프로그래밍\(C#\)](#)
- [비동기 반환 형식\(C#\)](#)

# 특성(C#)

2020-11-02 • 14 minutes to read • [Edit Online](#)

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. 자세한 내용은 [리플렉션\(C#\)](#)을 참조하세요.

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. 자세한 내용은 [사용자 지정 특성 만들기\(C#\)](#)를 참조하세요.
- 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. 자세한 내용은 [리플렉션을 사용하여 특성 액세스\(C#\)](#)를 참조하세요.

## 특성 사용

특정 특성은 유효한 선언 형식이 제한적일 수 있지만 거의 모든 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.

```
[Serializable]
public class SampleClass
{
    // Objects of this type can be serialized.
}
```

[DllImportAttribute](#) 특성을 사용하는 메서드는 다음 예제와 같이 선언됩니다.

```
[System.Runtime.InteropServices.DllImport("user32.dll")]
extern static void SampleMethod();
```

다음 예제와 같이 둘 이상의 특성을 하나의 선언에 추가할 수 있습니다.

```
using System.Runtime.InteropServices;
```

```
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.

```
[Conditional("DEBUG"), Conditional("TEST1")]
void TraceMethod()
{
    // ...
}
```

#### NOTE

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.

```
[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.

#### 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.

```
[target : attribute-list]
```

가능한 `target` 값 목록은 다음 표에 나와 있습니다.

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

대상 값	적용 대상
property	속성
return	메서드, 속성 인덱서 또는 <code>get</code> 속성 접근자의 반환 값
type	구조체, 클래스, 인터페이스, 열거형 또는 대리자

자동 구현 속성에 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.

```
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.

```
// 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; }
```

#### NOTE

`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\(C#\)](#)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 serialization을 위해 클래스 멤버 및 XML 노드 간을 맵핑하는 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

## 관련 단원

자세한 내용은 다음을 참조하세요.

- [사용자 지정 특성 만들기\(C#\)](#)
- [리플렉션을 사용하여 특성 액세스\(C#\)](#)
- [특성\(C#\)을 사용하여 C/C++ 공용 구조체를 만드는 방법](#)
- [공통 특성\(C#\)](#)
- [호출자 정보\(C#\)](#)

## 참조

- [C# 프로그래밍 가이드](#)
- [리플렉션\(C#\)](#)
- [특성](#)
- [C#에서 특성 사용](#)

# 사용자 지정 특성 만들기(C#)

2020-11-02 • 3 minutes to read • [Edit Online](#)

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.

```
[System.AttributeUsage(System.AttributeTargets.Class |
                      System.AttributeTargets.Struct)
]
public class AuthorAttribute : System.Attribute
{
    private string name;
    public double version;

    public AuthorAttribute(string name)
    {
        this.name = name;
        version = 1.0;
    }
}
```

클래스 이름 `AuthorAttribute` 는 특성의 이름인 `Author` 와 `Attribute` 접미사입니다. 이 클래스는 `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.

```
[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 사용 특성을 만듭니다.

```
[System.AttributeUsage(System.AttributeTargets.Class |
                      System.AttributeTargets.Struct,
                      AllowMultiple = true) // multiuse attribute
]
public class AuthorAttribute : System.Attribute
```

다음 코드 예제에서는 같은 형식의 여러 특성이 한 클래스에 적용됩니다.

```
[Author("P. Ackerman", version = 1.1)]
[Author("R. Koch", version = 1.2)]
class SampleClass
{
    // P. Ackerman's code goes here...
    // R. Koch's code goes here...
}
```

## 참조

- [System.Reflection](#)
- [C# 프로그래밍 가이드](#)
- [사용자 지정 특성 작성](#)
- [리플렉션\(C#\)](#)
- [특성\(C#\)](#)
- [리플렉션을 사용하여 특성 액세스\(C#\)](#)
- [AttributeUsage\(C#\)](#)

# 리플렉션을 사용하여 특성 액세스(C#)

2020-11-02 • 4 minutes to read • [Edit Online](#)

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 여러 개의 오버로드된 버전이 있습니다. 자세한 내용은 [Attribute](#)를 참조하세요.

다음과 같은 특성 사양은

```
[Author("P. Ackerman", version = 1.1)]
class SampleClass
```

다음과 개념적으로 동일합니다.

```
Author anonymousAuthorObject = new Author("P. Ackerman");
anonymousAuthorObject.version = 1.1;
```

하지만 `SampleClass`에서 특성을 쿼리할 때까지 코드가 실행되지 않습니다. `SampleClass`에서 `GetCustomAttributes`를 호출하면 `Author` 개체가 위와 같이 구성 및 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.

## 예제

아래는 완성된 예제입니다. 사용자 지정 특성이 정의되어 있고 여러 엔터티에 적용되었으며 리플렉션을 통해 검색합니다.

```
// Multiuse attribute.
[System.AttributeUsage(System.AttributeTargets.Class |
    System.AttributeTargets.Struct,
    AllowMultiple = true) // Multiuse attribute.
]
public class Author : System.Attribute
{
    string name;
    public double version;

    public Author(string name)
    {
        this.name = name;

        // Default value.
        version = 1.0;
    }

    public string GetName()
    {
        return name;
    }
}

// Class with the Author attribute.
```

```

// 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
{
    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 {0}", t);

        // Using reflection.
        System.Attribute[] attrs = System.Attribute.GetCustomAttributes(t); // Reflection.

        // Displaying output.
        foreach (System.Attribute attr in attrs)
        {
            if (attr is Author)
            {
                Author a = (Author)attr;
                System.Console.WriteLine("    {0}, version {1:f}", a.GetName(), a.version);
            }
        }
    }
}

/* 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](#)
- [C# 프로그래밍 가이드](#)
- [특성에 저장된 정보 검색](#)
- [리플렉션\(C#\)](#)
- [특성\(C#\)](#)

- 사용자 지정 특성 만들기(C#)

# 특성을 사용하여 C/C++ 공용 구조체 만드는 방법 (C#)

2020-11-02 • 2 minutes to read • [Edit Online](#)

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.

## 예제

이 코드 세그먼트에서 `TestUnion`의 모든 필드는 메모리의 같은 위치에서 시작합니다.

```
// Add a using directive for System.Runtime.InteropServices.  
  
[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;  
}
```

## 예제

다음은 명시적으로 설정된 다른 위치에서 필드가 시작하는 또 다른 예제입니다.

```
// Add a using directive for System.Runtime.InteropServices.  
  
[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 `lg` 와 동일한 메모리 위치를 공유합니다. 구조체 레이아웃에 대한 이러한 종류의 제어는 플랫폼 호출을 사용할 때 유용합니다.

## 참고 항목

- [System.Reflection](#)
- [Attribute](#)
- [C# 프로그래밍 가이드](#)
- [특성](#)
- [리플렉션\(C#\)](#)
- [특성\(C#\)](#)
- [사용자 지정 특성 만들기\(C#\)](#)
- [리플렉션을 사용하여 특성 액세스\(C#\)](#)

# 컬렉션(C#)

2020-11-02 • 29 minutes to read • [Edit Online](#)

대부분의 애플리케이션의 경우 관련 개체의 그룹을 만들고 관리하려고 합니다. 개체를 그룹화하는 방법에는 개체 배열을 만들거나 개체 컬렉션을 만드는 두 가지가 있습니다.

배열은 고정된 개수의 강력한 형식 개체를 만들고 작업하는 데 가장 유용합니다. 배열에 대한 자세한 내용은 [배열](#)을 참조하세요.

컬렉션은 개체 그룹에 대해 작업하는 보다 유연한 방법을 제공합니다. 배열과 달리, 애플리케이션의 요구가 변경됨에 따라 작업하는 개체 그룹이 동적으로 확장되거나 축소될 수 있습니다. 일부 컬렉션의 경우 키를 사용하여 개체를 신속하게 검색할 수 있도록 컬렉션에 추가하는 모든 개체에 키를 할당할 수 있습니다.

컬렉션은 클래스이므로 해당 컬렉션에 요소를 추가하려면 먼저 클래스 인스턴스를 선언해야 합니다.

컬렉션에 단일 데이터 형식의 요소만 포함된 경우 [System.Collections.Generic](#) 네임스페이스의 클래스 중 하나를 사용할 수 있습니다. 제네릭 컬렉션은 다른 데이터 형식을 추가할 수 없도록 형식 안전성을 적용합니다. 제네릭 컬렉션에서 요소를 검색하는 경우 해당 데이터 형식을 결정하거나 변환할 필요가 없습니다.

## NOTE

이 항목의 예제에서는 `System.Collections.Generic` 및 `System.Linq` 네임스페이스에 대한 `using` 지시문을 포함합니다.

## 항목 내용

- [간단한 컬렉션 사용](#)
- [컬렉션 종류](#)
  - [System.Collections.Generic](#) 클래스
  - [System.Collections.Concurrent](#) 클래스
  - [System.Collections](#) 클래스
- [키/값 쌍의 컬렉션 구현](#)
- [LINQ를 사용하여 컬렉션에 액세스](#)
- [컬렉션 정렬](#)
- [사용자 지정 컬렉션 정의](#)
- [반복기](#)

## 간단한 컬렉션 사용

이 섹션의 예제에서는 강력한 형식의 개체 목록을 사용할 수 있게 해주는 제네릭 `List<T>` 클래스를 사용합니다.

다음 예제에서는 문자열 목록을 만든 다음, `foreach` 문을 사용하여 문자열을 반복합니다.

```
// Create a list of strings.  
var salmons = new List<string>();  
salmons.Add("chinook");  
salmons.Add("coho");  
salmons.Add("pink");  
salmons.Add("sockeye");  
  
// Iterate through the list.  
foreach (var salmon in salmons)  
{  
    Console.WriteLine(salmon + " ");  
}  
// Output: chinook coho pink sockeye
```

컬렉션의 내용을 사전에 알고 있는 경우 [컬렉션 초기화](#)를 사용하여 컬렉션을 초기화할 수 있습니다. 자세한 내용은 [개체 및 컬렉션 초기화](#)를 참조하세요.

다음 예제는 컬렉션 초기화를 사용하여 컬렉션에 요소를 추가한다는 점을 제외하고 이전 예제와 같습니다.

```
// Create a list of strings by using a  
// collection initializer.  
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };  
  
// Iterate through the list.  
foreach (var salmon in salmons)  
{  
    Console.WriteLine(salmon + " ");  
}  
// Output: chinook coho pink sockeye
```

`foreach` 문 대신 `for` 문을 사용하여 컬렉션을 반복할 수 있습니다. 인덱스 위치에 따라 컬렉션 요소에 액세스하여 이 작업을 수행합니다. 요소의 인덱스는 0부터 시작하고 요소 개수-1에서 끝납니다.

다음 예제에서는 `foreach` 대신 `for`를 사용하여 컬렉션의 요소를 반복합니다.

```
// Create a list of strings by using a  
// collection initializer.  
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };  
  
for (var index = 0; index < salmons.Count; index++)  
{  
    Console.WriteLine(salmons[index] + " ");  
}  
// Output: chinook coho pink sockeye
```

다음 예제에서는 제거할 개체를 지정하여 컬렉션에서 요소를 제거합니다.

```

// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Remove an element from the list by specifying
// the object.
salmons.Remove("coho");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.WriteLine(salmon + " ");
}
// Output: chinook pink sockeye

```

다음 예제에서는 제네릭 목록에서 요소를 제거합니다. `foreach` 문 대신 내림차순으로 반복하는 `for` 문을 사용합니다. 이는 `RemoveAt` 메서드로 인해 제거된 요소 뒤의 요소가 더 낮은 인덱스 값을 갖기 때문입니다.

```

var numbers = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Remove odd numbers.
for (var index = numbers.Count - 1; index >= 0; index--)
{
    if (numbers[index] % 2 == 1)
    {
        // Remove the element by specifying
        // the zero-based index in the list.
        numbers.RemoveAt(index);
    }
}

// Iterate through the list.
// A lambda expression is placed in the ForEach method
// of the List(T) object.
numbers.ForEach(
    number => Console.WriteLine(number + " "));
// Output: 0 2 4 6 8

```

`List<T>`의 요소 형식에 대해 고유한 클래스를 정의할 수도 있습니다. 다음 예제에서, `List<T>`에서 사용되는 `Galaxy` 클래스는 코드에서 정의됩니다.

```

private static void IterateThroughList()
{
    var theGalaxies = new List<Galaxy>
    {
        new Galaxy() { Name="Tadpole", MegaLightYears=400},
        new Galaxy() { Name="Pinwheel", MegaLightYears=25},
        new Galaxy() { Name="Milky Way", MegaLightYears=0},
        new Galaxy() { Name="Andromeda", MegaLightYears=3}
    };

    foreach (Galaxy theGalaxy in theGalaxies)
    {
        Console.WriteLine(theGalaxy.Name + " " + theGalaxy.MegaLightYears);
    }

    // Output:
    // Tadpole 400
    // Pinwheel 25
    // Milky Way 0
    // Andromeda 3
}

public class Galaxy
{
    public string Name { get; set; }
    public int MegaLightYears { get; set; }
}

```

## 컬렉션 종류

.NET은 다양한 일반적인 컬렉션을 제공합니다. 컬렉션의 각 형식은 특정 목적에 맞게 설계되었습니다.

이 섹션에서는 다음 몇 가지 일반적인 컬렉션 클래스에 대해 설명합니다.

- [System.Collections.Generic](#) 클래스
- [System.Collections.Concurrent](#) 클래스
- [System.Collections](#) 클래스

### **System.Collections.Generic** 클래스

[System.Collections.Generic](#) 네임스페이스의 클래스 중 하나를 사용하여 제네릭 컬렉션을 만들 수 있습니다. 제네릭 컬렉션은 컬렉션의 모든 항목에 동일한 데이터 형식이 있는 경우에 유용합니다. 제네릭 컬렉션은 원하는 데이터 형식만 추가할 수 있도록 하여 강력한 형식 지정을 적용합니다.

다음 표에서는 자주 사용되는 [System.Collections.Generic](#) 네임스페이스 클래스 중 일부를 보여 줍니다.

클래스	DESCRIPTION
<a href="#">Dictionary&lt; TKey, TValue &gt;</a>	키에 따라 구성된 키/값 쌍의 컬렉션을 나타냅니다.
<a href="#">List&lt; T &gt;</a>	인덱스로 액세스할 수 있는 개체 목록을 나타냅니다. 목록의 검색, 정렬 및 수정에 사용할 수 있는 메서드를 제공합니다.
<a href="#">Queue&lt; T &gt;</a>	FIFO(선입선출) 방식의 개체 컬렉션을 나타냅니다.
<a href="#">SortedList&lt; TKey, TValue &gt;</a>	연관된 <a href="#">IComparer&lt; T &gt;</a> 구현을 기반으로 키에 따라 정렬된 키/값 쌍의 컬렉션을 나타냅니다.
<a href="#">Stack&lt; T &gt;</a>	LIFO(후입선출) 방식의 개체 컬렉션을 나타냅니다.

자세한 내용은 [일반적으로 사용되는 컬렉션 형식](#), [Collection 클래스 선택](#) 및 [System.Collections.Generic](#)을 참조하세요.

### System.Collections.Concurrent 클래스

.NET Framework 4 이상 버전에서 [System.Collections.Concurrent](#) 네임스페이스의 컬렉션은 여러 스레드에서 컬렉션 항목에 액세스하기 위한 효율적이고 스레드로부터 안전한 작업을 제공합니다.

여러 스레드가 동시에 컬렉션에 액세스할 때마다 [System.Collections.Generic](#) 및 [System.Collections](#)의 해당 형식 대신 [System.Collections.Concurrent](#) 네임스페이스의 클래스를 사용해야 합니다. 자세한 내용은 [스레드로부터 안전한 컬렉션](#) 및 [System.Collections.Concurrent](#)을 참조하세요.

[System.Collections.Concurrent](#) 네임스페이스에 포함된 일부 클래스는 [BlockingCollection<T>](#), [ConcurrentDictionary< TKey, TValue >](#), [ConcurrentQueue<T>](#) 및 [ConcurrentStack<T>](#)입니다.

### System.Collections 클래스

[System.Collections](#) 네임스페이스의 클래스는 구체적 형식의 개체가 아니라 [Object](#) 형식의 개체로 요소를 저장합니다.

가능하면 항상 [System.Collections](#) 네임스페이스의 레거시 형식 대신 [System.Collections.Generic](#) 네임스페이스 또는 [System.Collections.Concurrent](#) 네임스페이스의 제네릭 컬렉션을 사용해야 합니다.

다음 표에서는 자주 사용되는 [System.Collections](#) 네임스페이스 클래스 중 일부를 보여 줍니다.

클래스	설명
<a href="#">ArrayList</a>	필요에 따라 크기가 동적으로 증가하는 개체 배열을 나타냅니다.
<a href="#">Hashtable</a>	키의 해시 코드에 따라 구성된 키/값 쌍의 컬렉션을 나타냅니다.
<a href="#">Queue</a>	FIFO(선입선출) 방식의 개체 컬렉션을 나타냅니다.
<a href="#">Stack</a>	LIFO(후입선출) 방식의 개체 컬렉션을 나타냅니다.

[System.Collections.Specialized](#) 네임스페이스는 문자열 전용 컬렉션 및 연결된 목록과 하이브리드 사전 등의 특수한 강력한 형식의 컬렉션 클래스를 제공합니다.

### 키/값 쌍의 컬렉션 구현

[Dictionary< TKey, TValue >](#) 제네릭 컬렉션을 사용하면 각 요소의 키를 통해 컬렉션의 요소에 액세스할 수 있습니다. 사전에 추가하는 각 항목은 값과 관련 키로 이루어져 있습니다. [Dictionary](#) 클래스는 해시 테이블로 구현되므로 해당 키를 사용하여 값을 검색하는 것이 빠릅니다.

다음 예제에서는 [Dictionary](#) 컬렉션을 만들고 [foreach](#) 문을 사용하여 사전을 반복합니다.

```

private static void IterateThruDictionary()
{
    Dictionary<string, Element> elements = BuildDictionary();

    foreach (KeyValuePair<string, Element> kvp in elements)
    {
        Element theElement = kvp.Value;

        Console.WriteLine("key: " + kvp.Key);
        Console.WriteLine("values: " + theElement.Symbol + " " +
            theElement.Name + " " + theElement.AtomicNumber);
    }
}

private static Dictionary<string, Element> BuildDictionary()
{
    var elements = new Dictionary<string, Element>();

    AddToDictionary(elements, "K", "Potassium", 19);
    AddToDictionary(elements, "Ca", "Calcium", 20);
    AddToDictionary(elements, "Sc", "Scandium", 21);
    AddToDictionary(elements, "Ti", "Titanium", 22);

    return elements;
}

private static void AddToDictionary(Dictionary<string, Element> elements,
    string symbol, string name, int atomicNumber)
{
    Element theElement = new Element();

    theElement.Symbol = symbol;
    theElement.Name = name;
    theElement.AtomicNumber = atomicNumber;

    elements.Add(key: theElement.Symbol, value: theElement);
}

public class Element
{
    public string Symbol { get; set; }
    public string Name { get; set; }
    public int AtomicNumber { get; set; }
}

```

대신 컬렉션 인터페이스를 사용하여 `Dictionary` 컬렉션을 빌드하려면 `BuildDictionary` 및 `AddToDictionary` 메서드를 다음 메서드로 바꾸면 됩니다.

```

private static Dictionary<string, Element> BuildDictionary2()
{
    return new Dictionary<string, Element>
    {
        {"K",
            new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
        {"Ca",
            new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        {"Sc",
            new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        {"Ti",
            new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
}

```

다음 예제에서는 `Dictionary`의 `ContainsKey` 메서드 및 `Item[]` 속성을 사용하여 키를 통해 항목을 신속하게 찾습니다.

니다. `Item` 속성을 사용하면 C#에서 `elements[symbol]`를 통해 `elements` 컬렉션의 항목에 액세스할 수 있습니다.

```
private static void FindInDictionary(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

    if (elements.ContainsKey(symbol) == false)
    {
        Console.WriteLine(symbol + " not found");
    }
    else
    {
        Element theElement = elements[symbol];
        Console.WriteLine("found: " + theElement.Name);
    }
}
```

다음 예제에서는 대신 `TryGetValue` 메서드를 사용하여 키를 통해 항목을 신속하게 찾습니다.

```
private static void FindInDictionary2(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

    Element theElement = null;
    if (elements.TryGetValue(symbol, out theElement) == false)
        Console.WriteLine(symbol + " not found");
    else
        Console.WriteLine("found: " + theElement.Name);
}
```

## LINQ를 사용하여 컬렉션에 액세스

LINQ(통합 언어 쿼리)를 사용하여 컬렉션에 액세스할 수 있습니다. LINQ 쿼리는 필터링, 정렬 및 그룹화 기능을 제공합니다. 자세한 내용은 [C#에서 LINQ 시작](#)을 참조하세요.

다음 예제에서는 제네릭 `List`에 대해 LINQ 쿼리를 실행합니다. LINQ 쿼리는 결과를 포함하는 다른 컬렉션을 반환합니다.

```

private static void ShowLINQ()
{
    List<Element> elements = BuildList();

    // LINQ Query.
    var subset = from theElement in elements
                where theElement.AtomicNumber < 22
                orderby theElement.Name
                select theElement;

    foreach (Element theElement in subset)
    {
        Console.WriteLine(theElement.Name + " " + theElement.AtomicNumber);
    }

    // Output:
    // Calcium 20
    // Potassium 19
    // Scandium 21
}

private static List<Element> BuildList()
{
    return new List<Element>
    {
        { new Element() { Symbol="K", Name="Potassium", AtomicNumber=19} },
        { new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20} },
        { new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21} },
        { new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22} }
    };
}

public class Element
{
    public string Symbol { get; set; }
    public string Name { get; set; }
    public int AtomicNumber { get; set; }
}

```

## 컬렉션 정렬

다음 예제에서는 컬렉션 정렬 절차를 보여 줍니다. 예제에서는 `List<T>`에 저장된 `Car` 클래스 인스턴스를 정렬합니다. `Car` 클래스는 `CompareTo` 메서드가 구현되어야 하는 `IComparable<T>` 인터페이스를 구현합니다.

`CompareTo` 메서드를 호출할 때마다 정렬에 사용되는 단일 비교가 수행됩니다. `CompareTo` 메서드의 사용자 작성 코드는 다른 개체와 현재 개체의 각 비교에 대한 값을 반환합니다. 현재 개체가 다른 개체보다 작으면 반환되는 값이 0보다 작고, 현재 개체가 다른 개체보다 크면 0보다 크고, 같으면 0입니다. 이렇게 하면 보다 큼, 보다 작음 및 같음에 대한 조건을 코드에서 정의할 수 있습니다.

`ListCars` 메서드에서 `cars.Sort()` 문은 목록을 정렬합니다. `List<T>`의 `Sort` 메서드를 호출하면 `List`의 `Car` 개체에 대해 `CompareTo` 메서드가 자동으로 호출됩니다.

```

private static void ListCars()
{
    var cars = new List<Car>
    {
        { new Car() { Name = "car1", Color = "blue", Speed = 20} },
        { new Car() { Name = "car2", Color = "red", Speed = 50} },
        { new Car() { Name = "car3", Color = "green", Speed = 10} },
        { new Car() { Name = "car4", Color = "blue", Speed = 50} },
        { new Car() { Name = "car5", Color = "blue", Speed = 30} },
        { new Car() { Name = "car6", Color = "red", Speed = 60} },
        { new Car() { Name = "car7", Color = "green", Speed = 50} }
    };
}
```

```

};

// Sort the cars by color alphabetically, and then by speed
// in descending order.
cars.Sort();

// View all of the cars.
foreach (Car thisCar in cars)
{
    Console.Write(thisCar.Color.PadRight(5) + " ");
    Console.Write(thisCar.Speed.ToString() + " ");
    Console.WriteLine(thisCar.Name);
    Console.WriteLine();
}

// Output:
// blue 50 car4
// blue 30 car5
// blue 20 car1
// green 50 car7
// green 10 car3
// red 60 car6
// red 50 car2
}

public class Car : IComparable<Car>
{
    public string Name { get; set; }
    public int Speed { get; set; }
    public string Color { get; set; }

    public int CompareTo(Car other)
    {
        // A call to this method makes a single comparison that is
        // used for sorting.

        // Determine the relative order of the objects being compared.
        // Sort by color alphabetically, and then by speed in
        // descending order.

        // Compare the colors.
        int compare;
        compare = String.Compare(this.Color, other.Color, true);

        // If the colors are the same, compare the speeds.
        if (compare == 0)
        {
            compare = this.Speed.CompareTo(other.Speed);

            // Use descending order for speed.
            compare = -compare;
        }

        return compare;
    }
}

```

## 사용자 지정 컬렉션 정의

`IEnumerable<T>` 또는 `IEnumerable` 인터페이스를 구현하여 컬렉션을 정의할 수 있습니다.

사용자 지정 컬렉션을 정의할 수도 있지만, 일반적으로 이 문서의 앞부분에 있는 [컬렉션 종류](#)에서 설명한 .NET에 포함된 컬렉션을 대신 사용하는 것이 좋습니다.

다음 예제에서는 `AllColors`라는 사용자 지정 컬렉션 클래스를 정의합니다. 이 클래스는 `GetEnumerator` 메서드가 구현되어야 하는 `IEnumerable` 인터페이스를 구현합니다.

`GetEnumerator` 메서드는 `ColorEnumerator` 클래스의 인스턴스를 반환합니다. `ColorEnumerator`는 `Current` 속성, `MoveNext` 메서드 및 `Reset` 메서드가 구현되어야 하는 `IEnumerator` 인터페이스를 구현합니다.

```
private static void ListColors()
{
    var colors = new AllColors();

    foreach (Color theColor in colors)
    {
        Console.Write(theColor.Name + " ");
    }
    Console.WriteLine();
    // Output: red blue green
}

// Collection class.
public class AllColors : System.Collections.IEnumerable
{
    Color[] _colors =
    {
        new Color() { Name = "red" },
        new Color() { Name = "blue" },
        new Color() { Name = "green" }
    };

    public System.Collections.IEnumerator GetEnumerator()
    {
        return new ColorEnumerator(_colors);

        // Instead of creating a custom enumerator, you could
        // use the GetEnumerator of the array.
        //return _colors.GetEnumerator();
    }

    // Custom enumerator.
    private class ColorEnumerator : System.Collections.IEnumerator
    {
        private Color[] _colors;
        private int _position = -1;

        public ColorEnumerator(Color[] colors)
        {
            _colors = colors;
        }

        object System.Collections.IEnumerator.Current
        {
            get
            {
                return _colors[_position];
            }
        }

        bool System.Collections.IEnumerator.MoveNext()
        {
            _position++;
            return (_position < _colors.Length);
        }

        void System.Collections.IEnumerator.Reset()
        {
            _position = -1;
        }
    }
}

// Element class.
```

```
public class Color
{
    public string Name { get; set; }
}
```

## 반복기

반복기는 컬렉션에 대해 사용자 지정 반복을 수행하는 데 사용됩니다. 반복기는 메서드 또는 `get` 접근자일 수 있습니다. 반복기는 `yield return` 문을 사용하여 한 번에 하나씩 컬렉션의 각 요소를 반환합니다.

`foreach` 문을 사용하여 반복기를 호출합니다. 각각의 `foreach` 루프의 반복이 반복기를 호출합니다. `yield return` 문이 반복기 메서드에 도달하면 식이 반환되고 코드에서 현재 위치는 유지됩니다. 다음에 반복기가 호출되면 해당 위치에서 실행이 다시 시작됩니다.

자세한 내용은 [반복기\(C#\)](#)를 참조하세요.

다음 예제에서는 반복기 메서드를 사용합니다. 반복기 메서드는 `for` 루프 내부에 있는 `yield return` 문을 포함합니다. `ListEvenNumbers` 메서드에서 `foreach` 문 본문을 반복할 때마다 다음 `yield return` 문으로 진행하는 반복기 메서드에 대한 호출이 생성됩니다.

```
private static void ListEvenNumbers()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.WriteLine(number.ToString() + " ");
    }
    Console.WriteLine();
    // Output: 6 8 10 12 14 16 18
}

private static IEnumerable<int> EvenSequence(
    int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (var number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```

## 참조

- [개체 이니셜라이저 및 컬렉션 이니셜라이저](#)
- [프로그래밍 개념\(C#\)](#)
- [Option Strict 문](#)
- [LINQ to Objects\(C#\)](#)
- [PLINQ\(병렬 LINQ\)](#)
- [컬렉션 및 데이터 구조](#)
- [Collection 클래스 선택](#)
- [컬렉션 내에서 비교 및 정렬](#)
- [제네릭 컬렉션 사용 기준](#)

# 공변성(Covariance) 및 반공변성(Contravariance) (C#)

2020-11-02 • 7 minutes to read • [Edit Online](#)

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.

```
// 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.

```
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.

```

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.

```

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

제목	설명
<a href="#">제네릭 인터페이스의 가변성(C#)</a>	제네릭 인터페이스의 공변성(Covariance) 및 반공변성(Contravariance)에 대해 설명하고 .NET의 Variant 제네릭 인터페이스 목록을 제공합니다.
<a href="#">Variant 제네릭 인터페이스 만들기(C#)</a>	사용자 지정 variant 인터페이스를 만드는 방법을 보여 줍니다.
<a href="#">제네릭 컬렉션용 인터페이스의 가변성 사용(C#)</a>	IEnumerable<T> 및 IComparable<T> 인터페이스의 공변성(Covariance) 및 반공변성(Contravariance) 지원을 통해 코드를 다시 사용하는 방법을 보여 줍니다.
<a href="#">대리자의 가변성(C#)</a>	제네릭 및 제네릭이 아닌 대리자의 공변성(Covariance) 및 반공변성(Contravariance)을 설명하고 .NET의 Variant 제네릭 인터페이스 목록을 제공합니다.
<a href="#">대리자의 가변성 사용(C#)</a>	제네릭이 아닌 대리자의 공변성(Covariance) 및 반공변성(Contravariance) 지원을 사용하여 메서드 시그니처를 대리자 형식과 일치시키는 방법을 보여 줍니다.
<a href="#">Func 및 Action 제네릭 대리자에 가변성 사용(C#)</a>	Func 및 Action 대리자의 공변성(Covariance) 및 반공변성(Contravariance) 지원을 통해 코드를 다시 사용하는 방법을 보여 줍니다.

# 제네릭 인터페이스의 가변성(C#)

2020-11-02 • 6 minutes to read • [Edit Online](#)

.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>`(T는 공변(covariant)임)
- `IEnumerator<T>`(T는 공변(covariant)임)
- `IQueryable<T>`(T는 공변(covariant)임)
- `IGrouping< TKey, TElement >`(`TKey` 및 `TElement`는 공변(covariant)임)
- `IComparer<T>`(T는 반공변(contravariant)임)
- `IEqualityComparer<T>`(T는 반공변(contravariant)임)
- `IComparable<T>`(T는 반공변(contravariant)임)

.NET Framework 4.5부터 다음 인터페이스는 variant입니다.

- `IReadOnlyList<T>`(T는 공변(covariant)임)
- `IReadOnlyCollection<T>`(T는 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.

```
IEnumerable<String> strings = new List<String>();  
IEnumerable<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 비교하는데 `BaseComparer`를 사용할 수 있습니다. 다음 코드 예제에서 이를 확인할 수 있습니다.

```

// 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.

```

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 변환 할 수 없습니다. 다음 코드 예제에서 이 내용을 보여 줍니다.

```

// 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#)

2020-11-02 • 13 minutes to read • [Edit Online](#)

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.

## NOTE

.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.

## IMPORTANT

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.

```
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 대리자가 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.

```
interface ICovariant<out R>
{
    void DoSomething(Action<R> callback);
}
```

- 형식은 인터페이스 메서드에 대한 제네릭 제약 조건으로 사용되지 않습니다. 이는 다음 코드에 설명되어 있습니다.

```

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.

```

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.

```

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.

```

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.

```

// 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.

```

interface ICovariant<out T> { }
interface IInvariant<T> : ICovariant<T> { }
interface IExtCovariant<out T> : ICovariant<T> { }

```

두 인터페이스가 모두 같은 인터페이스를 확장하더라도 `IInvariant<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.

```

interface ICovariant<out T> { }
interface IContravariant<in T> { }
interface IInvariant<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.

```

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.

```

// 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#)

2020-11-02 • 4 minutes to read • [Edit Online](#)

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.

```
// 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 `IComparer<T>` 인터페이스에서 반공변성(Contravariance) 지원의 이점을 보여 줍니다.

`PersonComparer` 클래스가 `IComparer<Person>` 인터페이스를 구현합니다. 그러나 `Employee`는 `Person`을 상속하

므로 `Employee` 형식 개체의 시퀀스를 비교하기 위해 이 클래스를 다시 사용할 수 있습니다.

```
// 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#)

2020-11-02 • 13 minutes to read • [Edit Online](#)

.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.

```
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.

```
// 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.

```
// 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.

```
// 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.

```
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,TOutput>](#) 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.

```
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.

```
public delegate void DContravariant<in A>(A a);
```

#### IMPORTANT

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.

```
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.

```
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.

```
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.

```
// 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#)

2020-11-02 • 4 minutes to read • [Edit Online](#)

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

### 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

```
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.

```
public delegate void KeyEventHandler(object sender, KeyEventArgs e)
```

- `Button.MouseClick` 이벤트의 시그니처를 정의하는 `MouseEventHandler` 대리자. 해당 시그니처는 다음과 같습니다.

```
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. `EventArgs`는 `KeyEventArgs` 및 `MouseEventArgs`의 기본 형식이므로 이 작업을 수행할 수 있습니다.

## 코드

```
// 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;  
}
```

## 참조

- 대리자의 가변성(C#)
- `Func` 및 `Action` 제네릭 대리자에 가변성 사용(C#)

# Func 및 Action 제네릭 대리자에 가변성 사용(C#)

2020-11-02 • 3 minutes to read • [Edit Online](#)

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.

```
// 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.

```
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](#)

# 식 트리(C#)

2020-11-02 • 10 minutes to read • [Edit Online](#)

식 트리는 `x < y` 등의 이진 연산이나 메서드 호출과 같이 각 노드가 식인 트리 형식 데이터 구조의 코드를 표시합니다.

식 트리로 표시되는 코드를 컴파일하고 실행할 수 있습니다. 이렇게 하면 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\(C#\)](#)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 식에서 식 트리 만들기

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 생성할 수 있으며 문 람다(여러 줄 람다)는 구문 분석할 수 없습니다. C#의 람다 식에 대한 자세한 내용은 [람다 식](#)을 참조하세요.

다음 코드 예제에서는 C# 컴파일러에서 람다 식 `num => num < 5`를 나타내는 식 트리를 만드는 방법을 보여 줍니다.

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

## API를 사용하여 식 트리 만들기

API를 사용하여 식 트리를 만들려면 [Expression](#) 클래스를 사용합니다. 이 클래스는 변수나 매개 변수를 나타내는 [ParameterExpression](#) 또는 메서드 호출을 나타내는 [MethodCallExpression](#)과 같이 특정 형식의 식 트리 노드를 만드는 정적 팩터리 메서드를 포함합니다. [ParameterExpression](#), [MethodCallExpression](#) 및 기타 식 관련 형식도 [System.Linq.Expressions](#) 네임스페이스에서 정의됩니다. 이러한 형식은 추상 형식 [Expression](#)에서 파생됩니다.

다음 코드 예제에서는 API를 사용하여 람다 식 `num => num < 5`를 나타내는 식 트리를 만드는 방법을 보여 줍니다.

```

// Add the following using directive to your code file:
// using System.Linq.Expressions;

// 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 });

```

.NET Framework 4 이상에서는 식 트리 API가 루프, 조건부 블록, `try-catch` 블록 등의 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.

```

// 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.

## 식 트리 구문 분석

다음 코드 예제에서는 람다 식 `num => num < 5`를 나타내는 식 트리를 개별 구성 요소로 구성 해제하는 방법을 보여 줍니다.

```

// 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

```

## 변경 불가능한 식 트리

식 트리는 변경할 수 없어야 합니다. 즉, 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 트래버스할 수 있습니다. 자세한 내용은 [식 트리 수정 방법\(C#\)](#)을 참조하세요.

## 식 트리 컴파일

`Expression<TDelegate>` 형식은 식 트리로 표시되는 코드를 실행 가능한 대리자로 컴파일하는 `Compile` 메서드를 제공합니다.

다음 코드 예제에서는 식 트리를 컴파일하고 결과 코드를 실행하는 방법을 보여 줍니다.

```

// Creating an expression tree.
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.

```

자세한 내용은 [식 트리 실행 방법\(C#\)](#)을 참조하세요.

## 참조

- [System.Linq.Expressions](#)
- [식 트리 실행 방법\(C#\)](#)
- [식 트리 수정 방법\(C#\)](#)
- [람다 식](#)
- [동적 언어 런타임 개요](#)

- 프로그래밍 개념(C#)

# 식 트리 실행 방법(C#)

2020-11-02 • 3 minutes to read • [Edit Online](#)

이 항목에서는 식 트리를 실행하는 방법을 보여 줍니다. 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.

## NOTE

대리자의 형식을 알 수 없는 경우, 즉 람다 식이 [Expression<TDelegate>](#) 형식이 아니라 [LambdaExpression](#) 형식인 경우 대리자를 직접 호출하는 대신 대리자의 [DynamicInvoke](#) 메서드를 호출해야 합니다.

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.

## 예제

다음 코드 예제에서는 람다 식을 만들고 실행하여 숫자의 거듭제곱을 나타내는 식 트리를 실행하는 방법을 보여 줍니다. 숫자의 거듭제곱을 나타내는 결과가 표시됩니다.

```
// 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
```

## 코드 컴파일

- System.Linq.Expressions 네임스페이스를 포함합니다.

## 참조

- [식 트리\(C#\)](#)
- [식 트리 수정 방법\(C#\)](#)

# 식 트리 수정 방법(C#)

2020-11-02 • 5 minutes to read • [Edit Online](#)

이 항목에서는 식 트리를 수정하는 방법을 보여 줍니다. 식 트리는 변경할 수 없으며, 직접 수정할 수 없음을 의미합니다. 식 트리를 변경하려면 기존 식 트리의 복사본을 만들고, 해당 복사본을 만들 때 필요한 사항을 변경해야 합니다. [ExpressionVisitor](#) 클래스를 사용하여 기존 식 트리를 트래버스하고 방문하는 각 노드를 복사할 수 있습니다.

식 트리를 수정하려면

1. 콘솔 애플리케이션 프로젝트를 새로 만듭니다.
2. `System.Linq.Expressions` 네임스페이스에 대한 `using` 지시문을 파일에 추가합니다.
3. 프로젝트에 `AndAlsoModifier` 클래스를 추가합니다.

```
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 이 작업을 위해 클래스는 기본 형식의 `VisitBinary` 메서드를 재정의합니다. `VisitBinary` 메서드에서 전달된 식이 조건부 `AND` 작업을 나타내는 경우 코드는 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.

4. `System.Linq.Expressions` 네임스페이스에 대한 `using` 지시문을 파일에 추가합니다.
5. `Program.cs` 파일의 `Main` 메서드에 코드를 추가하여 식 트리를 만들고 이 식 트리를 수정할 메서드에 전달합니다.

```
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.

6. 애플리케이션을 컴파일하고 실행합니다.

## 참조

- [식 트리 실행 방법\(C#\)](#)
- [식 트리\(C#\)](#)

# 런타임 상태에 따라 쿼리(C#)

2021-02-18 • 18 minutes to read • [Edit Online](#)

데이터 소스에 대해 [IQueryable](#) 또는 [IQueryable<T>](#)을 정의하는 코드를 고려합니다.

```
var companyNames = new[] {
    "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"
};

// We're using 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);
```

해당 코드를 실행할 때마다 똑같은 쿼리가 실행됩니다. 코드에서 런타임에 조건에 따라 다른 쿼리를 실행하려고 할 수 있으므로 이 방식은 그다지 유용하지 않습니다. 이 문서에서는 런타임 상태에 따라 다른 쿼리를 실행하는 방법을 설명합니다.

## IQueryable/IQueryable<T> 및 식 트리

기본적으로 [IQueryable](#)에는 다음 두 가지 구성 요소가 있습니다.

- [Expression](#) — 식 트리 형식으로 현재 쿼리 구성 요소의 언어 및 데이터 소스 종립적 표현입니다.
- [Provider](#) — 현재 쿼리를 값 또는 값 세트로 구체화하는 방법을 인식하는 LINQ 공급자의 인스턴스입니다.

동적 쿼리 컨텍스트에서 공급자는 일반적으로 동일하게 유지됩니다. 쿼리의 식 트리는 쿼리별로 다릅니다.

식 트리는 변경할 수 없습니다. 다른 식 트리 — 및 이에 따라 다른 쿼리 — 가 필요한 경우에는 기존 식 트리를 새 식 트리로 변환하고 이에 따라 새 [IQueryable](#)로 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
- [Expression](#)에서 팩터리 메서드를 사용하여 [Expression<TDelegate>](#) 식 트리 생성
- [IQueryable](#)의 식 트리에 메서드 호출 노드 추가
- 문자열을 생성하고 [동적 LINQ 라이브러리](#) 사용

## 식 트리 내에서 런타임 상태 사용

LINQ 공급자가 지원하는 경우 동적으로 쿼리하는 가장 간단한 방법은 다음 코드 예제의 `length` 같이 둘러싸인 변수를 통해 쿼리에서 직접 런타임 상태를 참조하는 것입니다.

```

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 트리(및 이에 따라 쿼리)는 수정되지 않았습니다. `length` 값이 변경되었기 때문에 쿼리는 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 [IQueryable<T>](#) 반환 메서드의 결과로 바꿔서 새 쿼리를 가져올 수 있습니다. 다음 예제와 같이 런타임 상태에 따라 해당 작업을 수행할 수 있습니다.

```

// 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.

```

// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>> expr = (startsWith, endsWith) switch
{
    ("", "") => x => true,
    (_, "") => x => x.StartsWith(startsWith),
    ("", _) => x => x.EndsWith(endsWith),
    (_, _) => x => x.StartsWith(startsWith) || x.EndsWith(endsWith)
};

var qry = companyNamesSource.Where(expr);

```

[LinqKit](#)의 [PredicateBuilder](#)와 같은 타사 라이브러리를 사용하여 다양한 하위 식을 작성할 수도 있습니다.

```

// 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

지금까지 모든 예제에서 컴파일 시간의 요소 형식 `string` 및 이에 따른 쿼리 형식 `IQueryable<string>`을 알아보았습니다. 요소 형식의 쿼리에 구성 요소를 추가해야 할 수 있습니다. 요소 형식에 따라 다른 구성 요소를 추가해야 할 수 있습니다. `System.Linq.Expressions.Expression`에서 팩터리 메서드를 사용하여 처음부터 식 트리를 만들 수 있으므로 특정 요소 형식에 맞게 식을 조정할 수 있습니다.

### `Expression<TDelegate>` 생성

LINQ 메서드 중 하나에 전달할 식을 생성하는 경우 실제로는 `Expression<TDelegate>`의 인스턴스를 생성하는 것입니다. 여기서 `TDelegate`는 `Func<string, bool>`, `Action` 또는 사용자 지정 대리자 형식과 같은 대리자 형식입니다.

`Expression<TDelegate>`는 다음과 같은 전체 람다 식을 나타내는 `LambdaExpression`에서 상속됩니다.

```
Expression<Func<string, bool>> expr = x => x.StartsWith("a");
```

`LambdaExpression`에는 다음 두 가지 구성 요소가 있습니다.

- `Parameters` 속성으로 표시되는 매개 변수 목록 `(string x)`
- `Body` 속성으로 표시되는 본문 `x.StartsWith("a")`

`Expression<TDelegate>`를 생성하는 기본 단계는 다음과 같습니다.

- `Parameter` 팩터리 메서드를 사용하여 람다 식에서 각 매개 변수(있는 경우)에 대해 `ParameterExpression` 개체를 정의합니다.

```
ParameterExpression x = Parameter(typeof(string), "x");
```

- 정의한 `ParameterExpression`을 사용하여 `LambdaExpression` 본문을 생성합니다. 예를 들어 `x.StartsWith("a")`를 나타내는 식은 다음과 같이 생성할 수 있습니다.

```
Expression body = Call(
    x,
    typeof(string).GetMethod("StartsWith", new [] {typeof(string)}),
    Constant("a")
);
```

- 적절한 **Lambda** 팩터리 메서드 오버로드를 사용하여 컴파일 시간 형식의 **Expression<TDelegate>**로 매개 변수와 본문을 래핑합니다.

```
Expression<Func<string, bool>> expr = Lambda<Func<string, bool>>(body, prm);
```

다음 섹션에서는 LINQ 메서드에 전달할 **Expression<TDelegate>**를 생성하려고 하는 시나리오를 설명하고 팩터리 메서드를 사용하여 해당 작업을 수행하는 방법의 전체 예제를 제공합니다.

### 시나리오

여러 엔터티 형식이 있다고 가정해 보겠습니다.

```
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 반환하겠습니다. **Person**의 경우 **FirstName** 및 **LastName** 속성을 검색하려고 합니다.

```
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.

```
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.

예제

```

// 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", new[] { 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`만이 아니라 `IQueryable<T>`를 사용하고 반환하기 때문에 텍스트 필터 뒤에 컴파일 시간 형식의 쿼리 요소를 더 추가할 수 있습니다.

```

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);

```

## IQueryable의 식 트리에 메서드 호출 노드 추가

`IQueryable<T>` 대신 `IQueryable`이 있는 경우 제네릭 LINQ 메서드를 직접 호출할 수 없습니다. 한 가지 대안은 위와 같이 내부 식 트리를 빌드하고 리플렉션을 사용하여 식 트리를 전달하는 동안 적절한 LINQ 메서드를 호출하는 것입니다.

LINQ 메서드 호출을 나타내는 `MethodCallExpression`으로 전체 트리를 래핑하여 LINQ 메서드의 기능을 복제할 수도 있습니다.

```

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",
        new[] { 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 `Expression<TDelegate>` 대신 `LambdaExpression`을 생성하는 `Lambda` 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.

```
// 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);
}
```

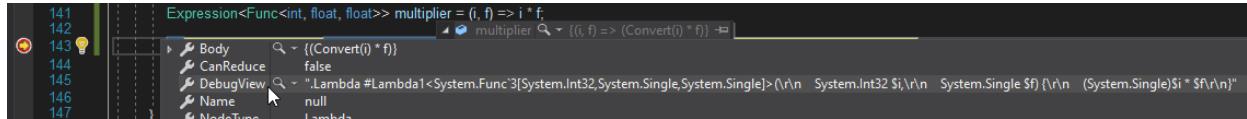
## 참조

- [식 트리\(C#\)](#)
- [식 트리 실행 방법\(C#\)](#)
- [런타임에 동적으로 조건자 필터 지정](#)

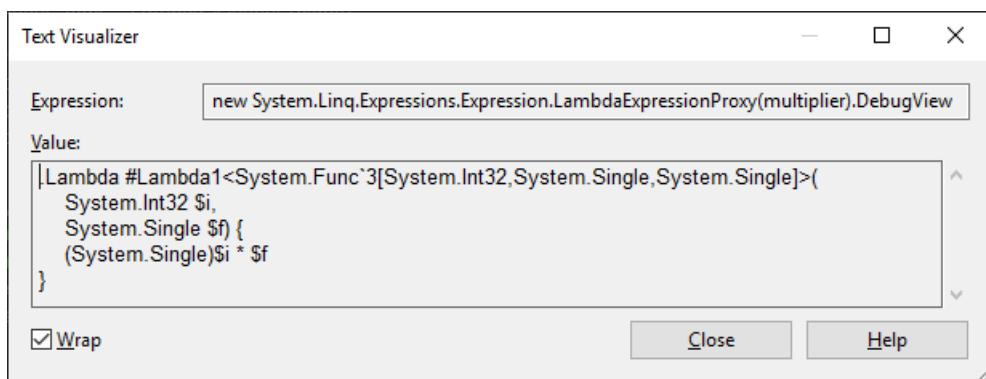
# Visual Studio에서 식 트리 디버그(C#)

2020-11-02 • 2 minutes to read • [Edit Online](#)

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.)

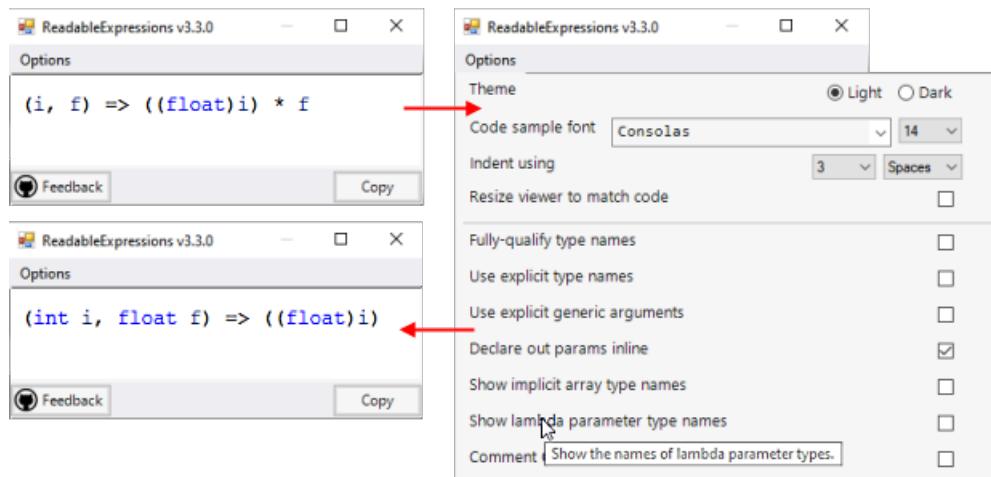


`DebugView`는 문자열이므로 `DebugView` 레이블 옆의 돋보기 아이콘에서 텍스트 시작화 도우미를 선택하여 [기본 제공 텍스트 시작화 도우미](#)를 사용하여 여러 줄에서 볼 수 있습니다.

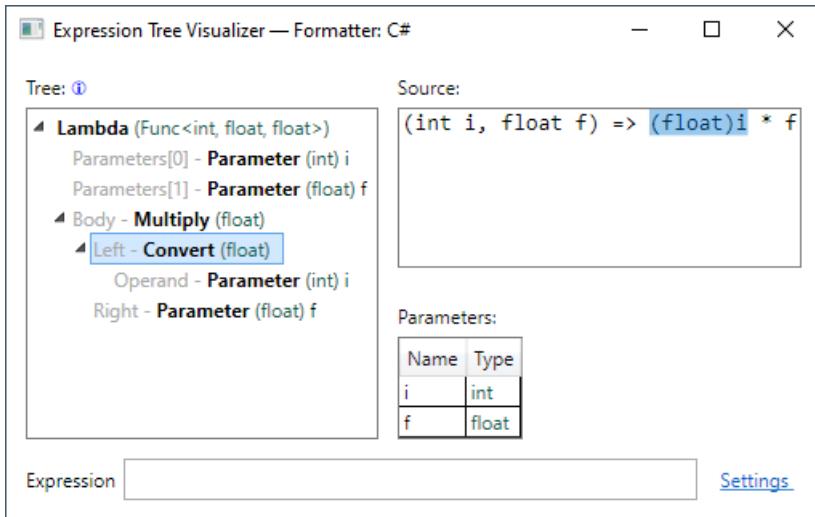


또는 다음과 같은 식 트리에 [사용자 지정 시작화 도우미](#)를 설치하여 사용할 수 있습니다.

- [읽을 수 있는 식\(MIT 라이선스, Visual Studio Marketplace\)](#)은 다양한 렌더링 옵션을 사용하여 식 트리를 테마 지정 가능 C# 코드로 렌더링 합니다.



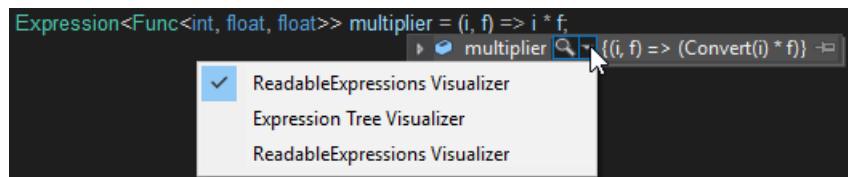
- [식 트리 시작화 도우미\(MIT 라이선스\)](#)에서는 식 트리 및 개별 노드의 트리 뷰를 제공합니다.



식 트리에 대한 시각화 도우미를 열려면

1. DataTips, 조사식 창, 자동 창 또는 지역 창의 식 트리 옆에 나타나는 돋보기 아이콘을 클릭합니다.

사용 가능한 시각화 도우미의 목록이 나타납니다.



2. 사용할 시각화 도우미를 클릭합니다.

## 참조

- [식 트리\(C#\)](#)
- [Visual Studio의 디버깅](#)
- [Create Custom Visualizers of Data](#)(데이터의 사용자 지정 시각화 도우미 만들기)
- [DebugView 구문](#)

# DebugView 구문

2020-05-20 • 4 minutes to read • [Edit Online](#)

`DebugView` 속성(디버깅 할 때만 사용 가능)은 식 트리의 문자열 렌더링을 제공합니다. 대부분의 구문은 이해하기 쉽습니다. 특별한 경우는 다음 섹션에서 설명합니다.

각 예제 다음에는 `DebugView` 를 포함한 블록 주석이 이어집니다.

## ParameterExpression

`System.Linq.Expressions.ParameterExpression` 변수 이름의 시작 부분에 `$` 기호가 표시됩니다.

매개 변수에 이름이 없으면 자동으로 생성된 이름이 할당됩니다(예: `$var1` 또는 `$var2`).

예

```
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
/*
    $num
*/

ParameterExpression numParam = Expression.Parameter(typeof(int));
/*
    $var1
*/
```

## ConstantExpression

정수 값, 문자열 및 `null` 을 나타내는 `System.Linq.Expressions.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.

형식	키워드	접미사
<code>System.UInt32</code>	<code>uint</code>	<code>U</code>
<code>System.Int64</code>	<code>long</code>	<code>L</code>
<code>System.UInt64</code>	<code>ulong</code>	<code>UL</code>
<code>System.Double</code>	<code>double</code>	<code>D</code>
<code>System.Single</code>	<code>float</code>	<code>F</code>
<code>System.Decimal</code>	<code>decimal</code>	<code>M</code>

예

```

int num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10
*/

double num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10D
*/

```

## BlockExpression

[System.Linq.Expressions.BlockExpression](#) 개체의 형식이 블록에 있는 마지막 식의 형식과 다를 경우 형식은 꺼쇠 괄호( `<` 및 `>` ) 안에 표시됩니다. 같은 경우 [BlockExpression](#) 개체의 형식이 표시되지 않습니다.

예

```

BlockExpression block = Expression.Block(Expression.Constant("test"));
/*
    .Block() {
        "test"
    }
*/

BlockExpression block = Expression.Block(typeof(Object), Expression.Constant("test"));
/*
    .Block<System.Object>() {
        "test"
    }
*/

```

## LambdaExpression

[System.Linq.Expressions.LambdaExpression](#) 개체는 대리자 형식과 함께 표시됩니다.

람다 식에 이름이 없으면 자동으로 생성된 이름이 할당됩니다(예: `#Lambda1` 또는 `#Lambda2` ).

예

```

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

[System.Linq.Expressions.LabelExpression](#) 개체의 기본값을 지정하면 이 값은 [System.Linq.Expressions.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 ).

예

```
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.

예

```
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
*/

```

# 반복기(C#)

2020-11-02 • 16 minutes to read • [Edit Online](#)

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.

```
static void Main()
{
    foreach (int number in SomeNumbers())
    {
        Console.WriteLine(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.

## NOTE

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.

```

static void Main()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.WriteLine(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 `DaysOfTheWeek` 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.

```

static void Main()
{
    DaysOfTheWeek days = new DaysOfTheWeek();

    foreach (string day in days)
    {
        Console.WriteLine(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.

```
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.Write(name + " ");
    }
    Console.WriteLine();
    // Output: Penguin Warbler

    foreach (string name in theZoo.Mammals)
    {
        Console.Write(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.

```

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];
            }
        }

        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }

        public IEnumerable<T> TopToBottom
        {
            get { return this; }
        }

        public IEnumerable<T> BottomToTop
        {
            get
            {
                for (int index = 0; index <= top - 1; index++)
                {
                    yield return values[index];
                }
            }
        }

        public IEnumerable<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 `IEnumerable<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`은 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 `Il2asm.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 때 까지 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`을 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 페이지링 폐치가 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`
- `yield`
- 배열에 `foreach` 사용
- 제네릭

# LINQ(Language-Integrated Query)

2020-11-02 • 10 minutes to read • [Edit Online](#)

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 데이터 원본의 각 유형에 대해 다른 쿼리 언어를 배워야 합니다. SQL 데이터베이스, XML 문서, 다양한 웹 서비스 등. LINQ를 사용할 경우 쿼리는 클래스, 메서드, 이벤트와 같은 고급 언어 구문이 됩니다. 언어 키워드 및 친숙한 연산자를 사용하여 강력한 형식의 개체 컬렉션에 대해 쿼리를 작성합니다. LINQ 기술은 개체(LINQ to Objects), 관계형 데이터베이스(LINQ to SQL) 및 XML(LINQ to XML)에 대한 일관성 있는 쿼리 환경을 제공합니다.

쿼리를 작성하는 개발자의 경우 LINQ에서 가장 눈에 잘 띠는 "언어 통합" 부분은 쿼리 식입니다. 쿼리 식은 선언적 쿼리 구문으로 작성됩니다. 쿼리 구문을 사용하면 최소한의 코드로 데이터 소스에 대해 필터링, 정렬 및 그룹화 작업을 수행할 수 있습니다. 동일한 기본 쿼리 식 패턴을 사용하여 SQL 데이터베이스, ADO.NET 데이터 세트, XML 문서 및 스트림, .NET 컬렉션에서 데이터를 쿼리하고 변환할 수 있습니다.

SQL Server 데이터베이스, XML 문서, ADO.NET 데이터 세트 및 `IEnumerable` 또는 제네릭 `IEnumerable<T>` 인터페이스를 지원하는 모든 개체 컬렉션에 대해 C#으로 LINQ 쿼리를 작성할 수 있습니다. LINQ는 많은 웹 서비스 및 기타 데이터베이스 구현을 위해 타사에서도 지원됩니다.

다음 예제에서는 전체 쿼리 작업을 보여 줍니다. 전체 작업에는 데이터 소스 만들기, 쿼리 식 정의 및 `foreach` 문의 쿼리 실행이 포함됩니다.

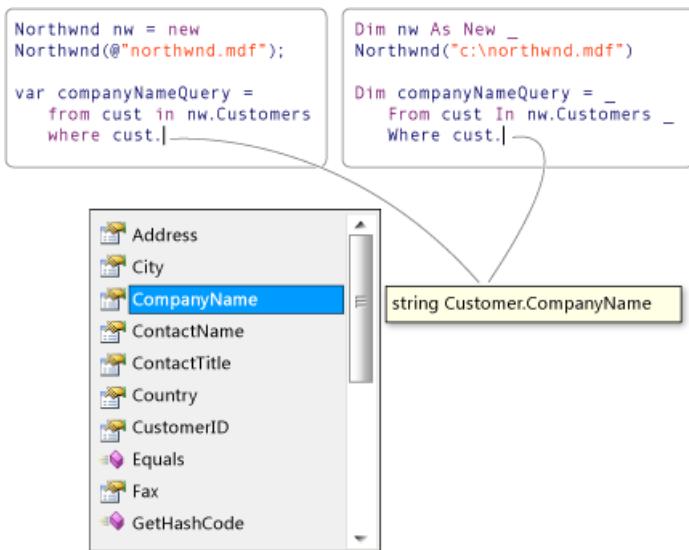
```
class LINQQueryExpressions
{
    static void Main()
    {

        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery =
            from score in scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in scoreQuery)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 97 92 81
```

다음 그림에서는 Visual Studio에서 전체 형식 검사 및 IntelliSense를 지원하는 C#과 Visual Basic을 사용하여 SQL Server 데이터베이스에 대해 부분적으로 완성된 LINQ 쿼리를 보여줍니다.



## 쿼리 식 개요

- 쿼리 식은 LINQ 사용 데이터 소스에서 데이터를 쿼리하고 변환하는 데 사용할 수 있습니다. 예를 들어 단일 쿼리로 SQL 데이터베이스에서 데이터를 검색하고 XML 스트림을 출력으로 생성할 수 있습니다.
- 쿼리 식은 익숙한 C# 언어 구문을 많이 사용하기 때문에 쉽게 마스터할 수 있습니다.
- 대부분의 경우 컴파일러가 형식을 유추할 수 있기 때문에 명시적으로 형식을 제공할 필요는 없지만 쿼리 식의 변수는 모두 강력한 형식을 갖습니다. 자세한 내용은 [LINQ 쿼리 작업의 형식 관계](#)를 참조하세요.
- 쿼리는 쿼리 변수를 반복할 때까지(예: `foreach` 문) 실행되지 않습니다. 자세한 내용은 [LINQ 쿼리 소개](#)를 참조하세요.
- 컴파일 타임에 쿼리 식은 C# 사양에 명시된 규칙에 따라 표준 쿼리 연산자 메서드 호출으로 변환됩니다. 쿼리 구문을 사용하여 표현할 수 있는 모든 쿼리는 메서드 구문으로도 표현할 수 있습니다. 그러나 대부분의 경우 쿼리 구문이 더 읽기 쉽고 간결합니다. 자세한 내용은 [C# 언어 사양](#) 및 [표준 쿼리 연산자 개요](#)를 참조하세요.
- 일반적으로 LINQ 쿼리를 작성하는 경우 가능하면 쿼리 구문을 사용하고 필요한 경우 메서드 구문을 사용하는 것이 좋습니다. 두 개의 다른 폼 간에 의미 체계 또는 성능상의 차이는 없습니다. 쿼리 식이 메서드 구문으로 작성된 동급의 식보다 읽기 쉬운 경우가 많습니다.
- `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. 자세한 내용은 [쿼리 구문과 메서드 구문 비교](#)를 참조하세요.
- 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.

## 다음 단계

LINQ에 대한 자세한 내용을 알아보려면 [쿼리 식 기본 사항](#)에서 몇 가지 기본 개념을 익힌 후 관심 있는 LINQ 기술에 대한 설명서를 읽어보세요.

- XML 문서: [LINQ to XML](#)
- ADO.NET Entity Framework: [LINQ to Entities](#)
- .NET 컬렉션, 파일, 문자열 등: [LINQ to Objects](#)

LINQ를 보다 깊이 있게 이해하려면 [C#의 LINQ](#)를 참조하세요.

C#에서 LINQ를 사용하려면 자습서 [LINQ 작업](#)을 참조하세요.

# LINQ 쿼리 소개(C#)

2020-11-02 • 16 minutes to read • [Edit Online](#)

쿼리는 데이터 소스에서 데이터를 검색하는 식입니다. 쿼리는 일반적으로 특수화된 쿼리 언어로 표현됩니다. 관계형 데이터베이스에는 SQL이 사용되고 XML에는 XQuery가 사용되는 것처럼 시간에 따라 다양한 형식의 데이터 소스에 대해 서로 다른 언어가 개발되었습니다. 따라서 개발자는 지원해야 하는 데이터 소스의 형식이나 데이터 형식에 따라 새로운 쿼리 언어를 배워야 했습니다. LINQ는 다양한 데이터 소스 및 형식에 사용할 수 있는 일관된 모델을 제공함으로써 이러한 상황을 단순화합니다. LINQ 쿼리에서는 항상 객체를 사용합니다. XML 문서, SQL 데이터베이스, ADO.NET 데이터 세트, .NET 컬렉션 및 LINQ 공급자를 사용할 수 있는 다른 모든 형식에서 데이터를 쿼리하고 변환하는 데 동일한 기본 코딩 패턴을 사용합니다.

## 쿼리 작업의 세 부분

모든 LINQ 쿼리 작업은 다음과 같은 세 가지 고유한 작업으로 구성됩니다.

1. 데이터 소스 가져오기.
2. 쿼리 만들기.
3. 쿼리 실행.

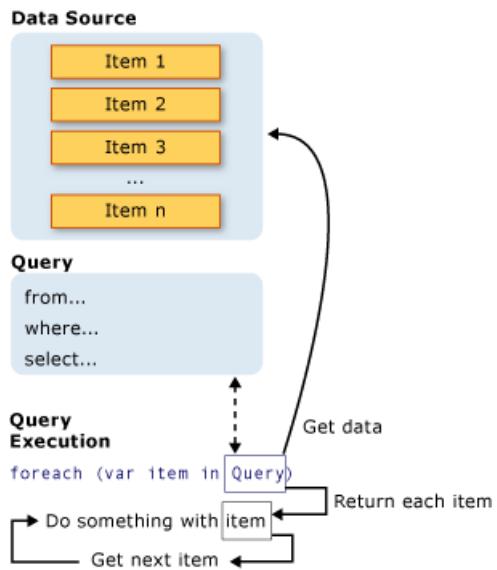
다음 예제에서는 쿼리 작업의 세 부분이 소스 코드로 표현되는 방식을 보여 줍니다. 예제에서는 편의상 정수 배열을 데이터 소스로 사용하지만 다른 데이터 소스에도 동일한 개념이 적용됩니다. 이 예제는 이 항목의 나머지 부분 전체에서 참조됩니다.

```
class IntroToLINQ
{
    static void Main()
    {
        // The Three Parts of a LINQ Query:
        // 1. Data source.
        int[] numbers = new int[7] { 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.WriteLine("{0,1} ", num);
        }
    }
}
```

다음 그림에서는 전체 쿼리 작업을 보여 줍니다. LINQ에서 쿼리 실행은 쿼리 자체와 다릅니다. 다시 말해 쿼리 변수를 만드는 것만으로는 어떤 데이터도 검색되지 않습니다.



## 데이터 소스

이전 예제에서는 데이터 소스가 배열이기 때문에 제네릭 `IEnumerable<T>` 인터페이스를 암시적으로 지원합니다. 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. 소스 데이터가 쿼리 가능 형식으로 메모리에 아직 없는 경우 LINQ 공급자도 그렇게 나타내야 합니다. 예를 들어 LINQ to XML은 XML 문서를 쿼리 가능 `XElement` 형식으로 로드합니다.

```
// Create a data source from an XML document.  
// using System.Xml.Linq;  
XElement contacts = XElement.Load(@"c:\myContactList.xml");
```

먼저 LINQ to SQL을 사용하여 디자인 타임에 수동으로 또는 [Visual Studio의 LINQ to SQL 도구](#)를 사용하여 개체 관계형 매핑을 만듭니다. 개체에 대해 쿼리를 작성하면 런타임에 LINQ to SQL에서 데이터베이스와의 통신을 처리합니다. 다음 예에서 `Customers`는 데이터베이스의 특정 테이블을 나타내며, `IQueryable<T>` 쿼리 결과 형식은 `IEnumerable<T>`에서 파생됩니다.

```
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 아주 간단합니다. LINQ 데이터 소스는 제네릭 `IEnumerable<T>` 인터페이스 또는 이 인터페이스에서 상속된 인터페이스를 지원하는 모든 개체입니다.

## NOTE

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. 필요한 경우 쿼리는 정보를 반환하기 전에 해당 정보를 정렬, 그룹화 및 구체화하는 방법도 지정합니다. 쿼리는 쿼리 변수에 저장되고 쿼리 식으로 초기화됩니다. 쿼리를 쉽게 작성할 수 있도록 C#에서는 새로운 쿼리 구문이 도입되었습니다.

이전 예제의 쿼리는 정수 배열에서 모든 짝수를 반환합니다. 쿼리 식에는 `from`, `where` 및 `select`의 세 가지 절이 포함됩니다. 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. 이러한 쿼리 절 및 다른 쿼리 절은 [LINQ\(Language Integrated Query\)](#) 섹션에서 자세히 설명합니다. 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. 백그라운드에서 쿼리를 생성하는 방법에 대한 자세한 내용은 [표준 쿼리 연산자 개요\(C#\)](#)를 참조하세요.

## NOTE

쿼리는 메서드 구문을 사용하여 표현할 수도 있습니다. 자세한 내용은 [LINQ의 쿼리 구문 및 메서드 구문](#)을 참조하세요.

## 쿼리 실행

### 지연된 실행

앞에서 설명한 대로 쿼리 변수 자체는 쿼리 명령을 저장할 뿐입니다. 실제 쿼리 실행은 `foreach` 문에서 쿼리 변수가 반복될 때까지 지연됩니다. 이 개념을 [지연된 실행](#)이라고 하며 다음 예제에서 보여 줍니다.

```
// Query execution.  
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 저장하지 않으므로 원하는 만큼 자주 실행할 수 있습니다. 예를 들어 별도의 애플리케이션에서 지속적으로 업데이트되는 데이터베이스가 있을 수 있습니다. 이 애플리케이션에서 최근 데이터를 검색하는 쿼리를 작성하고 이를 일정 간격을 두고 반복적으로 실행하여 매번 다른 결과를 검색할 수 있습니다.

### 즉시 실행 강제 적용

소스 요소 범위에 대해 집계 함수를 수행하는 쿼리는 먼저 해당 요소를 반복해야 합니다. 이러한 쿼리의 예로 `Count`, `Max`, `Average` 및 `First`가 있습니다. 이러한 쿼리는 쿼리 자체에서 결과를 반환하려면 `foreach`를 사용해야 하기 때문에 명시적 `foreach` 문 없이 실행됩니다. 또한 이러한 유형의 쿼리는 `IEnumerable` 컬렉션이 아니라 단일 값을 반환합니다. 다음 쿼리는 소스 배열에서 짝수의 개수를 반환합니다.

```
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.

```
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.

## 참조

- [C#에서 LINQ 시작](#)
- [연습: C#에서 쿼리 작성](#)
- [LINQ\(Language-Integrated Query\)](#)
- [foreach, in](#)
- [쿼리 키워드\(LINQ\)](#)

# LINQ 및 제네릭 형식(C#)

2020-11-02 • 5 minutes to read • [Edit Online](#)

LINQ 쿼리는 .NET Framework 버전 2.0에서 도입된 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.

```
IEnumerator<Customer> customerQuery =
    from cust in customers
    where cust.City == "London"
    select cust;

foreach (Customer customer in customerQuery)
{
    Console.WriteLine(customer.LastName + ", " + customer.FirstName);
}
```

자세한 내용은 [LINQ 쿼리 작업의 형식 관계](#)를 참조하세요.

## 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.

```
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 명시

적으로 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.

## 참조

- [제네릭](#)

# 기본 LINQ 쿼리 작업(C#)

2020-11-02 • 12 minutes to read • [Edit Online](#)

이 항목에서는 LINQ 쿼리 식 및 쿼리에서 수행하는 일부 일반적인 작업을 간단히 소개합니다. 자세한 내용은 다음 항목을 참조하세요.

## LINQ 쿼리 식

### 표준 쿼리 연산자 개요(C#)

### 연습: C#에서 쿼리 작성

#### NOTE

SQL 또는 XQuery와 같은 쿼리 언어를 이미 잘 알고 있으면 이 항목의 대부분을 건너뛸 수 있습니다. LINQ 쿼리 식에서 절의 순서를 알아보려면 "from" 절을 확인하세요.

## 데이터 소스 가져오기

LINQ 쿼리에서 첫 번째 단계는 데이터 소스를 지정하는 것입니다. 대부분의 프로그래밍 언어에서처럼 C#에서 변수는 선언된 후 사용되어야 합니다. LINQ 쿼리에서는 데이터 소스(`customers`) 및 범위 변수(`cust`)를 소개하기 위해 `from` 절이 먼저 나옵니다.

```
//queryAllCustomers is an IEnumerable<Customer>
var queryAllCustomers = from cust in customers
                        select cust;
```

쿼리 식에서는 실제 반복이 발생하지 않는다는 점을 제외하고 범위 변수는 `foreach` 루프의 반복 변수와 비슷합니다. 쿼리가 실행될 때 범위 변수는 `customers`에서 각 연속 요소에 대한 참조로 사용됩니다. 컴파일러에서 `cust` 형식을 유추할 수 있으므로 명시적으로 지정할 필요가 없습니다. 추가 범위 변수는 `let` 절에 의해 소개될 수 있습니다. 자세한 내용은 `let` 절을 참조하세요.

#### NOTE

`ArrayList`과 같은 제네릭이 아닌 데이터 소스의 경우 범위 변수를 명시적으로 형식화해야 합니다. 자세한 내용은 [LINQ를 사용하여 ArrayList를 쿼리하는 방법\(C#\)](#) 및 `from` 절을 참조하세요.

## 필터링

대부분의 일반적인 쿼리 작업에서는 부울 식 형태로 필터를 적용합니다. 필터를 사용하면 쿼리에서는 식이 `true`인 요소만 반환합니다. 결과는 `where` 절에 따라 반환됩니다. 적용되는 필터는 소스 시퀀스에서 제외할 요소를 지정합니다. 다음 예제에서는 London에 주소가 있는 `customers`만 반환됩니다.

```
var queryLondonCustomers = from cust in customers
                            where cust.City == "London"
                            select cust;
```

친숙한 C# 논리 `AND` 및 `OR` 연산자를 사용하여 `where` 절에서 필요한 만큼 필터 식을 적용할 수 있습니다. 예를 들어 "London"에 있고(`AND`) 이름이 "Devon"인 고객만 반환하려면 다음 코드를 작성합니다.

```
where cust.City == "London" && cust.Name == "Devon"
```

London 또는 Paris에 있는 고객을 반환하려면 다음 코드를 작성합니다.

```
where cust.City == "London" || cust.City == "Paris"
```

자세한 내용은 [where 절](#)을 참조하세요.

## 순서 지정

보통 반환된 데이터를 정렬하는 것이 편리합니다. `orderby` 절을 사용하면 반환된 시퀀스의 요소가 정렬되는 형식의 기본 비교자에 따라 정렬됩니다. 예를 들어 `Name` 속성에 따라 결과를 정렬하도록 다음 쿼리를 확장할 수 있습니다. `Name`이 문자열이면 기본 비교자는 A에서 Z까지 사전순 정렬을 수행합니다.

```
var queryLondonCustomers3 =
    from cust in customers
    where cust.City == "London"
    orderby cust.Name ascending
    select cust;
```

결과를 역순으로 정렬하려면 `orderby...descending` 절을 사용합니다.

자세한 내용은 [orderby 절](#)을 참조하세요.

## 그룹화

`group` 절을 사용하면 지정한 키를 기준으로 결과를 그룹화할 수 있습니다. 예를 들어 결과가 `city` 별로 그룹화되어 London 또는 Paris의 모든 고객이 개별 그룹에 포함되도록 지정할 수 있습니다. 이 경우 `cust.city`가 키입니다.

```
// queryCustomersByCity is an IEnumerable<IGrouping<string, Customer>>
var queryCustomersByCity =
    from cust in customers
    group cust by cust.City;

// customerGroup is an IGrouping<string, Customer>
foreach (var customerGroup in queryCustomersByCity)
{
    Console.WriteLine(customerGroup.Key);
    foreach (Customer customer in customerGroup)
    {
        Console.WriteLine("    {0}", customer.Name);
    }
}
```

쿼리를 `group` 절로 종료하면 결과에 여러 목록으로 구성된 목록 형식이 사용됩니다. 목록의 각 요소는 `Key` 멤버 및 해당 키로 그룹화된 요소 목록이 포함된 개체입니다. 그룹 시퀀스를 생성하는 쿼리를 반복할 경우 중첩된 `foreach` 루프를 사용해야 합니다. 외부 루프는 각 그룹을 반복하고 내부 루프는 각 그룹의 멤버를 반복합니다.

그룹 작업의 결과를 참조해야 할 경우 `into` 키워드를 사용하여 추가로 쿼리될 수 있는 식별자를 만들 수 있습니다. 다음 쿼리는 세 명 이상의 고객이 포함된 그룹만 반환합니다.

```
// custQuery is an IEnumerable<IGrouping<string, Customer>>
var custQuery =
    from cust in customers
    group cust by cust.City into custGroup
    where custGroup.Count() > 2
    orderby custGroup.Key
    select custGroup;
```

자세한 내용은 [group 절](#)을 참조하세요.

## 조인

조인 작업은 데이터 소스에서 명시적으로 모델링되지 않은 시퀀스 간 연결을 만듭니다. 예를 들어 같은 위치를 가진 모든 고객 및 배포자를 찾는 조인을 수행할 수 있습니다. LINQ에서 `join` 절은 항상 직접 데이터베이스 테이블이 아닌 개체 컬렉션에 대해 작동합니다.

```
var innerJoinQuery =
    from cust in customers
    join dist in distributors on cust.City equals dist.City
    select new { CustomerName = cust.Name, DistributorName = dist.Name };
```

LINQ의 외래 키는 개체 모델에서 항목 컬렉션을 포함하는 속성으로 표현되므로 LINQ에서는 SQL에서처럼 자주 `join`을 사용할 필요가 없습니다. 예를 들어 `Customer` 개체에는 `Order` 개체의 컬렉션이 포함됩니다. 조인을 수행하지 않고 점 표기법을 사용하여 주문에 액세스합니다.

```
from order in Customer.Orders...
```

자세한 내용은 [join 절](#)을 참조하세요.

## 선택(프로젝션)

`select` 절은 쿼리 결과를 생성하고 각 반환된 요소의 "모양" 또는 형식을 지정합니다. 예를 들어 결과가 계산 또는 새 개체 만들기에 따라 전체 `Customer` 개체, 하나의 멤버만, 멤버 하위 집합 또는 일부 완전히 다른 결과 형식으로 구성될지 지정할 수 있습니다. `select` 절이 소스 요소의 복사본이 아닌 다른 항목을 생성하는 경우 이 작업을 **프로젝션**이라고 합니다. 프로젝션을 사용하여 데이터를 변환하는 것은 LINQ 쿼리 식의 강력한 기능입니다. 자세한 내용은 [LINQ를 통한 데이터 변환\(C#\)](#) 및 [select 절](#)을 참조하세요.

## 참조

- [LINQ 쿼리 식](#)
- [연습: C#에서 쿼리 작성](#)
- [쿼리 키워드\(LINQ\)](#)
- [익명 형식](#)

# LINQ를 통한 데이터 변환(C#)

2020-11-02 • 12 minutes to read • [Edit Online](#)

LINQ(Language-Integrated Query)는 데이터 검색에만 관련된 것이 아닙니다. 데이터 변환을 위한 강력한 도구이기도 합니다. LINQ 쿼리를 사용하여 소스 시퀀스를 입력으로 사용하고 다양한 방법으로 수정하여 새 출력 시퀀스를 만들 수 있습니다. 정렬 및 그룹화를 통해 요소 자체를 수정하지 않고 시퀀스 자체를 수정할 수 있습니다. 하지만 LINQ 쿼리의 가장 강력한 기능은 새 형식을 만드는 기능일 것입니다. 이 작업은 `select` 절에서 수행합니다. 예를 들어, 아래와 같은 작업을 수행할 수 있습니다.

- 여러 입력 시퀀스를 새 형식을 가진 단일 출력 시퀀스로 병합합니다.
- 소스 시퀀스에 있는 각 요소의 속성 하나만으로 또는 여러 속성으로 구성된 출력 시퀀스를 만듭니다.
- 요소가 소스 데이터에서 수행된 작업의 결과로 구성된 출력 시퀀스를 만듭니다.
- 출력 시퀀스를 다른 형식으로 만듭니다. 예를 들어 데이터를 SQL 행 또는 텍스트 파일에서 XML로 변환 할 수 있습니다.

이것은 몇 가지 예일 뿐입니다. 물론 같은 쿼리에서 다양한 방법으로 이러한 변환을 결합할 수 있습니다. 또한 한 쿼리의 출력 시퀀스는 새 쿼리에 대한 입력 시퀀스로 사용할 수 있습니다.

## 여러 입력을 단일 출력 시퀀스로 결합

LINQ 쿼리를 사용하여 두 개 이상의 입력 시퀀스에서 생성된 요소가 포함된 출력 시퀀스를 만들 수 있습니다. 다음 예제에서는 두 개의 메모리 내 데이터 구조를 결합하는 방법을 보여 주지만 XML, SQL 또는 DataSet 소스에서 데이터를 결합하는 데는 같은 원칙을 적용할 수 있습니다. 다음 두 가지 클래스 형식을 가정합니다.

```
class Student
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public List<int> Scores;
}

class Teacher
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public string City { get; set; }
}
```

다음 예제에서는 쿼리를 보여 줍니다.

```

class DataTransformations
{
    static void Main()
    {
        // Create the first data source.
        List<Student> students = new List<Student>()
        {
            new Student { First="Svetlana",
                Last="Omelchenko",
                ID=111,
                Street="123 Main Street",
                City="Seattle",
                Scores= new List<int> { 97, 92, 81, 60 } },
            new Student { First="Claire",
                Last="O'Donnell",
                ID=112,
                Street="124 Main Street",
                City="Redmond",
                Scores= new List<int> { 75, 84, 91, 39 } },
            new Student { First="Sven",
                Last="Mortensen",
                ID=113,
                Street="125 Main Street",
                City="Lake City",
                Scores= new List<int> { 88, 94, 65, 91 } },
        };
    }

    // Create the second data source.
    List<Teacher> teachers = new List<Teacher>()
    {
        new Teacher { First="Ann", Last="Beebe", ID=945, City="Seattle" },
        new Teacher { First="Alex", Last="Robinson", ID=956, City="Redmond" },
        new Teacher { First="Michiyo", Last="Sato", ID=972, City="Tacoma" }
    };

    // Create the query.
    var peopleInSeattle = (from student in students
                           where student.City == "Seattle"
                           select student.Last)
                           .Concat(from teacher in teachers
                                   where teacher.City == "Seattle"
                                   select teacher.Last);

    Console.WriteLine("The following students and teachers live in Seattle:");
    // Execute the query.
    foreach (var person in peopleInSeattle)
    {
        Console.WriteLine(person);
    }

    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}

/*
 * Output:
 * The following students and teachers live in Seattle:
 * Omelchenko
 * Beebe
 */

```

자세한 내용은 [join 절 및 select 절](#)을 참조하세요.

## 각 소스 요소의 하위 집합 선택

소스 시퀀스에서 각 요소의 하위 집합을 선택하는 두 가지 기본 방법은 다음과 같습니다.

1. 소스 요소의 멤버를 하나만 선택하려면 점 작업을 사용합니다. 다음 예제에서는 `Customer` 개체에 `City` 문자열을 비롯한 여러 `public` 속성이 포함된다고 가정합니다. 실행될 경우 이 쿼리는 문자열의 출력 시퀀스를 생성합니다.

```
var query = from cust in Customers  
            select cust.City;
```

2. 소스 요소의 속성의 두 개 이상 포함된 요소를 만들려면 개체 이니셜라이저를 명명된 개체 또는 무명 형식과 함께 사용합니다. 다음 예제에서는 무명 형식을 사용하여 각 `Customer` 요소의 두 개 속성을 캡슐화하는 방법을 보여 줍니다.

```
var query = from cust in Customer  
            select new {Name = cust.Name, City = cust.City};
```

자세한 내용은 [개체 및 컬렉션 이니셜라이저](#) 및 [무명 형식](#)을 참조하세요.

## 메모리 내 개체를 XML로 변환

LINQ 쿼리를 통해 메모리 내 데이터 구조, SQL 데이터베이스, ADO.NET 데이터 세트 및 XML 스트림이나 문서 간에 손쉽게 데이터를 변환할 수 있습니다. 다음 예제에서는 메모리 내 데이터 구조의 개체를 XML 요소로 변환합니다.

```
class XMLTransform  
{  
    static void Main()  
    {  
        // Create the data source by using a collection initializer.  
        // The Student class was defined previously in this topic.  

```

이 코드에서는 다음 XML 출력을 생성합니다.

```
<Root>
  <student>
    <First>Svetlana</First>
    <Last>Omelchenko</Last>
    <Scores>97,92,81,60</Scores>
  </student>
  <student>
    <First>Claire</First>
    <Last>O'Donnell</Last>
    <Scores>75,84,91,39</Scores>
  </student>
  <student>
    <First>Sven</First>
    <Last>Mortensen</Last>
    <Scores>88,94,65,91</Scores>
  </student>
</Root>
```

자세한 내용은 [C#에서 XML 트리 만들기\(LINQ to XML\)](#)를 참조하세요.

## 소스 요소에서 작업 수행

출력 시퀀스에 소스 시퀀스의 요소 또는 요소 속성이 포함되어 있지 않을 수 있습니다. 대신에 출력이 소스 요소를 입력 인수로 사용하여 계산되는 값 시퀀스일 수 있습니다.

다음 쿼리는 원의 반지름을 나타내는 숫자의 시퀀스를 사용하여 각 반지름의 면적을 계산하고, 계산된 면적으로 형식이 지정된 문자열이 포함된 출력 시퀀스를 반환합니다.

출력 시퀀스의 각 문자열은 [문자열 보간](#)을 사용하여 형식이 지정됩니다. 보간된 문자열에는 문자열의 여는 따옴표 앞에 `$` 기호가 포함되며, 보간된 문자열 내에 배치된 중괄호 내에서 작업을 수행할 수 있습니다. 이러한 작업이 수행된 후에는 결과가 연결됩니다.

### NOTE

쿼리가 일부 다른 도메인으로 변환될 경우 쿼리 식에서 메서드를 호출할 수 없습니다. 예를 들어 SQL Server에는 관련 컨텍스트가 없으므로 LINQ to SQL에서 일반 C# 메서드를 호출할 수 없습니다. 그러나 저장 프로시저를 메서드에 매핑하고 저장 프로시저를 호출할 수 있습니다. 자세한 내용은 [저장 프로시저](#)를 참조하세요.

```

class FormatQuery
{
    static void Main()
    {
        // Data source.
        double[] radii = { 1, 2, 3 };

        // LINQ query using method syntax.
        IEnumerable<string> output =
            radii.Select(r => $"Area for a circle with a radius of '{r}' = {r * r * Math.PI:F2}");

        /*
        // LINQ query using query syntax.
        IEnumerable<string> output =
            from rad in radii
            select $"Area for a circle with a radius of '{rad}' = {rad * rad * Math.PI:F2}";
        */

        foreach (string s in output)
        {
            Console.WriteLine(s);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
   Area for a circle with a radius of '1' = 3.14
   Area for a circle with a radius of '2' = 12.57
   Area for a circle with a radius of '3' = 28.27
*/

```

## 참조

- [LINQ\(Language-Integrated Query\)\(C#\)](#)
- [LINQ to SQL](#)
- [LINQ to DataSet](#)
- [LINQ to XML\(C#\)](#)
- [LINQ 쿼리 식](#)
- [select 절](#)

# LINQ 쿼리 작업의 형식 관계(C#)

2020-11-02 • 7 minutes to read • [Edit Online](#)

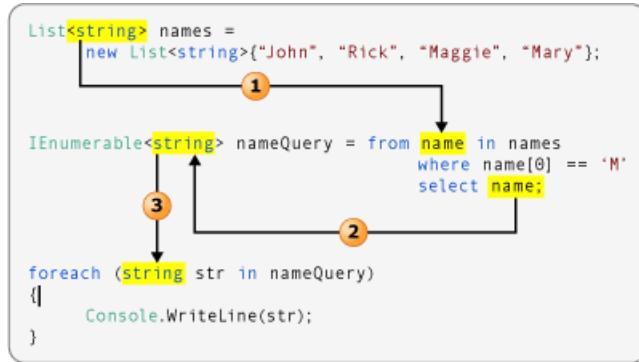
쿼리를 효과적으로 작성하려면 전체 쿼리 작업의 변수 형식이 모두 어떻게 서로 관련되는지를 이해해야 합니다. 이러한 관계를 이해하면 설명서의 LINQ 샘플 및 코드 예제를 더 쉽게 이해할 수 있습니다. 또한 `var`을 사용하여 변수를 암시적으로 형식화하는 경우 백그라운드에서 발생하는 상황을 이해할 수 있습니다.

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 `var`을 통해 암시적 형식화를 사용하는 경우에도 어떻게 동일한 원칙이 적용되는지를 보여 줍니다.

## 소스 데이터를 변환하지 않는 쿼리

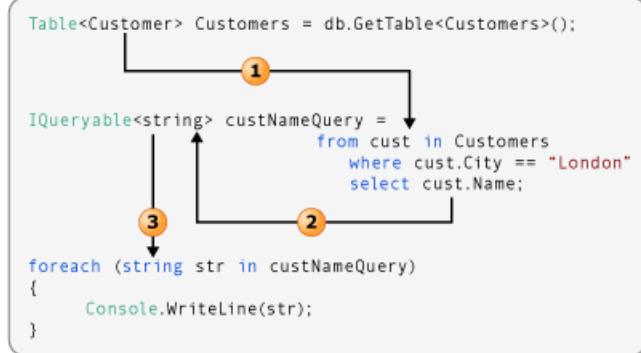
다음 그림에서는 데이터 변환을 수행하지 않는 LINQ to Objects 쿼리 작업을 보여 줍니다. 소스에는 문자열 시퀀스가 포함되어 있고, 쿼리 출력도 문자열 시퀀스입니다.



- 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 간단한 데이터 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.

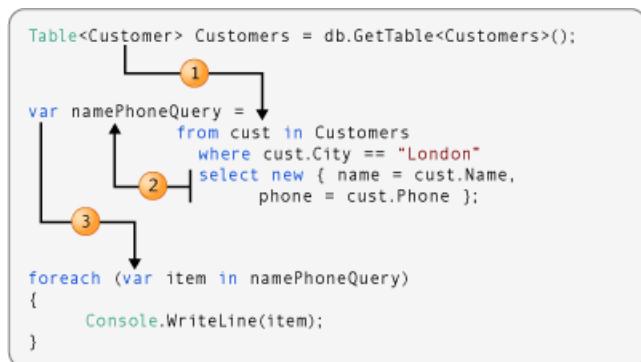


1. 데이터 소스의 형식 인수에 따라 범위 변수의 형식이 결정됩니다.

2. `select` 문은 전체 `Customer` 개체가 아니라 `Name` 속성을 반환합니다. `Name`이 문자열이므로 `custNameQuery`의 형식 인수는 `Customer`가 아니라 `string`입니다.

3. `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.



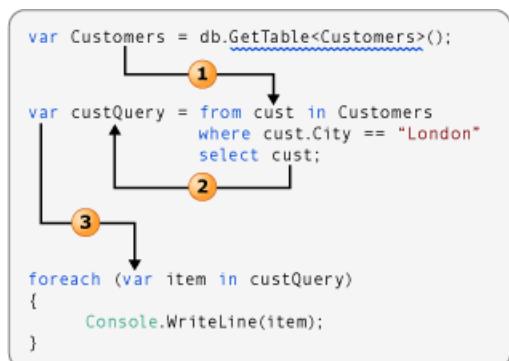
1. 데이터 소스의 형식 인수는 항상 쿼리의 범위 변수 형식입니다.

2. `select` 문이 무명 형식을 생성하기 때문에 `var`을 사용하여 쿼리 변수를 암시적으로 형식화해야 합니다.

3. 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.



`var`에 대한 자세한 내용은 [암시적 형식 지역 변수](#)를 참조하세요.

# LINQ의 쿼리 구문 및 메서드 구문(C#)

2020-11-02 • 12 minutes to read • [Edit Online](#)

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 의미상 동일하지만, 쿼리 구문이 더 간단하고 읽기 쉽다고 생각하는 사람이 많습니다. 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. 따라서 LINQ 쿼리를 작성하기 시작한 경우에도 쿼리 및 쿼리 식 자체에서 메서드 구문을 사용하는 방법을 잘 알고 있으면 유용합니다.

## 표준 쿼리 연산자 확장 메서드

다음 예제는 간단한 `쿼리 식` 및 `메서드 기반 쿼리`로서 작성된, 의미상 동등한 쿼리를 보여 줍니다.

```
class QueryVMethodSyntax
{
    static void Main()
    {
        int[] numbers = { 5, 10, 8, 3, 6, 12 };

        //Query syntax:
        IEnumerable<int> numQuery1 =
            from num in numbers
            where num % 2 == 0
            orderby num
            select num;

        //Method syntax:
        IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 == 0).OrderBy(n => n);

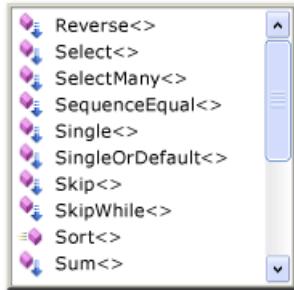
        foreach (int i in numQuery1)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine(System.Environment.NewLine);
        foreach (int i in numQuery2)
        {
            Console.Write(i + " ");
        }

        // Keep the console open in debug mode.
        Console.WriteLine(System.Environment.NewLine);
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/*
Output:
6 8 10 12
6 8 10 12
*/
```

두 예제에서 출력은 동일합니다. 쿼리 변수의 형식이 두 양식에서 모두 동일한 것을 알 수 있습니다 (`IEnumerable<T>`).

메서드 기반 쿼리를 이해할 수 있도록 더 자세히 살펴보겠습니다. 식의 오른쪽에서 `where` 절이 이제 `numbers` 개체의 인스턴스 메서드로 표현된 것을 알 수 있습니다. 이 개체를 다시 호출하면 `IEnumerable<int>` 형식을 갖게 됩니다. 제네릭 `IEnumerable<T>` 인터페이스에 대해 잘 알고 있다면 여기에 `Where` 메서드가 없다는 사실을 알 것입니다. 그러나 Visual Studio IDE에서 IntelliSense 완성 목록을 호출하면 `Where` 메서드뿐 아니라 `Select`, `SelectMany`, `Join`, `Orderby` 등의 다른 많은 메서드도 볼 수 있습니다. 이들은 모두 표준 쿼리 연산자입니다.

```
List<string> list = new List<string>();  
list. |
```



`IEnumerable<T>`이 이러한 추가 메서드를 포함하도록 다시 정의된 것처럼 보이지만 실제로는 그렇지 않습니다. 표준 쿼리 연산자는 확장 메서드라는 새로운 종류의 메서드로서 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.

LINQ를 사용하기 시작할 때 확장 메서드에 대해 알아야 할 것은, 정확한 `using` 지시문을 사용하여 이를 애플리케이션의 범위로 가져오는 방법입니다. 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. LINQ to SQL 및 LINQ to XML 같은 일부 LINQ 공급자는 `IEnumerable<T>` 이외의 다른 형식에 대해 자체 표준 쿼리 연산자 및 추가 확장 메서드를 구현합니다.

## 람다 식

앞의 예제에서 조건식(`num % 2 == 0`)은 `Where` 메서드(`Where(num => num % 2 == 0)`)에 인라인 인수로 전달됩니다. 이 인라인 식을 람다 식이라고 합니다. 이 방법을 사용하면 무명 메서드나 제네릭 대리자 또는 식 트리로서 좀 더 복잡한 형식으로 작성해야 하는 코드를 편리하게 작성할 수 있습니다. C#에서 `=>`는 "goes to"로 읽는 람다 연산자입니다. 연산자 왼쪽의 `num`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 구문의 식 또는 다른 C# 식이나 문의 식과 동일하며, 메서드 호출 및 기타 복잡한 논리를 포함할 수 있습니다. "반환 값"은 식 결과입니다.

LINQ 사용을 시작하기 위해 람다를 광범위하게 사용할 필요는 없습니다. 그러나 특정 쿼리는 메서드 구문으로만 표현할 수 있으며 그중 일부는 람다 식이 필요합니다. 람다에 익숙해지면 LINQ 도구 상자에서 람다가 강력하고 유연한 도구임을 알게 될 것입니다. 자세한 내용은 [람다 식](#)을 참조하세요.

## 쿼리 작성 가능성

이전 코드 예제에서는 `where` 호출 시 점 연산자를 사용하여 `OrderBy` 메서드를 호출합니다. `Where`가 필터링된 시퀀스를 생성하면 `Orderby`는 이를 정렬하여 해당 시퀀스에서 작동합니다. 쿼리는 `IEnumerable`을 반환하기 때문에, 사용자는 메서드 호출을 함께 연결하여 메서드 구문에서 쿼리를 작성합니다. 사용자가 쿼리 구문을 사용하여 쿼리를 작성할 때 백그라운드에서는 컴파일러가 이 작업을 수행합니다. 쿼리 변수는 쿼리 결과를 저장하지 않기 때문에 언제든지 수정할 수 있으며, 실행한 후에도 언제든지 새 쿼리의 기반으로 사용할 수 있습니다.

# LINQ를 지원하는 C# 기능

2020-11-02 • 10 minutes to read • [Edit Online](#)

다음 섹션에서는 C# 3.0에 도입된 새로운 언어 구문을 소개합니다. 이러한 새 기능은 모두 LINQ 쿼리에서 어느 정도 사용되지만 LINQ로 제한되지 않고 유용한 모든 컨텍스트에서 사용할 수 있습니다.

## 쿼리 식

쿼리 식은 SQL이나 XQuery와 유사한 선언적 구문을 사용하여 `IEnumerable` 컬렉션을 쿼리합니다. 컴파일 시간에 쿼리 구문은 LINQ 공급자의 표준 쿼리 연산자 확장 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 그룹화하여 그룹의 순서를 지정합니다.

```
var query = from str in stringArray
            group str by str[0] into stringGroup
            orderby stringGroup.Key
            select stringGroup;
```

자세한 내용은 [LINQ 쿼리 식](#)을 참조하세요.

## 암시적으로 형식화된 변수(var)

변수를 선언하고 초기화할 때 명시적으로 형식을 지정하는 대신 다음과 같이 `var` 한정자를 사용하여 형식을 유추하고 할당하도록 컴파일러에 지시할 수 있습니다.

```
var number = 5;
var name = "Virginia";
var query = from str in stringArray
            where str[0] == 'm'
            select str;
```

`var`로 선언된 변수는 형식을 명시적으로 지정한 변수만큼 강력한 형식입니다. `var`을 사용하면 무명 형식을 만들 수 있지만 지역 변수에만 사용할 수 있습니다. 배열은 암시적 형식 지정을 사용하여 선언할 수도 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.

```
var cust = new Customer { Name = "Mike", Phone = "555-1212" };
```

계속해서 `Customer` 클래스를 사용하여 `IncomingOrders`라는 데이터 소스가 있다고 가정하고 큰 `OrderSize`가 있는 각 주문의 경우 해당 주문을 기반으로 새 `Customer`를 만들려고 한다고 가정합니다. LINQ 쿼리는 이 데이터 소스에서 실행하고 개체 초기화를 사용하여 컬렉션을 채울 수 있습니다.

```
var newLargeOrderCustomers = from o in IncomingOrders
                             where o.OrderSize > 5
                             select new Customer { Name = o.Name, Phone = o.Phone };
```

데이터 소스에는 `Customer` 클래스보다 더 많은 속성(예: `OrderSize`)이 있을 수 있지만, 개체 초기화를 사용하면 쿼리에서 반환된 데이터가 원하는 데이터 형식으로 모델링되므로 클래스와 관련된 데이터를 선택합니다. 따라서 이제 `IEnumerable` 이 원하는 새 `Customer`로 채워집니다. 위의 내용은 LINQ 메서드 구문으로 작성할 수도 있습니다.

```
var newLargeOrderCustomers = IncomingOrders.Where(x => x.OrderSize > 5).Select(y => new Customer { Name = y.Name, Phone = y.Phone });
```

자세한 내용은 다음을 참조하세요.

- [개체 이니셜라이저 및 컬렉션 이니셜라이저](#)
- [표준 쿼리 연산자의 쿼리 식 구문](#)

## 익명 형식

무명 형식은 컴파일러에서 생성되며 형식 이름은 컴파일러에서만 사용할 수 있습니다. 무명 형식은 별도의 명명된 형식을 정의하지 않고 쿼리 결과에서 일시적으로 속성 집합을 그룹화하는 편리한 방법을 제공합니다. 무명 형식은 다음과 같이 새로운 식과 개체 이니셜라이저를 사용하여 초기화됩니다.

```
select new {name = cust.Name, phone = cust.Phone};
```

자세한 내용은 [무명 형식](#)을 참조하세요.

## 확장명 메서드

확장 메서드는 형식과 연결하여 형식에 대한 인스턴스 메서드인 것처럼 호출할 수 있는 정적 메서드입니다. 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.

자세한 내용은 [확장 메서드](#)를 참조하세요.

## 람다 식

람다 식은 => 연산자를 사용하여 함수 본문에서 입력 매개 변수를 구분하는 인라인 함수이며 컴파일 시간에 대리자나 식 트리로 변환할 수 있습니다. LINQ 프로그래밍에서는 표준 쿼리 연산자에 대한 메서드를 직접 호출할 때 람다 식이 나타납니다.

자세한 내용은 다음을 참조하세요.

- [익명 함수](#)
- [람다 식](#)
- [식 트리\(C#\)](#)

## 참조

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

# 연습: C#에서 쿼리 작성(LINQ)

2020-11-02 • 22 minutes to read • [Edit Online](#)

이 연습에서는 LINQ 쿼리 식을 작성하는 데 사용되는 C # 언어 기능을 보여 줍니다.

## C# 프로젝트 만들기

### NOTE

다음 지침은 Visual Studio용입니다. 다른 개발 환경을 사용하는 경우 System.Core.dll에 대한 참조와 [System.Linq](#) 네임스페이스에 대한 `using` 지시문을 사용하여 콘솔 프로젝트를 만듭니다.

**Visual Studio**에서 프로젝트를 만들려면

1. Visual Studio를 시작합니다.
2. 메뉴 모음에서 파일, 새로 만들기, 프로젝트를 차례로 선택합니다.  
새 프로젝트 대화 상자가 열립니다.
3. 설치됨, 템플릿, **Visual C#** 을 차례로 확장하고 콘솔 애플리케이션을 선택합니다.
4. 이름 텍스트 상자에 다른 이름을 입력하거나 기본 이름을 선택한 다음 확인 단추를 선택합니다.  
솔루션 탐색기에 새 프로젝트가 표시됩니다.
5. 프로젝트에 System.Core.dll에 대한 참조 및 [System.Linq](#) 네임스페이스에 대한 `using` 지시문이 있습니다.

## 메모리 내 데이터 소스 만들기

쿼리의 데이터 소스는 간단한 `Student` 개체 목록입니다. 각 `Student` 레코드에는 이름, 성 및 클래스의 테스트 점수를 나타내는 정수 배열이 있습니다. 프로젝트에 이 코드를 복사합니다. 다음 특성에 주의합니다.

- `Student` 클래스는 자동으로 구현된 속성으로 구성됩니다.
- 목록의 각 학생은 개체 이니셜라이저로 초기화됩니다.
- 목록 자체는 컬렉션 이니셜라이저로 초기화됩니다.

이 전체 데이터 구조는 생성자 또는 명시적 멤버 액세스에 대한 명시적 호출 없이 초기화되고 인스턴스화됩니다. 이러한 새로운 기능에 대한 자세한 내용은 [자동으로 구현된 속성 및 개체 및 컬렉션 이니셜라이저](#)를 참조하세요.

데이터 소스를 추가하려면

- `Student` 클래스 및 초기화된 학생 목록을 프로젝트의 `Program` 클래스에 추가합니다.

```

public class Student
{
    public string First { get; set; }
    public string Last { get; set; }
    public int ID { get; set; }
    public List<int> Scores;
}

// Create a data source by using a collection initializer.
static List<Student> students = new List<Student>
{
    new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 92, 81, 60}},
    new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},
    new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {88, 94, 65, 91}},
    new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {97, 89, 85, 82}},
    new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {35, 72, 91, 70}},
    new Student {First="Fadi", Last="Fakhouri", ID=116, Scores= new List<int> {99, 86, 90, 94}},
    new Student {First="Hanying", Last="Feng", ID=117, Scores= new List<int> {93, 92, 80, 87}},
    new Student {First="Hugo", Last="Garcia", ID=118, Scores= new List<int> {92, 90, 83, 78}},
    new Student {First="Lance", Last="Tucker", ID=119, Scores= new List<int> {68, 79, 88, 92}},
    new Student {First="Terry", Last="Adams", ID=120, Scores= new List<int> {99, 82, 81, 79}},
    new Student {First="Eugene", Last="Zabokritski", ID=121, Scores= new List<int> {96, 85, 91, 60}},
    new Student {First="Michael", Last="Tucker", ID=122, Scores= new List<int> {94, 92, 91, 91}}
};

```

학생 목록에 새 학생을 추가하려면

- 새 `Student` 를 `Students` 목록에 추가하고 원하는 이름 및 시험 점수를 사용합니다. 개체 이니셜라이저의 구문을 더 잘 알 수 있도록 새로운 학생 정보를 모두 입력해 보세요.

## 쿼리 만들기

단순 쿼리를 작성하려면

- 애플리케이션의 `Main` 메서드에서, 실행 시 첫 번째 테스트의 점수가 90보다 큰 모든 학생의 목록을 생성하는 간단한 쿼리를 만듭니다. 전체 `Student` 개체가 선택되므로 쿼리의 형식은 `IEnumerable<Student>`입니다. `var` 키워드를 사용하여 코드에서 암시적 형식을 사용할 수도 있지만, 결과를 명확하게 설명하기 위해 명시적 형식이 사용됩니다. `var`에 대한 자세한 내용은 [암시적으로 형식화된 지역 변수](#)를 참조하세요.

또한 쿼리의 범위 변수 `student`는 소스의 각 `Student`에 대한 참조로 사용되며, 각 개체에 대한 멤버 액세스를 제공합니다.

```

// 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;

```

## 쿼리 실행

쿼리를 실행하려면

- 이제 쿼리를 실행하도록 할 `foreach` 루프를 작성합니다. 다음은 코드에 대한 유의 사항입니다.

- 반환된 시퀀스의 각 요소는 `foreach` 루프의 반복 변수를 통해 액세스됩니다.
- 이 변수의 형식은 `Student`이며, 쿼리 변수 형식은 `IEnumerable<Student>`과 호환됩니다.

- 이 코드를 추가한 후 애플리케이션을 빌드하고 실행하고 콘솔 창에서 결과를 확인하세요.

```

// Execute the query.
// var could be used here also.
foreach (Student student in studentQuery)
{
    Console.WriteLine("{0}, {1}", student.Last, student.First);
}

// Output:
// Omelchenko, Svetlana
// Garcia, Cesar
// Fakhouri, Fadi
// Feng, Hanying
// Garcia, Hugo
// Adams, Terry
// Zabokritski, Eugene
// Tucker, Michael

```

다른 필터 조건을 추가하려면

- 쿼리를 구체화하기 위해 `where` 절에서 여러 부울 조건을 결합할 수 있습니다. 다음 코드는 쿼리를 실행하여 첫 번째 점수가 90을 초과하고 마지막 점수가 80 미만인 학생들을 반환하도록 하는 조건을 추가합니다. `where` 절은 다음 코드와 유사합니다.

```
where student.Scores[0] > 90 && student.Scores[3] < 80
```

자세한 내용은 [where 절](#)을 참조하세요.

## 쿼리 수정

결과를 정렬하려면

- 특정 순서로 되어 있는 경우 결과를 더 쉽게 검색할 수 있습니다. 반환된 시퀀스를 소스 요소에서 액세스 가능한 필드 기준으로 정렬할 수 있습니다. 예를 들어, 다음 `orderby` 절은 각 학생의 성에 따라 결과를 사전순으로 정렬합니다. `where` 문 바로 다음과 `select` 문 앞에서 다음 `orderby` 절을 쿼리에 추가합니다.

```
orderby student.Last ascending
```

- 이제 가장 높은 점수에서 가장 낮은 점수까지 첫 번째 테스트의 점수에 따라 역순으로 결과를 정렬하도록 `orderby` 절을 변경합니다.

```
orderby student.Scores[0] descending
```

- 점수를 볼 수 있도록 `WriteLine` 형식 문자열을 변경합니다.

```
Console.WriteLine("{0}, {1} {2}", student.Last, student.First, student.Scores[0]);
```

자세한 내용은 [orderby 절](#)을 참조하세요.

결과를 그룹화하려면

- 그룹화는 쿼리 식의 강력한 기능입니다. 그룹 절이 있는 쿼리는 그룹 시퀀스를 생성하며, 각 그룹 자체는 `Key` 및 해당 그룹의 모든 멤버로 구성된 시퀀스를 포함합니다. 다음과 같은 새 쿼리는 학생 성의 첫 글자를 키로 사용하여 학생들을 그룹화합니다.

```
// studentQuery2 is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0];
```

2. 이제 쿼리 형식이 변경되었습니다. 이제 쿼리를 실행하면 `char` 형식을 키로 가지고 있고 `Student` 개체의 시퀀스를 가지고 있는 그룹의 시퀀스가 생성됩니다. 쿼리 형식이 변경되었으므로 다음 코드는 `foreach` 실행 루프도 변경합니다.

```
// studentGroup is a IGrouping<char, Student>
foreach (var studentGroup in studentQuery2)
{
    Console.WriteLine(studentGroup.Key);
    foreach (Student student in studentGroup)
    {
        Console.WriteLine(" {0}, {1}",
                          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
```

3. 애플리케이션을 실행하고 콘솔 창에서 결과를 봅니다.

자세한 내용은 [group 절](#)을 참조하세요.

변수를 암시적으로 형식화하려면

1. `IGroupings` 의 `IEnumerables` 를 명시적으로 코딩하는 작업은 지루할 수 있습니다. `var` 을 사용하여 동일한 쿼리 및 `foreach` 루프를 훨씬 더 편리하게 작성할 수 있습니다. `var` 키워드는 개체 형식을 변경하지 않고, 형식을 추론하도록 컴파일러에 지시합니다. `studentQuery` 의 형식 및 `group` 반복 변수를 `var` 로 변경하고 쿼리를 다시 실행합니다. 내부 `foreach` 루프에서 반복 변수의 형식은 여전히 `Student` 로 지정되며 쿼리는 이전과 마찬가지로 작동합니다. `s` 반복 변수를 `var` 로 변경하고 쿼리를 다시 실행합니다. 정확히 동일한 결과가 표시됩니다.

```

var studentQuery3 =
    from student in students
    group student by student.Last[0];

foreach (var groupOfStudents in studentQuery3)
{
    Console.WriteLine(groupOfStudents.Key);
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine("  {0}, {1}",
            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

```

`var`에 대한 자세한 내용은 [암시적으로 형식화된 지역 변수](#)를 참조하세요.

키 값을 기준으로 그룹을 정렬하려면

1. 이전 쿼리를 실행하면 그룹이 사전순으로 표시되지 않습니다. 이를 변경하려면 `group` 절 뒤에 `orderby` 절을 제공해야 합니다. 그러나 `orderby` 절을 사용하려면 우선 `group` 절로 만든 그룹에 대한 참조 역할을 하는 식별자가 필요합니다. 다음과 같이 `into` 키워드를 사용하여 식별자를 제공합니다.

```

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(" {0}, {1}",
            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

```

이 쿼리를 실행하면 이제 그룹이 사전순으로 정렬되는 것을 알 수 있습니다.

**let**을 사용하여 식별자를 소개하려면

1. **let** 키워드를 사용하여 쿼리 식의 식 결과에 대한 식별자를 소개할 수 있습니다. 이 식별자는 다음 예제에서처럼 편리하게 사용할 수도 있고, 여러 번 계산할 필요가 없도록 표현식의 결과를 저장하여 성능을 향상시킬 수도 있습니다.

```

// studentQuery5 is an IEnumerable<string>
// 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

```

자세한 내용은 [let 절을 참조하세요](#).

쿼리 식에서 메서드 구문을 사용하려면

1. [LINQ의 쿼리 구문 및 메서드 구문](#)에 설명된 대로 일부 쿼리 작업은 메서드 구문을 사용해야만 표현할 수 있습니다. 다음 코드는 소스 시퀀스의 각 `student`에 대한 총 점수를 계산한 다음, 해당 쿼리의 결과에 대해 `Average()` 메서드를 호출하여 클래스의 평균 점수를 계산합니다.

```

var studentQuery6 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    select totalScore;

double averageScore = studentQuery6.Average();
Console.WriteLine("Class average score = {0}", averageScore);

// Output:
// Class average score = 334.16666666666667

```

`select` 절에서 변환 또는 프로젝션 하려면

1. 쿼리가 소스 시퀀스의 요소와 다른 요소를 갖는 시퀀스를 생성하는 것은 매우 일반적입니다. 이전 쿼리 및 실행 루프를 삭제하거나 주석으로 처리하고 다음 코드로 바꿉니다. 쿼리는 문자열 시퀀스(`Students` 아님)를 반환하며 이 사실은 `foreach` 루프에 반영됩니다.

```

IEnumerable<string> studentQuery7 =
    from student in students
    where student.Last == "Garcia"
    select student.First;

Console.WriteLine("The Garcias in the class are:");
foreach (string s in studentQuery7)
{
    Console.WriteLine(s);
}

// Output:
// The Garcias in the class are:
// Cesar
// Debra
// Hugo

```

2. 이 연습의 앞부분에 있는 코드는 평균 클래스 점수가 약 334임을 나타냅니다. 총점이 클래스 평균보다 큰 `Students`의 시퀀스를 `Student ID`와 함께 생성하려면 `select` 문에서 무명 형식을 사용할 수 있습니다.

```

var studentQuery8 =
    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 studentQuery8)
{
    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

```

## 다음 단계

C#에서 쿼리 작업의 기본 사항에 익숙해지면 관심이 있는 특정 LINQ 공급자 형식에 대한 설명서와 샘플을 읽을 준비가 된 것입니다.

[LINQ to SQL](#)

[LINQ to DataSet](#)

[LINQ to XML\(C#\)](#)

[LINQ to Objects\(C#\)](#)

## 참조

- [LINQ\(Language-Integrated Query\)\(C#\)](#)
- [LINQ 쿼리 식](#)

# 표준 쿼리 연산자 개요(C#)

2020-11-02 • 9 minutes to read • [Edit Online](#)

표준 쿼리 연산자는 LINQ 패턴을 형성하는 메서드입니다. 이 메서드 중 대부분은 시퀀스에서 작동합니다. 여기서 시퀀스란 `IEnumerable<T>` 인터페이스 또는 `IQueryable<T>` 인터페이스를 구현하는 형식의 개체를 의미합니다. 표준 쿼리 연산자는 필터링, 프로젝션, 집계, 정렬 등을 포함하여 다양한 쿼리 기능을 제공합니다.

`IEnumerable<T>` 형식의 개체와 작동하는 연산자와 `IQueryable<T>` 형식의 개체와 작동하는 연산자로 두 가지 LINQ 표준 쿼리 연산자 집합이 있습니다. 각 집합을 구성하는 메서드는 각각 `Enumerable` 및 `Queryable` 클래스의 정적 멤버입니다. 작동하는 형식의 확장 메서드로 정의됩니다. 확장 메서드는 정적 메서드 구문 또는 인스턴스 메서드 구문을 사용하여 호출할 수 있습니다.

또한 여러 표준 쿼리 연산자 메서드는 `IEnumerable<T>` 또는 `IQueryable<T>`을 기반으로 하는 형식이 아닌 다른 형식에서 작동합니다. `Enumerable` 형식은 `IEnumerable` 형식의 개체에서 작동하는 이러한 두 메서드를 정의합니다. 두 메서드 `Cast<TResult>(IEnumerable)` 및 `OfType<TResult>(IEnumerable)`을 사용하면 LINQ 패턴에서 매개 변수가 없는 컬렉션이나 제네릭이 아닌 컬렉션을 쿼리할 수 있습니다. 이 작업을 위해 강력한 형식의 개체 컬렉션을 만듭니다. `Queryable` 클래스는 `Queryable` 형식의 개체에서 작동하는 두 개의 유사한 메서드 `Cast<TResult>(IQueryable)` 및 `OfType<TResult>(IQueryable)`을 정의합니다.

표준 쿼리 연산자는 단일 값을 반환하는지 또는 시퀀스를 반환하는지에 따라 실행되는 타이밍이 다릅니다. singleton 값(예: `Average` 및 `Sum`)을 반환하는 이러한 메서드는 즉시 실행됩니다. 시퀀스를 반환하는 메서드는 쿼리 실행을 지연하고 열거 가능한 개체를 반환합니다.

메모리 내 컬렉션에 대해 작동하는 메서드, 즉 `IEnumerable<T>`을 확장하는 메서드의 경우 반환된 열거 가능한 개체는 메서드에 전달된 인수를 캡처합니다. 해당 개체를 열거하는 경우 쿼리 연산자의 논리가 사용되며 쿼리 결과가 반환됩니다.

반면 `IQueryable<T>`을 확장하는 메서드는 쿼리 동작을 구현하지 않고, 수행 할 쿼리를 나타내는 식 트리를 빌드합니다. 쿼리 처리는 소스 `IQueryable<T>` 개체에 의해 처리됩니다.

쿼리 메서드 호출을 연결하여 임의로 복잡해질 수 있는 한 쿼리로 연결될 수 있습니다.

다음 코드 예제에서는 표준 쿼리 연산자를 사용하여 시퀀스에 대한 정보를 가져올 수 있는 방법을 보여 줍니다.

```

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

```

## 쿼리 식 구문

자주 사용되는 표준 쿼리 연산자 중 일부에는 '쿼리 식'의 일부로 호출할 수 있는 전용 C# 및 Visual Basic 언어 키워드 구문이 있습니다. 전용 키워드와 해당 구문이 있는 표준 쿼리 연산자에 대한 자세한 내용은 [표준 쿼리 연산자에 대한 쿼리 식 구문\(C#\)](#)을 참조하세요.

## 표준 쿼리 연산자 확장

대상 도메인 또는 기술에 적합한 도메인 특정 메서드를 만들어 표준 쿼리 연산자 집합을 강화할 수 있습니다. 또한 원격 평가, 쿼리 변환, 최적화 등의 추가 서비스를 제공하는 고유한 구현으로 표준 쿼리 연산자를 바꿀 수도 있습니다. 예제는 [AsEnumerable](#)을 참조하세요.

## 관련 단원

다음 링크를 누르면 기능에 따라 다양한 표준 쿼리 연산자에 대한 추가 정보를 제공하는 문서로 이동합니다.

[데이터 정렬\(C#\)](#)

[집합 작업\(C#\)](#)

[데이터 필터링\(C#\)](#)

[수량자 작업\(C#\)](#)

[프로젝션 작업\(C#\)](#)

[데이터 분할\(C#\)](#)

[조인 작업\(C#\)](#)

[데이터 그룹화\(C#\)](#)

[생성 작업\(C#\)](#)

[같은 연산\(C#\)](#)

[요소 작업\(C#\)](#)

[데이터 형식 변환\(C#\)](#)

[연결 작업\(C#\)](#)

[집계 작업\(C#\)](#)

## 참조

- [Enumerable](#)
- [Queryable](#)
- [LINQ 쿼리 소개\(C#\)](#)
- [표준 쿼리 연산자의 쿼리 식 구문\(C#\)](#)
- [실행 방식에 따라 표준 쿼리 연산자 분류\(C#\)](#)
- [확장명 메서드](#)

# 표준 쿼리 연산자의 쿼리 식 구문(C#)

2020-11-02 • 3 minutes to read • [Edit Online](#)

자주 사용되는 표준 쿼리 연산자 중 일부에는 쿼리 식의 일부로 호출할 수 있는 전용 C# 언어 키워드 구문이 있습니다. 쿼리 식은 [메서드 기반](#) 양식과는 다른, 가독성이 더 우수한 쿼리 표현 양식입니다. 쿼리 식 절은 컴파일 시간에 쿼리 메서드 호출로 변환됩니다.

## 쿼리 식 구문 표

다음 표에는 동등한 쿼리 식 절이 있는 표준 쿼리 연산자가 나열되어 있습니다.

메서드	C# 쿼리 식 구문
Cast	예를 들어 명시적으로 형식화된 범위 변수를 사용합니다.  <code>from int i in numbers</code>  자세한 내용은 <a href="#">from</a> 절을 참조하세요.
GroupBy	<code>group ... by</code>  또는  <code>group ... by ... into ...</code>  자세한 내용은 <a href="#">group</a> 절을 참조하세요.
GroupJoin<TOuter,TInner,TKey,TResult> (IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>,TResult>)	<code>join ... in ... on ... equals ... into ...</code>  자세한 내용은 <a href="#">join</a> 절을 참조하세요.
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)	<code>join ... in ... on ... equals ...</code>  자세한 내용은 <a href="#">join</a> 절을 참조하세요.
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	<code>orderby</code>  자세한 내용은 <a href="#">orderby</a> 절을 참조하세요.
OrderByDescending<TSource,TKey> (IEnumerable<TSource>, Func<TSource,TKey>)	<code>orderby ... descending</code>  자세한 내용은 <a href="#">orderby</a> 절을 참조하세요.
Select	<code>select</code>  자세한 내용은 <a href="#">select</a> 절을 참조하세요.
SelectMany	여러 <code>from</code> 절.  자세한 내용은 <a href="#">from</a> 절을 참조하세요.

메서드	C# 쿼리 식 구문
ThenBy<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	<code>orderby ... , ...</code> 자세한 내용은 <a href="#">orderby 절</a> 을 참조하세요.
ThenByDescending<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	<code>orderby ... , ... descending</code> 자세한 내용은 <a href="#">orderby 절</a> 을 참조하세요.
Where	<code>where</code> 자세한 내용은 <a href="#">where 절</a> 을 참조하세요.

## 참조

- [Enumerable](#)
- [Queryable](#)
- 표준 쿼리 연산자 개요(C#)
- 실행 방식에 따라 표준 쿼리 연산자 분류(C#)

# 실행 방식에 따라 표준 쿼리 연산자 분류(C#)

2020-11-02 • 8 minutes to read • [Edit Online](#)

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. 여러 쿼리 연산자가 어떻게 실행되는지 알고 있으면 제공된 쿼리에서 얻을 결과를 이해하는데 도움이 될 수 있습니다. 데이터 소스가 변경되거나 다른 쿼리 위에 쿼리를 빌드할 경우 특히 도움이 됩니다. 이 항목에서는 실행 방식에 따라 표준 쿼리 연산자를 분류합니다.

## 실행 방식

### 직접 실행

즉시 실행은 코드의 쿼리가 선언되는 지점에서 데이터 소스를 읽고 작업이 수행됨을 의미합니다. 하나의 비열거형 결과를 반환하는 모든 표준 쿼리 연산자는 즉시 실행됩니다.

### 연기됨

지연된 실행은 코드의 쿼리가 선언되는 지점에서 작업이 수행되지 않음을 의미합니다. 예를 들어 `foreach` 문을 사용하여 쿼리 변수가 열거될 경우에만 작업이 수행됩니다. 즉, 쿼리 실행 결과는 쿼리가 정의될 때가 아니라 쿼리가 실행될 때 데이터 소스의 내용에 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.

지연된 실행을 사용하는 쿼리 연산자는 스트리밍 및 비스트리밍으로 추가로 분류할 수 있습니다.

### 스트리밍

스트리밍 연산자는 요소를 생성하기 전에 모든 소스 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 소스 데이터를 읽어야 합니다. 정렬 또는 그룹화 등의 작업은 이 범주로 분류됩니다. 실행 시 비스트리밍 쿼리 연산자는 모든 소스 데이터를 읽고, 데이터 구조에 넣고, 작업을 수행하고, 결과 요소를 생성합니다.

## 분류 표

다음 표에서는 실행 방법에 따라 각 표준 쿼리 연산자 메서드를 분류합니다.

### NOTE

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.

표준 쿼리 연산자	반환 형식	즉시 실행	지연된 스트리밍 실행	지연된 비스트리밍 실행
Aggregate	TSource	x		
All	Boolean	x		

표준 쿼리 연산자	반환 형식	즉시 실행	지연된 스트리밍 실행	지연된 비스트리밍 실행
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		
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		

표준 쿼리 연산자	반환 형식	즉시 실행	지연된 스트리밍 실행	지연된 비스트리밍 실행
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	
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	

표준 쿼리 연산자	반환 형식	즉시 실행	지연된 스트리밍 실행	지연된 비스트리밍 실행
Where	IEnumerable<T>		X	

## 참조

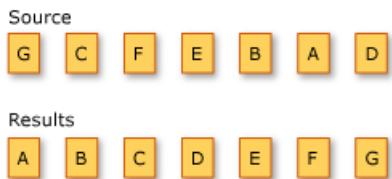
- [Enumerable](#)
- [표준 쿼리 연산자 개요\(C#\)](#)
- [표준 쿼리 연산자의 쿼리 식 구문\(C#\)](#)
- [LINQ to Objects\(C#\)](#)

# 데이터 정렬(C#)

2020-11-02 • 5 minutes to read • [Edit Online](#)

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.

다음 그림은 문자 시퀀스에 대한 사전순 정렬 작업의 결과를 보여 줍니다.



다음 섹션에는 데이터를 정렬하는 표준 쿼리 연산자 메서드가 나와 있습니다.

## 메서드

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
OrderBy	값을 오름차순으로 정렬합니다.	<code>orderby</code>	<a href="#">Enumerable.OrderBy</a> <a href="#">Queryable.OrderBy</a>
OrderByDescending	값을 내림차순으로 정렬합니다.	<code>orderby ... descending</code>	<a href="#">Enumerable.OrderByDescending</a> <a href="#">Queryable.OrderByDescending</a>
ThenBy	2차 정렬을 오름차순으로 수행합니다.	<code>orderby ..., ...</code>	<a href="#">Enumerable.ThenBy</a> <a href="#">Queryable.ThenBy</a>
ThenByDescending	2차 정렬을 내림차순으로 수행합니다.	<code>orderby ..., ... descending</code>	<a href="#">Enumerable.ThenByDescending</a> <a href="#">Queryable.ThenByDescending</a>
Reverse	컬렉션에서 요소의 순서를 반대로 바꿉니다.	해당 사용 없음.	<a href="#">Enumerable.Reverse</a> <a href="#">Queryable.Reverse</a>

## 쿼리 식 구문 예제

### 1차 정렬 예제

#### 1차 오름차순 정렬

다음 예제에서는 LINQ 쿼리에 `orderby` 절을 사용하여 배열의 문자열을 문자열 길이의 오름차순으로 정렬하는 방법을 보여 줍니다.

```

string[] words = { "the", "quick", "brown", "fox", "jumps" };

IQueryable<string> query = from word in words
                           orderby word.Length
                           select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

the
fox
quick
brown
jumps
*/

```

## 1차 내림차순 정렬

다음 예제에서는 LINQ 쿼리에 `orderby descending` 절을 사용하여 문자열을 첫 글자의 내림차순으로 정렬하는 방법을 보여 줍니다.

```

string[] words = { "the", "quick", "brown", "fox", "jumps" };

IQueryable<string> query = from word in words
                           orderby word.Substring(0, 1) descending
                           select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

the
quick
jumps
fox
brown
*/

```

## 2차 정렬 예제

### 2차 오름차순 정렬

다음 예제에서는 LINQ 쿼리에 `orderby` 절을 사용하여 배열의 문자열에 대해 1차 및 2차 정렬을 수행하는 방법을 보여 줍니다. 문자열은 길이의 오름차순으로 1차 정렬된 다음 문자열 첫 글자의 오름차순으로 2차 정렬됩니다.

```

string[] words = { "the", "quick", "brown", "fox", "jumps" };

IQueryable<string> query = from word in words
                           orderby word.Length, word.Substring(0, 1)
                           select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

fox
the
brown
jumps
quick
*/

```

## 2차 내림차순 정렬

다음 예제에서는 LINQ 쿼리에 `orderby descending` 절을 사용하여 1차 정렬을 오름차순으로 수행한 다음 2차 정렬을 내림차순으로 수행하는 방법을 보여 줍니다. 문자열은 길이순으로 1차 정렬된 다음 문자열 첫 글자순으로 2차 정렬됩니다.

```

string[] words = { "the", "quick", "brown", "fox", "jumps" };

IQueryable<string> query = from word in words
                           orderby word.Length, word.Substring(0, 1) descending
                           select word;

foreach (string str in query)
    Console.WriteLine(str);

/* This code produces the following output:

the
fox
quick
jumps
brown
*/

```

## 참조

- [System.Linq](#)
- [표준 쿼리 연산자 개요\(C#\)](#)
- [orderby 절](#)
- [Join 절 결과를 서순대로 정렬](#)
- [단어 또는 필드에 따라 텍스트 데이터를 정렬하거나 필터링하는 방법\(LINQ\)\(C#\)](#)

# 집합 작업(C#)

2020-11-02 • 4 minutes to read • [Edit Online](#)

LINQ의 집합 작업은 동일 컬렉션이나 별개 컬렉션(또는 집합)에 동등한 요소가 있는지 여부에 따라 결과 집합을 생성하는 쿼리 작업을 가리킵니다.

다음 섹션에는 집합 작업을 수행하는 표준 쿼리 연산자 메서드가 나와 있습니다.

## 메서드

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
Distinct	컬렉션에서 중복 값을 제거합니다.	해당 사항 없음.	<a href="#">Enumerable.Distinct</a> <a href="#">Queryable.Distinct</a>
Except	두 번째 컬렉션에 표시되지 않는 한 컬렉션의 요소를 의미하는 차집합을 반환합니다.	해당 사항 없음.	<a href="#">Enumerable.Except</a> <a href="#">Queryable.Except</a>
Intersect	두 컬렉션에 각각 표시되는 요소를 의미하는 교집합을 반환합니다.	해당 사항 없음.	<a href="#">Enumerable.Intersect</a> <a href="#">Queryable.Intersect</a>
Union	두 컬렉션 중 하나에 표시되는 고유한 요소를 의미하는 합집합을 반환합니다.	해당 사항 없음.	<a href="#">Enumerable.Union</a> <a href="#">Queryable.Union</a>

## 집합 작업 비교

### Distinct

다음 예제에서는 문자 시퀀스에 대한 [Enumerable.Distinct](#) 메서드의 동작을 보여줍니다. 반환된 시퀀스에는 입력 시퀀스의 고유한 요소가 포함됩니다.



```

string[] planets = { "Mercury", "Venus", "Venus", "Earth", "Mars", "Earth" };

IEnumerable<string> query = from planet in planets.Distinct()
                             select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Venus
 * Earth
 * Mars
 */

```

## Except

다음 예제에서는 [Enumerable.Except](#)의 동작을 보여줍니다. 반환된 시퀀스에는 두 번째 입력 시퀀스에 없는 첫 번째 입력 시퀀스의 요소만 포함됩니다.



```

string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IEnumerable<string> query = from planet in planets1.Except(planets2)
                             select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Venus
 */

```

## Intersect

다음 예제에서는 [Enumerable.Intersect](#)의 동작을 보여줍니다. 반환된 시퀀스에는 입력 시퀀스 둘 다에 공통적으로 있는 요소가 포함됩니다.



```

string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IQueryable<string> query = from planet in planets1.Intersect(planets2)
                            select planet;

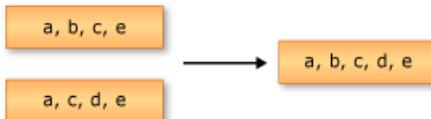
foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Earth
 * Jupiter
 */

```

## Union

다음 예제에서는 두 개의 문자 시퀀스에 대한 합집합을 보여줍니다. 반환된 시퀀스에는 두 입력 시퀀스의 고유한 요소가 모두 포함됩니다.



```

string[] planets1 = { "Mercury", "Venus", "Earth", "Jupiter" };
string[] planets2 = { "Mercury", "Earth", "Mars", "Jupiter" };

IQueryable<string> query = from planet in planets1.Union(planets2)
                            select planet;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * Mercury
 * Venus
 * Earth
 * Jupiter
 * Mars
 */

```

## 참조

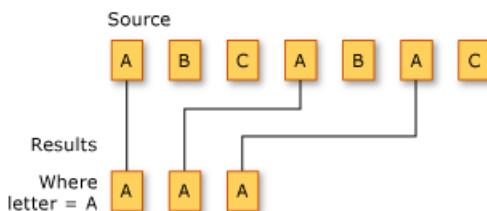
- [System.Linq](#)
- [표준 쿼리 연산자 개요\(C#\)](#)
- [문자열 컬렉션의 결합 및 비교 방법\(LINQ\)\(C#\)](#)
- [두 목록 간의 차집합을 구하는 방법\(LINQ\)\(C#\)](#)

# 데이터 필터링(C#)

2020-11-02 • 2 minutes to read • [Edit Online](#)

필터링은 지정된 조건을 충족하는 요소만 포함하도록 결과 집합을 제한하는 작업을 가리킵니다. 필터링은 선택이라고도 합니다.

다음 그림에서는 문자 시퀀스를 필터링한 결과를 보여 줍니다. 필터링 작업에 대한 조건자는 문자가 'A'가 되도록 지정합니다.



선택을 수행하는 표준 쿼리 연산자 메서드는 다음 섹션에 나열됩니다.

## 메서드

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
OfType	지정된 형식으로 캐스트할 수 있는지 여부에 따라 값을 선택합니다.	해당 사항 없음.	<a href="#">Enumerable.OfType</a> <a href="#">Queryable.OfType</a>
Where	조건자 함수를 기반으로 하는 값을 선택합니다.	where	<a href="#">Enumerable.Where</a> <a href="#">Queryable.Where</a>

## 쿼리 식 구문 예제

다음 예제에서는 `where` 절을 사용하여 배열에서 특정 길이의 문자열을 필터링합니다.

```
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
*/
```

## 참조

- [System.Linq](#)
- [표준 쿼리 연산자 개요\(C#\)](#)

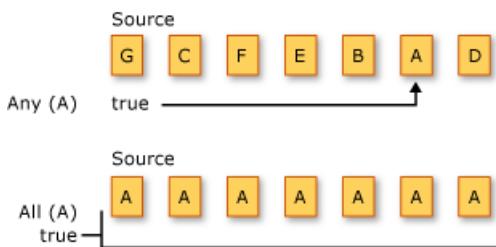
- `where` 절
- 런타임에 동적으로 조건자 필터 지정
- 리플렉션을 사용하여 어셈블리의 메타데이터를 쿼리하는 방법(LINQ)(C#)
- 지정된 특성 또는 이름을 사용하여 파일을 쿼리하는 방법(C#)
- 단어 또는 필드에 따라 텍스트 데이터를 정렬하거나 필터링하는 방법(LINQ)(C#)

# 수량자 작업(C#)

2020-11-02 • 5 minutes to read • [Edit Online](#)

수량자 작업은 시퀀스에서 조건을 충족하는 요소가 일부인지 전체인지를 나타내는 Boolean 값을 반환합니다.

다음 그림은 두 개의 서로 다른 소스 시퀀스에 대한 두 개의 서로 다른 수량자 작업을 보여 줍니다. 첫 번째 작업은 요소 중 하나 이상이 문자 'A'이고 결과가 true인지 물입니다. 두 번째 작업은 모든 요소가 문자 'A'이고 결과가 true인지 물입니다.



다음 섹션에는 수량자 작업을 수행하는 표준 쿼리 연산자 메서드가 나와 있습니다.

## 메서드

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
모두	시퀀스의 모든 요소가 조건을 만족하는지를 확인합니다.	해당 사항 없음.	<a href="#">Enumerable.All</a> <a href="#">Queryable.All</a>
임의의 값	시퀀스의 임의의 요소가 조건을 만족하는지를 확인합니다.	해당 사항 없음.	<a href="#">Enumerable.Any</a> <a href="#">Queryable.Any</a>
포함	시퀀스에 지정된 요소가 들어 있는지를 확인합니다.	해당 사항 없음.	<a href="#">Enumerable.Contains</a> <a href="#">Queryable.Contains</a>

## 쿼리 식 구문 예제

### 모두

다음 예제에서는 `All`을 사용하여 모든 문자열이 특정 길이인지 확인합니다.

```

class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market have all fruit names length equal to 5
    IEnumerable<string> names = from market in markets
                                 where market.Items.All(item => item.Length == 5)
                                 select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Kim's market
}

```

## 임의의 값

다음 예제에서는 `Any` 를 사용하여 모든 문자열이 'o'로 시작하는지 확인합니다.

```

class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market have any fruit names start with 'o'
    IEnumerable<string> names = from market in markets
                                 where market.Items.Any(item => item.StartsWith("o"))
                                 select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Kim's market
    // Adam's market
}

```

## 포함

다음 예제에서는 `Contains` 를 사용하여 배열에 특정 요소가 있는지 확인합니다.

```
class Market
{
    public string Name { get; set; }
    public string[] Items { get; set; }
}

public static void Example()
{
    List<Market> markets = new List<Market>
    {
        new Market { Name = "Emily's", Items = new string[] { "kiwi", "cheery", "banana" } },
        new Market { Name = "Kim's", Items = new string[] { "melon", "mango", "olive" } },
        new Market { Name = "Adam's", Items = new string[] { "kiwi", "apple", "orange" } },
    };

    // Determine which market contains fruit names equal 'kiwi'
    IEnumerable<string> names = from market in markets
                                  where market.Items.Contains("kiwi")
                                  select market.Name;

    foreach (string name in names)
    {
        Console.WriteLine($"{name} market");
    }

    // This code produces the following output:
    //
    // Emily's market
    // Adam's market
}
```

## 참조

- [System.Linq](#)
- [표준 쿼리 연산자 개요\(C#\)](#)
- [런타임에 동적으로 조건자 필터 지정](#)
- [지정된 단어 집합이 들어 있는 문장을 쿼리하는 방법\(LINQ\)\(C#\)](#)

# 프로젝션 작업(C#)

2020-11-02 • 7 minutes to read • [Edit Online](#)

프로젝션은 주로 이후에 사용할 속성으로만 구성된 새 양식으로 개체를 변환하는 작업을 가리킵니다. 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.

다음 섹션에는 프로젝션을 수행하는 표준 쿼리 연산자 메서드가 나와 있습니다.

## 메서드

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
선택	변환 함수를 기반으로 하는 값을 프로젝션합니다.	<code>select</code>	<a href="#">Enumerable.Select</a> <a href="#">Queryable.Select</a>
SelectMany	변환 함수를 기반으로 하는 값의 시퀀스를 프로젝션한 다음 하나의 시퀀스로 평면화합니다.	여러 <code>from</code> 절 사용	<a href="#">Enumerable.SelectMany</a> <a href="#">Queryable.SelectMany</a>

## 쿼리 식 구문 예제

### 선택

다음 예제에서는 `select` 절을 사용하여 문자열 목록의 각 문자열에서 첫 글자를 프로젝션합니다.

```
List<string> words = new List<string>() { "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
*/
```

### SelectMany

다음 예제에서는 여러 `from` 절을 사용하여 문자열 목록의 각 문자열에서 각 단어를 프로젝션합니다.

```

List<string> phrases = new List<string>() { "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
*/

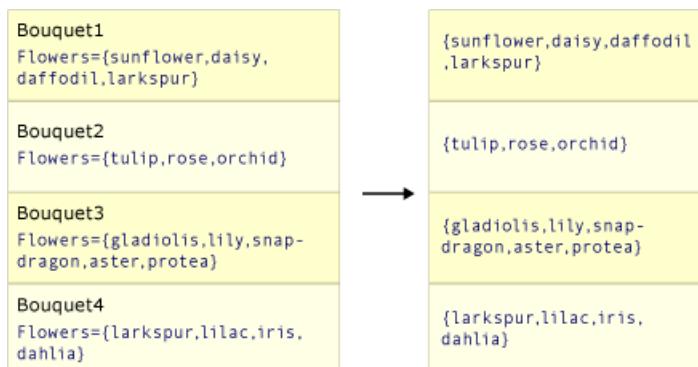
```

## Select 및 SelectMany

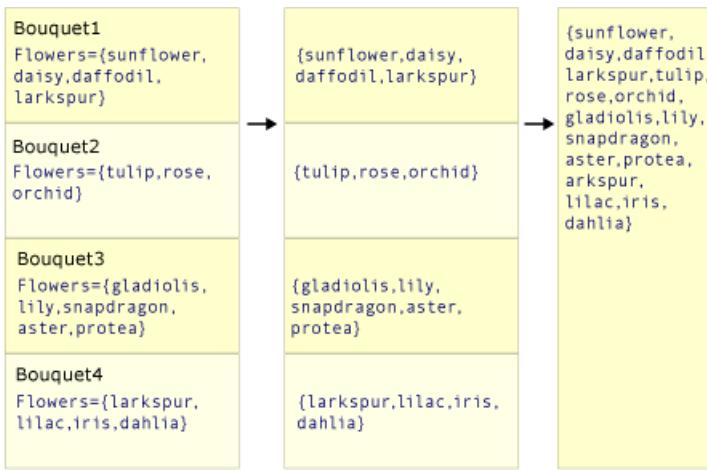
`Select()` 및 `SelectMany()` 둘 다의 작업은 소스 값에서 결과 값을 생성하는 것입니다. `Select()`는 모든 소스 값에 대해 하나의 결과 값을 생성합니다. 따라서 전체 결과는 소스 컬렉션과 동일한 개수의 요소가 들어 있는 컬렉션입니다. 반면, `SelectMany()`는 각 소스 값에서 연결된 하위 컬렉션을 포함하는 하나의 전체 결과를 생성합니다. `SelectMany()`에 대한 인수로 전달되는 변환 함수는 각 소스 값에 대해 열거 가능한 값 시퀀스를 반환해야 합니다. 이러한 열거 가능한 시퀀스는 `SelectMany()`에 의해 연결되어 하나의 큰 시퀀스를 만듭니다.

다음 두 그림은 이러한 두 메서드의 작업 간에 개념적 차이를 보여 줍니다. 각각의 경우에서 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 처음 두 항목을 사용하여 꽃 "부케"를 만듭니다. 이 예제에서 변환 함수 `Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>)`가 사용하는 "단일 값"은 값 컬렉션입니다. 이 경우 각 하위 시퀀스의 각 문자열을 열거하기 위해 `foreach` 루프가 추가로 필요합니다.

```

class Bouquet
{
    public List<string> Flowers { get; set; }
}

static void SelectVsSelectMany()
{
    List<Bouquet> bouquets = new List<Bouquet>() {
        new Bouquet { Flowers = new List<string> { "sunflower", "daisy", "daffodil", "larkspur" } },
        new Bouquet { Flowers = new List<string> { "tulip", "rose", "orchid" } },
        new Bouquet { Flowers = new List<string> { "gladiolus", "lily", "snapdragon", "aster", "protea" } },
        new Bouquet { Flowers = new List<string> { "larkspur", "lilac", "iris", "dahlia" } }
    };

    // ***** Select *****
    IEnumerable<List<string>> query1 = bouquets.Select(bq => bq.Flowers);

    // ***** SelectMany *****
    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);

    /* This code produces the following output:

    Results by using Select():
    sunflower
    daisy
    daffodil
    larkspur
    tulip
    rose
    orchid
    gladiolus
    lily
    snapdragon
    aster
    */
}

```

```
protea
larkspur
lilac
iris
dahlia

Results by using SelectMany():
sunflower
daisy
daffodil
larkspur
tulip
rose
orchid
gladiolus
lily
snapdragon
aster
protea
larkspur
lilac
iris
dahlia
*/
}

}
```

## 참조

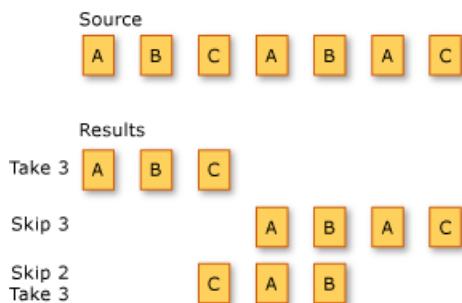
- [System.Linq](#)
- [표준 쿼리 연산자 개요\(C#\)](#)
- [select 절](#)
- [여러 소스로 개체 컬렉션을 채우는 방법\(LINQ\)\(C#\)](#)
- [그룹을 사용하여 파일을 여러 파일로 분할하는 방법\(LINQ\)\(C#\)](#)

# 데이터 분할(C#)

2020-11-02 • 2 minutes to read • [Edit Online](#)

LINQ의 분할은 요소를 다시 정렬한 후 섹션 중 하나를 반환하지 않고 입력 시퀀스를 두 개의 섹션으로 나누는 작업을 가리킵니다.

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

연산자 이름	설명	C# 쿼리 식 구문	추가 정보
Skip	시퀀스에서 지정한 위치까지 요소를 건너뜁니다.	해당 사항 없음.	<a href="#">Enumerable.Skip</a> <a href="#">Queryable.Skip</a>
SkipWhile	요소가 조건을 충족하지 않을 때까지 조건자 함수를 기반으로 하여 요소를 건너뜁니다.	해당 사항 없음.	<a href="#">Enumerable.SkipWhile</a> <a href="#">Queryable.SkipWhile</a>
Take	시퀀스에서 지정된 위치까지 요소를 사용합니다.	해당 사항 없음.	<a href="#">Enumerable.Take</a> <a href="#">Queryable.Take</a>
TakeWhile	요소가 조건을 충족하지 않을 때까지 조건자 함수를 기반으로 하여 요소를 사용합니다.	해당 사항 없음.	<a href="#">Enumerable.TakeWhile</a> <a href="#">Queryable.TakeWhile</a>

## 참조

- [System.Linq](#)
- [표준 쿼리 연산자 개요\(C#\)](#)

# Join 작업(C#)

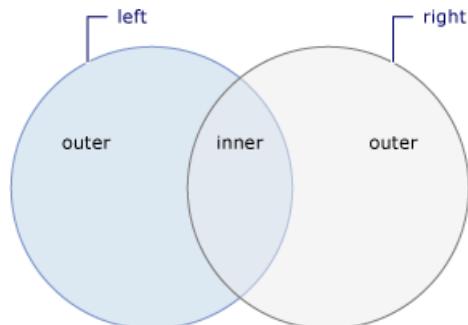
2020-11-02 • 8 minutes to read • [Edit Online](#)

두 데이터 소스를 조인하는 것은 한 데이터 소스의 개체를 공통 특성을 공유하는 다른 데이터 소스의 개체와 연결하는 것입니다.

서로 간의 관계를 직접 적용할 수 없는 데이터 원본을 대상으로 하는 쿼리에서는 Join이 중요한 작업입니다. 개체 지향 프로그래밍에서 데이터 소스 간의 관계를 직접 적용할 수 없다는 것은 모델링되지 않은 개체 간에 상관관계가 있음을 의미할 수 있습니다(예: 단방향 관계에서 반대 방향을 사용). 단방향 관계의 예로는 Customer 클래스가 City 형식 속성을 포함하는데 City 클래스는 Customer 개체의 컬렉션인 속성을 포함하지 않는 경우가 있습니다. City 개체 목록이 있는 경우 각 구/군/시의 모든 고객을 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 '같음'이 아닌 '보다 작음' 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

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>	<a href="#">Enumerable.Join</a> <a href="#">Queryable.Join</a>
GroupJoin	키 선택기 함수를 기준으로 두 시퀀스를 Join한 다음 결과로 생성된 일치 항목을 요소마다 그룹화합니다.	<code>join ... in ... on ... equals ... into ...</code>	<a href="#">Enumerable.GroupJoin</a> <a href="#">Queryable.GroupJoin</a>

## 쿼리 식 구문 예제

### Join

다음 예제에서는 `join ... in ... on ... equals ...` 절을 사용하여 특정 값을 기준으로 두 시퀀스를 조인합니다.

```

class Product
{
    public string Name { get; set; }
    public int CategoryId { get; set; }
}

class Category
{
    public int Id { get; set; }
    public string CategoryName { get; set; }
}

public static void Example()
{
    List<Product> products = new List<Product>
    {
        new Product { Name = "Cola", CategoryId = 0 },
        new Product { Name = "Tea", CategoryId = 0 },
        new Product { Name = "Apple", CategoryId = 1 },
        new Product { Name = "Kiwi", CategoryId = 1 },
        new Product { Name = "Carrot", CategoryId = 2 },
    };

    List<Category> categories = new List<Category>
    {
        new Category { Id = 0, CategoryName = "Beverage" },
        new Category { Id = 1, CategoryName = "Fruit" },
        new Category { Id = 2, CategoryName = "Vegetable" }
    };

    // Join products and categories based on CategoryId
    var query = from product in products
                join category in categories on product.CategoryId equals category.Id
                select new { product.Name, category.CategoryName };

    foreach (var item in query)
    {
        Console.WriteLine($"{item.Name} - {item.CategoryName}");
    }

    // This code produces the following output:
    //
    // Cola - Beverage
    // Tea - Beverage
    // Apple - Fruit
    // Kiwi - Fruit
    // Carrot - Vegetable
}

```

## GroupJoin

다음 예제에서는 `join ... in ... on ... equals ... into ...` 절을 사용하여 특정 값을 기준으로 두 시퀀스를 조인하고 각 요소에 대해 결과 일치 항목을 그룹화합니다.

```

class Product
{
    public string Name { get; set; }
    public int CategoryId { get; set; }
}

class Category
{
    public int Id { get; set; }
    public string CategoryName { get; set; }
}

public static void Example()
{
    List<Product> products = new List<Product>
    {
        new Product { Name = "Cola", CategoryId = 0 },
        new Product { Name = "Tea", CategoryId = 0 },
        new Product { Name = "Apple", CategoryId = 1 },
        new Product { Name = "Kiwi", CategoryId = 1 },
        new Product { Name = "Carrot", CategoryId = 2 },
    };

    List<Category> categories = new List<Category>
    {
        new Category { Id = 0, CategoryName = "Beverage" },
        new Category { Id = 1, CategoryName = "Fruit" },
        new Category { Id = 2, CategoryName = "Vegetable" }
    };

    // Join categories and product based on CategoryId and grouping result
    var productGroups = from category in categories
                        join product in products on category.Id equals product.CategoryId into productGroup
                        select productGroup;

    foreach (IEnumerable<Product> productGroup in productGroups)
    {
        Console.WriteLine("Group");
        foreach (Product product in productGroup)
        {
            Console.WriteLine($"{product.Name,8}");
        }
    }

    // This code produces the following output:
    //
    // Group
    //     Cola
    //     Tea
    // Group
    //     Apple
    //     Kiwi
    // Group
    //     Carrot
}

```

## 참조

- [System.Linq](#)
- [표준 쿼리 연산자 개요\(C#\)](#)
- [익명 형식](#)
- [Join 및 교차곱 쿼리 작성](#)
- [join 절](#)

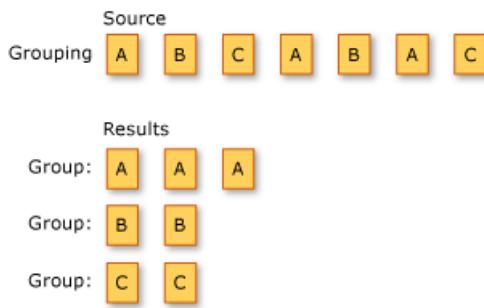
- Join 복합 키를 사용하여
- 서로 다른 파일의 콘텐츠를 조인하는 방법(LINQ)(C#)
- Join 절 결과를 서순대로 정렬
- 사용자 지정 조인 작업 수행
- 그룹화 조인 수행
- 내부 조인 수행
- 왼쪽 우선 외부 조인 수행
- 여러 소스로 개체 컬렉션을 채우는 방법(LINQ)(C#)

# 데이터 그룹화(C#)

2020-11-02 • 2 minutes to read • [Edit Online](#)

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.



데이터 요소를 그룹화하는 표준 쿼리 연산자 메서드가 다음 섹션에 나와 있습니다.

## 메서드

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
GroupBy	공통 특성을 공유하는 요소를 그룹화합니다. 각 그룹은 <code>IGrouping&lt; TKey, TElement &gt;</code> 개체로 표시됩니다.	<code>group ... by</code> 또는 <code>group ... by ... into ...</code>	<a href="#">Enumerable.GroupBy</a> <a href="#">Queryable.GroupBy</a>
ToLookup	키 선택기 함수에 따라 <code>Lookup&lt; TKey, TElement &gt;</code> (일대다 사전)에 요소를 삽입합니다.	해당 사항 없음.	<a href="#">Enumerable.ToLookup</a>

## 쿼리 식 구문 예제

다음 코드 예제에서는 `group by` 절을 사용하여 짹수 또는 홀수인지에 따라 목록에서 정수를 그룹화합니다.

```

List<int> numbers = new List<int>() { 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);
}

/* This code produces the following output:

Odd numbers:
35
3987
199
329

Even numbers:
44
200
84
4
446
208
*/

```

## 참조

- [System.Linq](#)
- [표준 쿼리 연산자 개요\(C#\)](#)
- [group 절](#)
- [중첩 그룹 만들기](#)
- [확장명에 따라 파일을 그룹화하는 방법\(LINQ\)\(C#\)](#)
- [쿼리 결과 그룹화](#)
- [그룹화 작업에서 하위 쿼리 수행](#)
- [그룹을 사용하여 파일을 여러 파일로 분할하는 방법\(LINQ\)\(C#\)](#)

# 생성 작업(C#)

2020-11-02 • 2 minutes to read • [Edit Online](#)

생성은 값의 새 시퀀스를 만드는 작업을 나타냅니다.

생성을 수행하는 표준 쿼리 연산자 메서드는 다음 섹션에 나열됩니다.

## 메서드

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
DefaultIfEmpty	빈 컬렉션을 기본값을 갖는 singleton 컬렉션으로 바꿉니다.	해당 사항 없음.	<a href="#">Enumerable.DefaultIfEmpty</a> <a href="#">Queryable.DefaultIfEmpty</a>
Empty	비어 있는 컬렉션을 반환합니다.	해당 사항 없음.	<a href="#">Enumerable.Empty</a>
범위	일련의 숫자를 포함하는 컬렉션을 생성합니다.	해당 사항 없음.	<a href="#">Enumerable.Range</a>
Repeat	반복되는 값이 하나 들어 있는 컬렉션을 생성합니다.	해당 사항 없음.	<a href="#">Enumerable.Repeat</a>

## 참조

- [System.Linq](#)
- [표준 쿼리 연산자 개요\(C#\)](#)

# 같음 연산(C#)

2020-11-02 • 2 minutes to read • [Edit Online](#)

해당 요소가 동일하고 같은 수의 요소를 포함 하는 두 시퀀스는 같은 것으로 간주됩니다.

## 메서드

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
SequenceEqual	쌍 단위 방식으로 요소를 비교하여 두 시퀀스가 서로 같은지 확인합니다.	해당 사항 없음.	<a href="#">Enumerable.SequenceEqual</a> <a href="#">Queryable.SequenceEqual</a>

## 참조

- [System.Linq](#)
- [표준 쿼리 연산자 개요\(C#\)](#)
- [두 폴더의 내용을 비교하는 방법\(LINQ\)\(C#\)](#)

# 요소 작업(C#)

2020-11-02 • 3 minutes to read • [Edit Online](#)

요소 작업은 시퀀스에서 특정 단일 요소를 반환합니다.

다음 섹션에는 요소 작업을 수행하는 표준 쿼리 연산자 메서드가 나열되어 있습니다.

## 메서드

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
ElementAt	컬렉션의 지정된 인덱스에 있는 요소를 반환합니다.	해당 사항 없음.	<a href="#">Enumerable.ElementAt</a> <a href="#">Queryable.ElementAt</a>
ElementAtOrDefault	컬렉션의 지정된 인덱스에 있는 요소를 반환하거나 인덱스가 범위를 벗어나면 기본값을 반환합니다.	해당 사항 없음.	<a href="#">Enumerable.ElementAtOrDefault</a> <a href="#">Queryable.ElementAtOrDefault</a>
First	컬렉션의 첫 번째 요소 또는 특정 조건에 맞는 첫 번째 요소를 반환합니다.	해당 사항 없음.	<a href="#">Enumerable.First</a> <a href="#">Queryable.First</a>
FirstOrDefault	컬렉션의 첫 번째 요소 또는 특정 조건에 맞는 첫 번째 요소를 반환합니다. 이러한 요소가 없으면 기본값을 반환합니다.	해당 사항 없음.	<a href="#">Enumerable.FirstOrDefault</a> <a href="#">Queryable.FirstOrDefault</a> <a href="#">Queryable.FirstOrDefault&lt;TSource&gt;</a> <a href="#">(IQueryable&lt;TSource&gt;)</a>
마지막	컬렉션의 마지막 요소 또는 특정 조건에 맞는 마지막 요소를 반환합니다.	해당 사항 없음.	<a href="#">Enumerable.Last</a> <a href="#">Queryable.Last</a>
LastOrDefault	컬렉션의 마지막 요소 또는 특정 조건에 맞는 마지막 요소를 반환합니다. 이러한 요소가 없으면 기본값을 반환합니다.	해당 사항 없음.	<a href="#">Enumerable.LastOrDefault</a> <a href="#">Queryable.LastOrDefault</a>
Single	컬렉션의 유일한 요소 또는 특정 조건에 맞는 유일한 요소를 반환합니다. 반환할 요소가 없거나 두 개 이상 있는 경우 <a href="#">InvalidOperationException</a> 을 throw합니다.	해당 사항 없음.	<a href="#">Enumerable.Single</a> <a href="#">Queryable.Single</a>

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
SingleOrDefault	컬렉션의 유일한 요소 또는 특정 조건에 맞는 유일한 요소를 반환합니다. 반환할 요소가 없는 경우 기본값을 반환합니다. 반환할 요소가 두 개 이상 있는 경우 <a href="#">InvalidOperationException</a> 을 throw합니다.	해당 사항 없음.	<a href="#">Enumerable.SingleOrDefault</a> <a href="#">Queryable.SingleOrDefault</a>

## 참조

- [System.Linq](#)
- [표준 쿼리 연산자 개요\(C#\)](#)
- [디렉터리 트리에서 가장 큰 파일을 하나 이상 쿼리하는 방법\(LINQ\)\(C#\)](#)

# 데이터 형식 변환(C#)

2020-11-02 • 5 minutes to read • [Edit Online](#)

변환 메서드는 입력 객체의 형식을 변경합니다.

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 실행하는데 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.

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
AsEnumerable	IEnumerable<T>로 형식화된 입력을 반환합니다.	해당 사항 없음.	<a href="#">Enumerable.AsEnumerable</a>
AsQueryable	(제네릭) IEnumerable을 (제네릭) IQueryable로 변환합니다.	해당 사항 없음.	<a href="#">Queryable.AsQueryable</a>
Cast	컬렉션의 요소를 지정된 형식으로 캐스트합니다.	명시적 형식 범위 변수를 사용합니다. 예를 들어: <pre>from string str in words</pre>	<a href="#">Enumerable.Cast</a> <a href="#">Queryable.Cast</a>
OfType	지정된 형식으로 캐스트할 수 있는지 여부에 따라 값을 필터링합니다.	해당 사항 없음.	<a href="#">Enumerable.OfType</a> <a href="#">Queryable.OfType</a>
ToArray	컬렉션을 배열로 변환합니다. 이 메서드는 쿼리를 강제로 실행합니다.	해당 사항 없음.	<a href="#">Enumerable.ToArray</a>
ToDictionary	키 선택기 함수에 따라 Dictionary< TKey, TValue >에 요소를 배치합니다. 이 메서드는 쿼리를 강제로 실행합니다.	해당 사항 없음.	<a href="#">Enumerable.ToDictionary</a>

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
ToList	컬렉션을 <a href="#">List&lt;T&gt;</a> 로 변환합니다. 이 메서드는 쿼리를 강제로 실행합니다.	해당 사항 없음.	<a href="#">Enumerable.ToList</a>
ToLookup	키 선택기 함수에 따라 <a href="#">Lookup&lt; TKey, TElement &gt;</a> (일대 다 사전)에 요소를 배치합니다. 이 메서드는 쿼리를 강제로 실행합니다.	해당 사항 없음.	<a href="#">Enumerable.ToLookup</a>

## 쿼리 식 구문 예제

다음 코드 예제에서는 명시적 형식 범위 변수를 사용하여 하위 형식에서만 사용할 수 있는 멤버에 액세스하기 전에 형식을 하위 형식으로 캐스트합니다.

```
class Plant
{
    public string Name { get; set; }
}

class CarnivorousPlant : Plant
{
    public string TrapType { get; set; }
}

static void Cast()
{
    Plant[] plants = new Plant[] {
        new CarnivorousPlant { Name = "Venus Fly Trap", TrapType = "Snap Trap" },
        new CarnivorousPlant { Name = "Pitcher Plant", TrapType = "Pitfall Trap" },
        new CarnivorousPlant { Name = "Sundew", TrapType = "Flypaper Trap" },
        new CarnivorousPlant { Name = "Waterwheel Plant", TrapType = "Snap Trap" }
    };

    var query = from CarnivorousPlant cPlant in plants
               where cPlant.TrapType == "Snap Trap"
               select cPlant;

    foreach (Plant plant in query)
        Console.WriteLine(plant.Name);

    /* This code produces the following output:
     *
     * Venus Fly Trap
     * Waterwheel Plant
     */
}
```

## 참조

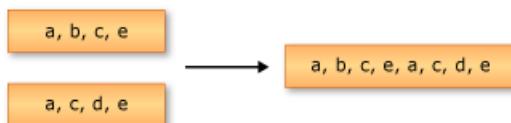
- [System.Linq](#)
- [표준 쿼리 연산자 개요\(C#\)](#)
- [from 절](#)
- [LINQ 쿼리 식](#)
- [LINQ를 사용하여 ArrayList를 쿼리하는 방법\(C#\)](#)

# 연결 작업(C#)

2020-11-02 • 2 minutes to read • [Edit Online](#)

연결은 한 시퀀스를 다른 시퀀스에 추가하는 작업을 나타냅니다.

다음 그림은 두 개의 문자 시퀀스에 대한 연결 작업을 보여 줍니다.



다음 섹션에는 연결을 수행하는 표준 쿼리 연산자 메서드가 나와 있습니다.

## 메서드

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
Concat	두 시퀀스를 연결하여 하나의 시퀀스를 구성합니다.	해당 사항 없음.	<a href="#">Enumerable.Concat</a> <a href="#">Queryable.Concat</a>

## 참조

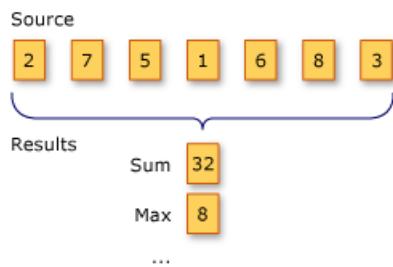
- [System.Linq](#)
- [표준 쿼리 연산자 개요\(C#\)](#)
- [문자열 컬렉션의 결합 및 비교 방법\(LINQ\)\(C#\)](#)

# 집계 작업(C#)

2020-11-02 • 3 minutes to read • [Edit Online](#)

집계 작업에서는 값의 컬렉션에서 하나의 값을 계산합니다. 예를 들어 1달 동안의 일일 온도 값에서 평균 일일 온도를 계산하는 것이 집계 작업입니다.

다음 그림은 숫자 시퀀스에 대한 두 가지 집계 작업의 결과를 보여 줍니다. 첫 번째 작업은 숫자의 합계를 계산합니다. 두 번째 작업은 시퀀스의 최대 값을 반환합니다.



다음 섹션에는 집계 작업을 수행하는 표준 쿼리 연산자 메서드가 나와 있습니다.

## 메서드

메서드 이름	설명	C# 쿼리 식 구문	추가 정보
Aggregate	컬렉션 값에 대해 사용자 지정 집계 작업을 수행합니다.	해당 사항 없음.	<a href="#">Enumerable.Aggregate</a> <a href="#">Queryable.Aggregate</a>
평균	값 컬렉션의 평균 값을 계산합니다.	해당 사항 없음.	<a href="#">Enumerable.Average</a> <a href="#">Queryable.Average</a>
개수	컬렉션에서 요소(선택적으로 조건자 함수를 총족하는 요소만) 개수를 계산합니다.	해당 사항 없음.	<a href="#">Enumerable.Count</a> <a href="#">Queryable.Count</a>
LongCount	큰 컬렉션에서 요소(선택적으로 조건자 함수를 총족하는 요소만) 개수를 계산합니다.	해당 사항 없음.	<a href="#">Enumerable.LongCount</a> <a href="#">Queryable.LongCount</a>
최대	컬렉션의 최대값을 확인합니다.	해당 사항 없음.	<a href="#">Enumerable.Max</a> <a href="#">Queryable.Max</a>
최소	컬렉션의 최소값을 확인합니다.	해당 사항 없음.	<a href="#">Enumerable.Min</a> <a href="#">Queryable.Min</a>
Sum	컬렉션에 있는 값의 합계를 계산합니다.	해당 사항 없음.	<a href="#">Enumerable.Sum</a> <a href="#">Queryable.Sum</a>

## 참조

- [System.Linq](#)
- [표준 쿼리 연산자 개요\(C#\)](#)
- [CSV 텍스트 파일의 열 값을 컴퓨팅하는 방법\(LINQ\)\(C#\)](#)
- [디렉터리 트리에서 가장 큰 파일을 하나 이상 쿼리하는 방법\(LINQ\)\(C#\)](#)
- [폴더 집합의 전체 바이트 수를 쿼리하는 방법\(LINQ\)\(C#\)](#)

# LINQ to Objects(C#)

2020-11-02 • 5 minutes to read • [Edit Online](#)

"LINQ to Objects"라는 용어는 중간 LINQ 공급자 또는 [LINQ to SQL](#), [LINQ to XML](#) 등의 API를 사용하지 않고 모든 [IEnumerable](#) 또는 [IEnumerable<T>](#) 컬렉션에 대해 LINQ 쿼리를 직접 사용하는 것입니다. LINQ를 사용하면 [List<T>](#), [Array](#), [Dictionary< TKey, TValue >](#) 등의 모든 열거 가능 컬렉션을 쿼리할 수 있습니다. 컬렉션은 사용자가 정의할 수도 있고 .NET API에서 반환할 수도 있습니다.

기본적으로 LINQ to Objects는 새로운 컬렉션 방식을 나타냅니다. 이전에는 컬렉션에서 데이터를 검색하는 방법을 지정하는 복잡한 `foreach` 루프를 작성해야 했습니다. 그러나 LINQ 방식에서는 검색 할 항목을 설명하는 선언적 코드를 작성합니다.

또한 LINQ 쿼리는 기존의 `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.

일반적으로는 수행하려는 데이터 작업이 복잡할수록 기존의 반복 기법 대신 LINQ를 사용하면 더 큰 이점을 얻을 수 있습니다.

이 섹션에서는 몇 가지 예제를 통해 LINQ 방식에 대해 설명합니다. 여기서 설명하는 방식 외에도 다양한 방식을 사용할 수 있습니다.

## 섹션 내용

### [LINQ 및 문자열\(C#\)](#)

LINQ를 사용하여 문자열 및 문자열 컬렉션을 쿼리하고 변환하는 방법을 설명합니다. 또한 이러한 원칙을 설명하는 문서의 링크도 제공합니다.

### [LINQ 및 리플렉션\(C#\)](#)

LINQ에서 리플렉션을 사용하는 방식을 보여 주는 샘플 링크를 제공합니다.

### [LINQ 및 파일 딕렉터리\(C#\)](#)

LINQ를 사용하여 파일 시스템을 조작하는 방법을 설명합니다. 또한 이러한 개념을 설명하는 문서의 링크도 제공합니다.

### [LINQ를 사용하여 ArrayList를 쿼리하는 방법\(C#\)](#)

C#에서 ArrayList를 쿼리하는 방법을 보여 줍니다.

### [LINQ 쿼리용 사용자 지정 메서드를 추가하는 방법\(C#\)](#)

[IEnumerable<T>](#) 인터페이스에 확장 메서드를 추가하여 LINQ 쿼리에 대해 사용할 수 있는 메서드 집합 확장 방법을 설명합니다.

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

LINQ 관련 설명과 쿼리를 수행하는 코드 예제를 제공하는 문서의 링크가 나와 있습니다.

# LINQ 및 문자열(C#)

2020-11-02 • 8 minutes to read • [Edit Online](#)

LINQ를 사용하여 문자열 및 문자열 컬렉션을 쿼리하고 변환할 수 있습니다. 텍스트 파일의 반구조적 데이터에 특히 유용할 수 있습니다. LINQ 쿼리에 기존의 문자열 함수 및 정규식을 결합할 수 있습니다. 예를 들어 `String.Split` 또는 `Regex.Split` 메서드를 사용하여 문자열 배열을 만든 다음 LINQ를 사용하여 쿼리하거나 수정할 수 있습니다. LINQ 쿼리의 `where` 절에 `Regex.IsMatch` 메서드를 사용할 수 있습니다. 또한 LINQ를 사용하여 정규식에서 반환된 `MatchCollection` 결과를 쿼리하거나 수정할 수 있습니다.

이 섹션에 설명된 기법을 사용하여 반구조적 텍스트 데이터를 XML로 변환할 수도 있습니다. 자세한 내용은 [CSV 파일에서 XML을 생성하는 방법](#)을 참조하세요.

이 섹션의 예제는 다음 두 가지 범주로 구분됩니다.

## 텍스트 블록 쿼리

`String.Split` 메서드 또는 `Regex.Split` 메서드를 사용하여 더 작은 문자열의 쿼리 가능 배열로 분할하면 텍스트 블록을 쿼리, 분석, 수정할 수 있습니다. 소스 텍스트를 단어, 문장, 단락, 페이지 또는 기타 기준으로 분할한 다음 쿼리에 필요한 경우 추가 분할을 수행할 수 있습니다.

- [문자열에서 단어가 나오는 횟수를 세는 방법\(LINQ\)\(C#\)](#)  
간단한 텍스트 쿼리를 위해 LINQ를 사용하는 방법을 보여 줍니다.
- [지정된 단어 집합이 들어 있는 문장을 쿼리하는 방법\(LINQ\)\(C#\)](#)  
임의의 경계에서 텍스트 파일을 분할하는 방법 및 각 부분에 대해 쿼리를 수행하는 방법을 보여 줍니다.
- [문자열의 문자를 쿼리하는 방법\(LINQ\)\(C#\)](#)  
문자열이 쿼리 가능한 형식인지를 보여 줍니다.
- [LINQ 쿼리와 정규식 결합 방법\(C#\)](#)  
필터링된 쿼리 결과에 대한 복잡한 패턴 일치를 위해 LINQ 쿼리에서 정규식을 사용하는 방법을 보여 줍니다.

## 텍스트 형식의 반구조적 데이터 쿼리

다양한 형식의 텍스트 파일은 대체로 탭 또는 쉼표로 구분된 파일 또는 고정 길이 줄과 같은 유사한 형식을 가진 일련의 줄로 이루어져 있습니다. 이러한 텍스트 파일을 메모리로 읽어온 후 LINQ를 사용하여 줄을 쿼리 및/또는 수정할 수 있습니다. 또한 LINQ 쿼리는 여러 소스의 데이터를 결합하는 작업을 간소화합니다.

- [두 목록 간의 차집합을 구하는 방법\(LINQ\)\(C#\)](#)  
한 목록에는 있지만 다른 목록에는 없는 모든 문자열을 찾는 방법을 보여 줍니다.
- [단어 또는 필드에 따라 텍스트 데이터를 정렬하거나 필터링하는 방법\(LINQ\)\(C#\)](#)  
단어 또는 필드에 따라 텍스트 줄을 정렬하는 방법을 보여 줍니다.
- [구분된 파일의 필드를 다시 정렬하는 방법\(LINQ\)\(C#\)](#)  
.csv 파일에서 줄의 필드 순서를 변경하는 방법을 보여 줍니다.
- [문자열 컬렉션의 결합 및 비교 방법\(LINQ\)\(C#\)](#)

다양한 방법으로 문자열 목록을 조합하는 방법을 보여 줍니다.

- [여러 소스로 개체 컬렉션을 채우는 방법\(LINQ\)\(C#\)](#)

여러 텍스트 파일을 데이터 소스로 사용하여 개체 컬렉션을 만드는 방법을 보여 줍니다.

- [서로 다른 파일의 콘텐츠를 조인하는 방법\(LINQ\)\(C#\)](#)

일치하는 키를 사용하여 두 목록의 문자열을 단일 문자열로 결합하는 방법을 보여 줍니다.

- [그룹을 사용하여 파일을 여러 파일로 분할하는 방법\(LINQ\)\(C#\)](#)

단일 파일을 데이터 소스로 사용하여 새 파일을 만드는 방법을 보여 줍니다.

- [CSV 텍스트 파일의 열 값을 컴퓨팅하는 방법\(LINQ\)\(C#\)](#)

.csv 파일의 텍스트 데이터에 대해 수학적 계산을 수행하는 방법을 보여 줍니다.

## 참조

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

- [CSV 파일에서 XML을 생성하는 방법](#)

# 문자열에서 단어가 나오는 횟수를 세는 방법 (LINQ)(C#)

2020-11-02 • 3 minutes to read • [Edit Online](#)

이 예제에서는 LINQ 쿼리를 사용하여 문자열에서 지정된 단어의 발생 수를 계산하는 방법을 보여 줍니다. 계산을 수행하려면 먼저 [Split](#) 메서드를 호출하여 단어 배열을 만듭니다. [Split](#) 메서드를 사용하는 경우 성능이 저하됩니다. 문자열의 유일한 작업이 단어 개수 계산인 경우 [Matches](#) 또는 [IndexOf](#) 메서드를 대신 사용하는 것이 좋습니다. 그러나 성능이 중요한 문제가 아니거나 다른 유형의 쿼리를 수행하기 위해 이미 문장을 분할한 경우 LINQ를 사용하여 단어 또는 구도 계산하는 것이 좋습니다.

## 예제

```
class CountWords
{
    static void Main()
    {
        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
        string[] source = text.Split(new char[] { '.', '?', '!', ' ', ';' , ':' , ',' }, StringSplitOptions.RemoveEmptyEntries);

        // Create the query. Use ToLowerInvariant to match "data" and "Data"
        var matchQuery = from word in source
                        where word.ToLowerInvariant() == searchTerm.ToLowerInvariant()
                        select word;

        // Count the matches, which executes the query.
        int wordCount = matchQuery.Count();
        Console.WriteLine("{0} occurrence(s) of the search term \"{1}\" were found.", wordCount,
            searchTerm);

        // Keep console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* Output:
   3 occurrence(s) of the search term "data" were found.
*/
```

## 코드 컴파일

System.Linq 및 System.IO 네임스페이스에 대한 `using` 지시문을 통해 C# 콘솔 애플리케이션 프로젝트를 만듭니다.

## 참조

- LINQ 및 문자열(C#)

# 지정된 단어 집합이 들어 있는 문장을 쿼리하는 방법(LINQ)(C#)

2020-11-02 • 4 minutes to read • [Edit Online](#)

이 예제에서는 지정된 각 단어 집합과 일치하는 항목이 포함된 문장을 텍스트 파일에서 찾는 방법을 보여 줍니다. 이 예제에서는 검색어 배열이 하드 코드되어 있지만 런타임에 동적으로 채워질 수도 있습니다. 이 예제에서 쿼리는 "Historically", "data" 및 "integrated" 단어가 포함된 문장을 반환합니다.

## 예제

```
class FindSentences
{
    static void Main()
    {
        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(new char[] { '.', '?', '!' });

        // Define the search terms. This list could also be dynamically populated at runtime.
        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.
        var sentenceQuery = from sentence in sentences
                            let w = sentence.Split(new char[] { '.', '?', '!', ' ', ';', ':', ',' },
                                StringSplitOptions.RemoveEmptyEntries)
                            where w.Distinct().Intersect(wordsToMatch).Count() == wordsToMatch.Count()
                            select sentence;

        // Execute the query. Note that you can explicitly type
        // the iteration variable here even though sentenceQuery
        // was implicitly typed.
        foreach (string str in sentenceQuery)
        {
            Console.WriteLine(str);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}

/* Output:
Historically, the world of data and the world of objects have not been well integrated
*/
```

쿼리에서는 먼저 텍스트를 문장으로 분할한 다음 문장을 각 단어가 포함된 문자열 배열로 분할합니다. 각 배열

에 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** 호출에서는 문자열의 구분 기호를 제거하기 위해 문장 부호가 구분 기호로 사용되었습니다. 이렇게 하지 않았다면, 예를 들어 `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.

## 코드 컴파일

System.Linq 및 System.IO 네임스페이스에 대한 `using` 지시문을 통해 C# 콘솔 애플리케이션 프로젝트를 만듭니다.

## 참조

- [LINQ 및 문자열\(C#\)](#)

# 문자열의 문자를 쿼리하는 방법(LINQ)(C#)

2020-11-02 • 2 minutes to read • [Edit Online](#)

`String` 클래스는 제네릭 `IEnumerable<T>` 인터페이스를 구현하기 때문에 모든 문자열을 문자 시퀀스로 쿼리할 수 있습니다. 그러나 LINQ는 일반적으로 이 용도로 사용되지 않습니다. 복잡한 패턴 일치 작업의 경우 `Regex` 클래스를 사용합니다.

## 예제

다음 예제에서는 문자열을 쿼리하여 문자열에 포함된 숫자 자릿수를 확인합니다. 쿼리가 처음 실행된 후 "다시 사용"됩니다. 이 작업은 쿼리 자체가 실제 결과를 저장하지 않기 때문에 가능합니다.

```
class QueryAString
{
    static void Main()
    {
        string aString = "ABCDE99F-J74-12-89A";

        // Select only those characters that are numbers
        IEnumerable<char> 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 = {0}", count);

        // Select all characters before the first '-'
        IEnumerable<char> stringQuery2 = aString.TakeWhile(c => c != '-');

        // Execute the second query
        foreach (char c in stringQuery2)
            Console.Write(c);

        Console.WriteLine(System.Environment.NewLine + "Press any key to exit");
        Console.ReadKey();
    }
}

/* Output:
Output: 9 9 7 4 1 2 8 9
Count = 8
ABCDE99F
*/
```

## 코드 컴파일

`System.Linq` 및 `System.IO` 네임스페이스에 대한 `using` 지시문을 통해 C# 콘솔 애플리케이션 프로젝트를 만듭니다.

## 참조

- LINQ 및 문자열(C#)
- LINQ 쿼리와 정규식 결합 방법(C#)

# LINQ 쿼리와 정규식 결합 방법(C#)

2020-11-02 • 3 minutes to read • [Edit Online](#)

이 예제에서는 [Regex](#) 클래스를 사용하여 더 복잡한 텍스트 문자열 일치를 찾는 정규식을 작성하는 방법을 보여 줍니다. LINQ 쿼리를 사용하면 쉽게 정규식을 통해 검색하려는 파일을 정확히 필터링하고 결과를 구성할 수 있습니다.

## 예제

```
class QueryWithRegEx
{
    public static void Main()
    {
        // Modify this path as necessary so that it accesses your version of Visual Studio.
        string startFolder = @"C:\Program Files (x86)\Microsoft Visual Studio 14.0\";
        // One of the following paths may be more appropriate on your computer.
        //string startFolder = @"C:\Program Files (x86)\Microsoft Visual Studio\2017\";

        // Take a snapshot of the file system.
        IEnumerable<System.IO.FileInfo> fileList = GetFiles(startFolder);

        // Create the regular expression to find all things "Visual".
        System.Text.RegularExpressions.Regex searchTerm =
            new System.Text.RegularExpressions.Regex(@"Visual (Basic|C#|C\+\+|Studio)");

        // 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 == ".htm"
            let fileText = System.IO.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 \"{0}\" was found in:", searchTerm.ToString());

        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);
            }
        }
    }
}
```

```

        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // This method assumes that the application has discovery
    // permissions for all folders under the specified path.
    static IEnumerable<System.IO.FileInfo> GetFiles(string path)
    {
        if (!System.IO.Directory.Exists(path))
            throw new System.IO.DirectoryNotFoundException();

        string[] fileNames = null;
        List<System.IO.FileInfo> files = new List<System.IO.FileInfo>();

        fileNames = System.IO.Directory.GetFiles(path, "*.*", System.IO.SearchOption.AllDirectories);
        foreach (string name in fileNames)
        {
            files.Add(new System.IO.FileInfo(name));
        }
        return files;
    }
}

```

**RegEx** 검색에서 반환되는 **MatchCollection** 객체를 쿼리할 수도 있습니다. 이 예제에서는 각 일치 항목의 값만 결과로 생성됩니다. 그러나 LINQ를 사용하여 해당 컬렉션에 대한 모든 종류의 필터링, 정렬 및 그룹화를 수행할 수도 있습니다. **MatchCollection**은 제네릭이 아닌 **IEnumerable** 컬렉션이므로 쿼리에 범위 변수의 형식을 명시적으로 기술해야 합니다.

## 코드 컴파일

System.Linq 및 System.IO 네임 스페이스에 대한 **using** 지시문을 통해 C# 콘솔 애플리케이션 프로젝트를 만듭니다.

## 참조

- [LINQ 및 문자열\(C#\)](#)
- [LINQ 및 파일 딕터리\(C#\)](#)

# 두 목록 간의 차집합을 구하는 방법(LINQ)(C#)

2020-11-02 • 2 minutes to read • [Edit Online](#)

이 예제에서는 LINQ를 사용하여 두 개의 문자열 목록을 비교하고 names1.txt에 있지만 names2.txt에는 없는 줄만 출력하는 방법을 보여 줍니다.

데이터 파일을 만들려면

1. [문자열 컬렉션을 결합 및 비교 방법\(LINQ\)\(C#\)](#)에 표시된 대로 names1.txt 및 names2.txt를 솔루션 폴더에 복사합니다.

## 예제

```
class CompareLists
{
    static void Main()
    {
        // Create the IEnumerable data sources.
        string[] names1 = System.IO.File.ReadAllLines(@"../../names1.txt");
        string[] names2 = System.IO.File.ReadAllLines(@"../../names2.txt");

        // Create the query. Note that method syntax must be used here.
        IEnumerable<string> 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);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
/* 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, Concat 등 C#에서 일부 유형의 쿼리 작업은 메서드 기반 구문으로만 표현할 수 있습니다.

## 코드 컴파일

System.Linq 및 System.IO 네임스페이스에 대한 `using` 지시문을 통해 C# 콘솔 애플리케이션 프로젝트를 만듭니다.

## 참조

- [LINQ 및 문자열\(C#\)](#)

# 단어 또는 필드에 따라 텍스트 데이터를 정렬하거나 필터링하는 방법(LINQ)(C#)

2020-11-02 • 2 minutes to read • [Edit Online](#)

다음 예제에서는 줄의 필드를 기준으로 쉼표로 구분된 값 등의 구조적 텍스트 줄을 정렬하는 방법을 보여 줍니다. 필드가 런타임에 동적으로 지정될 수도 있습니다. scores.csv의 필드가 학생의 ID 번호와 일련의 시험 성적 4개를 나타낸다고 가정합니다.

데이터가 포함된 파일을 만들려면

1. [서로 다른 파일의 콘텐츠를 조인하는 방법\(LINQ\)\(C#\)](#) 항목에서 scores.csv 데이터를 복사하여 솔루션 폴더에 저장합니다.

## 예제

```

public class SortLines
{
    static void Main()
    {
        // Create an IEnumerable data source
        string[] scores = System.IO.File.ReadAllLines(@"../../../../scores.csv");

        // Change this to any value from 0 to 4.
        int sortField = 1;

        Console.WriteLine("Sorted highest to lowest by field [{0}]:", sortField);

        // Demonstrates how to return query from a method.
        // The query is executed here.
        foreach (string str in RunQuery(scores, sortField))
        {
            Console.WriteLine(str);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Returns the query variable, not query results!
    static IEnumerable<string> RunQuery(IEnumerable<string> source, int num)
    {
        // Split the string and sort on field[num]
        var scoreQuery = from line in source
                         let fields = line.Split(',')
                         orderby fields[num] descending
                         select line;

        return scoreQuery;
    }
}

/* 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
*/

```

이 예제에서는 메서드에서 쿼리 변수를 반환하는 방법도 보여 줍니다.

## 코드 컴파일

System.Linq 및 System.IO 네임스페이스에 대한 `using` 지시문을 통해 C# 콘솔 애플리케이션 프로젝트를 만듭니다.

## 참조

- [LINQ 및 문자열\(C#\)](#)

# 구분된 파일의 필드를 다시 정렬하는 방법(LINQ) (C#)

2020-11-02 • 3 minutes to read • [Edit Online](#)

쉼표로 구분된 값(CSV) 파일은 스프레드시트 데이터 또는 행과 열로 표현되는 다른 테이블 형식 데이터를 저장하는 데 자주 사용되는 텍스트 파일입니다. [Split](#) 메서드를 사용하여 필드를 구분하면 LINQ를 사용하여 쉽게 CSV 파일을 쿼리하고 조작할 수 있습니다. 실제로 동일한 방법을 사용하여 모든 구조적 텍스트 줄의 일부를 다시 정렬할 수 있습니다. CSV 파일로 제한되지 않습니다.

다음 예제에서는 세 개의 열이 학생의 "last name", "first name" 및 "ID"를 나타낸다고 가정합니다. 필드는 학생의 성을 기준으로 알파벳 순서로 나열됩니다. 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 저장되고 원래 데이터가 수정되지 않습니다.

데이터 파일을 만들려면

1. spreadsheet1.csv라는 일반 텍스트 파일에 다음 줄을 복사합니다. 프로젝트 폴더에 파일을 저장합니다.

```
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
```

예제

```

class CSVFiles
{
    static void Main(string[] args)
    {
        // Create the IEnumerable data source
        string[] lines = System.IO.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 x = line.Split(',')
            orderby x[2]
            select x[2] + ", " + (x[1] + " " + x[0]);

        // Execute the query and write out the new file. Note that WriteAllLines
        // takes a string[], so ToArray is called on the query.
        System.IO.File.WriteAllLines(@"../../../../spreadsheet2.csv", query.ToArray());

        Console.WriteLine("Spreadsheet2.csv written to disk. Press any key to exit");
        Console.ReadKey();
    }
}

/* 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
*/

```

## 코드 컴파일

System.Linq 및 System.IO 네임스페이스에 대한 `using` 지시문을 통해 C# 콘솔 애플리케이션 프로젝트를 만듭니다.

## 참조

- [LINQ 및 문자열\(C#\)](#)
- [LINQ 및 파일 딕렉터리\(C#\)](#)
- [CSV 파일에서 XML을 생성하는 방법\(C#\)](#)

# 문자열 컬렉션의 결합 및 비교 방법(LINQ)(C#)

2020-11-02 • 3 minutes to read • [Edit Online](#)

이 예제에서는 텍스트 줄이 포함된 파일을 병합하고 결과를 정렬하는 방법을 보여 줍니다. 특히, 두 개의 텍스트 줄 집합에 대한 단순 연결, 합집합 및 교집합을 수행하는 방법을 보여 줍니다.

프로젝트 및 텍스트 파일을 설정하려면

1. 이러한 이름을 names1.txt 텍스트 파일에 복사하고 파일을 프로젝트 폴더에 저장합니다.

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

2. 이러한 이름을 names2.txt 텍스트 파일에 복사하고 파일을 프로젝트 폴더에 저장합니다. 두 파일의 일부 이름에는 공통점이 있습니다.

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkengne
El Yassir, Mehdi
```

## 예제

```
class MergeStrings
{
    static void Main(string[] args)
    {
        //Put text files in your solution folder
        string[] fileA = System.IO.File.ReadAllLines(@"../../../../names1.txt");
        string[] fileB = System.IO.File.ReadAllLines(@"../../../../names2.txt");

        //Simple concatenation and sort. Duplicates are preserved.
        IEnumerable<string> 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.
        IEnumerable<string> 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).
IEnumerable<string> 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";

IEnumerable<String> tempQuery1 =
    from name in fileA
    let n = name.Split(',')
    where n[0] == nameMatch
    select name;

IEnumerable<string> tempQuery2 =
    from name2 in fileB
    let n2 = name2.Split(',')
    where n2[0] == nameMatch
    select name2;

IEnumerable<string> nameMatchQuery =
    tempQuery1.Concat(tempQuery2).OrderBy(s => s);
OutputQueryResults(nameMatchQuery, $"Concat based on partial name match \'{nameMatch}\':");

// Keep the console window open in debug mode.
Console.WriteLine("Press any key to exit");
Console.ReadKey();
}

static void OutputQueryResults(IEnumerable<string> query, string message)
{
    Console.WriteLine(System.Environment.NewLine + message);
    foreach (string item in query)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("{0} total names in list", query.Count());
}
}

/* 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, Nkenge
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, Nkenge
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
```

```
*/
```

## 코드 컴파일

System.Linq 및 System.IO 네임스페이스에 대한 `using` 지시문을 통해 C# 콘솔 애플리케이션 프로젝트를 만듭니다.

## 참조

- [LINQ 및 문자열\(C#\)](#)
- [LINQ 및 파일 딕렉터리\(C#\)](#)

# 여러 소스로 개체 컬렉션을 채우는 방법(LINQ) (C#)

2020-11-02 • 7 minutes to read • [Edit Online](#)

이 예제에서는 여러 소스의 데이터를 새 형식의 시퀀스에 병합하는 방법을 보여 줍니다.

## NOTE

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. 데이터베이스의 데이터를 메모리 내 데이터에 조인하려면 먼저 데이터베이스 쿼리에서 `ToList` 또는 `ToArray`를 호출한 다음 반환된 컬렉션에서 조인을 수행합니다.

## 데이터 파일을 만들려면

[서로 다른 파일의 콘텐츠를 조인하는 방법\(LINQ\)\(C#\)](#)에 설명된 대로 `names.csv` 및 `scores.csv` 파일을 해당 프로젝트 폴더에 복사합니다.

## 예제

다음 예제에서는 명명된 형식 `Student`를 사용하여 스프레드시트 데이터를 시뮬레이트하는 두 개의 메모리 내 문자열 컬렉션의 병합된 데이터를 `.csv` 형식으로 저장하는 방법을 보여 줍니다. 첫 번째 문자열 컬렉션은 학생 이름과 ID를 나타내고, 두 번째 컬렉션은 학생 ID(첫 번째 열)와 4개의 시험 점수를 나타냅니다. ID는 외래 키로 사용됩니다.

```
using System;
using System.Collections.Generic;
using System.Linq;

class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int ID { get; set; }
    public List<int> ExamScores { get; set; }
}

class PopulateCollection
{
    static void Main()
    {
        // These data files are defined in How to join content from
        // dissimilar files (LINQ).

        // 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 = System.IO.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 = System.IO.File.ReadAllLines(@"../../scores.csv");

        // Merge the data sources using a named type.
        // var could be used instead of an explicit type. Note the dynamic
    }
}
```

```

// This could be done instead by an explicit query like this:
// 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(',')
    where Convert.ToInt32(splitName[2]) == 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)).
                      ToList()
    };
}

// 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 {0} {1} is {2}.",
        student.FirstName, student.LastName,
        student.ExamScores.Average());
}

//Keep console window open in debug mode
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}

/*
 * 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 개체 이니셜라이저는 두 소스의 데이터를 사용하여 새 `Student` 개체를 각각 인스턴스화하는데 사용됩니다.

쿼리 결과를 저장할 필요가 없는 경우 무명 형식이 명명된 형식보다 더 편리할 수 있습니다. 명명된 형식은 쿼리가 실행되는 메서드 외부로 쿼리 결과를 전달하는 경우에 필요합니다. 다음 예제는 앞의 예제와 동일한 작업을 실행하지만 명명된 형식 대신 무명 형식을 사용합니다.

```

// 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 new
    {
        First = splitName[0],
        Last = 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 {0} {1} is {2}.",
                      student.First, student.Last, student.ExamScores.Average());
}

```

## 참조

- [LINQ 및 문자열\(C#\)](#)
- [개체 이니셜라이저 및 컬렉션 이니셜라이저](#)
- [익명 형식](#)

# 그룹을 사용하여 파일을 여러 파일로 분할하는 방법(LINQ)(C#)

2020-11-02 • 3 minutes to read • [Edit Online](#)

이 예제에서는 두 파일의 내용을 병합한 다음 새로운 방식으로 데이터를 구성하는 새 파일 집합을 만드는 한 가지 방법을 보여 줍니다.

데이터 파일을 만들려면

1. 이러한 이름을 names1.txt 텍스트 파일에 복사하고 파일을 프로젝트 폴더에 저장합니다.

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

2. 이러한 이름을 names2.txt 텍스트 파일에 복사하고 파일을 프로젝트 폴더에 저장합니다. 두 파일에서 일부 이름은 공통됩니다.

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

## 예제

```
class SplitWithGroups
{
    static void Main()
    {
        string[] fileA = System.IO.File.ReadAllLines(@"../../../../names1.txt");
        string[] fileB = System.IO.File.ReadAllLines(@"../../../../names2.txt");

        // Concatenate and remove duplicate names based on
        // default string comparer
        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(',')
                         group name by n[0][0] into g
                         orderby g.Key
                         select g;
```

```

// Create a new file for each group that was created
// Note that nested foreach loops are required to access
// individual items with each group.
foreach (var g in groupQuery)
{
    // Create the new file name.
    string fileName = @"../../testFile_" + g.Key + ".txt";

    // Output to display.
    Console.WriteLine(g.Key);

    // Write file.
    using (System.IO.StreamWriter sw = new System.IO.StreamWriter(fileName))
    {
        foreach (var item in g)
        {
            sw.WriteLine(item);
            // Output to console for example purposes.
            Console.WriteLine("    {0}", item);
        }
    }
}

// Keep console window open in debug mode.
Console.WriteLine("Files have been written. Press any key to exit");
Console.ReadKey();
}

/* 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
*/

```

프로그램에서 데이터 파일과 동일한 폴더에 각 그룹에 대한 별도 파일을 작성합니다.

## 코드 컴파일

System.Linq 및 System.IO 네임스페이스에 대한 `using` 지시문을 통해 C# 콘솔 애플리케이션 프로젝트를 만듭니다.

## 참조

- LINQ 및 문자열(C#)
- LINQ 및 파일 딕렉터리(C#)

# 서로 다른 파일의 콘텐츠를 조인하는 방법(LINQ) (C#)

2020-11-02 • 4 minutes to read • [Edit Online](#)

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.

## 데이터 파일을 만들려면

1. 다음 줄을 *scores.csv* 파일에 복사하고 파일을 프로젝트 폴더에 저장합니다. 파일은 스프레드시트 데이터를 나타냅니다. 열 1은 학생 ID이고, 열 2-5는 시험 점수입니다.

```
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
```

2. 다음 줄을 *names.csv* 파일에 복사하고 파일을 프로젝트 폴더에 저장합니다. 파일은 학생의 성, 이름 및 학생 ID를 포함하는 스프레드시트를 나타냅니다.

```
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
```

## 예제

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
  
class JoinStrings  
{  
    static void Main()  
    {  
        // Join content from dissimilar files that contain
```

```

// Join content from dissimilar files that contain
// related information. File names.csv contains the student
// name plus an ID number. File scores.csv contains the ID
// and a set of four test scores. The following query joins
// the scores to the student names by using ID as a
// matching key.

string[] names = System.IO.File.ReadAllLines(@"../../names.csv");
string[] scores = System.IO.File.ReadAllLines(@"../../scores.csv");

// Name:    Last[0],      First[1],  ID[2]
//          Omelchenko,   Svetlana,  11
// Score:   StudentID[0], Exam1[1]  Exam2[2],  Exam3[3],  Exam4[4]
//          111,           97,        92,       81,       60

// This query joins two dissimilar spreadsheets based on common ID value.
// Multiple from clauses are used instead of a join clause
// in order to store results of id.Split.
IEnumerable<string> scoreQuery1 =
    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];

// Pass a query variable to a method and execute it
// in the method. The query itself is unchanged.
OutputQueryResults(scoreQuery1, "Merge two spreadsheets:");

// Keep console window open in debug mode.
Console.WriteLine("Press any key to exit");
Console.ReadKey();
}

static void OutputQueryResults(IEnumerable<string> query, string message)
{
    Console.WriteLine(System.Environment.NewLine + message);
    foreach (string item in query)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("{0} total names in list", query.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
*/

```

## 참고 항목

- [LINQ 및 문자열\(C#\)](#)
- [LINQ 및 파일 딕렉터리\(C#\)](#)



# CSV 텍스트 파일의 열 값을 컴퓨팅하는 방법 (LINQ)(C#)

2020-11-02 • 5 minutes to read • [Edit Online](#)

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.

## 소스 파일을 만들려면

1. 다음 줄을 scores.csv 파일에 복사하고 파일을 프로젝트 폴더에 저장합니다. 첫 번째 열은 학생 ID를 나타내고 후속 열은 4개 시험의 점수를 나타낸다고 가정합니다.

```
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
```

## 예제

```
class SumColumns
{
    static void Main(string[] args)
    {
        string[] lines = System.IO.File.ReadAllLines(@"../../scores.csv");

        // 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);

        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    static void SingleColumn(IEnumerable<string> strs, int examNum)
    {
        Console.WriteLine("Single Column Query:");

        // Parameter examNum specifies the column to
        // run the calculations on. This value could be
```

```

// passed in dynamically at runtime.

// Variable columnQuery is an IEnumerable<int>.
// The following query performs two steps:
// 1) use Split to break each row (a string) into an array
//    of strings,
// 2) convert the element at position examNum to an int
//    and select it.
var columnQuery =
    from line in strs
    let elements = line.Split(',')
    select Convert.ToInt32(elements[examNum]);

// Execute the query and cache the results to improve
// performance. This is helpful only with very large files.
var results = columnQuery.ToList();

// Perform aggregate calculations Average, Max, and
// Min on the column specified by examNum.
double average = results.Average();
int max = results.Max();
int min = results.Min();

Console.WriteLine("Exam #{0}: Average:{1:#.#} High Score:{2} Low Score:{3}",
    examNum, average, max, min);
}

static void MultiColumns(IEnumerable<string> strs)
{
    Console.WriteLine("Multi Column Query:");

    // Create a query, multiColQuery. Explicit typing is used
    // to make clear that, when executed, multiColQuery produces
    // nested sequences. However, you get the same results by
    // using 'var'.

    // The multiColQuery query performs the following steps:
    // 1) use Split to break each row (a string) into an array
    //    of strings,
    // 2) use Skip to skip the "Student ID" column, and store the
    //    rest of the row in scores.
    // 3) convert each score in the current row from a string to
    //    an int, and select that entire sequence as one row
    //    in the results.
    IEnumerable<IEnumerable<int>> multiColQuery =
        from line in strs
        let elements = line.Split(',')
        let scores = elements.Skip(1)
        select (from str in scores
                select Convert.ToInt32(str));

    // Execute the query and cache the results to improve
    // performance.
    // ToArray could be used instead ofToList.
    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 #{0} Average: {1:##.##} High Score: {2} Low Score: {3}",
                           column + 1, average, max, 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
*/

```

쿼리는 [Split](#) 메서드를 사용하여 텍스트의 각 줄을 배열로 변환하는 방식으로 작동합니다. 각 배열 요소는 열을 나타냅니다. 마지막으로 각 열의 텍스트가 숫자 표현으로 변환됩니다. 탭으로 구분된 파일인 경우 [Split](#) 메서드의 인수를 `\t`로 업데이트하세요.

## 코드 컴파일

`System.Linq` 및 `System.IO` 네임스페이스에 대한 `using` 지시문을 통해 C# 콘솔 애플리케이션 프로젝트를 만듭니다.

## 참조

- [LINQ 및 문자열\(C#\)](#)
- [LINQ 및 파일 딕렉터리\(C#\)](#)

# 리플렉션을 사용하여 어셈블리의 메타데이터를 쿼리하는 방법(LINQ)(C#)

2020-11-02 • 3 minutes to read • [Edit Online](#)

.NET 리플렉션 API는 .NET 어셈블리에서 메타데이터를 검사하고 해당 어셈블리에 없는 형식, 형식 멤버, 매개 변수 등의 컬렉션을 만드는데 사용할 수 있습니다. 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 열거 가능한 형식을 반환하는 모든 메서드의 이름을 어셈블리에서 검색합니다.

## 예제

```
using System;
using System.Linq;
using System.Reflection;

class ReflectionHowTO
{
    static void Main()
    {
        Assembly assembly = Assembly.Load("System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089");
        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);
            }
        }

        Console.WriteLine("Press any key to exit... ");
        Console.ReadKey();
    }
}
```

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.

## 참조

- LINQ to Objects(C#)

# 리플렉션을 사용하여 어셈블리의 메타데이터를 쿼리하는 방법(LINQ)(C#)

2020-11-02 • 3 minutes to read • [Edit Online](#)

.NET 리플렉션 API는 .NET 어셈블리에서 메타데이터를 검사하고 해당 어셈블리에 없는 형식, 형식 멤버, 매개 변수 등의 컬렉션을 만드는데 사용할 수 있습니다. 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 열거 가능한 형식을 반환하는 모든 메서드의 이름을 어셈블리에서 검색합니다.

## 예제

```
using System;
using System.Linq;
using System.Reflection;

class ReflectionHowTO
{
    static void Main()
    {
        Assembly assembly = Assembly.Load("System.Core, Version=3.5.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089");
        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);
            }
        }

        Console.WriteLine("Press any key to exit... ");
        Console.ReadKey();
    }
}
```

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.

## 참조

- LINQ to Objects(C#)

# LINQ 및 파일 딕렉터리(C#)

2020-11-02 • 7 minutes to read • [Edit Online](#)

많은 파일 시스템 작업은 기본적으로 쿼리이므로 LINQ 접근 방식에 적합합니다.

이 섹션에서 쿼리는 비파괴적입니다. 쿼리는 원본 파일이나 폴더의 내용을 변경하는 데 사용되지 않으며, 쿼리로 인해 의도하지 않은 결과가 발생하지 않아야 한다는 규칙을 따릅니다. 일반적으로 소스 데이터를 수정하는 모든 코드(만들기/업데이트/삭제 작업을 수행하는 쿼리 포함)는 데이터를 쿼리만하는 코드와 별도로 유지되어야 합니다.

이 단원에는 다음 항목이 포함되어 있습니다.

## 지정된 특성 또는 이름을 갖는 파일을 쿼리하는 방법(C#)

하나 이상의 [FileInfo](#) 개체 속성을 검사하여 파일을 검색하는 방법을 보여 줍니다.

## 확장명에 따라 파일을 그룹화하는 방법(LINQ)(C#)

파일 이름 확장명에 따라 [FileInfo](#) 개체의 그룹을 반환하는 방법을 보여 줍니다.

## 폴더 집합의 전체 바이트 수를 쿼리하는 방법(LINQ)(C#)

지정된 딕렉터리 트리에 있는 모든 파일에서 전체 바이트 수를 반환하는 방법을 보여 줍니다.

## 두 폴더의 내용을 비교하는 방법(LINQ)(C#)

두 개의 지정된 폴더에 있는 모든 파일뿐만 아니라 특정 폴더에만 있는 모든 파일도 반환하는 방법을 보여 줍니다.

## 디렉터리 트리에서 가장 큰 파일을 하나 이상 쿼리하는 방법(LINQ)(C#)

디렉터리 트리에서 가장 크거나 가장 작은 파일 또는 지정한 파일 수를 반환하는 방법을 보여 줍니다.

## 디렉터리 트리의 중복 파일을 쿼리하는 방법(LINQ)(C#)

지정된 딕렉터리 트리에서 둘 이상의 위치에 나타나는 모든 파일 이름을 그룹화하는 방법을 보여 줍니다. 또한 사용자 지정 비교자에 따라 보다 복잡한 비교를 수행하는 방법을 보여 줍니다.

## 폴더의 파일 내용을 쿼리하는 방법(LINQ)(C#)

트리의 폴더를 반복하고, 각 파일을 열고, 파일의 내용을 쿼리하는 방법을 보여 줍니다.

## 주석

파일 시스템의 내용을 정확하게 나타내고 예외를 정상적으로 처리하는 데이터 소스 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](#) 개체에서 파일 시스템에 액세스하여 `Length`의 값을 업데이트합니다. 파일이 더 이상 존재하지 않는 경우 파일 시스템을 직접 쿼리하지 않아도 쿼리에서 [FileNotFoundException](#)을 가져옵니다. 이 섹션의 일부 쿼리는 특정한 경우에 이러한 특정 예외를 사용하는 별도의 메서드를 사용합니다. 또 다른 옵션은 [FileSystemWatcher](#)를 사용하여 데이터 소스가 동적으로 업데이트되도록 하는 것입니다.

## 참조

- [LINQ to Objects\(C#\)](#)

# 지정된 특성 또는 이름을 갖는 파일을 쿼리하는 방법(C#)

2020-11-02 • 2 minutes to read • [Edit Online](#)

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.

## 예제

```
class FindFileByExtension
{
    // This query will produce the full path for all .txt files
    // under the specified folder including subfolders.
    // It orders the list according to the file name.
    static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        //Create the query
        IEnumerable<System.IO.FileInfo> fileQuery =
            from file in fileList
            where file.Extension == ".txt"
            orderby file.Name
            select file;

        //Execute the query. This might write out a lot of files!
        foreach (System.IO.FileInfo fi in fileQuery)
        {
            Console.WriteLine(fi.FullName);
        }

        // Create and execute a new query by using the previous
        // query as a starting point. fileQuery is not
        // executed again until the call to Last()
        var newestFile =
            (from file in fileQuery
            orderby file.CreationTime
            select new { file.FullName, file.CreationTime })
            .Last();

        Console.WriteLine("\r\nThe newest .txt file is {0}. Creation time: {1}",
            newestFile.FullName, newestFile.CreationTime);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

## 코드 컴파일

System.Linq 및 System.IO 네임스페이스에 대한 `using` 지시문을 통해 C# 콘솔 애플리케이션 프로젝트를 만듭니다.

## 참조

- [LINQ to Objects\(C#\)](#)
- [LINQ 및 파일 디렉터리\(C#\)](#)

# 확장명에 따라 파일을 그룹화하는 방법(LINQ)(C#)

2020-11-02 • 5 minutes to read • [Edit Online](#)

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 페이징하는 방법을 보여 줍니다.

## 예제

다음 쿼리는 지정된 디렉터리 트리의 내용을 파일 이름 확장명으로 그룹화하는 방법을 보여 줍니다.

```
class GroupByExtension
{
    // This query will sort all the files under the specified folder
    // and subfolder into groups keyed by the file extension.
    private static void Main()
    {
        // Take a snapshot of the file system.
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\Common7";

        // Used in WriteLine to trim output lines.
        int trimLength = startFolder.Length;

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles(".*",
System.IO.SearchOption.AllDirectories);

        // Create the query.
        var queryGroupByExt =
            from file in fileList
            group file by file.Extension.ToLower() into fileGroup
            orderby fileGroup.Key
            select fileGroup;

        // Display one group at a time. If the number of
        // entries is greater than the number of lines
        // in the console window, then page the output.
        PageOutput(trimLength, queryGroupByExt);
    }

    // This method specifically handles group queries of FileInfo objects with string keys.
    // It can be modified to work for any long listings of data. Note that explicit typing
    // must be used in method signatures. The groupbyExtList parameter is a query that produces
    // groups of FileInfo objects with string keys.
    private static void PageOutput(int rootLength,
                                    IEnumerable<System.Linq.IGrouping<string, System.IO.FileInfo>>
groupByExtList)
    {
        // Flag to break out of paging loop.
        bool goAgain = true;

        // "3" = 1 line for extension + 1 for "Press any key" + 1 for input cursor.
        int numLines = Console.WindowHeight - 3;

        // Iterate through the outer collection of groups.
        foreach (var filegroup in groupByExtList)
        {
            // Start a new extension at the top of a page.
            //
```

```

int currentLine = 0;

// Output only as many lines of the current group as will fit in the window.
do
{
    Console.Clear();
    Console.WriteLine(filegroup.Key == String.Empty ? "[none]" : filegroup.Key);

    // Get 'numLines' number of items starting at number 'currentLine'.
    var resultPage = filegroup.Skip(currentLine).Take(numLines);

    //Execute the resultPage query
    foreach (var f in resultPage)
    {
        Console.WriteLine("\t{0}", f.FullName.Substring(rootLength));
    }

    // Increment the line counter.
    currentLine += numLines;

    // Give the user a chance to escape.
    Console.WriteLine("Press any key to continue or the 'End' key to break...");
    ConsoleKey key = Console.ReadKey().Key;
    if (key == ConsoleKey.End)
    {
        goAgain = false;
        break;
    }
} while (currentLine < filegroup.Count());

if (goAgain == false)
    break;
}
}
}

```

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 보여 줍니다. Windows 및 웹 애플리케이션에 동일한 기법을 적용할 수 있습니다. 코드에서 그룹의 항목을 페이지하기 때문에 중첩된 `foreach` 루프가 필요합니다. 목록에서 현재 위치를 컴퓨팅하며 사용자가 페이지를 중지하고 프로그램을 종료할 수 있도록 하는 몇 가지 추가 논리도 있습니다. 이 특정 사례에서는 페이지 쿼리가 원래 쿼리에서 캐시된 결과에 대해 실행됩니다. LINQ to SQL 등의 다른 컨텍스트에서는 이러한 캐싱이 필요하지 않습니다.

## 코드 컴파일

`System.Linq` 및 `System.IO` 네임스페이스에 대한 `using` 지시문을 통해 C# 콘솔 애플리케이션 프로젝트를 만듭니다.

## 참조

- [LINQ to Objects\(C#\)](#)
- [LINQ 및 파일 딕렉터리\(C#\)](#)

# 폴더 집합의 전체 바이트 수를 쿼리하는 방법 (LINQ)(C#)

2020-11-02 • 5 minutes to read • [Edit Online](#)

이 예제에서는 지정된 폴더 및 모든 하위 폴더의 모든 파일에서 사용된 총 바이트 수를 검색하는 방법을 보여 줍니다.

## 예제

`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 쉽게 수정할 수 있습니다.

```

class QuerySize
{
    public static void Main()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\VC#";

        // Take a snapshot of the file system.
        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<string> fileList = System.IO.Directory.GetFiles(startFolder, "*.*",
System.IO.SearchOption.AllDirectories);

        var fileQuery = from file in fileList
                        select GetFileLength(file);

        // 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 {0} bytes in {1} files under {2}",
            totalBytes, fileList.Count(), startFolder);
        Console.WriteLine("The largest files is {0} bytes.", largestFile);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // This method is used to swallow the possible exception
    // that can be raised when accessing the System.IO.FileInfo.Length property.
    static long GetFileLength(string filename)
    {
        long retval;
        try
        {
            System.IO.FileInfo fi = new System.IO.FileInfo(filename);
            retval = fi.Length;
        }
        catch (System.IO.FileNotFoundException)
        {
            // If a file is no longer present,
            // just add zero bytes to the total.
            retval = 0;
        }
        return retval;
    }
}

```

지정된 디렉터리 트리의 바이트 수만 계산하면 되는 경우 데이터 소스로 목록 컬렉션을 만드는 오버헤드를 유발하는 LINQ 쿼리를 만들지 않고 보다 효율적으로 이 작업을 수행할 수 있습니다. LINQ 방식의 유용성은 쿼리가 더 복잡함에 따라 또는 동일한 데이터 소스에 대해 여러 쿼리를 실행해야 하는 경우에 증가합니다.

쿼리는 파일 길이를 가져오기 위해 별도 메서드를 호출합니다. `GetFiles` 호출에서 `FileInfo` 개체가 생성된 후 파일이 다른 스레드에서 삭제된 경우 발생할 수 있는 예외를 처리하기 위해 이 작업을 수행합니다. `FileInfo` 개체가 이미 생성된 경우에도 `FileInfo` 개체는 속성에 처음 액세스할 때 최신 길이를 사용하여 해당 `Length` 속성의 새로 고침을 시도하기 때문에 예외가 발생할 수 있습니다. 코드는 이 작업을 쿼리 외부의 try-catch 블록에 배치하여, 부작용을 일으킬 수 있는 작업을 쿼리에서 방지하는 규칙을 따릅니다. 일반적으로, 예외를 처리할 때는 애플리케이션이 알 수 없는 상태로 남지 않도록 주의해야 합니다.

## 코드 컴파일

System.Linq 및 System.IO 네임스페이스에 대한 `using` 지시문을 통해 C# 콘솔 애플리케이션 프로젝트를 만듭니다.

## 참조

- [LINQ to Objects\(C#\)](#)
- [LINQ 및 파일 딕렉터리\(C#\)](#)

# 두 폴더의 내용을 비교하는 방법(LINQ)(C#)

2020-11-02 • 4 minutes to read • [Edit Online](#)

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.

## NOTE

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 사용되지 않습니다. 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.

## 예제

```
namespace QueryCompareTwoDirs
{
    class CompareDirs
    {

        static void Main(string[] args)
        {

            // Create two identical or different temporary folders
            // on a local drive and change these file paths.
            string pathA = @"C:\TestDir";
            string pathB = @"C:\TestDir2";

            System.IO.DirectoryInfo dir1 = new System.IO.DirectoryInfo(pathA);
            System.IO.DirectoryInfo dir2 = new System.IO.DirectoryInfo(pathB);

            // Take a snapshot of the file system.
            IEnumerable<System.IO.FileInfo> list1 = dir1.GetFiles(".*",
System.IO.SearchOption.AllDirectories);
            IEnumerable<System.IO.FileInfo> list2 = dir2.GetFiles(".*",
System.IO.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:");
        foreach (var v in queryCommonFiles)
        {
            Console.WriteLine(v.FullName); //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.
    // For this example we only check one way.
    var queryList1Only = (from file in list1
                           select file).Except(list2, myFileCompare);

    Console.WriteLine("The following files are in list1 but not list2:");
    foreach (var v in queryList1Only)
    {
        Console.WriteLine(v.FullName);
    }

    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}

// 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 : System.Collections.Generic.IEqualityComparer<System.IO.FileInfo>
{
    public FileCompare() { }

    public bool Equals(System.IO.FileInfo f1, System.IO.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(System.IO.FileInfo fi)
    {
        string s = $"{fi.Name}{fi.Length}";
        return s.GetHashCode();
    }
}
}

```

## 코드 컴파일

System.Linq 및 System.IO 네임스페이스에 대한 `using` 지시문을 통해 C# 콘솔 애플리케이션 프로젝트를 만듭니다.

## 참조

- [LINQ to Objects\(C#\)](#)
- [LINQ 및 파일 딕터리\(C#\)](#)

# 디렉터리 트리에서 가장 큰 파일을 하나 이상 쿼리하는 방법(LINQ)(C#)

2020-11-02 • 7 minutes to read • [Edit Online](#)

이 예제에서는 파일 크기(바이트)와 관련된 다섯 개의 쿼리를 보여 줍니다.

- 가장 큰 파일의 크기(바이트)를 검색하는 방법입니다.
- 가장 작은 파일의 크기(바이트)를 검색하는 방법입니다.
- 지정된 루트 폴더 아래의 하나 이상 폴더에서 [FileInfo](#) 객체의 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.

## 예제

다음 예제에서는 파일 크기(바이트)에 따라 파일을 쿼리 및 그룹화하는 방법을 보여 주는 5개의 개별 쿼리가 포함되어 있습니다. 쿼리가 [FileInfo](#) 객체의 다른 일부 속성을 기반으로 하도록 이러한 예제를 쉽게 수정할 수 있습니다.

```
class QueryBySize
{
    static void Main(string[] args)
    {
        QueryFilesBySize();
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    private static void QueryFilesBySize()
    {
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
        System.IO.SearchOption.AllDirectories);

        //Return the size of the largest file
        long maxSize =
            (from file in fileList
            let len = GetFileLength(file)
            select len)
            .Max();

        Console.WriteLine("The length of the largest file under {0} is {1}",
            startFolder, maxSize);

        // Return the FileInfo object for the largest file
        // by sorting and selecting from beginning of list
        System.IO.FileInfo longestFile =
    }
```

```

(from file in fileList
let len = GetFileLength(file)
where len > 0
orderby len descending
select file)
.First();

Console.WriteLine("The largest file under {0} is {1} with a length of {2} bytes",
                  startFolder, longestFile.FullName, longestFile.Length);

//Return the FileInfo of the smallest file
System.IO.FileInfo smallestFile =
(from file in fileList
let len = GetFileLength(file)
where len > 0
orderby len ascending
select file).First();

Console.WriteLine("The smallest file under {0} is {1} with a length of {2} bytes",
                  startFolder, smallestFile.FullName, smallestFile.Length);

//Return the FileInfos for the 10 largest files
// queryTenLargest is an IEnumerable<System.IO.FileInfo>
var queryTenLargest =
(from file in fileList
let len = GetFileLength(file)
orderby len descending
select file).Take(10);

Console.WriteLine("The 10 largest files under {0} are:", startFolder);

foreach (var v in queryTenLargest)
{
    Console.WriteLine("{0}: {1} bytes", v.FullName, v.Length);
}

// Group the files according to their size, leaving out
// files that are less than 200000 bytes.
var querySizeGroups =
    from file in fileList
    let len = GetFileLength(file)
    where len > 0
    group file by (len / 100000) into fileGroup
    where fileGroup.Key >= 2
    orderby fileGroup.Key descending
    select fileGroup;

foreach (var filegroup in querySizeGroups)
{
    Console.WriteLine(filegroup.Key.ToString() + "0000");
    foreach (var item in filegroup)
    {
        Console.WriteLine("\t{0}: {1}", item.Name, item.Length);
    }
}
}

// This method is used to swallow the possible exception
// that can be raised when accessing the FileInfo.Length property.
// In this particular case, it is safe to swallow the exception.
static long GetFileLength(System.IO.FileInfo fi)
{
    long retval;
    try
    {
        retval = fi.Length;
    }
    catch (System.IO.FileNotFoundException)
    {

```

```
// If a file is no longer present,  
// just add zero bytes to the total.  
retval = 0;  
}  
return retval;  
}  
  
}
```

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.

`GetFiles` 호출에서 [FileInfo](#) 개체가 생성된 이후 기간 내에 파일이 다른 스레드에서 삭제된 경우 발생할 수 있는 예외를 처리하기 위해 쿼리에서 별도 메서드를 호출하여 파일 크기(바이트)를 가져옵니다. [FileInfo](#) 개체가 이미 생성된 경우에도 [FileInfo](#) 개체는 속성에 처음 액세스할 때 최신 크기(바이트)를 사용하여 해당 [Length](#) 속성의 새로 고침을 시도하기 때문에 예외가 발생할 수 있습니다. 이 작업을 쿼리 외부의 try-catch 블록에 배치하여, 부작용을 일으킬 수 있는 작업을 쿼리에서 방지하는 규칙을 따릅니다. 일반적으로, 예외를 처리할 때는 애플리케이션이 알 수 없는 상태로 남지 않도록 주의해야 합니다.

## 코드 컴파일

System.Linq 및 System.IO 네임스페이스에 대한 `using` 지시문을 통해 C# 콘솔 애플리케이션 프로젝트를 만듭니다.

## 참조

- [LINQ to Objects\(C#\)](#)
- [LINQ 및 파일 딕렉터리\(C#\)](#)

# 디렉터리 트리의 중복 파일을 쿼리하는 방법 (LINQ)(C#)

2020-11-02 • 5 minutes to read • [Edit Online](#)

동일한 이름을 가진 파일이 둘 이상의 폴더에 있는 경우도 있습니다. 예를 들어 Visual Studio 설치 폴더 아래의 여러 폴더에 readme.htm 파일이 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.

## 예제

```
class QueryDuplicateFileNames
{
    static void Main(string[] args)
    {
        // Uncomment QueryDuplicates2 to run that query.
        QueryDuplicates();
        // QueryDuplicates2();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void QueryDuplicates()
    {
        // Change the root drive or folder if necessary
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
        System.IO.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;

        // Pass the query to a method that will
        // output one page at a time.
        PageOutput<string, string>(queryDupNames);
    }

    // A Group key that can be passed to a separate method.
    // Override Equals and GetHashCode to define equality for the key.
    // Override ToString to provide a friendly name for Key.ToString()
    class PortableKey
    {
        public string Name { get; set; }
        public DateTime LastWriteTime { get; set; }
    }
}
```

```

public long Length { get; set; }

public override bool Equals(object obj)
{
    PortableKey other = (PortableKey)obj;
    return other.LastWriteTime == this.LastWriteTime &&
           other.Length == this.Length &&
           other.Name == this.Name;
}

public override int GetHashCode()
{
    string str = $"{this.LastWriteTime}{this.Length}{this.Name}";
    return str.GetHashCode();
}

public override string ToString()
{
    return $"{this.Name} {this.Length} {this.LastWriteTime}";
}

static void QueryDuplicates2()
{
    // Change the root drive or folder if necessary.
    string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\Common7";

    // Make the lines shorter for the console display
    int charsToSkip = startFolder.Length;

    // Take a snapshot of the file system.
    System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);
    IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
        System.IO.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
            new PortableKey { Name = file.Name, LastWriteTime = file.LastWriteTime, Length = file.Length
    } into fileGroup
        where fileGroup.Count() > 1
        select fileGroup;

    var list = queryDupFiles.ToList();

    int i = queryDupFiles.Count();

    PageOutput<PortableKey, string>(queryDupFiles);
}

// A generic method to page the output of the QueryDuplications methods
// Here the type of the group must be specified explicitly. "var" cannot
// be used in method signatures. This method does not display more than one
// group per page.
private static void PageOutput<K, V>(IEnumerable<System.Linq.IGrouping<K, V>> groupByExtList)
{
    // Flag to break out of paging loop.
    bool goAgain = true;

    // "3" = 1 line for extension + 1 for "Press any key" + 1 for input cursor.
    int numLines = Console.WindowHeight - 3;

    // Iterate through the outer collection of groups.
    foreach (var filegroup in groupByExtList)
    {

```

```

// Start a new extension at the top of a page.
int currentLine = 0;

// Output only as many lines of the current group as will fit in the window.
do
{
    Console.Clear();
    Console.WriteLine("Filename = {0}", filegroup.Key.ToString() == String.Empty ? "[none]" :
filegroup.Key.ToString());

    // Get 'numLines' number of items starting at number 'currentLine'.
    var resultPage = filegroup.Skip(currentLine).Take(numLines);

    //Execute the resultPage query
    foreach (var fileName in resultPage)
    {
        Console.WriteLine("\t{0}", fileName);
    }

    // Increment the line counter.
    currentLine += numLines;

    // Give the user a chance to escape.
    Console.WriteLine("Press any key to continue or the 'End' key to break...");
    ConsoleKey key = Console.ReadKey().Key;
    if (key == ConsoleKey.End)
    {
        goAgain = false;
        break;
    }
} while (currentLine < filegroup.Count());

if (goAgain == false)
    break;
}
}
}

```

첫 번째 쿼리는 단순 키를 사용하여 일치하는 항목을 확인합니다. 이름은 같지만 내용이 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.

## 코드 컴파일

System.Linq 및 System.IO 네임스페이스에 대한 `using` 지시문을 통해 C# 콘솔 애플리케이션 프로젝트를 만듭니다.

## 참조

- [LINQ to Objects\(C#\)](#)
- [LINQ 및 파일 딕렉터리\(C#\)](#)

# 폴더의 텍스트 파일 내용을 쿼리하는 방법(LINQ) (C#)

2020-11-02 • 2 minutes to read • [Edit Online](#)

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. 자세한 내용은 [LINQ 쿼리와 정규식 결합 방법\(C#\)](#)을 참조하세요.

## 예제

```

class QueryContents
{
    public static void Main()
    {
        // Modify this path as necessary.
        string startFolder = @"c:\program files\Microsoft Visual Studio 9.0\";

        // Take a snapshot of the file system.
        System.IO.DirectoryInfo dir = new System.IO.DirectoryInfo(startFolder);

        // This method assumes that the application has discovery permissions
        // for all folders under the specified path.
        IEnumerable<System.IO.FileInfo> fileList = dir.GetFiles("*.*",
System.IO.SearchOption.AllDirectories);

        string searchTerm = @"Visual Studio";

        // Search the contents of each file.
        // A regular expression created with the Regex class
        // could be used instead of the Contains method.
        // queryMatchingFiles is an IEnumerable<string>.
        var queryMatchingFiles =
            from file in fileList
            where file.Extension == ".htm"
            let fileText = GetFileText(file.FullName)
            where fileText.Contains(searchTerm)
            select file.FullName;

        // Execute the query.
        Console.WriteLine("The term \"{0}\" was found in:", searchTerm);
        foreach (string filename in queryMatchingFiles)
        {
            Console.WriteLine(filename);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }

    // Read the contents of the file.
    static string GetFileText(string name)
    {
        string fileContents = String.Empty;

        // If the file has been deleted since we took
        // the snapshot, ignore it and return the empty string.
        if (System.IO.File.Exists(name))
        {
            fileContents = System.IO.File.ReadAllText(name);
        }
        return fileContents;
    }
}

```

## 코드 컴파일

System.Linq 및 System.IO 네임스페이스에 대한 `using` 지시문을 통해 C# 콘솔 애플리케이션 프로젝트를 만듭니다.

## 참조

- [LINQ 및 파일 딕렉터리\(C#\)](#)

- [LINQ to Objects\(C#\)](#)

# LINQ를 사용하여 ArrayList를 쿼리하는 방법(C#)

2020-11-02 • 3 minutes to read • [Edit Online](#)

LINQ를 사용하여 [ArrayList](#) 등의 제네릭이 아닌 [IEnumerable](#) 컬렉션을 쿼리하는 경우 컬렉션에 있는 개체의 특정 형식을 반영하도록 범위 변수의 형식을 명시적으로 선언해야 합니다. 예를 들어 `Student` 개체의 [ArrayList](#)가 있는 경우 `from` 절은 다음과 같아야 합니다.

```
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 없는 경우 [Cast](#)에서 예외를 throw합니다. [Cast](#) 및 [OfType](#)은 제네릭이 아닌 [IEnumerable](#) 형식에서 작동하는 두 가지 표준 쿼리 연산자 메서드입니다. 자세한 내용은 [LINQ 쿼리 작업의 형식 관계](#)를 참조하세요.

## 예제

다음 예제에서는 [ArrayList](#)에 대한 단순 쿼리를 보여 줍니다. 이 예제에서는 코드가 [Add](#) 메서드를 호출할 때 개체 이니셜라이저를 사용하지만 요구 사항은 아닙니다.

```

using System;
using System.Collections;
using System.Linq;

namespace NonGenericLINQ
{
    public class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int[] Scores { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arrList = new ArrayList();
            arrList.Add(
                new Student
                {
                    FirstName = "Svetlana", LastName = "Omelchenko", Scores = new int[] { 98, 92, 81, 60
                });
            arrList.Add(
                new Student
                {
                    FirstName = "Claire", LastName = "O'Donnell", Scores = new int[] { 75, 84, 91, 39 }
                });
            arrList.Add(
                new Student
                {
                    FirstName = "Sven", LastName = "Mortensen", Scores = new int[] { 88, 94, 65, 91 }
                });
            arrList.Add(
                new Student
                {
                    FirstName = "Cesar", LastName = "Garcia", Scores = new int[] { 97, 89, 85, 82 }
                });

            var query = from Student student in arrList
                       where student.Scores[0] > 95
                       select student;

            foreach (Student s in query)
                Console.WriteLine(s.LastName + ": " + s.Scores[0]);

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}

/* Output:
   Omelchenko: 98
   Garcia: 97
*/

```

## 참조

- [LINQ to Objects\(C#\)](#)

# LINQ 쿼리용 사용자 지정 메서드를 추가하는 방법 (C#)

2020-11-02 • 11 minutes to read • [Edit Online](#)

`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 메서드 추가

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.

```
public static class LINQExtension
{
    public static double Median(this IEnumerable<double>? source)
    {
        if (!(source?.Any() ?? false))
        {
            throw new InvalidOperationException("Cannot compute median for a null or empty set.");
        }

        var sortedList = (from number in source
                         orderby number
                         select 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.

```

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 형식을 허용하도록 집계 메서드 오버로드

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.

각 형식에 대한 오버로드를 만들려면

지원하려는 각 형식에 대한 특정 오버로드를 만들 수 있습니다. 다음 코드 예제에서는 `int` 형식에 대한 `Median` 메서드의 오버로드를 보여 줍니다.

```

//int overload
public static double Median(this IEnumerable<int> source) =>
    (from num in source select (double)num).Median();

```

이제 다음 코드와 같이 `integer` 및 `double` 형식에 대한 `Median` 오버로드를 호출할 수 있습니다.

```

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
Integer: Median = 3
*/

```

## 제네릭 오버로드를 만들려면

제네릭 개체의 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.

```

// Generic overload.
public static double Median<T>(this IEnumerable<T> numbers,
                                Func<T, double> selector) =>
    (from num in numbers select selector(num)).Median();

```

이제 임의의 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.

```
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:

 Integer: Median = 3
 String: Median = 4
 */
```

## 시퀀스를 반환하는 메서드 추가

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.

```
// 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 i = 0;
    foreach (var element in source)
    {
        if (i % 2 == 0)
        {
            yield return element;
        }
        i++;
    }
}
```

다음 코드와 같이 `IEnumerable<T>` 인터페이스에서 다른 메서드를 호출할 때와 같은 방법으로 열거 가능한 모든 컬렉션에 대해 이 확장 메서드를 호출할 수 있습니다.

```
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
*/
```

## 참조

- [IEnumerable<T>](#)
- [확장명 메서드](#)

# LINQ to ADO.NET(포털 페이지)

2020-11-02 • 5 minutes to read • [Edit Online](#)

LINQ to ADO.NET을 사용하면 LINQ(Language-Integrated Query) 프로그래밍 모델을 통해 ADO.NET의 열거 가능한 개체를 쿼리할 수 있습니다.

## NOTE

LINQ to ADO.NET 설명서는 .NET Framework SDK의 ADO.NET 섹션에 있습니다. [LINQ 및 ADO.NET](#).

ADO.NET LINQ(Language-Integrated Query) 기술에는 LINQ to DataSet, LINQ to SQL 및 LINQ to Entities의 세 가지 독립적인 기술이 있습니다. LINQ to DataSet은 다양한 기능을 사용하여 [DataSet](#)에 대해 최적화된 쿼리를 제공하고, LINQ to SQL는 SQL Server 데이터베이스 스키마를 직접 쿼리하는 데 사용되며, LINQ to Entities는 엔티티 데이터 모델을 쿼리하는 데 사용됩니다.

## LINQ to DataSet

[DataSet](#)는 ADO.NET에서 가장 널리 사용되는 구성 요소 중 하나이며, ADO.NET의 기반이 되는 연결되지 않은 프로그래밍 모델의 핵심 요소입니다. 그러나 이러한 탁월함에도 불구하고 [DataSet](#)의 쿼리 기능에는 한계가 있습니다.

LINQ to DataSet을 사용하면 다른 많은 데이터 원본에 사용되는 것과 같은 쿼리 기능을 사용하여 더 다양한 쿼리 기능을 [DataSet](#)에 빌드할 수 있습니다.

자세한 내용은 [LINQ to DataSet](#)을 참조하세요.

## LINQ to SQL

LINQ to SQL은 관계형 데이터를 개체로 관리하는 데 필요한 런타임 인프라를 제공합니다. LINQ to SQL에서 관계형 데이터베이스의 데이터 모델은 개발자의 프로그래밍 언어로 표현되는 개체 모델에 매핑됩니다. 애플리케이션을 실행하면 LINQ to SQL에서 개체 모델의 언어 통합 쿼리를 SQL로 변환하여 실행을 위해 데이터베이스로 전송합니다. 데이터베이스가 결과를 반환하면 LINQ to SQL에서 조작할 수 있는 개체로 다시 변환합니다.

LINQ to SQL에는 데이터베이스의 저장 프로시저 및 사용자 정의 함수와 개체 모델의 상속을 지원합니다.

자세한 내용은 [LINQ to SQL](#)을 참조하세요.

## LINQ to Entities

엔티티 데이터 모델을 통해 관계형 데이터는 .NET 환경에서 개체로 공개됩니다. 이를 통해 개체 계층은 LINQ 지원을 위한 이상적인 대상이 되므로 개발자는 비즈니스 논리를 개발할 때 사용한 언어로 데이터베이스에 대한 쿼리를 작성할 수 있습니다. 이 기능은 LINQ to Entities로 알려져 있습니다. 자세한 내용은 [LINQ to Entities](#)를 참조하세요.

## 참조

- [LINQ 및 ADO.NET](#)
- [LINQ\(Language-Integrated Query\)\(C#\)](#)

# LINQ 쿼리에 대한 데이터 소스 활성화

2021-02-18 • 9 minutes to read • [Edit Online](#)

다양한 방법으로 LINQ를 확장하여 LINQ 패턴에서 원하는 데이터 소스를 쿼리할 수 있습니다. 데이터 소스의 예를 몇 가지 들자면 데이터 구조, 웹 서비스, 파일 시스템 또는 데이터베이스가 있습니다. 쿼리의 구문과 패턴은 변경되지 않으므로 LINQ 패턴을 사용하면 LINQ 쿼리가 사용 설정된 데이터 소스를 클라이언트가 쉽게 쿼리할 수 있습니다. 다음은 LINQ를 이러한 데이터 소스로 확장할 수 있는 방법입니다.

- 특정 형식에 이 형식의 LINQ to Objects 쿼리를 활성화하는 `IEnumerable<T>` 인터페이스 구현
- 형식을 확장하는 `Where` 및 `Select` 같은 표준 쿼리 연산자 메서드를 만들어 해당 형식의 사용자 지정 LINQ 쿼리 사용 설정
- `IQueryable<T>` 인터페이스를 구현하는 데이터 소스에 대한 공급자 만들기. 이 인터페이스를 구현하는 공급자는 원격 실행과 같이 사용자 지정 방식으로 실행할 수 있는 식 트리의 형태로 LINQ 쿼리를 받습니다.
- 기존 LINQ 기술을 이용하는 데이터 소스에 대한 공급자 만들기. 이러한 공급자를 사용하면 쿼리뿐만 아니라 사용자 정의 형식에 대한 삽입, 업데이트 및 삭제 작업과 매핑을 수행할 수 있습니다.

이 항목에서는 이러한 옵션에 대해 설명합니다.

## 데이터 소스의 LINQ 쿼리를 활성화하는 방법

### 메모리 내 데이터

메모리 내 데이터의 LINQ 쿼리를 사용 설정하는 방법에는 두 가지가 있습니다. 데이터가 `IEnumerable<T>`을 구현하는 형식이라면 LINQ to Objects를 사용하여 데이터를 쿼리할 수 있습니다. 하지만 `IEnumerable<T>` 인터페이스를 구현하여 형식의 열거형을 사용 설정하는 것이 바람직하지 않다면 해당 형식에서 LINQ 표준 쿼리 연산자 메서드를 정의하거나 해당 형식을 확장하는 LINQ 표준 쿼리 연산자 메서드를 만들 수 있습니다. 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. 하지만 이 방식은 데이터 소스에서 LINQ to SQL 같은 공급자를 확장하는 방식과 다릅니다.

## IQueryable LINQ 공급자

`IQueryable<T>`을 구현하는 LINQ 공급자의 복잡성은 경우에 따라 크게 다릅니다. 이 단원에서는 이러한 복잡성의 차이에 대해 설명합니다.

덜 복잡한 `IQueryable` 공급자는 웹 서비스의 단일 메서드와 인터페이스 할 수도 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 호출 식만 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. 공급자가 웹 서비스를 대상으로 하는 경우 둘 이상의 웹 서비스 메서드에 대한 인터페이스를 제공하고 쿼리가 노출하는 질문에 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 이 공급자는 이동 가능한 일대다 관계가 있는 형식을 노출할 수 있지만 사용자 정의 형식에 대한 매핑 기술을 제공하지 않습니다.

LINQ to SQL 공급자와 같은 복잡한 `IQueryable` 공급자가 전체 LINQ 쿼리를 SQL과 같이 표현이 가능한 쿼리 언어로 변환할 수 있습니다. 복잡한 공급자가 덜 복잡한 공급자보다 더욱 다양하고 방대한 질문을 쿼리로 처리할 수 있기 때문에 더 일반적이라 할 수 있습니다. 또한 개방형 형식 시스템을 가지므로 사용자 정의 형식을 매핑하는 확장 인프라를 포함해야 합니다. 복잡성이 높은 공급자를 개발하려면 상당한 노력이 필요합니다.

## 참조

- [IQueryable<T>](#)
- [IEnumerable<T>](#)
- [Enumerable](#)
- [표준 쿼리 연산자 개요\(C#\)](#)
- [LINQ to Objects\(C#\)](#)

# LINQ에 대한 Visual Studio IDE 및 도구 지원(C#)

2020-11-02 • 2 minutes to read • [Edit Online](#)

Visual Studio IDE(통합 개발 환경)는 LINQ 애플리케이션 개발을 지원하는 다음과 같은 기능을 제공합니다.

## Object Relational Designer

개체 관계형 디자이너는 기본 데이터베이스의 관계형 데이터를 나타내는 C#의 클래스를 생성하기 위해 [LINQ to SQL](#) 애플리케이션에서 사용할 수 있는 시각적 디자인 도구입니다. 자세한 내용은 [Visual Studio의 LINQ to SQL 도구](#)를 참조하세요.

## SQLMetal 명령줄 도구

SQLMetal은 LINQ to SQL 애플리케이션에서 사용할 기존 데이터베이스로부터 클래스를 생성하기 위해 빌드 프로세스에서 사용할 수 있는 명령줄 도구입니다. 자세한 내용은 [SqlMetal.exe\(코드 생성 도구\)](#)를 참조하세요.

## LINQ 인식 코드 편집기

C# 코드 편집기는 IntelliSense 및 서식 지정 기능으로 LINQ를 광범위하게 지원합니다.

## Visual Studio 디버거 지원

Visual Studio 디버거는 쿼리 식의 디버깅을 지원합니다. 자세한 내용은 [LINQ 디버깅](#)을 참조하세요.

## 참조

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

# 리플렉션(C#)

2021-02-18 • 4 minutes to read • [Edit Online](#)

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.

## NOTE

`using System;` 및 `using System.Reflection;` 을 .cs 파일의 맨 위에 추가해야 합니다.

```
// 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.

```
// Using Reflection to get information of an Assembly:  
Assembly info = typeof(int).Assembly;  
Console.WriteLine(info);
```

출력은 `System.Private.CoreLib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e`입니다.

## NOTE

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.

## 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](#) 항목을 참조하세요.

## 관련 단원

## 추가 정보

- 리플렉션
- 형식 정보 보기
- 리플렉션 및 제네릭 형식
- System.Reflection.Emit
- 특성에 저장된 정보 검색

## 참고 항목

- C# 프로그래밍 가이드
- .NET 어셈블리

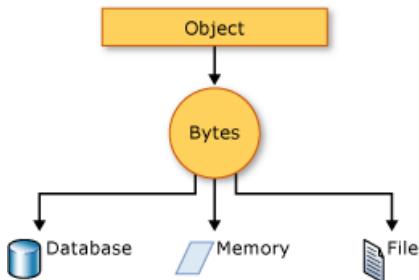
# Serialization(C#)

2020-11-02 • 13 minutes to read • [Edit Online](#)

Serialization은 개체를 저장하거나 메모리, 데이터베이스 또는 파일로 전송하기 위해 개체를 바이트 스트림으로 변환하는 프로세스입니다. 주 목적은 필요할 때 다시 개체로 만들 수 있도록 개체의 상태를 저장하는 것입니다. 역 프로세스를 deserialization이라고 합니다.

## Serialization 작동 방법

아래 그림에서는 serialization의 전체 프로세스 과정을 보여 줍니다:



개체는 데이터를 전달하는 스트림으로 직렬화됩니다. 스트림에는 버전, 문화권 및 어셈블리 이름과 같은 개체 형식 정보가 포함될 수도 있습니다. 해당 스트림의 개체를 데이터베이스, 파일 또는 메모리에 저장할 수 있습니다.

### Serialization 용도

Serialization은 개발자가 개체의 상태를 저장하고 필요에 따라 개체를 다시 만들 수 있게 함으로써 데이터 교환뿐 아니라 개체 스토리지 기능도 제공합니다. Serialization을 통해 개발자는 다음과 같은 작업을 수행할 수 있습니다.

- 웹 서비스를 사용하여 원격 애플리케이션에 개체 보내기
- 한 도메인에서 다른 도메인으로 개체 전달
- 방화벽을 통해 JSON 또는 XML 문자열로 개체 전달
- 애플리케이션 간에 보안 또는 사용자 관련 정보 유지

## JSON serialization

[System.Text.Json](#) 네임스페이스에는 JSON(JavaScript Object Notation) serialization 및 deserialization 클래스가 포함되어 있습니다. JSON은 웹에서 데이터를 공유하는데 일반적으로 사용되는 개방형 표준입니다.

JSON serialization은 개체의 퍼블릭 속성을 [RFC 8259 JSON 사양](#)을 따르는 문자열, 바이트 배열 또는 스트림으로 직렬화합니다. [JsonSerializer](#)가 클래스 인스턴스를 직렬화 또는 역직렬화하는 방식을 제어하려면 다음을 수행합니다.

- [JsonSerializerOptions](#) 개체 사용
- 클래스 또는 속성에 [System.Text.Json.Serialization](#) 네임스페이스의 특성 적용
- [사용자 지정 변환기 구현](#)

## 이진 및 XML Serialization

[System.Runtime.Serialization](#) 네임스페이스에는 이진 및 XML serialization 및 deserialization 클래스가 포함되어 있습니다.

이진 serialization은 이진 인코딩을 사용하여 스토리지 또는 스트림 기반 네트워크 스트림과 같은 용도로 사용할 수 있는 압축 serialization을 생성합니다. 이진 Serialization에서 멤버가 읽기 전용이더라도 모든 멤버가 Serialize되고 성능이 향상됩니다.

#### WARNING

이진 serialization은 위험할 수 있습니다. 자세한 내용은 [Binaryformatter 보안 가이드](#)를 참조 하세요.

XML serialization은 개체의 public 필드와 속성 또는 메서드의 매개 변수와 반환 값을 특정 XSD(XML 스키마 정의 언어) 문서와 일치하는 XML 스트림으로 serialize합니다. XML serialization을 사용하면 XML로 변환되는 public 속성 및 필드가 있는 강력한 형식의 클래스가 만들어집니다. [System.Xml.Serialization](#)에는 XML을 직렬화 및 역직렬화하기 위한 클래스가 포함되어 있습니다. [XmlSerializer](#)가 클래스 인스턴스를 직렬화 또는 역직렬화하는 방법을 제어하기 위해 클래스 및 클래스 멤버에 특성을 적용합니다.

#### 개체를 **Serialize** 가능하게 만들기

이진 또는 XML serialization의 경우 다음이 필요합니다.

- 직렬화할 개체
- 직렬화된 개체를 포함할 스트림
- [System.Runtime.Serialization.Formatter](#) 인스턴스

형식에 [SerializableAttribute](#) 특성을 적용하여 이 형식의 인스턴스를 직렬화할 수 있음을 나타냅니다. 형식에 [SerializableAttribute](#) 특성이 없는 경우 직렬화하려고 하면 예외가 throw됩니다.

필드가 직렬화되지 않도록 하려면 [NonSerializedAttribute](#) 특성을 적용합니다. serialize 가능한 형식의 필드에 특정 환경과 관련된 포인터, 핸들 또는 다른 데이터 구조가 포함되어 있고 필드를 다른 환경에서 의미 있게 재구성 할 수 없으면 serialize할 수 없게 만들 수 있습니다.

직렬화된 클래스에 [SerializableAttribute](#)가 표시된 다른 클래스의 개체에 대한 참조가 포함된 경우 해당 개체도 직렬화됩니다.

#### 기본 및 사용자 지정 **Serialization**

이진 및 XML serialization은 기본 및 사용자 지정의 두 가지 방법으로 수행할 수 있습니다.

기본 serialization은 .NET을 사용하여 개체를 자동으로 직렬화합니다. 유일한 요구 사항은 클래스에 [SerializableAttribute](#) 특성이 적용되어야 한다는 것입니다. [NonSerializedAttribute](#)는 특정 필드가 직렬화되지 않도록 하는 데 사용할 수 있습니다.

기본 Serialization을 사용하면 개체의 버전 관리에 문제가 발생할 수 있습니다. 버전 관리 문제가 중요하면 사용자 지정 Serialization을 사용합니다. 기본 serialization은 serialization을 수행하는 가장 쉬운 방법이지만 프로세스를 강력하게 제어하기는 어렵습니다.

사용자 지정 serialization에서는 serialize될 개체 및 수행 방법을 정확하게 지정할 수 있습니다. 클래스에 [SerializableAttribute](#)가 표시되어야 하고 [ISerializable](#) 인터페이스를 구현해야 합니다. 개체를 사용자 지정 방식으로도 역직렬화하려는 경우 사용자 지정 생성자를 사용합니다.

## 디자이너 **Serialization**

디자이너 Serialization은 개발 도구와 관련해서 개체 지속성이 적용되는 특수한 형태의 Serialization입니다. 디자이너 serialization은 개체 그래프를 소스 파일로 변환하여 나중에 개체 그래프를 복구하는 데 사용할 수 있도록 하는 프로세스입니다. 소스 파일에는 코드, 태그 또는 심지어 SQL 테이블 정보도 포함될 수 있습니다.

## 관련 항목 및 예제

[System.Text.Json 개요](#) [System.Text.Json](#) 라이브러리를 가져오는 방법을 보여 줍니다.

.NET에서 JSON을 직렬화 및 역직렬화하는 방법 `JsonSerializer` 클래스를 사용하여 JSON에서 개체 데이터를 읽고 쓰는 방법을 보여 줍니다.

#### [연습: Visual Studio에서 개체 유지\(C#\)](#)

`serialization`을 사용하여 인스턴스 간에 개체의 데이터를 유지함으로써 다음에 개체를 인스턴스화할 때 값을 저장하고 검색하는 방식을 보여 줍니다.

#### [XML 파일에서 개체 데이터를 읽는 방법\(C#\)](#)

`XmlSerializer` 클래스를 사용하여 이전에 XML 파일에 기록된 개체 데이터를 읽는 방법을 보여 줍니다.

#### [XML 파일에 개체 데이터를 쓰는 방법\(C#\)](#)

`XmlSerializer` 클래스를 사용하여 클래스의 개체를 XML 파일에 쓰는 방법을 보여 줍니다.

# XML 파일에 개체 데이터를 쓰는 방법(C#)

2020-11-02 • 2 minutes to read • [Edit Online](#)

이 예제에서는 [XmlSerializer](#) 클래스를 사용하여 XML 파일에 클래스의 개체를 씁니다.

## 예제

```
public class XMLWrite
{
    static void Main(string[] args)
    {
        WriteXML();
    }

    public class Book
    {
        public String title;
    }

    public static void WriteXML()
    {
        Book overview = new Book();
        overview.title = "Serialization Overview";
        System.Xml.Serialization.XmlSerializer writer =
            new System.Xml.Serialization.XmlSerializer(typeof(Book));

        var path = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) +
        "//SerializationOverview.xml";
        System.IO.FileStream file = System.IO.File.Create(path);

        writer.Serialize(file, overview);
        file.Close();
    }
}
```

## 코드 컴파일

직렬화되는 클래스에는 매개 변수가 없는 공용 생성자가 있어야 합니다.

## 강력한 프로그래밍

다음 조건에서 예외가 발생합니다.

- serialize되는 클래스에 매개 변수가 없는 public 생성자가 없는 경우
- 파일이 있지만 읽기 전용인 경우([IOException](#))
- 경로가 너무 긴 경우([PathTooLongException](#))
- 디스크가 꽉 찬 경우([IOException](#))

## .NET 보안

이 예제에서는 파일이 아직 없는 경우 새 파일을 만듭니다. 애플리케이션에서 파일을 만들어야 하는 경우 해당 애플리케이션에 폴더에 대한 [Create](#) 권한이 있어야 합니다. 파일이 이미 있는 경우 애플리케이션에 더 낮은 권

한인 `Write` 권한만 있으면 됩니다. 가능한 경우 배포하는 동안 파일을 만들고, 폴더에 대한 `Create` 권한 대신 단일 파일에 대해 `Read` 권한만 부여하는 것이 더 안전합니다.

## 참조

- [StreamWriter](#)
- [XML 파일에서 개체 데이터를 읽는 방법\(C#\)](#)
- [Serialization\(C#\)](#)

# XML 파일에서 개체 데이터를 읽는 방법(C#)

2020-11-02 • 2 minutes to read • [Edit Online](#)

이 예제에서는 [XmlSerializer](#) 클래스를 사용하여 이전에 XML 파일에 기록된 개체 데이터를 읽습니다.

## 예제

```
public class Book
{
    public String title;
}

public void ReadXML()
{
    // First write something so that there is something to read ...
    var b = new Book { title = "Serialization Overview" };
    var writer = new System.Xml.Serialization.XmlSerializer(typeof(Book));
    var wfile = new System.IO.StreamWriter(@"c:\temp\SerializationOverview.xml");
    writer.Serialize(wfile, b);
    wfile.Close();

    // Now we can read the serialized book ...
    System.Xml.Serialization.XmlSerializer reader =
        new System.Xml.Serialization.XmlSerializer(typeof(Book));
    System.IO.StreamReader file = new System.IO.StreamReader(
        @"c:\temp\SerializationOverview.xml");
    Book overview = (Book)reader.Deserialize(file);
    file.Close();

    Console.WriteLine(overview.title);
}
```

## 코드 컴파일

파일 이름 "c:\temp\SerializationOverview.xml"을 `serialize`된 데이터가 포함된 파일 이름으로 바꿉니다. 데이터를 직렬화하는 작업에 대한 자세한 내용은 [XML 파일에 개체 데이터를 쓰는 방법\(C#\)](#)을 참조하세요.

클래스에는 매개 변수가 없는 `public` 생성자가 있어야 합니다.

`public` 속성과 필드만 역직렬화됩니다.

## 강력한 프로그래밍

다음 조건에서 예외가 발생합니다.

- `serialize`되는 클래스에 매개 변수가 없는 `public` 생성자가 없는 경우
- 파일의 데이터가 역직렬화할 클래스의 데이터를 나타내지 않는 경우
- 파일이 없는 경우([IOException](#))

## .NET 보안

항상 입력을 확인하고, 신뢰할 수 없는 소스의 데이터를 역직렬화하지 마세요. 다시 생성된 개체는 역직렬화한 코드의 사용 권한으로 로컬 컴퓨터에서 실행됩니다. 애플리케이션에서 데이터를 사용하기 전에 모든 입력을 확

인해야 합니다.

## 참조

- [StreamWriter](#)
- [XML 파일에 개체 데이터를 쓰는 방법\(C#\)](#)
- [Serialization\(C#\)](#)
- [C# 프로그래밍 가이드](#)

# C#을 사용하여 개체 유지

2020-11-02 • 12 minutes to read • [Edit Online](#)

serialization을 사용하면 인스턴스 간에 개체의 데이터를 유지할 수 있으므로, 다음에 개체를 인스턴스화할 때 값을 저장하고 검색할 수 있습니다.

이 연습에서는 기본 `Loan` 개체를 만들고 데이터를 파일에 유지합니다. 그런 다음 개체를 다시 만들 때 파일에서 데이터를 검색합니다.

## IMPORTANT

이 예제에서는 파일이 아직 없는 경우 새 파일을 만듭니다. 애플리케이션에서 파일을 만들어야 하는 경우 해당 애플리케이션은 폴더에 대한 `Create` 권한이 있어야 합니다. 권한은 액세스 제어 목록을 사용하여 설정됩니다. 파일이 이미 있는 경우, 애플리케이션에는 더 낮은 권한인 `Write` 권한만 필요합니다. 가능한 경우 배포하는 동안 파일을 만드는 것이 안전하며, 폴더에 대한 `Create` 권한 대신 단일 파일에 대해 `Read` 권한만 부여하는 것이 더 안전합니다. 또한 루트 폴더나 Program Files 폴더보다 사용자 폴더에 데이터를 쓰는 것이 더 안전합니다.

## IMPORTANT

이 예제에서는 이진 형식 파일의 데이터를 저장합니다. 이러한 형식은 암호 또는 신용 카드 정보와 같은 중요한 데이터에 사용하면 안 됩니다.

## 사전 요구 사항

- 빌드하고 실행하려면 [.NET Core SDK](#)를 설치합니다.
- 아직 없는 경우 즐겨 찾는 코드 편집기를 설치합니다.

## TIP

코드 편집기를 설치해야 하나요? [Visual Studio](#)를 체험해 보세요.

- 예제는 C# 7.3이 필요합니다. [C# 언어 버전 선택](#)을 참조하세요.

.NET 샘플 GitHub 리포지토리에서 온라인으로 샘플 코드를 검사할 수 있습니다.

## Loan 개체 만들기

첫 번째 단계는 `Loan` 클래스와 이 클래스를 사용하는 콘솔 애플리케이션을 만드는 것입니다.

- 새 애플리케이션을 만듭니다. `dotnet new console -o serialization` 을 입력하여 `serialization`이라는 하위 디렉터리에서 새 콘솔 애플리케이션을 만듭니다.
- 편집기에서 애플리케이션을 열고 `Loan.cs`라는 새 클래스를 추가합니다.
- `Loan` 클래스에 다음 코드를 추가합니다.

```

public class Loan : INotifyPropertyChanged
{
    public double LoanAmount { get; set; }
    public double InterestRatePercent { get; set; }

    [field:NonSerialized()]
    public DateTime TimeLastLoaded { get; set; }

    public int Term { get; set; }

    private string customer;
    public string Customer
    {
        get { return customer; }
        set
        {
            customer = value;
            PropertyChanged?.Invoke(this,
                new PropertyChangedEventArgs(nameof(Customer)));
        }
    }

    [field: NonSerialized()]
    public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;

    public Loan(double loanAmount,
               double interestRate,
               int term,
               string customer)
    {
        this.LoanAmount = loanAmount;
        this.InterestRatePercent = interestRate;
        this.Term = term;
        this.customer = customer;
    }
}

```

`Loan` 클래스를 사용하는 애플리케이션도 만들어야 합니다.

## Loan 개체 직렬화

- `Program.cs` 를 엽니다. 다음 코드를 추가합니다.

```
Loan TestLoan = new Loan(10000.0, 7.5, 36, "Neil Black");
```

`PropertyChanged` 이벤트에 대한 이벤트 처리기를 추가하고 `Loan` 개체를 수정하고 변경 내용을 표시하는 몇 줄을 추가합니다. 다음 코드에서 추가된 기능을 확인할 수 있습니다.

```

TestLoan.PropertyChanged += (_, __) => Console.WriteLine($"New customer value: {TestLoan.Customer}");

TestLoan.Customer = "Henry Clay";
Console.WriteLine(TestLoan.InterestRatePercent);
TestLoan.InterestRatePercent = 7.1;
Console.WriteLine(TestLoan.InterestRatePercent);

```

이 시점에서 코드를 실행하고 현재 출력을 확인할 수 있습니다.

```
New customer value: Henry Clay
```

```
7.5
```

```
7.1
```

이 애플리케이션을 반복해서 실행하면 항상 동일한 값을 씁니다. 프로그램을 실행할 때마다 새로운 Loan 객체가 생성됩니다. 현실 세계에서 금리는 주기적으로 변경되지만, 애플리케이션이 실행될 때마다 변경되는 것은 아닙니다. Serialization 코드는 애플리케이션의 인스턴스 간에 가장 최근 이자율을 유지함을 의미합니다. 다음 단계에서는 Loan 클래스에 serialization을 추가하여 이를 수행합니다.

## Serialization을 사용하여 객체 유지

Loan 클래스의 값을 유지하려면 먼저 클래스를 `Serializable` 속성으로 표시해야 합니다. Loan 클래스 선언 위에 다음 코드를 추가합니다.

```
[Serializable()]
```

`SerializableAttribute`는 클래스의 모든 내용을 파일에 유지할 수 있음을 컴파일러에 알립니다. `PropertyChanged` 이벤트가 저장되어야 하는 개체 그래프의 일부를 나타내지 않기 때문에 직렬화되지 않아야 합니다. 그러면 해당 이벤트에 연결된 모든 개체를 직렬화합니다. `PropertyChanged` 이벤트 처리기의 필드 선언에 `NonSerializedAttribute`를 추가할 수 있습니다.

```
[field: NonSerialized()]
public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;
```

C# 7.3부터는 `field` 대상 값을 사용하여 자동 구현 속성의 지원 필드에 특성을 연결할 수 있습니다. 다음 코드에서는 `TimeLastLoaded` 속성을 추가하고 직렬화할 수 없음으로 표시합니다.

```
[field:NonSerialized()]
public DateTime TimeLastLoaded { get; set; }
```

다음 단계는 LoanApp 애플리케이션에 serialization 코드를 추가하는 것입니다. 클래스를 serialize하여 파일에 쓰려면 `System.IO` 및 `System.Runtime.Serialization.Formatters.Binary` 네임스페이스를 사용합니다. 정규화된 이름을 입력하지 않으려면 필요한 다음 코드에 표시된 대로 필요한 네임스페이스에 참조를 추가할 수 있습니다.

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

다음 단계는 객체를 만들 때 파일에서 객체를 역직렬화할 코드를 추가하는 것입니다. 다음 코드에 표시된 대로 직렬화된 데이터의 파일 이름에 대한 클래스에 상수를 추가합니다.

```
const string FileName = @"../../../../SavedLoan.bin";
```

다음으로 `TestLoan` 객체를 만든 줄 뒤에 다음 코드를 추가합니다.

```
if (File.Exists(FileName))
{
    Console.WriteLine("Reading saved file");
    Stream openFileStream = File.OpenRead(FileName);
    BinaryFormatter deserializer = new BinaryFormatter();
    TestLoan = (Loan)deserializer.Deserialize(openFileStream);
    TestLoan.TimeLastLoaded = DateTime.Now;
    openFileStream.Close();
}
```

먼저 파일이 있는지를 확인해야 합니다. 파일이 있으면 이진 파일을 읽는 [Stream](#) 클래스와 파일을 변환하는 [BinaryFormatter](#) 클래스를 만듭니다. 또한 스트림 형식에서 Loan 개체 형식으로 변환해야 합니다.

다음으로 클래스를 직렬화하는 코드를 파일에 추가해야 합니다. [Main](#) 메서드에서 기존 코드 뒤에 다음 코드를 추가합니다.

```
Stream SaveFileStream = File.Create(FileName);
BinaryFormatter serializer = new BinaryFormatter();
serializer.Serialize(SaveFileStream, TestLoan);
SaveFileStream.Close();
```

이 시점에서 다시 애플리케이션을 빌드 및 실행할 수 있습니다. 처음으로 실행되면 이자율이 7.5에서 시작한다음, 7.1로 변경됩니다. 애플리케이션을 닫았다가 다시 엽니다. 이제 애플리케이션이 저장된 파일을 읽는 메시지를 인쇄하고 이자율은 코드가 변경하기 전에도 7.1입니다.

## 참조

- [Serialization\(C#\)](#)
- [C# 프로그래밍 가이드](#)

# 문, 식, 연산자(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

애플리케이션을 구성하는 C# 코드는 키워드, 식, 연산자가 포함된 문으로 구성됩니다. 이 섹션에서는 C# 프로그램의 이러한 기본 요소에 대한 정보를 소개합니다.

자세한 내용은 다음을 참조하세요.

- [문](#)
- [연산자 및 식](#)
- [식 본문 멤버](#)
- [익명 함수](#)
- [같음 비교](#)

## 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](#)
- [캐스팅 및 형식 변환](#)

# 문(C# 프로그래밍 가이드)

2020-11-02 • 14 minutes to read • [Edit Online](#)

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 보여줍니다.

```
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.

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 다음 항목을 참조하세요. <ul style="list-style-type: none"> <li>• <a href="#">if</a></li> <li>• <a href="#">else</a></li> <li>• <a href="#">switch</a></li> <li>• <a href="#">case</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 다음 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="#">in</a></li> <li>• <a href="#">while</a></li> </ul>
점프 문	점프 문은 컨트롤을 다른 코드 섹션으로 전송합니다. 자세한 내용은 다음 항목을 참조하세요. <ul style="list-style-type: none"> <li>• <a href="#">break</a></li> <li>• <a href="#">continue</a></li> <li>• <a href="#">default</a></li> <li>• <a href="#">goto</a></li> <li>• <a href="#">return</a></li> <li>• <a href="#">yield</a></li> </ul>
예외 처리 문	예외 처리 문을 사용하면 런타임 시 발생하는 예외 조건에서 정상적으로 복구할 수 있습니다. 자세한 내용은 다음 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>
Checked 및 Unchecked	checked 및 unchecked 문을 사용하면 결과가 저장되는 변수가 너무 작아서 결과 값을 수용할 수 없는 경우 수치 연산에서 오버플로우가 발생할 수 있는지 여부를 지정할 수 있습니다. 자세한 내용은 <a href="#">checked</a> 및 <a href="#">unchecked</a> 를 참조하세요.

<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.

```

// 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.

```

void ProcessMessages()
{
    while (ProcessMessage())
        ; // Statement needed here.
}

void F()
{
    //...
    if (done) goto exit;
//...
exit:
    ; // Statement needed here.
}

```

## 포함 문

[do](#), [while](#), [for](#) 및 [foreach](#)를 비롯한 일부 문은 항상 포함 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.

```

// 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.

```

if(pointB == true)
    //Error CS1023:
    int radius = 5;

```

오류를 해결하려면 포함 문을 블록에 배치합니다.

```
if (b == true)
{
    // OK:
    System.DateTime d = System.DateTime.Now;
    System.Console.WriteLine(d.ToString("yyyy-MM-dd HH:mm:ss"));
}
```

## 중첩된 문 블록

다음 코드와 같이 문 블록을 중첩할 수 있습니다.

```
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.

```
// 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 사양](#)의 문 섹션을 참조하세요.

## 참조

- [C# 프로그래밍 가이드](#)
- [문 키워드](#)
- [C# 연산자 및 식](#)

# 식 본문 멤버(C# 프로그래밍 가이드)

2020-11-02 • 8 minutes to read • [Edit Online](#)

식 본문 정의를 사용하면 간결하고 읽을 수 있는 형식으로 멤버 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.

```
member => expression;
```

여기서 *expression*은 유효한 식입니다.

C# 6에서는 메서드 및 읽기 전용 속성에 식 본문 정의 지원이 제공되었으며, C# 7.0에서는 해당 지원이 확장되었습니다. 다음 표에 나열된 형식 멤버와 함께 식 본문 정의를 사용할 수 있습니다.

멤버	지원 버전
메서드	C# 6
읽기 전용 속성	C# 6
Property	C# 7.0
생성자	C# 7.0
종료자	C# 7.0
인덱서	C# 7.0

## 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.

```

using System;

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

C# 6부터는 읽기 전용 속성을 구현하기 위해 식 본문 정의를 사용할 수 있습니다. 이를 위해 사용하는 구문은 다음과 같습니다.

```
PropertyType PropertyName => expression;
```

다음 예제에서는 `Location` 클래스를 정의합니다. 이 클래스의 읽기 전용 `Name` 속성은 `locationName` 비공개 필드의 값을 반환하는 식 본문 정의로 구현됩니다.

```

public class Location
{
    private string locationName;

    public Location(string name)
    {
        locationName = name;
    }

    public string Name => locationName;
}

```

속성에 대한 자세한 내용은 [속성\(C# 프로그래밍 가이드\)](#)를 참조하세요.

## 속성

C# 7.0부터는 식 본문 정의를 사용하여 `get` 및 `set` 속성 접근자를 구현할 수 있습니다. 다음 예제에서는 이를 수행하는 방법을 보여줍니다.

```
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 할당식 또는 메서드 호출로 구성됩니다.

다음 예제에서는 생성자에 *name*이라는 단일 문자열 매개 변수가 있는 `Location` 클래스를 정의합니다. 식 본문 정의에서 `Name` 속성에 인수를 할당합니다.

```
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.

```
using System;

public class Destroyer
{
    public override string ToString() => GetType().Name;

    ~Destroyer() => Console.WriteLine($"The {ToString()} destructor is executing.");
}
```

자세한 내용은 [종료자\(C# 프로그래밍 가이드\)](#)를 참조하세요.

## 인덱서

속성과 마찬가지로, `get` 접근자가 값을 반환하는 단일 식으로 구성되거나 `set` 접근자가 단순 할당을 수행하

는 경우 인덱서의 `get` 및 `set` 접근자는 식 본문 정의로 구성됩니다.

다음 예제에서는 다양한 스포츠의 이름이 포함된 내부 `String` 배열을 포함하는 `Sports`라는 클래스를 정의합니다. 인덱서의 `get` 및 `set` 접근자는 둘 다 식 본문 정의로 구현됩니다.

```
using System;
using System.Collections.Generic;

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.

# 익명 함수(C# 프로그래밍 가이드)

2020-11-02 • 4 minutes to read • [Edit Online](#)

익명 함수는 대리자 형식이 예상되는 곳에서 항상 사용할 수 있는 “인라인” 문 또는 식입니다. 이를 사용하여 명명된 대리자를 초기화하거나 명명된 대리자 형식 대신 이를 메서드 매개 변수로 전달할 수 있습니다.

람다 식 또는 무명 메서드를 사용하여 익명 함수를 만들 수 있습니다. 인라인 코드를 작성하는 더 간결하고 이해하기 쉬운 방법을 제공하는 람다 식을 사용하는 것이 좋습니다. 무명 메서드와 달리 일부 형식의 람다 식은 식트리 형식으로 변환될 수 있습니다.

## C#에서 대리자의 발전

C# 1.0에서는 코드의 다른 위치에 정의된 메서드를 사용하여 명시적으로 초기화하는 방식으로 대리자의 인스턴스를 만들었습니다. C# 2.0에서는 대리자 호출에서 실행될 수 있는 이름 없는 인라인 문 블록을 작성하는 방법으로 무명 메서드의 개념을 소개했습니다. C# 3.0에서는 개념적으로 무명 메서드와 비슷하지만 더 간결하고 표현이 다양한 람다 식을 소개했습니다. 이러한 두 기능을 함께 익명 함수라고 합니다. 일반적으로 .NET Framework 3.5 이상을 대상으로 하는 애플리케이션은 람다 식을 사용해야 합니다.

다음 예제에서는 C# 1.0에서 C# 3.0까지 대리자 만들기의 발전을 보여 줍니다.

```

class Test
{
    delegate void TestDelegate(string s);
    static void M(string s)
    {
        Console.WriteLine(s);
    }

    static void Main(string[] args)
    {
        // Original delegate syntax required
        // initialization with a named method.
        TestDelegate testDelA = new TestDelegate(M);

        // C# 2.0: A delegate can be initialized with
        // inline code, called an "anonymous method." This
        // method takes a string as an input parameter.
        TestDelegate testDelB = delegate(string s) { Console.WriteLine(s); };

        // C# 3.0. A delegate can be initialized with
        // a lambda expression. The lambda also takes a string
        // as an input parameter (x). The type of x is inferred by the compiler.
        TestDelegate testDelC = (x) => { Console.WriteLine(x); };

        // Invoke the delegates.
        testDelA("Hello. My name is M and I write lines.");
        testDelB("That's nothing. I'm anonymous and ");
        testDelC("I'm a famous author.");

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
Hello. My name is M and I write lines.
That's nothing. I'm anonymous and
I'm a famous author.
Press any key to exit.
*/

```

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 익명 함수 식](#) 섹션을 참조하세요.

## 참조

- 문, 식, 연산자
- 람다 식
- 대리자
- 식 트리(C#)

# 쿼리에 람다 식을 사용하는 방법(C# 프로그래밍 가이드)

2020-11-02 • 4 minutes to read • [Edit Online](#)

람다 식은 쿼리 구문에 직접 사용하지 않고 메서드 호출에 사용하며, 쿼리 식에 메서드 호출이 포함될 수 있습니다. 실제로 일부 쿼리 작업은 메서드 구문으로만 표현할 수 있습니다. 쿼리 구문과 메서드 구문 간의 차이점에 대한 자세한 내용은 [LINQ의 쿼리 구문 및 메서드 구문](#)을 참조하세요.

## 예제

다음 예제에서는 `Enumerable.Where` 표준 쿼리 연산자를 사용하여 메서드 기반 쿼리에 람다 식을 사용하는 방법을 보여 줍니다. 이 예제의 `Where` 메서드에는 대리자 형식 `Func<T,TResult>`의 입력 매개 변수가 있으며, 해당 대리자는 정수를 입력으로 사용하고 부울을 반환합니다. 람다 식을 해당 대리자로 변환할 수 있습니다.

`Queryable.Where` 메서드를 사용하는 LINQ to SQL 쿼리인 경우 매개 변수 형식은 `Expression<Func<int,bool>>`이지만 람다 식은 정확히 동일하게 표시됩니다. 식 형식에 대한 자세한 내용은 [System.Linq.Expressions.Expression](#)을 참조하세요.

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

        // Data source.
        int[] scores = { 90, 71, 82, 93, 75, 82 };

        // The call to Count forces iteration of the source
        int highScoreCount = scores.Where(n => n > 80).Count();

        Console.WriteLine("{0} scores are greater than 80", highScoreCount);

        // Outputs: 4 scores are greater than 80
    }
}
```

## 예제

다음 예제에서는 쿼리 식의 메서드 호출에 람다 식을 사용하는 방법을 보여 줍니다. 쿼리 구문을 사용하여 `Sum` 표준 쿼리 연산자를 호출할 수 없기 때문에 람다가 필요합니다.

쿼리는 먼저 `GradeLevel` 열거형에 정의된 성적 수준에 따라 학생을 그룹화합니다. 그런 다음 각 그룹에 대해 각 학생의 총 점수를 더합니다. 이 경우 두 개의 `sum` 연산이 필요합니다. 한쪽 `sum`은 각 학생의 총 점수를 계산하고, 바깥쪽 `sum`은 그룹에 속한 모든 학생에 대해 합산된 누계를 유지합니다.

```

private static void TotalsByGradeLevel()
{
    // This query retrieves the total scores for First Year students, Second Years, and so on.
    // The outer Sum method uses a lambda in order to specify which numbers to add together.
    var categories =
        from student in students
        group student by student.Year into studentGroup
        select new { GradeLevel = studentGroup.Key, TotalScore = studentGroup.Sum(s => s.ExamScores.Sum()) };

    // Execute the query.
    foreach (var cat in categories)
    {
        Console.WriteLine("Key = {0} Sum = {1}", cat.GradeLevel, cat.TotalScore);
    }
}
/*
Outputs:
Key = SecondYear Sum = 1014
Key = ThirdYear Sum = 964
Key = FirstYear Sum = 1058
Key = FourthYear Sum = 974
*/

```

## 코드 컴파일

이 코드를 실행하려면 메서드를 복사하고 [개체 컬렉션 쿼리](#)에 제공된 `StudentClass`에 붙여넣은 다음 `Main` 메서드에서 호출합니다.

## 참조

- 람다 식
- 식 트리(C#)

# 같은 비교(C# 프로그래밍 가이드)

2020-11-02 • 8 minutes to read • [Edit Online](#)

두 값이 같은지를 비교해야 하는 경우가 있습니다. 때로는 두 변수에 포함된 값이 같음을 의미하는 값 같은(동등이라고도 함)을 테스트합니다. 다른 경우에는 두 변수가 메모리에서 동일한 기본 개체를 참조하는지 여부를 확인해야 합니다. 이 유형의 같은을 참조 같은 또는 *ID*라고 합니다. 이 항목에서는 이러한 두 종류의 같은을 설명하고 자세한 정보가 있는 다른 항목에 대한 링크를 제공합니다.

## 참조 같은

참조 같은은 두 개체 참조가 동일한 기본 개체를 참조함을 의미합니다. 이러한 상황은 다음 예제와 같이 단순 할당을 통해 발생할 수 있습니다.

```
using System;
class Test
{
    public int Num { get; set; }
    public string Str { get; set; }

    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);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

이 코드에서는 두 개체가 생성되지만 대입문 이후 두 참조가 동일한 개체를 참조합니다. 따라서 참조 같은이 있습니다. 두 참조가 동일한 개체를 참조하는지 확인하려면 [ReferenceEquals](#) 메서드를 사용합니다.

참조 같은의 개념은 참조 형식에만 적용됩니다. 값 형식의 인스턴스가 변수에 할당될 때 값의 복사본이 생성되기 때문에 값 형식 개체에는 참조 같은이 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 같은\(ID\)을 테스트하는 방법](#)을 참조하세요.

## 값 같은

값 같은은 두 개체에 동일한 값이 포함되어 있음을 의미합니다. `int` 또는 `bool`과 같은 기본 값 형식의 경우 값 같은 테스트가 간단합니다. 다음 예제와 같이 `==` 연산자를 사용할 수 있습니다.

```

int a = GetOriginalValue();
int b = GetCurrentValue();

// Test for value equality.
if (b == a)
{
    // The two integers are equal.
}

```

대부분의 다른 형식에서는 값 같은 테스트가 더 복잡한데, 형식에서 정의된 방식을 알아야 하기 때문입니다. 여러 필드나 속성이 있는 클래스 및 구조체의 경우 값 같은은 대체로 모든 필드 또는 속성에 동일한 값이 있다는 의미로 정의됩니다. 예를 들어 pointA.X가 pointB.X와 같고 pointA.Y가 pointB.Y와 같으면 두 `Point` 개체가 동일한 것으로 정의할 수 있습니다.

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 같은을 정의하는 방법에 대한 자세한 내용은 [형식의 값 같은을 정의하는 방법](#)을 참조하세요.

### 부동 소수점 값에 대한 값 같은

이진 컴퓨터의 부정확한 부동 소수점 연산 때문에 부동 소수점 값(`double` 및 `float`)의 같은 비교에서 문제가 발생합니다. 자세한 내용은 [System.Double](#) 항목의 설명을 참조하세요.

## 관련 항목

제목	설명
<a href="#">참조 같은(ID)을 테스트하는 방법</a>	두 변수에 참조 같은이 있는지를 확인하는 방법을 설명 합니다.
<a href="#">형식의 값 같은을 정의하는 방법</a>	형식에 대한 값 같은의 사용자 지정 정의를 제공하는 방법을 설명합니다.
<a href="#">C# 프로그래밍 가이드</a>	.NET을 통해 C#에 제공되는 기능 및 중요한 C# 언어 기능에 대한 자세한 정보 링크를 제공합니다.
<a href="#">유형</a>	C# 형식 시스템에 대한 정보 및 추가 정보 링크를 제공합니다.

## 참조

- [C# 프로그래밍 가이드](#)

# 형식에 대한 값 같음을 정의하는 방법(C# 프로그래밍 가이드)

2020-11-02 • 15 minutes to read • [Edit Online](#)

클래스 또는 구조체를 정의할 때 형식에 대한 값 같음(또는 동등)의 사용자 지정 정의를 만드는 것이 적합한지 결정합니다. 일반적으로 형식의 개체를 일종의 컬렉션에 추가해야 하는 경우 또는 주요 용도가 필드 또는 속성 집합 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 사항을 따라야 합니다(다음 규칙의 경우 `x`, `y` 및 `z`가 Null이 아닌 것으로 가정).

1. `x.Equals(x)` 가 `true` 를 반환하는 경우 이를 반사 속성이라고 합니다.
2. `x.Equals(y)` 는 `y.Equals(x)` 와 동일한 값을 반환합니다. 이를 대칭 속성이라고 합니다.
3. `(x.Equals(y) && y.Equals(z))` 가 `true` 를 반환하면 `x.Equals(z)` 가 `true` 를 반환합니다. 이를 전이적 속성이라고 합니다.
4. `x.Equals(y)` 의 연속 호출은 `x` 및 `y`에서 참조하는 개체가 수정되지 않는 한 동일한 값이 반환됩니다.
5. Null이 아닌 값은 Null과 같지 않습니다. 그러나 CLR은 모든 메서드 호출에서 Null을 확인하고 `this` 참조가 Null인 경우 `NullReferenceException` 을 throw합니다. 따라서 `x` Null인 경우 `x.Equals(y)` 는 예외를 throw합니다. `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 )` 구현에 서 `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 같은 정의를 결정할 수도 있습니다. `Equals` 에서 예외를 throw하지 않습니다. 클래스만 해당: 이 메서드는 클래스에 선언된 필드만 검사해야 합니다. `base.Equals` 를 호출하여 기본 클래스에 있는 필드를 검사해야 합니다. 형식이 `Object`에서 직접 상속하는 경우에는 이 작업을 수행하지 마세요. `Object.Equals(Object)`의 `Object` 구현에서 참조 같음 검사를 수행합니다.
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.

**NOTE**

C# 9.0부터 레코드를 사용하여 불필요한 상용구 코드 없이 값 같음 의미 체계를 가져올 수 있습니다.

## 클래스 예제

다음 예제에서는 클래스(참조 형식)에서 값 같음을 구현하는 방법을 보여 줍니다.

```
namespace ValueEquality
{
    using System;
    class TwoDPoint : IEquatable<TwoDPoint>
    {
        // Readonly auto-implemented properties.
        public int X { get; private set; }
        public int Y { get; private set; }

        // Set the properties in the constructor.
        public TwoDPoint(int x, int y)
        {
            if ((x < 1) || (x > 2000) || (y < 1) || (y > 2000))
            {
                throw new System.ArgumentException("Point must be in range 1 - 2000");
            }
            this.X = x;
            this.Y = y;
        }

        public override bool Equals(object obj)
        {
            return this.Equals(obj as TwoDPoint);
        }

        public bool Equals(TwoDPoint p)
        {
            // If parameter is null, return false.
            if (Object.ReferenceEquals(p, 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()
        {
            return X * 0x00010000 + Y;
        }

        public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)
```

```

{
    // Check for null on left side.
    if (Object.ReferenceEquals(lhs, null))
    {
        if (Object.ReferenceEquals(rhs, null))
        {
            // null == null = true.
            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)
{
    return !(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 System.ArgumentException("Point must be in range 1 - 2000");
        }
        this.Z = z;
    }

    public override bool Equals(object obj)
    {
        return this.Equals(obj as ThreeDPoint);
    }

    public bool Equals(ThreeDPoint p)
    {
        // If parameter is null, return false.
        if (Object.ReferenceEquals(p, 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()
        {
            return (X * 0x100000) + (Y * 0x1000) + Z;
        }

        public static bool operator ==(ThreeDPoint lhs, ThreeDPoint rhs)
        {
            // Check for null.
            if (Object.ReferenceEquals(lhs, null))
            {
                if (Object.ReferenceEquals(rhs, 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)
    {
        return !(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

```

```

        compare to some other type - raise
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.

## 구조체 예제

다음 예제에서는 구조체(값 형식)에서 값 같음을 구현하는 방법을 보여 줍니다.

```

using System;
struct TwoDPoint : IEquatable<TwoDPoint>
{
    // Read/write auto-implemented properties.
    public int X { get; private set; }
    public int Y { get; private set; }

    public TwoDPoint(int x, int y)
        : this()
    {
        X = x;
        Y = x;
    }

    public override bool Equals(object obj)
    {
        if (obj is TwoDPoint)
        {
            return this.Equals((TwoDPoint)obj);
        }
        return false;
    }

    public bool Equals(TwoDPoint p)
    {
        return (X == p.X) && (Y == p.Y);
    }

    public override int GetHashCode()
    {
        return X ^ Y;
    }

    public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)
    {
        return lhs.Equals(rhs);
    }

    public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs)
    {
        return !(lhs.Equals(rhs));
    }
}

class Program
{

```

```

    static void Main(string[] args)
    {
        TwoDPoint pointA = new TwoDPoint(3, 4);
        TwoDPoint pointB = new TwoDPoint(3, 4);
        int i = 5;

        // Compare using virtual Equals, static Equals, and == and != operators.
        // 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);

        // 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
   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.

## 참조

- [같음 비교](#)
- [C# 프로그래밍 가이드](#)

# 참조 같음(ID)을 테스트하는 방법(C# 프로그래밍 가이드)

2021-02-18 • 6 minutes to read • [Edit Online](#)

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.

## 예제

```
using System;
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 = \"{0}\" strB = \"{1}\\"", 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
*/

```

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.

## 참조

- [같음 비교](#)

# 형식(C# 프로그래밍 가이드)

2021-02-18 • 35 minutes to read • [Edit Online](#)

## 형식, 변수 및 값

C#은 강력한 형식의 언어입니다. 모든 변수 및 상수에는 값으로 계산되는 모든 식을 실행하는 형식이 있습니다. 모든 메서드 선언은 각 입력 매개 변수와 반환 값에 대해 이름, 매개 변수 수, 형식 및 종류(값, 참조 또는 출력)를 지정합니다. .NET 클래스 라이브러리는 기본 제공 숫자 형식 집합 및 파일 시스템, 네트워크 연결, 컬렉션, 개체 배열, 날짜 등의 다양한 논리 구문을 나타내는 더 복잡한 형식을 정의합니다. 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.
- 런타임에 변수에 대한 메모리가 할당될 위치.
- 허용되는 작업 유형.

컴파일러는 형식 정보를 사용하여 코드에서 수행되는 모든 작업의 형식이 안전한지 확인합니다. 예를 들어 `int` 형식의 변수를 선언할 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.

```
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;
```

### NOTE

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

컴파일러는 형식 정보를 실행 파일에 메타데이터로 포함합니다. CLR(공용 언어 런타임)은 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.

```

// 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`가 필요하고 문자열을 반환하는 메서드를 보여 줍니다.

```

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`를 선언하고 `true`의 부울 값을 여기에 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#에서는 정수, 부동 소수점 값, 부울 식, 텍스트 문자, 10진수 값 및 기타 데이터 형식을 표현하는 기본 제공 형식의 표준 세트를 제공합니다. 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` 및 `enum` 구문을 사용하여 자체 사용자 지정 형식을 만듭니다. .NET 클래스 라이브러리 자체는 자체 애플리케이션에서 사용할 수 있는 Microsoft에서 제공되는 사용자 지정 형식의 컬렉션입니다. 기본적으로 클래스 라이브러리의 가장 자주 사용되는 형식을 모든 C# 프로그램에서 사용할 수 있습니다. 기타 형식은 정의되어 있는 어셈블리에 대한 프로젝트 참조를 명시적으로 추가할 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)

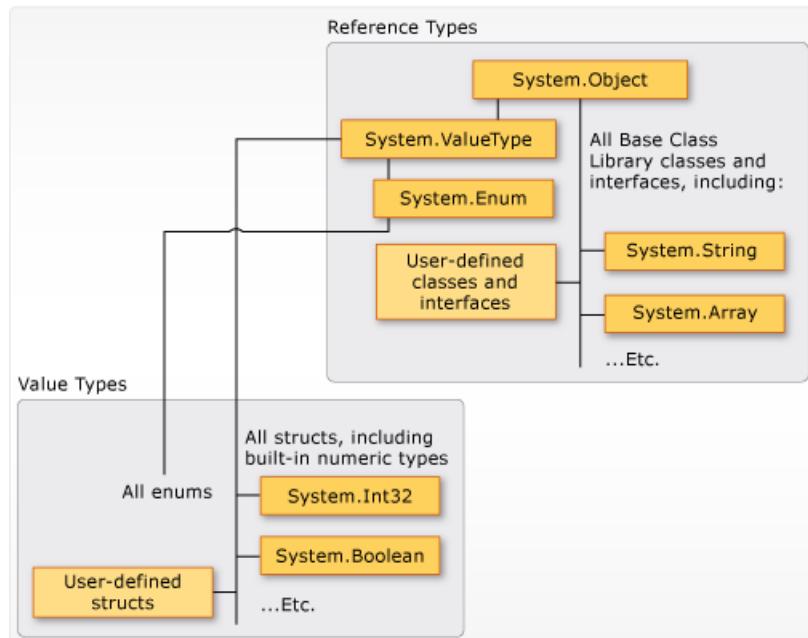
.NET의 형식 시스템에 대한 다음과 같은 두 가지 기초 사항을 이해해야 합니다.

- 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`)와 같은 기본 제공 숫자 형식을 포함한 모든 형식은 기본적으로 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`를 사용하여 정의한 형식은 값 형식이고, 모든 기본 제공 숫자 형식은 `structs`입니다. `class` 키워드를 사용하여 정의한 형식은 참조 형식입니다. 참조 형식과 값 형식의 컴파일 타입 규칙 및 런타임 동작은 서로 다릅니다.

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



#### NOTE

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.

## 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 포함되므로, 변수가 선언된 컨텍스트에 관계없이 메모리가 인라인으로 할당됩니다. 값 형식 변수에 대한 별도 힙 할당이나 가비지 수집 오버헤드는 없습니다.

값 형식에는 `struct` 및 `enum`의 두 가지 범주가 있습니다.

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

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

하지만 단순 비집계 형식처럼 값을 선언하고 변수에 할당합니다.

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

값 형식은 *sealed*입니다. 즉, `System.Int32`와 같은 값 형식에서 형식을 도출할 수 없습니다. 구조체는 `System.ValueType`에서만 상속할 수 있기 때문에 사용자 정의 클래스 또는 구조체에서 상속하는 구조체를 정의

할 수 없습니다. 그러나 구조체는 하나 이상의 인터페이스를 구현할 수 있습니다. 구조체 형식을 구조체가 구현하는 인터페이스 형식으로 캐스트할 수 있습니다. 이 캐스트로 인해 *boxing* 작업이 관리되는 힙의 참조 형식 자체 내에 구조체를 래핑합니다. *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.

```
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.

```
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

[클래스](#), [대리자](#), [배열](#) 또는 [인터페이스](#)로 정의되는 형식은 참조 형식입니다. 런타임에 참조 형식의 변수를 선언하면 `new` 연산자를 사용하여 개체를 명시적으로 만들거나 다음 예제와 같이 `new`를 사용하여 다른 곳에서 만들어진 개체를 할당할 때까지 변수에는 `null` 값이 포함됩니다.

```
MyClass mc = new MyClass();
MyClass mc2 = mc;
```

인터페이스는 구현하는 클래스 개체와 함께 초기화되어야 합니다. `MyClass` 가 [IMyInterface](#) 를 구현하는 경우 다음 예제와 같이 [IMyInterface](#) 의 인스턴스를 만듭니다.

```
IMyInterface iface = new MyClass();
```

개체가 만들어지면 관리되는 힙에 메모리가 할당되고 변수에는 개체 위치에 대한 참조만 포함됩니다. 관리되는

힙의 형식은 할당될 때, 그리고 가비지 수집이라는 CLR의 자동 메모리 관리 기능에 의해 회수될 때 오버헤드가 필요합니다. 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 파생되지만 다음 예제와 같이 C#에서 제공하는 단순한 구문을 사용하여 배열을 선언하고 사용할 수 있습니다.

```
// 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 상속될 수 있고 기타 클래스는 직접 만든 클래스에서 상속되고 가상 메서드를 재정의할 수 있습니다. 클래스를 직접 만드는 방법에 대한 자세한 내용은 [클래스 및 구조체](#)를 참조하세요. 상속 및 가상 메서드에 대한 자세한 내용은 [상속](#)을 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 4.56이 float로 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.

```
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

클라이언트 코드가 형식의 인스턴스를 만들 때 제공하는 실제 형식(구체적 형식)에 대한 자리 표시자로 사용되는 하나 이상의 형식 매개 변수를 사용하여 형식을 선언할 수 있습니다. 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: 문자열)을 지정합니다.

```
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 [개체](#)로 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

앞에서 설명한 대로 `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 형식을 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.

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

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

```
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.

- [캐스팅 및 형식 변환](#)
- [boxing 및 unboxing](#)
- [dynamic 형식 사용](#)
- [값 형식](#)
- [참조 형식](#)
- [클래스 및 구조체](#)
- [익명 형식](#)
- [제네릭](#)

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [XML 데이터 형식 변환](#)
- [정수 형식](#)

# 캐스팅 및 형식 변환(C# 프로그래밍 가이드)

2020-11-02 • 13 minutes to read • [Edit Online](#)

C#은 컴파일 시간에 정적으로 형식화되므로 변수가 선언된 후에는 다시 선언되거나 다른 형식의 값이 할당될 수 없습니다. 단, 형식이 변수의 형식으로 암시적으로 변환될 수 있는 경우는 예외입니다. 예를 들어 `string`은 `int`로 암시적으로 변환될 수 없습니다. 따라서 `i`를 `int`로 선언한 후에는 다음 코드와 같이 문자열 "Hello"를 할당할 수 없습니다.

```
int i;

// error CS0029: Cannot 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 변환**이라고 합니다. C#에서는 다음과 같은 변환을 수행할 수 있습니다.

- **암시적 변환**: 변환은 항상 성공하고 데이터가 손실되지 않으므로 특수 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 간의 명시적 및 암시적 변환을 사용하도록 정의할 수 있는 특수 메서드를 통해 수행됩니다. 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 및 기본 제공 숫자 형식(예: `Int32.Parse`)의 `Parse` 메서드를 사용하면 됩니다. 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.

```
// 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.

```
Derived d = new Derived();

// Always OK.
Base b = d;
```

## 명시적 변환

그러나 변환을 수행할 때 정보 손실의 위험이 있는 경우에는 컴파일러에서 **캐스트**라는 명시적 변환을 수행해야 합니다. 캐스트는 변환을 수행하고자 하고 데이터 손실이 발생하거나 런타임에 캐스트가 실패할 가능성을 알고 있음을 컴파일러에 명시적으로 알리는 방법입니다. 캐스트를 수행하려면 변환할 값 또는 변수 앞에 캐스트 할 형식을 괄호 안에 지정합니다. 다음 프로그램은 **double**을 **int**로 캐스트합니다. 캐스트가 없으면 프로그램이 컴파일되지 않습니다.

```
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.

```
// 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 간 캐스트 작업은 기본 개체의 런타임 형식을 변경하지 않습니다. 참조로 사용되는 값의 형식을 해당 개체로만 변경합니다. 자세한 내용은 [다형성](#)을 참조하세요.

## 런타임의 형식 변환 예외

일부 참조 형식 변환 시에는 컴파일러에서 캐스트가 유효한지 여부를 확인할 수 없습니다. 제대로 컴파일되는 캐스트 작업이 런타임에 실패할 수 있습니다. 다음 예제와 같이 런타임에 형식 캐스트가 실패하면 [InvalidCastException](#)이 throw됩니다.

```

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 생성하지 않고 형식을 확인하는 것이 더 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 사양](#)의 [전환](#) 섹션을 참조하세요.

## 참조

- [C# 프로그래밍 가이드](#)
- [유형](#)
- [캐스트 식](#)
- [사용자 정의 전환 연산자](#)
- [일반화된 형식 변환](#)
- [문자열을 숫자로 변환하는 방법](#)

# Boxing 및 Unboxing(C# 프로그래밍 가이드)

2020-11-02 • 9 minutes to read • [Edit Online](#)

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.

```
int i = 123;
// The following line boxes i.
object o = i;
```

그런 다음 `o` 개체를 unboxing하고 정수 변수 `i`에 할당할 수 있습니다.

```
o = 123;
i = (int)o; // unboxing
```

다음 예제에서는 C#에서 boxing이 사용되는 방법을 보여 줍니다.

```
byte[] array = { 0x64, 0x6f, 0x74, 0x63, 0x65, 0x74 };

string hexValue = Convert.ToString(array);
Console.WriteLine(hexValue);

/*Output:
646f74636574
*/
```

## 성능

단순 할당에서는 boxing과 unboxing을 수행하는 데 많은 계산 과정이 필요합니다. 값 형식을 boxing할 때는 새로운 개체를 할당하고 생성해야 합니다. 정도는 약간 덜 하지만 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.

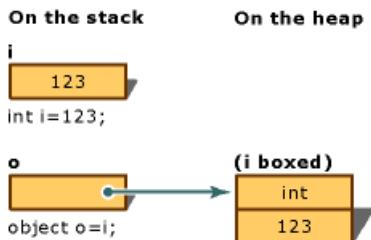
다음과 같이 값 형식 변수를 선언합니다.

```
int i = 123;
```

다음 문에서는 변수 `i`에 암시적으로 boxing 연산을 적용합니다.

```
// 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 값은 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.

```
int i = 123;
object o = (object)i; // explicit boxing
```

## 설명

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.

## 예제

```
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 형식`으로 명시적으로 변환하는 프로세스입니다. unboxing 연산 과정은 다음과 같습니다.

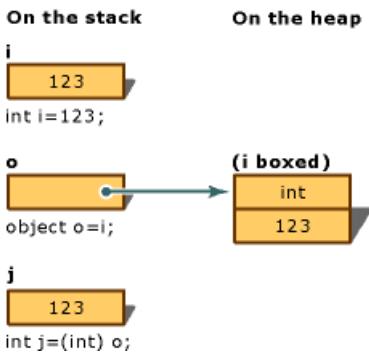
- 개체 인스턴스가 지정한 값 형식을 boxing한 값인지 확인합니다.

- 인스턴스의 값을 값 형식 변수에 복사합니다.

다음 문은 boxing 및 unboxing 연산을 모두 보여 줍니다.

```
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 `InvalidOperationException`을 보여 줍니다. 이 예제에서는 `try` 및 `catch`를 사용하여 오류가 발생할 때 오류 메시지를 표시합니다.

```
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

```
int j = (short) o;
```

다음과 같이 변경합니다.

```
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 소스 됩니다.

### 참조

- [C# 프로그래밍 가이드](#)
- [참조 형식](#)
- [값 형식](#)

# 바이트 배열을 int로 변환하는 방법(C# 프로그래밍 가이드)

2021-02-18 • 3 minutes to read • [Edit Online](#)

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.

반환되는 형식	메서드
<code>bool</code>	<code>ToBoolean(Byte[], Int32)</code>
<code>char</code>	<code>ToChar(Byte[], Int32)</code>
<code>double</code>	<code>ToDouble(Byte[], Int32)</code>
<code>short</code>	<code>ToInt16(Byte[], Int32)</code>
<code>int</code>	<code>ToInt32(Byte[], Int32)</code>
<code>long</code>	<code>ToInt64(Byte[], Int32)</code>
<code>float</code>	<code>ToSingle(Byte[], Int32)</code>
<code>ushort</code>	<code>ToUInt16(Byte[], Int32)</code>
<code>uint</code>	<code>ToUInt32(Byte[], Int32)</code>
<code>ulong</code>	<code>ToUInt64(Byte[], Int32)</code>

## 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.

### NOTE

출력은 컴퓨터 아키텍처의 endianness에 따라 달라질 수 있습니다.

```
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
```

## 예제

이 예제에서는 [BitConverter](#) 클래스의 [GetBytes\(Int32\)](#) 메서드를 호출하여 `int` 를 바이트 배열로 변환합니다.

### NOTE

출력은 컴퓨터 아키텍처의 endianness에 따라 달라질 수 있습니다.

```
byte[] bytes = BitConverter.GetBytes(201805978);
Console.WriteLine("byte array: " + BitConverter.ToString(bytes));
// Output: byte array: 9A-50-07-0C
```

## 참조

- [BitConverter](#)
- [IsLittleEndian](#)
- [형식](#)

# 문자열을 숫자로 변환하는 방법(C# 프로그래밍 가이드)

2021-02-18 • 9 minutes to read • [Edit Online](#)

다양한 숫자 형식(`int`, `long`, `double` 등)에 있는 `Parse` 또는 `TryParse` 메서드를 호출하거나 `System.Convert` 클래스의 메서드를 사용하여 문자열을 숫자로 변환할 수 있습니다.

`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 `Boolean` 값을 반환하고 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 이벤트에서 `FormatException`을 catch해야 합니다.

## Parse 및 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 10 을 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 보여 줍니다.

```

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 포함된 문자 수가 적으므로 예제에서는 [String.Concat](#) 메서드를 호출하

여 새 문자열에 유효한 문자를 할당합니다. 더 큰 문자열의 경우 [StringBuilder](#) 클래스를 대신 사용할 수 있습니다.

```
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
    }
}
```

## 변환 메서드 호출

다음 표에는 문자열을 숫자로 변환하는 데 사용할 수 있는 [Convert](#) 클래스의 일부 메서드가 나와 있습니다.

숫자 형식	메서드
<code>decimal</code>	<a href="#">ToDecimal(String)</a>
<code>float</code>	<a href="#">ToSingle(String)</a>

double	ToDouble(String)
short	ToInt16(String)
int	ToInt32(String)
long	ToInt64(String)
ushort	ToUInt16(String)
uint	ToUInt32(String)
ulong	ToUInt64(String)

다음 예제에서는 [Convert.ToInt32\(String\)](#) 메서드를 호출하여 입력 문자열을 [int](#)로 변환합니다. 예제에서는 이 메서드에서 throw 할 수 있는 두 가지 가장 일반적인 예외인 [FormatException](#)과 [OverflowException](#)을 catch합니다. [Int32.MaxValue](#)을 초과하지 않고 결과 숫자를 증분할 수 있는 경우 예제에서는 결과에 1을 더하고 출력을 표시합니다.

```

using System;

public class ConvertStringExample1
{
    static void Main(string[] args)
    {
        int numVal = -1;
        bool repeat = true;

        while (repeat)
        {
            Console.WriteLine("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

```

## 참조

- 형식
- 문자열이 숫자 값을 나타내는지 확인 방법
- 샘플:.NET Core WinForms 서식 유틸리티(C#)

# 16진수 문자열과 숫자 형식 간 변환 방법(C# 프로그래밍 가이드)

2021-02-18 • 6 minutes to read • [Edit Online](#)

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.

```
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
*/
```

## 예제

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`으로 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.

```

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 !
*/

```

## 예제

이 예제에서는 [Parse\(String, NumberStyles\)](#) 메서드를 호출하여 16진수 `string` 을 정수로 변환하는 방법을 보여 줍니다.

```

string hexString = "8E2";
int num = Int32.Parse(hexString, System.Globalization.NumberStyles.HexNumber);
Console.WriteLine(num);
//Output: 2274

```

## 예제

다음 예제에서는 [System.BitConverter](#) 클래스 및 [UInt32.Parse](#) 메서드를 사용하여 16진수 `string` 을 `float`로 변환하는 방법을 보여 줍니다.

```

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

```

## 예제

다음 예제에서는 [System.BitConverter](#) 클래스를 사용하여 `byte` 배열을 16진수 문자열로 변환하는 방법을 보여 줍니다.

```
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
*/
```

## 예제

다음 예제에서는 .NET 5.0에 도입된 [Convert.ToString](#) 메서드를 호출하여 byte 배열을 16진수 문자열로 변환하는 방법을 보여 줍니다.

```
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](#)

# dynamic 형식 사용(C# 프로그래밍 가이드)

2020-11-02 • 12 minutes to read • [Edit Online](#)

C# 4에서는 새로운 `dynamic` 형식이 도입되었습니다. 이 형식은 정적 형식이지만 `dynamic` 형식의 개체가 정적 형식 검사를 건너뜁니다. 대부분의 경우 이 형식은 `object` 형식을 가지고 있는 것처럼 작동합니다. 컴파일 시간에 `dynamic` 형식의 요소는 모든 연산을 지원하는 것으로 간주됩니다. 따라서 개체가 값을 COM API, IronPython 같은 동적 언어, HTML DOM(문서 개체 모델), 리플렉션 또는 프로그램의 다른 곳 등 어디서 가져오든 신경을 쓸 필요가 없습니다. 그러나 코드가 유효하지 않으면 런타임 시 오류가 catch됩니다.

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. 컴파일러는 메서드 `dynamic_ec.exampleMethod1(10, 4)`에 대한 두 번째 호출을 확인하지 않습니다. `dynamic_ec`의 형식이 `dynamic`이기 때문입니다. 따라서 컴파일러 오류가 보고되지 않습니다. 그러나 이 오류는 알림을 무기한 이스케이프하지 않고, 런타임에 catch되며 런타임 예외를 일으킵니다.

```
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();
}
```

```
class ExampleClass
{
    public ExampleClass() { }
    public ExampleClass(int v) { }

    public void exampleMethod1(int i) { }

    public void exampleMethod2(string str) { }
}
```

이 예제에서 컴파일러의 역할은 `dynamic`으로 형식이 지정된 개체 또는 식에 대해 각 문이 해야 할 일에 대한 정보를 패키지하는 것입니다. 런타임에는 저장된 정보의 검사가 수행되며, 유효하지 않은 문에서 런타임 예외가 발생합니다.

대부분의 동적 작업은 결과 그 자체가 `dynamic`입니다. 다음 예제에서 `testSum`이 사용된 곳에 마우스 포인터를 올려두면 IntelliSense에서 (지역 변수) `dynamic testSum` 형식을 표시합니다.

```
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 다음을 포함합니다.

- `dynamic` 에서 다른 형식으로의 전환.
- `dynamic` 형식의 인수를 포함하는 생성자 호출.

예를 들어 다음 선언에서 `testInstance` 의 형식은 `dynamic` 이 아니라 `ExampleClass` 입니다.

```
var testInstance = new ExampleClass(d);
```

변환 예제는 다음 섹션인 "변환"에 나와 있습니다.

## 변환

동적 개체와 다른 형식 간에 손쉽게 변환할 수 있습니다. 따라서 개발자는 동적 동작과 비동적 동작 간에 전환할 수 있습니다.

다음 예제와 같이 개체를 동적 형식으로 암시적으로 변환할 수 있습니다.

```
dynamic d1 = 7;
dynamic d2 = "a string";
dynamic d3 = System.DateTime.Today;
dynamic d4 = System.Diagnostics.Process.GetProcesses();
```

반대로, 암시적 변환을 `dynamic` 형식의 식에 동적으로 적용할 수 있습니다.

```
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 예제에서 액세스 가능한 유일한 `exampleMethod2` 메서드가 문자열 인수를 사용하도록 정의되는 경우, `d1`을 인수로서 전송하면 컴파일러 오류는 발생하지 않지만 런타임 예외가 발생합니다. `d1`의 런타임 형식은 `int` 인데 `exampleMethod2`에는 문자열이 필요하므로 오버로드 확인이 런타임에 실패합니다.

```
// 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 .NET Framework 4에 도입된 API입니다. DLR은 C#에서 `dynamic` 형식을 지원하는 인프라를 제공하며, IronPython 및 IronRuby와 같은 동적 프로그래밍 언어를 구현합니다. DLR에 대한 자세한 내용은 [동적 언어 런타임 개요](#)를 참조하세요.

## COM interop

C# 4에는 Office Automation API 등의 COM API와 상호 운용 환경을 개선하는 몇 가지 기능이 포함되어 있습니다. 개선 사항 중에는 `dynamic` 형식의 사용 및 명명된 인수 및 선택적 인수의 사용이 포함됩니다.

많은 COM 메서드는 형식을 `object`로 지정하여 인수 형식 및 반환 형식의 변환을 허용합니다. C#에서는 강력한 형식의 변수로 조정하기 위해 값을 명시적으로 캐스팅해야 합니다. [-link\(C# 컴파일러 옵션\)](#) 옵션을 사용하여 컴파일하는 경우 `dynamic` 형식을 사용하면 COM 서명에서 `object`의 발생을 마치 `dynamic` 형식인 것처럼 취급하여 캐스팅을 상당 부분 피할 수 있습니다. 예를 들어 다음 문은 `dynamic` 형식은 있고 `dynamic` 형식은 없는 Microsoft Office Excel 스프레드시트의 셀에 액세스하는 방법과 대조됩니다.

```
// Before the introduction of dynamic.  
((Excel.Range)excelApp.Cells[1, 1]).Value2 = "Name";  
Excel.Range range2008 = (Excel.Range)excelApp.Cells[1, 1];
```

```
// After the introduction of dynamic, the access to the Value property and  
// the conversion to Excel.Range are handled by the run-time COM binder.  
excelApp.Cells[1, 1].Value = "Name";  
Excel.Range range2010 = excelApp.Cells[1, 1];
```

## 관련 항목

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 만드는 데 필요한 단계별 지침을 제공합니다.
<a href="#">C# 기능을 사용하여 Office interop 개체에 액세스하는 방법</a>	명명된 인수와 선택적 인수, <code>dynamic</code> 형식, Office API 개체에 대한 액세스를 간소화하는 기타 향상된 기능을 사용하는 프로젝트를 만드는 방법을 보여 줍니다.

# 연습: 동적 개체 만들기 및 사용(C# 및 Visual Basic)

2021-02-18 • 26 minutes to read • [Edit Online](#)

동적 개체는 컴파일 시간이 아닌 런타임에 속성 및 메서드와 같은 멤버를 노출합니다. 이를 통해 형식 또는 정적 형식과 일치하지 않는 구조와 작동할 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 참조 할 수 있습니다.

동적 개체를 사용하면 IronPython 및 IronRuby와 같은 동적 언어에 편리하게 액세스할 수 있습니다. 동적 개체를 사용하면 런타임에 해석되는 동적 스크립트를 참조할 수 있습니다.

런타임에 바인딩을 사용하여 동적 개체를 참조합니다. C#에서는 런타임에 바인딩된 개체의 형식을 `dynamic`으로 지정합니다. Visual Basic에서는 런타임에 바인딩된 개체의 형식을 `Object`으로 지정합니다. 자세한 내용은 `dynamic` 및 [초기 바인딩 및 런타임에 바인딩](#)을 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 연습에서는 다음 작업을 수행합니다.

- 텍스트 파일의 내용을 개체의 속성으로서 동적으로 노출하는 사용자 지정 개체를 만듭니다.
- `IronPython` 라이브러리를 사용하는 프로젝트를 만듭니다.

## 사전 요구 사항

이 연습을 완료하려면 .NET용 `IronPython`이 필요합니다. [다운로드 페이지](#)로 이동하여 최신 버전을 다운로드하세요.

### NOTE

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.

## 사용자 지정 동적 개체 만들기

이 연습에서 만드는 첫 번째 프로젝트는 텍스트 파일의 내용을 검색하는 사용자 지정 동적 개체를 정의합니다. 검색할 텍스트는 동적 속성의 이름으로 지정됩니다. 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 `dynamicFile.Sample(StringSearchOption.StartsWith, false)`을 지정하면 동적 클래스는 각

줄의 시작 부분에서 "Sample"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 클래스를 만들려면

1. Visual Studio를 시작합니다.
2. 파일 메뉴에서 새로 만들기 를 가리킨 다음 프로젝트 를 클릭합니다.
3. 새 프로젝트 대화 상자의 프로젝트 형식 창에서 Windows 가 선택되었는지 확인합니다. 템플릿 창에서 콘솔 애플리케이션 을 선택합니다. 이름 상자에 DynamicSample 를 입력한 다음 확인 을 클릭합니다. 새 프로젝트가 만들어집니다.
4. DynamicSample 프로젝트를 마우스 오른쪽 단추로 클릭하고 추가 를 가리킨 다음 클래스 를 클릭합니다. 이름 상자에 ReadOnlyFile 을 입력한 다음 확인 을 클릭합니다. ReadOnlyFile 클래스가 포함된 새 파일이 추가됩니다.
5. ReadOnlyFile.cs 또는 ReadOnlyFile.vb 파일 맨 위에 다음 코드를 추가하여 System.IO 및 System.Dynamic 네임스페이스를 가져옵니다.

```
using System.IO;
using System.Dynamic;
```

```
Imports System.IO
Imports System.Dynamic
```

6. 사용자 지정 동적 개체는 열거형을 사용하여 검색 기준을 결정합니다. Class 문 앞에 다음 열거형 정의를 추가합니다.

```
public enum StringSearchOption
{
    StartsWith,
    Contains,
    EndsWith
}
```

```
Public Enum StringSearchOption
    StartsWith
    Contains
    EndsWith
End Enum
```

7. 다음 코드 예제와 같이, DynamicObject 클래스를 상속하도록 class 문을 업데이트합니다.

```
class ReadOnlyFile : DynamicObject
```

```
Public Class ReadOnlyFile
    Inherits DynamicObject
```

8. 다음 코드를 ReadOnlyFile 클래스에 추가하여 파일 경로의 전용 필드 및 ReadOnlyFile 클래스의 생성자를 정의합니다.

```
// 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;  
}
```

```
' Store the path to the file and the initial line count value.  
Private p_filePath As String  
  
' Public constructor. Verify that file exists and store the path in  
' the private variable.  
Public Sub New(ByVal filePath As String)  
    If Not File.Exists(filePath) Then  
        Throw New Exception("File path does not exist.")  
    End If  
  
    p_filePath = filePath  
End Sub
```

9. 다음 `GetPropertyValues` 메서드를 `ReadOnlyFile` 클래스에 추가합니다. `GetPropertyValues` 메서드는 검색 기준을 입력으로 가져오고 해당 검색 기준과 일치하는 텍스트 파일로부터 줄을 반환합니다.
- `ReadOnlyFile` 클래스가 제공하는 동적 메서드는 `GetPropertyValues` 메서드를 호출하여 각 결과를 검색합니다.

```
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;
}
```

```

Public Function GetPropertyValue(ByVal propertyName As String,
                               Optional ByVal StringSearchOption As StringSearchOption =
StringSearchOption.StartsWith,
                               Optional ByVal trimSpaces As Boolean = True) As List(Of String)

    Dim sr As StreamReader = Nothing
    Dim results As New List(Of String)
    Dim line = ""
    Dim testLine = ""

    Try
        sr = New StreamReader(p_filePath)

        While Not sr.EndOfStream
            line = sr.ReadLine()

            ' Perform a case-insensitive search by using the specified search options.
            testLine = UCASE(line)
            If trimSpaces Then testLine = Trim(testLine)

            Select Case StringSearchOption
                Case StringSearchOption.StartsWith
                    If testLine.StartsWith(UCASE(propertyName)) Then results.Add(line)
                Case StringSearchOption.Contains
                    If testLine.Contains(UCASE(propertyName)) Then results.Add(line)
                Case StringSearchOption.EndsWith
                    If testLine.EndsWith(UCASE(propertyName)) Then results.Add(line)
            End Select
        End While
    Catch
        ' Trap any exception that occurs in reading the file and return Nothing.
        results = Nothing
    Finally
        If sr IsNot Nothing Then sr.Close()
    End Try

    Return results
End Function

```

10. `GetPropertyValue` 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.

```

// 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;
}

```

```

' Implement the TryGetMember method of the DynamicObject class for dynamic member calls.
Public Overrides Function TryGetMember(ByVal binder As GetMemberBinder,
                                       ByRef result As Object) As Boolean
    result = GetPropertyValue(binder.Name)
    Return If(result Is Nothing, False, True)
End Function

```

11. `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 값인 경우 결과를 검색할 수 있도록 `GetPropertyValues` 메서드로 전달됩니다.

```
// 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 = GetPropertyValues(binder.Name, StringSearchOption, trimSpaces);

    return result == null ? false : true;
}
```

```

' Implement the TryInvokeMember method of the DynamicObject class for
' dynamic member calls that have arguments.
Public Overrides Function TryInvokeMember(ByName binder As InvokeMemberBinder,
                                         ByVal args() As Object,
                                         ByRef result As Object) As Boolean

    Dim StringSearchOption As StringSearchOption = StringSearchOption.StartsWith
    Dim trimSpaces = True

    Try
        If args.Length > 0 Then StringSearchOption = CType(args(0), StringSearchOption)
    Catch
        Throw New ArgumentException("StringSearchOption argument must be a StringSearchOption enum
value.")
    End Try

    Try
        If args.Length > 1 Then trimSpaces = CType(args(1), Boolean)
    Catch
        Throw New ArgumentException("trimSpaces argument must be a Boolean value.")
    End Try

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces)

    Return If(result Is Nothing, False, True)
End Function

```

## 12. 파일을 저장한 후 닫습니다.

샘플 텍스트 파일을 만들려면

- DynamicSample 프로젝트를 마우스 오른쪽 단추로 클릭하고 추가 를 가리킨 다음 새 항목 을 클릭합니다. 설치된 템플릿 창에서 일반 을 선택한 다음 텍스트 파일 템플릿을 선택합니다. 이름 상자에 있는 기본 이름 TextFile1.txt를 그대로 두고 추가 를 클릭합니다. 새 텍스트 파일이 프로젝트에 추가됩니다.
- TextFile1.txt 파일에 다음 텍스트를 복사합니다.

```

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

```

## 3. 파일을 저장한 후 닫습니다.

사용자 지정 동적 개체를 사용하는 샘플 애플리케이션을 만들려면

- 솔루션 탐색기 에서, Visual Basic을 사용 중인 경우 Module1.vb 파일, Visual C#을 사용 중인 경우 Program.cs 파일을 두 번 클릭합니다.
- Main 프로시저에 다음 코드를 추가하여 TextFile1.txt 파일에 대한 `ReadOnlyFile` 클래스의 인스턴스를 만듭니다. 코드는 런타임에 바인딩을 사용하여 동적 멤버를 호출하고 "Customer" 문자열이 포함된 텍스트의 줄을 검색합니다.

```

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);
}

```

```

Dim rFile As Object = New ReadOnlyFile("../..\\TextFile1.txt")
For Each line In rFile.Customer
    Console.WriteLine(line)
Next
Console.WriteLine("-----")
For Each line In rFile.Customer(StringSearchOption.Contains, True)
    Console.WriteLine(line)
Next

```

- 파일을 저장한 다음 Ctrl+F5를 눌러 애플리케이션을 빌드하고 실행합니다.

## 동적 언어 라이브러리 호출

이 연습에서 만드는 새 프로젝트는 동적 언어 IronPython에서 작성된 라이브러리에 액세스합니다.

사용자 지정 동적 클래스를 만들려면

- Visual Studio의 파일 메뉴에서 새로 만들기 를 가리킨 다음 프로젝트 를 클릭합니다.
- 새 프로젝트 대화 상자의 프로젝트 형식 창에서 Windows 가 선택되었는지 확인합니다. 템플릿 창에서 콘솔 애플리케이션 을 선택합니다. 이름 상자에 DynamicIronPythonSample 를 입력한 다음 확인 을 클릭합니다. 새 프로젝트가 만들어집니다.
- Visual Basic을 사용 중인 경우 DynamicIronPythonSample 프로젝트를 마우스 오른쪽 단추로 클릭하고 속성을 클릭합니다. 참조 탭을 클릭합니다. 추가 단추를 클릭합니다. Visual C#을 사용 중인 경우 솔루션 탐색기 에서 References 폴더를 마우스 오른쪽 단추로 클릭한 다음 참조 추가 를 클릭합니다.
- 찾아보기 탭에서 IronPython 라이브러리가 설치된 폴더로 이동합니다. 예를 들어 .NET Framework 4.0의 경우 C:\Program Files\IronPython 2.6입니다. IronPython.dll, IronPython.Modules.dll, Microsoft.Scripting.dll 및 Microsoft.Dynamic.dll 라이브러리를 선택합니다. 확인 을 클릭합니다.
- Visual Basic을 사용 중인 경우 Module1.vb 파일을 편집합니다. Visual C#을 사용 중인 경우 Program.cs 파일을 편집합니다.
- 파일 맨 위에 다음 코드를 추가하여 IronPython 라이브러리에서 Microsoft.Scripting.Hosting 및 IronPython.Hosting 네임스페이스를 가져옵니다.

```

using Microsoft.Scripting.Hosting;
using IronPython.Hosting;

```

```

Imports Microsoft.Scripting.Hosting
Imports IronPython.Hosting

```

- Main 메서드에서 다음 코드를 추가하여 IronPython 라이브러리를 호스트하기 위한 새 Microsoft.Scripting.Hosting.ScriptRuntime 개체를 만듭니다. ScriptRuntime 개체는 IronPython 라이브러리 모듈 random.py를 로드합니다.

```

// Set the current directory to the IronPython libraries.
System.IO.Directory.SetCurrentDirectory(
    Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles) +
    @"\IronPython 2.6 for .NET 4.0\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.");

```

```

' Set the current directory to the IronPython libraries.
My.Computer.FileSystem.CurrentDirectory =
    My.Computer.FileSystem.SpecialDirectories.ProgramFiles &
    "\IronPython 2.6 for .NET 4.0\Lib"

' Create an instance of the random.py IronPython library.
Console.WriteLine("Loading random.py")
Dim py = Python.CreateRuntime()
Dim random As Object = py.UseFile("random.py")
Console.WriteLine("random.py loaded.")

```

8. 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.

```

// 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("-----");
}

```

```

' Initialize an enumerable set of integers.
Dim items = Enumerable.Range(1, 7).ToArray()

' Randomly shuffle the array of integers by using IronPython.
For i = 0 To 4
    random.shuffle(items)
    For Each item In items
        Console.WriteLine(item)
    Next
    Console.WriteLine("-----")
Next

```

9. 파일을 저장한 다음 Ctrl+F5를 눌러 애플리케이션을 빌드하고 실행합니다.

## 참조

- [System.Dynamic](#)
- [System.Dynamic.DynamicObject](#)
- [dynamic 형식 사용](#)

- 초기 바인딩 및 런타임에 바인딩
- dynamic
- 동적 인터페이스 구현(Microsoft TechNet에서 다운로드 가능한 PDF)

# 클래스 및 구조체(C# 프로그래밍 가이드)

2020-11-02 • 18 minutes to read • [Edit Online](#)

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. 데이터 및 동작은 클래스 또는 구조체의 멤버로, 이 항목의 뒷부분에 나오는 것처럼 메서드, 속성 및 이벤트 등을 포함합니다.

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

클래스는 참조 형식입니다. 클래스의 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.

일반적으로 클래스는 좀 더 복잡한 동작이나 클래스 개체를 만든 후 수정하려는 데이터를 모델링하는데 사용됩니다. 구조체는 구조체를 만든 후에 수정하지 않으려는 데이터를 주로 포함하는 작은 데이터 구조에 가장 적합합니다.

자세한 내용은 [클래스, 개체 및 구조 형식](#)을 참조하세요.

## 예제

다음 예제에서 `ProgrammingGuide` 네임스페이스의 `CustomClass`에는 세 개의 멤버, 즉 인스턴스 생성자, `Number`라는 속성 및 `Multiply`이라는 메서드가 있습니다. `Program` 클래스의 `Main` 메서드는 `CustomClass`의 인스턴스(개체)를 만들고 개체의 메서드 및 속성을 점 표기법을 통해 액세스합니다.

```

using System;

namespace ProgrammingGuide
{
    // Class definition.
    public class CustomClass
    {
        // Class members.
        //
        // Property.
        public int Number { get; set; }

        // Method.
        public int Multiply(int num)
        {
            return num * Number;
        }

        // Instance Constructor.
        public CustomClass()
        {
            Number = 0;
        }
    }

    // Another class definition that contains Main, the program entry point.
    class Program
    {
        static void Main(string[] args)
        {
            // Create an object of type CustomClass.
            CustomClass custClass = new CustomClass();

            // Set the value of the public property.
            custClass.Number = 27;

            // Call the public method.
            int result = custClass.Multiply(4);
            Console.WriteLine($"The result is {result}.");
        }
    }
}

// The example displays the following output:
//     The result is 108.

```

## 캡슐화

캡슐화는 경우에 따라 개체 지향 프로그래밍의 첫 번째 pillar 또는 원리로 인식됩니다. 캡슐화의 원리에 따라 클래스 또는 구조체는 클래스 또는 구조체 외부의 코드에 각 멤버가 액세스하는 방법을 지정할 수 있습니다. 코딩 오류 또는 악의적인 악용 가능성을 제한하려면 클래스 또는 어셈블리 외부에서 사용하지 않으려는 메서드 및 변수는 숨길 수 있습니다.

클래스에 대한 자세한 내용은 [클래스](#) 및 [개체](#)를 참조하세요.

### 멤버

모든 메서드, 필드, 상수, 속성 및 이벤트는 형식 내에서 선언되어야 합니다. 이것을 형식의 **멤버**라고 합니다. 다른 언어에는 있지만 C#에는 전역 변수 또는 메서드가 없습니다. 프로그램의 진입점인 `Main` 메서드까지도 클래스 또는 구조체 내에서 선언되어야 합니다. 다음은 클래스 또는 구조체에서 선언될 수 있는 모든 다양한 종류의 멤버입니다.

- [필드](#)

- [상수](#)

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

### 액세스 가능성

일부 메서드 및 속성은 [클라이언트 코드](#)라고 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. 형식 및 해당 멤버가 클라이언트 코드에 액세스하는 방법은 액세스 한정자 [public](#), [protected](#), [internal](#), [protected internal](#), [private](#) 및 [private protected](#)를 사용하여 지정합니다. 기본 액세스 가능성은 [private](#)입니다. 자세한 내용은 [액세스 한정자](#)를 참조하세요.

### 상속

클래스(구조체는 아님)는 상속 개념을 지원합니다. 다른 클래스(기본 클래스)에서 파생되는 클래스는 생성자와 종료자를 제외하고 기본 클래스의 모든 [public](#), [protected](#) 및 [internal](#) 멤버를 자동으로 포함합니다. 자세한 내용은 [상속](#) 및 [다형성](#)을 참조하세요.

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 [Abstract 및 Sealed 클래스와 클래스 멤버](#)를 참조하세요.

### 인터페이스

클래스 및 구조체는 여러 인터페이스에서 상속할 수 있습니다. 인터페이스에서 상속하는 것은 형식이 해당 인터페이스에 정의된 모든 메서드를 구현한다는 것입니다. 자세한 내용은 [인터페이스](#)를 참조하세요.

### 제네릭 형식

하나 이상의 형식 매개 변수를 사용하여 클래스 및 구조체를 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(구조체는 아님)를 [static](#)으로 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. 클래스와 구조체 둘 다 정적 멤버를 포함할 수 있습니다. 자세한 내용은 [static 클래스 및 static 클래스 멤버](#)를 참조하세요.

### 중첩 형식

클래스 또는 구조체는 다른 클래스 또는 구조체 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 [Partial 클래스 및 메서드](#)를 참조하세요.

## 개체 이니셜라이저

해당 생성자를 명시적으로 호출하지 않고 클래스 또는 구조체 개체, 개체 컬렉션을 인스턴스화하고 초기화할 수 있습니다. 자세한 내용은 [개체 및 컬렉션 이니셜라이저](#)를 참조하세요.

## 익명 형식

유지하거나 다른 메서드에 전달할 필요가 없는 데이터 구조로 목록을 채우는 경우처럼 명명된 클래스를 만드는 것이 불편하거나 필요하지 않은 상황에서는 무명 형식을 사용합니다. 자세한 내용은 [무명 형식](#)을 참조하세요.

## 확장명 메서드

마치 원래 형식에 속하는 것처럼 해당 메서드를 호출할 수 있는 별도 형식을 만들면 파생 클래스를 만들지 않고도 클래스를 “확장”할 수 있습니다. 자세한 내용은 [확장 메서드](#)를 참조하세요.

## 암시적으로 형식화한 지역 변수

클래스 또는 구조체 메서드 내에서 암시적 형식 지정을 사용하여 컴파일러가 컴파일 타임에 올바른 형식을 결정하도록 할 수 있습니다. 자세한 내용은 [암시적으로 형식화된 지역 변수](#)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](#)

# 클래스(C# 프로그래밍 가이드)

2021-02-18 • 13 minutes to read • [Edit Online](#)

## 참조 형식

클래스로 정의된 형식은 참조 형식입니다. 런타임에 참조 형식의 변수를 선언하면 `new` 연산자를 사용하여 클래스의 인스턴스를 명시적으로 만들거나 다음 예제와 같이 다른 곳에서 만들어진 호환성 있는 형식의 개체를 할당할 때까지 변수에는 `null` 값이 포함됩니다.

```
//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. 관리되는 힙의 형식은 할당될 때, 그리고 가비지 수집이라는 CLR의 자동 메모리 관리 기능에 의해 회수될 때 오버헤드가 필요합니다. 그러나 가비지 수집은 고도로 최적화되고 대부분 시나리오에서 성능 문제를 일으키지 않습니다. 가비지 수집에 대한 자세한 내용은 [자동 메모리 관리 및 가비지 수집](#)을 참조하세요.

## 클래스 선언

클래스는 다음 예제와 같이 `class` 키워드 다음에 고유 식별자를 사용하여 선언됩니다.

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

`class` 키워드는 액세스 수준 뒤에 옵니다. 이 경우 `public`이 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

클래스와 개체는 바꿔 사용되기도 하지만 서로 다른 항목입니다. 클래스는 개체의 형식을 정의하지만 개체 자체는 아닙니다. 개체는 클래스에 기반을 둔 구체적 엔터티이고 클래스의 인스턴스라고도 합니다.

개체를 만들려면 다음과 같이 `new` 키워드 뒤에 개체의 기반이 되는 클래스의 이름을 사용합니다.

```
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 자체를 포함하지 않습니다. 실제로 개체를 만들지 않고도 개체 참조를 만들 수 있습니다.

```
Customer object2;
```

런타임에는 참조를 통한 개체 액세스 시도에 실패하므로 개체를 참조하지 않는 이와 같은 개체 참조는 만들지 않는 것이 좋습니다. 그러나 새 개체를 만들거나 다음과 같이 기존 개체를 할당하여 개체를 참조하는 참조를 만들 수 있습니다.

```
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.

## 클래스 상속

클래스는 개체 지향 프로그래밍의 기본적인 특성인 '상속'을 완전히 지원합니다. 클래스를 만들 때 `sealed`로 정의되지 않은 다른 클래스에서 상속될 수 있으며 다른 클래스는 직접 만든 클래스에서 상속되고 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 클래스를 사용하여 선언됩니다. 다음과 같이 파생 클래스 이름 뒤에 콜론 및 기본 클래스 이름을 추가하여 기본 클래스를 지정합니다.

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

기본 클래스를 선언하는 클래스는 생성자를 제외하고 기본 클래스의 모든 멤버를 상속합니다. 자세한 내용은 [상속](#)을 참조하세요.

C++와 달리 C#의 클래스는 하나의 기본 클래스에서만 직접 상속될 수 있습니다. 그러나 기본 클래스 자체는 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`로 선언될 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. 이와 달리 `sealed` 클래스에서는 다른 클래스가 파생될 수 없습니다. 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.

## 예제

다음 예제에서는 [자동 구현 속성](#), 메서드 및 생성자라는 특수 메서드를 포함하는 공용 클래스를 정의합니다. 자세한 내용은 [속성, 메서드 및 생성자](#) 항목을 참조하세요. 그런 다음, 클래스의 인스턴스는 `new` 키워드를 사용하여 인스턴스화됩니다.

```

using System;

public class Person
{
    // Constructor that takes no arguments:
    public Person()
    {
        Name = "unknown";
    }

    // Constructor that takes one argument:
    public Person(string name)
    {
        Name = name;
    }

    // Auto-implemented readonly property:
    public string Name { get; }

    // Method that overrides the base class (System.Object) implementation.
    public override string ToString()
    {
        return Name;
    }
}
class TestPerson
{
    static void Main()
    {
        // Call the constructor that has no parameters.
        var person1 = new Person();
        Console.WriteLine(person1.Name);

        // Call the constructor that has one parameter.
        var person2 = new Person("Sarah Jones");
        Console.WriteLine(person2.Name);
        // Get the string representation of the person2 instance.
        Console.WriteLine(person2);

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// unknown
// Sarah Jones
// Sarah Jones

```

## C# 언어 사양

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

## 참조

- [C# 프로그래밍 가이드](#)
- [개체 지향 프로그래밍](#)
- [다형성](#)
- [식별자 이름](#)
- [멤버](#)
- [메서드](#)

- 생성자
- 종료자
- 개체

# 개체(C# 프로그래밍 가이드)

2021-02-18 • 12 minutes to read • [Edit Online](#)

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.

## NOTE

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 개체가 첫 번째 개체에 할당되면 두 변수가 모두 해당 주소의 개체를 참조합니다. 이 내용에 대해서는 이 항목의 뒷 부분에서 자세히 설명합니다.

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

```

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);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
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.

```

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);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Output:
p1 Name = Alex Age = 9
p2 Name = Spencer Age = 7
p1 Name = Alex Age = 9
*/

```

`p1` 및 `p2` 둘 다에 대한 메모리가 스레드 스택에서 할당됩니다. 해당 메모리가 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++에서와 같이 클래스 개체를 자동으로 제거할 수 없습니다. .NET의 가비지 수집에 대한 자세한 내용은 [가비지 수집](#)을 참조하세요.

#### NOTE

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 [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.

```
// 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;
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 같은 알고리즘을 제공하는 방법에 대한 자세한 내용은 [형식의 값 같음을 정의하는 방법](#)을 참조하세요.

- 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 단원

추가 정보

- [클래스](#)
- [생성자](#)
- [종료자](#)
- [이벤트](#)

## 참조

- [C# 프로그래밍 가이드](#)
- [object](#)
- [상속](#)
- [class](#)

- 구조체 형식
- new 연산자
- 공용 형식 시스템

# 상속(C# 프로그래밍 가이드)

2020-11-02 • 16 minutes to read • [Edit Online](#)

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 클래스라고 하고 해당 멤버를 상속하는 클래스를 파생 클래스라고 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.

## NOTE

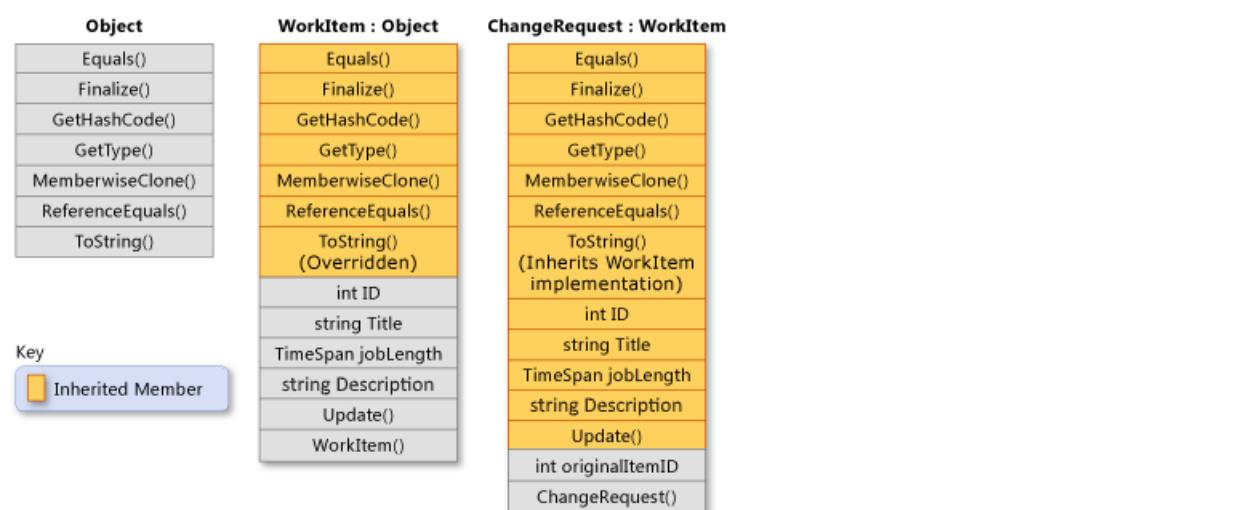
구조체는 상속을 지원하지 않지만 인터페이스를 구현할 수 있습니다. 자세한 내용은 [인터페이스](#)를 참조하세요.

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 언어 참조 섹션에서 [인터페이스](#)에 대한 문서를 참조하세요.

다른 클래스에서 파생 할 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` 은 고유한 멤버 5개를 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.

```

// 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 GetNextID 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.

```

// 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. 자세한 내용은 [Abstract 및 Sealed 클래스와 클래스 멤버](#)를 참조하세요.

## 인터페이스

'인터페이스'는 멤버 집합을 정의하는 참조 형식입니다. 인터페이스를 구현하는 모든 클래스와 구조체는 해당 멤버 집합을 구현해야 합니다. 인터페이스는 임의 또는 모든 구성원에 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. 자세한 내용은 [Abstract 및 Sealed 클래스와 클래스 멤버](#)를 참조하세요.

## 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.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [클래스 및 구조체](#)
- [class](#)

# 다형성(C# 프로그래밍 가이드)

2020-11-02 • 19 minutes to read • [Edit Online](#)

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 타임에는 사용자가 어떤 특정 형식의 도형을 만들지 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.

```

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.

```

// 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 다형 형식입니다.

## 다형성 개요

### 가상 멤버

기본 클래스에서 파생 클래스가 상속되면 파생 클래스는 기본 클래스의 모든 메서드, 필드, 속성 및 이벤트를 얻습니다. 파생 클래스의 디자이너는 가상 메서드의 동작에 대한 다양한 선택 사항이 있습니다.

- 파생 클래스가 기본 클래스의 가상 멤버를 재정의하여 새로운 동작을 정의할 수 있습니다.
- 파생 클래스가 가장 가까운 기본 클래스 메서드를 재정의하지 않고 상속하여 기존 동작은 보존하되 다른 파생 클래스가 해당 메서드를 재정의할 수 있도록 지원합니다.
- 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.

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

BaseClass A = (BaseClass)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. 자세한 내용은 [인터페이스](#)를 참조하세요.

#### 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 바깥 클래스 멤버의 반환 형식 앞에 배치됩니다. 다음 코드는 예제를 제공합니다.

```
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:

```
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 `A` 가 가상 멤버를 선언하고, 클래스 `B` 가 `A`에서 파생되며, 클래스 `C` 가 `B`에서 파생되면 클래스 `C` 는 클래스 `B` 가 해당 멤버에 대한 재정의를 선언했는지 여부와 관계없이 가상 멤버를 상속하며, 해당 가상 멤버를 재정의할 수 있습니다. 다음 코드는 예제를 제공합니다.

```

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

```

파생 클래스는 재정의를 **sealed**로 선언하여 가상 상속을 중지할 수 있습니다. 가상 상속을 중지하려면 클래스 멤버 선언에서 **override** 키워드 앞에 **sealed** 키워드를 배치해야 합니다. 다음 코드는 예제를 제공합니다.

```

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.

```

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

```

이 경우 형식 **D** 변수를 사용하여 **D**에 대해 **DoWork**을 호출하면 새 **DoWork** 가 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.

```

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.

#### NOTE

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.

## 단원 내용

- Override 및 New 키워드를 사용하여 버전 관리
- Override 및 New 키워드를 사용해야 하는 경우
- ToString 메서드 재정의 방법

## 참고 항목

- C# 프로그래밍 가이드
- 상속
- 추상/봉인된 클래스 및 클래스 멤버
- 메서드
- 이벤트
- 속성
- 인덱서
- 형식

# Override 및 New 키워드를 사용하여 버전 관리(C# 프로그래밍 가이드)

2020-11-02 • 14 minutes to read • [Edit Online](#)

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 클래스에 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 `기상`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.

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
}
```

회사에서 이 클래스를 사용하며, 사용자는 이를 사용하여 고유한 클래스를 파생시키고 새 메서드를 추가합니다.

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public void DrawRectangle() { }
}
```

회사 A에서 `GraphicsClass`의 새 버전을 출시할 때까지 애플리케이션은 문제없이 사용됩니다. 새 버전은 다음

코드와 유사합니다.

```
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.

```
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.

```
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.

```
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 호출 할 수 있는 최상의 메서드를 선택합니다. 다음 메서드는 호환 가능합니다.

```
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:

```
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:

```
((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](#)
- [클래스 및 구조체](#)
- [메서드](#)
- [상속](#)

# Override 및 New 키워드를 사용해야 하는 경우(C# 프로그래밍 가이드)

2020-11-02 • 20 minutes to read • [Edit Online](#)

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.

```
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.

```

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.

```

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.

```

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.

```

// 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.

```
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.

```
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.

```
public virtual void Method1()
{
    Console.WriteLine("Base - Method1");
}
```

프로젝트를 다시 실행합니다. 특히 다음 출력의 마지막 두 줄을 확인합니다.

```
// Output:
// Base - Method1
// Base - Method2
// Derived - Method1
// Derived - Method2
// Derived - Method1
// Base - Method2
```

`override` 한정자를 사용하면 `bcdc`가 `DerivedClass`에 정의된 `Method1` 메서드에 액세스할 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.

```

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 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.

```
// 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.

```

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.

```

// 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.

```

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.

```
// 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.

```
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.

```
// TestCars3
// -----
// A roof that opens up.
// Carries seven people.

// TestCars4
// -----
// Standard transportation.
// Carries seven people.
```

다음 코드에서는 전체 프로젝트와 해당 출력을 보여 줍니다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace OverridingAndNew
{
    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](#)
- [클래스 및 구조체](#)
- [Override 및 New 키워드를 사용하여 버전 관리](#)
- [base](#)
- [abstract](#)

# ToString 메서드 재정의 방법(C# 프로그래밍 가이드)

2021-02-18 • 3 minutes to read • [Edit Online](#)

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.

```
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.

## IMPORTANT

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.

```
public override string ToString(){}  
//
```

2. 문자열을 반환하도록 메서드를 구현합니다.

다음 예제에서는 클래스의 특정 인스턴스와 관련된 데이터뿐 아니라 클래스의 이름을 반환합니다.

```
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.

```
Person person = new Person { Name = "John", Age = 12 };
Console.WriteLine(person);
// Output:
// Person: John 12
```

## 참고 항목

- [IFormattable](#)
- [C# 프로그래밍 가이드](#)
- [클래스 및 구조체](#)
- [문자열](#)
- [string](#)
- [override](#)
- [virtual](#)
- [형식 서식 지정](#)

# 멤버(C# 프로그래밍 가이드)

2020-11-02 • 5 minutes to read • [Edit Online](#)

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 파생 클래스에서 액세스할 수 없습니다.

다음 표에는 클래스 또는 구조체에 포함될 수 있는 멤버 종류가 나와 있습니다.

멤버	설명
필드	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.
상수	상수는 해당 값이 컴파일 시간에 설정되며 변경할 수 없는 필드입니다.
속성	속성은 해당 클래스의 필드처럼 액세스되는 클래스의 매서드입니다. 속성은 클래스 필드에 대한 보호를 제공하여 개체 모르게 필드가 변경되지 않도록 할 수 있습니다.
메서드	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.
이벤트	이벤트는 단추 클릭, 성공적인 메서드 완료 등의 발생에 대한 알림을 다른 개체에 제공합니다. 이벤트는 대리자를 사용하여 정의 및 트리거됩니다.
연산자	오버로드된 연산자는 형식 멤버로 간주됩니다. 연산자를 오버로드하는 경우 유형에서 공용 정적 메서드로 정의합니다. 자세한 내용은 <a href="#">연산자 오버로드</a> 를 참조하세요.
인덱서	인덱서를 사용하면 배열과 유사한 방식으로 개체를 인덱싱할 수 있습니다.
생성자	생성자는 개체를 처음 만들 때 호출되는 메서드입니다. 대체로 개체의 데이터를 초기화하는 데 사용됩니다.
종료자	종료자는 C#에서 매우 드물게 사용됩니다. 메모리에서 개체를 제거할 때 런타임 실행 엔진이 호출하는 메서드입니다. 일반적으로 해제해야 하는 리소스가 적절하게 처리되도록 하는 데 사용됩니다.
중첩 형식	중첩 형식은 다른 형식 내에서 선언된 형식입니다. 중첩 형식은 대체로 개체를 포함하는 형식에서만 사용되는 개체를 설명하는 데 사용됩니다.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [클래스](#)

# 추상 및 봉인 클래스와 클래스 멤버(C# 프로그래밍 가이드)

2020-11-02 • 7 minutes to read • [Edit Online](#)

**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. 예를 들어:

```
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. 예를 들어:

```
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. 추상 클래스에서 기본 클래스의 가상 메서드를 상속하는 경우 추상 클래스에서는 추상 메서드를 사용하여 가상 메서드를 재정의할 수 있습니다. 예를 들어:

```
// 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 액세스할 수 없습니다. 앞의 예제에서 F 클래스의 `DoWork`에서는 D 클래스의 `DoWork`을 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. 예를 들어:

```
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 없으므로 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. 예를 들면 다음과 같습니다.

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

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [클래스 및 구조체](#)
- [상속](#)
- [메서드](#)
- [필드](#)

- 추상 속성 정의 방법

# 정적 클래스 및 정적 클래스 멤버(C# 프로그래밍 가이드)

2021-02-18 • 14 minutes to read • [Edit Online](#)

정적 클래스는 기본적으로 비정적 클래스와 동일하지만, 정적 클래스는 인스턴스화할 수 없다는 한 가지 차이점이 있습니다. 즉, `new` 연산자를 사용하여 클래스 형식의 변수를 만들 수 없습니다. 인스턴스 변수가 없기 때문에 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.

```
UtilityClass.MethodA();
```

정적 클래스는 입력 매개 변수에 대해서만 작동하고 내부 인스턴스 필드를 가져오거나 설정할 필요가 없는 메서드 집합에 대한 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.

```
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 형식과 마찬가지로, 정적 클래스에 대한 형식 정보는 클래스를 참조하는 프로그램이 로드될 때 .NET 런타임에 의해 로드됩니다. 프로그램에서 클래스가 로드되는 시기를 정확하게 지정할 수는 없습니다. 그러나 클래스가 로드되도록 하고, 프로그램에서 클래스를 처음으로 참조하기 전에 해당 필드가 초기화되고 정적 생성자가 호출되도록 합니다. 정적 생성자는 한 번만 호출되며, 프로그램이 있는 애플리케이션 도메인의 수명 동안 정적 클래스가 메모리에 유지됩니다.

## NOTE

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 생성자](#)를 포함할 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 없도록 합니다.

정적 클래스는 봉인되므로 상속할 수 없습니다. [Object](#)를 제외하고 어떤 클래스에서도 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 생성자](#)를 참조하세요.

## 예제

온도를 섭씨에서 화씨로, 화씨에서 섭씨로 변환하는 두 메서드를 포함하는 정적 클래스의 예는 다음과 같습니다.

```
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());
                Console.WriteLine("Temperature in Fahrenheit: {0:F2}", F);
                break;

            case "2":
                Console.Write("Please enter the Fahrenheit temperature: ");
                C = TemperatureConverter.FahrenheitToCelsius(Console.ReadLine());
                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 클래스 인스턴스가 생성되지 않은 경우에도 클래스에 대해 호출할 수 있습니다. 정적 멤버는 항상 인스턴스 이름이 아니라 클래스 이름으로 액세스됩니다. 생성된 클래스 인스턴스 개수에 관계없이 정적 멤버의 복사본 한 개만 있습니다. 정적 메서드 및 속성은 포함하는 형식의 비정적 필드 및 이벤트에 액세스할 수 없으며, 메서드 매개 변수에 명시적으로 전달되지 않은 경우 개체의 인스턴스 변수에 액세스할 수 없습니다.

전체 클래스를 `static`으로 선언하는 것보다 일부 정적 멤버가 포함된 비정적 클래스를 선언하는 것이 더 일반적입니다. 정적 필드의 두 가지 일반적인 사용은 인스턴스화된 개체 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 `static const`로 선언할 수는 없지만, `const` 필드는 기본적으로 정적으로 동작합니다. 형식의 인스턴스가 아니라 형식에 속합니다. 따라서 정적 필드에 사용되는 것과 동일한 `className.MemberName` 표기법을 사용하여 `const` 필드에 액세스할 수 있습니다. 개체 인스턴스는 필요하지 않습니다.

C#은 정적 지역 변수(즉, 메서드 범위 내에서 선언된 변수)를 지원하지 않습니다.

다음 예제와 같이 멤버의 반환 형식 앞에 `static` 키워드를 사용하여 정적 클래스 멤버를 선언합니다.

```

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 멤버를 처음으로 액세스하기 전, 정적 생성자가 있을 경우 호출되기 전에 초기화됩니다. 정적 클래스 멤버에 액세스하려면 다음 예제와 같이 변수 이름 대신 클래스 이름을 사용하여 멤버의 위치를 지정합니다.

```
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 간의 성능 차이는 크지 않습니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)에서 정적 클래스 및 정적 및 인스턴스 멤버를 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [static](#)
- [클래스](#)
- [class](#)
- [정적 생성자](#)
- [인스턴스 생성자](#)

# 액세스 한정자(C# 프로그래밍 가이드)

2021-02-18 • 12 minutes to read • [Edit Online](#)

모든 형식과 형식 멤버에는 액세스 수준이 있습니다. 액세스 수준은 사용 중인 어셈블리나 다른 어셈블리에서 형식 또는 형식 멤버를 사용할 수 있는지 여부를 제어합니다. 다음 액세스 한정자를 사용하여 형식 또는 멤버를 선언할 때 해당 항목의 액세스 수준을 지정할 수 있습니다.

- **public**: 동일한 어셈블리의 다른 코드나 해당 어셈블리를 참조하는 다른 어셈블리의 코드에서 형식이나 멤버에 액세스할 수 있습니다.
- **private**: 같은 `class` 또는 `struct`의 코드에서만 형식 또는 멤버에 액세스할 수 있습니다.
- **protected**: 같은 `class` 또는 해당 `class`에서 파생된 `class`의 코드에서만 형식 또는 멤버에 액세스할 수 있습니다.
- **internal**: 동일한 어셈블리의 코드에서는 형식이나 멤버에 액세스할 수 있지만 다른 어셈블리의 코드에서는 액세스할 수 없습니다.
- **protected internal**: 형식 또는 멤버가 선언된 어셈블리의 모든 코드에서 또는 다른 어셈블리의 파생 `class` 내에서 형식 또는 멤버에 액세스할 수 있습니다.
- **private protected**: 형식 또는 멤버가 선언된 어셈블리, 같은 `class`나 해당 `class`에서 파생된 형식의 코드에서만 형식 또는 멤버에 액세스할 수 있습니다.

다음 예제에서는 형식 및 멤버에 대해 액세스 한정자를 지정하는 방법을 보여 줍니다.

```
public class Bicycle
{
    public void Pedal() { }
}
```

모든 액세스 한정자를 모든 컨텍스트에서 모든 형식 또는 멤버에서 사용할 수 있는 것은 아닙니다. 형식 멤버의 액세스 수준이 해당 형식 멤버를 포함하는 형식의 액세스 수준으로 제한되는 경우도 있습니다.

## 클래스 및 구조체 액세스 수준

네임스페이스 내에서 직접 선언된(즉, 다른 클래스 또는 구조체 내에 종첩되지 않은) 클래스와 구조체는 `public`과 `internal` 중 하나일 수 있습니다. 액세스 한정자가 지정되지 않은 경우 기본값은 `internal`입니다.

구조체 멤버(종첩 클래스 및 구조체 포함)는 `public`, `internal` 또는 `private`으로 선언할 수 있습니다. 클래스 멤버(종첩 클래스 포함)는 `public`, `protected internal`, `protected`, `internal`, `private protected` 또는 `private`으로 선언할 수 있습니다. 클래스 멤버와 구조체 멤버(종첩 클래스 및 구조체 포함)의 액세스 수준은 기본적으로 `private`입니다. `private` 종첩 형식은 해당 형식을 포함하는 형식 외부에서 액세스할 수 없습니다.

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.

## 클래스 및 구조체 멤버 액세스 수준

종첩 클래스 및 구조체를 포함한 클래스 멤버는 6가지 액세스 형식으로 선언될 수 있습니다. 구조체는 상속을 지원하지 않으므로 구조체 멤버는 `protected`, `protected internal` 또는 `private protected`로 선언할 수 없습니다.

다.

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.

종료자에는 접근성 한정자를 사용할 수 없습니다.

`class` 또는 `struct` 멤버의 액세스 수준을 설정하려면 다음 예제와 같이 멤버 선언에 해당 키워드를 추가하세요.

```
// 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; }  
    }  
}
```

## 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 액세스하는데 사용되므로 인터페이스 멤버는 기본적으로 `public`입니다. 인터페이스 멤버 선언은 모든 액세스 한정자를 포함할 수 있습니다. 이는 정적 메서드가 모든 클래스 구현자에 필요한 공통된 구현을 제공하도록 할 때 가장 유용합니다.

열거형 멤버는 항상 `public`이고 어떤 액세스 한정자도 적용할 수 없습니다.

대리자는 클래스 및 구조체처럼 동작합니다. 기본적으로 대리자는 네임스페이스 내에서 직접 선언된 경우 `internal` 액세스를 갖고 중첩된 경우 `private` 액세스를 갖습니다.

## C# 언어 사양

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

## 참조

- [C# 프로그래밍 가이드](#)
- [클래스 및 구조체](#)

- 인터페이스
- private
- public
- internal
- protected
- protected internal
- private protected
- class
- struct
- interface

# 필드(C# 프로그래밍 가이드)

2021-02-18 • 9 minutes to read • [Edit Online](#)

필드는 [클래스](#) 또는 [구조체](#)에서 직접 선언되는 모든 형식의 변수입니다. 필드는 포함하는 형식의 [멤버](#)입니다.

클래스 또는 구조체에는 인스턴스 필드, 정적 필드 또는 둘 다 있을 수 있습니다. 인스턴스 필드는 형식의 인스턴스와 관련이 있습니다. 인스턴스 필드가 F인 T 클래스가 있는 경우 형식이 T인 개체 두 개를 만들고 각 개체에서 다른 개체의 값에 영향을 주지 않고 F 값을 수정할 수 있습니다. 반면 정적 필드는 클래스 자체에 속하며 해당 클래스의 모든 인스턴스에서 공유됩니다. 클래스 이름을 사용해야만 정적 필드에 액세스할 수 있습니다. 인스턴스 이름으로 정적 필드에 액세스 하는 경우 [CS0176](#) 컴파일 시간 오류가 발생합니다.

일반적으로 필드는 액세스 가능성이 `private` 또는 `protected`인 변수에만 사용해야 합니다. 클래스에서 클라이언트 코드에 노출하는 데이터는 [메서드](#), [속성](#) 및 [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.

필드는 일반적으로 둘 이상의 클래스 메서드에서 액세스할 수 있고 단일 메서드의 수명보다 오랫동안 저장되어야 하는 데이터를 저장합니다. 예를 들어 달력 날짜를 나타내는 클래스에는 각각 월, 일 및 연도에 대한 세 개의 정수 필드가 있을 수 있습니다. 단일 메서드 범위 내에서만 사용되는 변수는 메서드 본문 자체 내에 [지역 변수](#)로 선언해야 합니다.

필드는 필드의 액세스 수준을 지정한 다음 필드의 형식, 필드의 이름순으로 지정하여 클래스 블록에서 선언됩니다. 예들 들어 다음과 같습니다.

```

public class CalendarEntry
{
    // private field
    private DateTime date;

    // public field (Generally not recommended.)
    public string day;

    // 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();
            }
        }
    }

    // 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();
        }
    }

    public TimeSpan GetTimeSpan(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        if (dt.Ticks < date.Ticks)
        {
            return date - dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException();
        }
    }
}

```

개체의 필드에 액세스하려면 `objectname.fieldname` 와 같이 개체 이름과 필드 이름 뒤에 마침표를 추가합니다.  
예를 들어:

```
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.

```
public class CalendarDateWithInitialization
{
    public string day = "Monday";
    //...
}
```

필드는 개체 인스턴스에 대한 생성자를 호출하기 직전에 초기화됩니다. 생성자가 필드의 값을 할당하면 필드 선언 중에 지정된 모든 값을 덮어씁니다. 자세한 내용은 [생성자 사용](#)을 참조하세요.

#### NOTE

필드 이니셜라이저는 다른 인스턴스 필드를 참조할 수 없습니다.

필드는 `public`, `private`, `protected`, `internal`, `protected internal` 또는 `private protected`로 표시될 수 있습니다. 이러한 액세스 한정자는 클래스 사용자가 필드에 액세스 하는 방법을 정의합니다. 자세한 내용은 [액세스 한정자](#)를 참조하세요.

필요에 따라 필드를 `static`으로 선언할 수 있습니다. 그러면 클래스의 인스턴스가 없는 경우에도 언제든지 호출자가 필드를 사용할 수 있습니다. 자세한 내용은 [static 클래스 및 static 클래스 멤버](#)를 참조하세요.

필드를 `readonly`로 선언할 수 있습니다. 읽기 전용 필드는 초기화 중이나 생성자에서만 값을 할당할 수 있습니다. C# 컴파일러가 컴파일 시간에는 정적 읽기 전용 필드의 값에 액세스할 수 없고 런타임 시에만 액세스할 수 있다는 점을 제외하면 `static readonly` 필드는 상수와 매우 유사합니다. 자세한 내용은 [상수](#)를 참조하세요.

## C# 언어 사양

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

## 참조

- [C# 프로그래밍 가이드](#)
- [클래스 및 구조체](#)
- [생성자 사용](#)
- [상속](#)
- [액세스 한정자](#)
- [추상/봉인된 클래스 및 클래스 멤버](#)

# 상수(C# 프로그래밍 가이드)

2021-02-18 • 6 minutes to read • [Edit Online](#)

상수는 컴파일 시간에 알려진 변경할 수 없는 값입니다. 프로그램 수명 동안 변경하지 마세요. 상수는 `const` 한정자로 선언됩니다. C# 기본 제공 형식(`System.Object` 제외)만 `const`로 선언할 수 있습니다. 클래스, 구조체 및 배열을 비롯한 사용자 정의 형식은 `const` 가 될 수 없습니다. `readonly` 한정자를 사용하여 런타임에 한번 초기화되고(예: 생성자에서) 이후 변경할 수 없는 클래스, 구조체 또는 배열을 만들습니다.

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. 예를 들어:

```
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.

## NOTE

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.

```
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. 예를 들어:

```
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. 이러

한 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 선언하는데 **static** 키워드를 사용하지 않습니다. 상수를 정의하는 클래스에 포함되지 않은 식은 상수에 액세스 할 때 클래스 이름, 마침표 및 상수 이름을 사용해야 합니다. 예를 들어:

```
int birthstones = Calendar.Months;
```

## C# 언어 사양

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

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [클래스 및 구조체](#)
- [속성](#)
- [형식](#)
- [readonly](#)
- [C# 파트 1의 불변성: 불변성의 종류](#)

# 추상 속성 정의 방법(C# 프로그래밍 가이드)

2021-02-18 • 5 minutes to read • [Edit Online](#)

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 객체의 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.

```

// 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. 예를 들어:

```
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.

## 예제

다음 코드에서는 `Shape`의 세 가지 서브클래스와 이러한 서브클래스에서 `Area` 속성을 재정의하여 고유한 구현을 제공하는 방법을 보여 줍니다.

```

// 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;
        }
    }
}

```

예제

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# 프로그래밍 가이드](#)
- [클래스 및 구조체](#)
- [추상/봉인된 클래스 및 클래스 멤버](#)
- [속성](#)

# C#에서 상수 정의 방법

2021-02-18 • 2 minutes to read • [Edit Online](#)

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.

## NOTE

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

```
using System;

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

- [클래스 및 구조체](#)

# 속성(C# 프로그래밍 가이드)

2020-11-02 • 11 minutes to read • [Edit Online](#)

속성은 전용 필드의 값을 읽거나 쓰거나 계산하는 유연한 메커니즘을 제공하는 멤버입니다. 공용 데이터 멤버인 것처럼 속성을 사용할 수 있지만, 실제로 접근자라는 특수 메서드입니다. 이렇게 하면 데이터에 쉽게 액세스 할 수 있으며 메서드의 안전성과 유연성 수준을 올리는 데에도 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 반환하는 데 사용되고 `set` 속성 접근자는 새 값을 할당하는 데 사용됩니다. 이러한 접근자는 각기 다른 액세스 수준을 가질 수 있습니다. 자세한 내용은 [접근자 액세스 가능성 제한](#)을 참조하세요.
- `value` 키워드는 `set` 접근자가 할당하는 값을 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 속성으로 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 수행 할 수 있습니다. 또한 두 접근자 모두 데이터를 저장 또는 반환하기 전에 데이터에 대한 변환이나 계산을 수행 할 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 지정할 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.

```

using System;

class TimePeriod
{
    private double _seconds;

    public double Hours
    {
        get { return _seconds / 3600; }
        set {
            if (value < 0 || value > 24)
                throw new ArgumentOutOfRangeException(
                    $"{nameof(value)} must be between 0 and 24.");

            _seconds = value * 3600;
        }
    }
}

class Program
{
    static void Main()
    {
        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.

C# 6부터 읽기 전용 속성에서 `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.

```
using System;

public class Person
{
    private string _firstName;
    private string _lastName;

    public Person(string first, string last)
    {
        _firstName = first;
        _lastName = last;
    }

    public string Name => $"{_firstName} {_lastName}";
}

public class Example
{
    public static void Main()
    {
        var person = new Person("Magnus", "Hedlund");
        Console.WriteLine(person.Name);
    }
}

// The example displays the following output:
//      Magnus Hedlund
```

C# 7.0부터 `get` 및 `set` 접근자 모두를 식 본문 멤버로 구현할 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 사용하지 않습니다.

```

using System;

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;
    }
}

class Program
{
    static void Main(string[] args)
    {
        var item = new SaleItem("Shoes", 19.95m);
        Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
    }
}
// The example displays output like the following:
//     Shoes: sells for $19.95

```

## 자동으로 구현된 속성

경우에 따라 `get` 속성과 `set` 접근자에서 지원 필드에 값을 할당하거나 지원 필드에서 값을 검색하기만 하고 추가 논리를 포함하지 않을 수 있습니다. 자동 구현 속성을 사용하면 코드를 간소화할 수 있을 뿐 아니라 C# 컴파일러에서 지원 필드를 투명하게 제공하도록 할 수 있습니다.

속성에 `get` 및 `set` 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.

```
using System;

public class SaleItem
{
    public string Name
    { get; set; }

    public decimal Price
    { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        var item = new SaleItem{ Name = "Shoes", Price = 19.95m };
        Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
    }
}
// The example displays output like the following:
//     Shoes: sells for $19.95
```

## 관련 단원

- [속성 사용](#)
- [인터페이스 속성](#)
- [속성 및 인덱서 비교](#)
- [접근자 접근성 제한](#)
- [자동으로 구현된 속성](#)

## C# 언어 사양

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

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [속성 사용](#)
- [인덱서](#)
- [get 키워드](#)
- [set 키워드](#)

# 속성 사용(C# 프로그래밍 가이드)

2020-11-02 • 18 minutes to read • [Edit Online](#)

속성은 필드 및 메서드 모두의 측면을 결합합니다. 개체의 사용자에게 속성은 필드로 표시되며, 속성에 액세스 하려면 동일한 구문이 필요합니다. 클래스의 구현자에게 속성은 `get` 접근자 및/또는 `set` 접근자를 나타내는 하나 또는 두 개의 코드 블록입니다. `get` 접근자에 대한 코드 블록은 속성을 읽을 때 실행되고, `set` 접근자에 대한 코드 블록은 속성에 새 값을 할당할 때 실행됩니다. `set` 접근자가 없는 속성은 읽기 전용으로 간주됩니다. `get` 접근자가 없는 속성은 쓰기 전용으로 간주됩니다. 두 접근자가 모두 있는 속성은 읽기/쓰기입니다.

필드와 달리 속성은 변수로 분류되지 않습니다. 따라서 `ref` 또는 `out` 매개 변수로 속성을 전달할 수 없습니다.

속성은 여러 가지 용도로 사용됩니다. 변경을 허용하기 전에 데이터의 유효성을 검사하고, 데이터가 실제로 데이터베이스 등의 다른 소스에서 검색되는 클래스에 데이터를 투명하게 공개하고, 데이터 변경 시 이벤트 발생, 다른 필드의 값 변경 등의 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. 예를 들어:

```
public class Date
{
    private int _month = 7; // Backing store

    public int Month
    {
        get => _month;
        set
        {
            if ((value > 0) && (value < 13))
            {
                _month = value;
            }
        }
    }
}
```

이 예제에서 `Month`은 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 `private`로 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 속성](#)을 참조하세요.

## get 접근자

`get` 접근자 본문은 메서드 본문과 유사합니다. 속성 형식의 값을 반환해야 합니다. `get` 접근자의 실행은 필드 값을 읽는 것과 같습니다. 예를 들어 `get` 접근자에서 `private` 변수를 반환하고 최적화가 사용되는 경우 `get` 접근자 메서드 호출이 컴파일러에서 인라인되므로 메서드 호출 오버헤드가 없습니다. 그러나 가상 `get` 접근자 메서드는 컴파일러에서 런타임 시 실제로 호출될 수 있는 메서드를 컴파일 시간에 알 수 없기 때문에 인라인할 수 없습니다. 다음은 전용 필드 `_name`의 값을 반환하는 `get` 접근자입니다.

```
class Person
{
    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. 예를 들어:

```
Person person = new Person();
//...

System.Console.WriteLine(person.Name); // the get accessor is invoked here
```

`get` 접근자는 `return` 또는 `throw` 문으로 끝나야 하며, 제어가 접근자 본문을 벗어날 수 없습니다.

`get` 접근자를 사용하여 개체의 상태를 변경하는 것은 잘못된 프로그래밍 스타일입니다. 예를 들어 다음 접근자는 `_number` 필드에 액세스할 때마다 개체의 상태가 변경되는 부작용을 생성합니다.

```
private int _number;
public int Number => _number++; // Don't do this
```

`get` 접근자를 사용하여 필드 값을 반환하거나 계산한 후 반환할 수 있습니다. 예를 들어:

```
class Employee
{
    private string _name;
    public string Name => _name != null ? _name : "NA";
}
```

앞의 코드 세그먼트에서 `Name` 속성에 값을 할당하지 않으면 `NA` 값이 반환됩니다.

## set 접근자

`set` 접근자는 반환 형식이 `void`인 메서드와 비슷합니다. 형식이 속성의 형식인 `value`라는 암시적 매개 변수를 사용합니다. 다음 예제에서는 `set` 접근자가 `Name` 속성에 추가됩니다.

```
class Person
{
    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. 예를 들어:

```
Person person = new Person();
person.Name = "Joe"; // the set accessor is invoked here

System.Console.WriteLine(person.Name); // the get accessor is invoked here
```

`set` 접근자의 지역 변수 선언에 대해 암시적 매개 변수 이름 `value`를 사용하면 오류가 발생합니다.

## 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` 접근자가 서로 다른 액세스 한정자를 가질 수 있습니다. 예를 들어 `get`은 형식 외부에서 읽기 전용 액세스를 허용하도록 `public`이 되고, `set`은 `private` 또는 `protected`가 될 수 있습니다. 자세한 내용은 [액세스 한정자](#)를 참조하세요.

`static` 키워드를 사용하여 속성을 정적 속성으로 선언할 수 있습니다. 그러면 클래스의 인스턴스가 없는 경우에도 언제든지 호출자가 속성을 사용할 수 있습니다. 자세한 내용은 [static 클래스 및 static 클래스 멤버](#)를 참조하세요.

`virtual` 키워드를 사용하여 속성을 가상 속성으로 표시할 수 있습니다. 그러면 파생 클래스에서 `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. 즉, 클래스에 구현이 없으며 파생 클래스가 자체 구현을 작성해야 합니다. 이러한 옵션에 대한 자세한 내용은 [추상 및 봉인 클래스와 클래스 멤버](#)를 참조하세요.

### NOTE

`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.

```

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:
}

class TestEmployee
{
    static void Main()
    {
        Employee.NumberOfEmployees = 107;
        Employee e1 = new Employee();
        e1.Name = "Claude Vige";

        System.Console.WriteLine("Employee number: {0}", Employee.Counter);
        System.Console.WriteLine("Employee name: {0}", e1.Name);
    }
}
/* Output:
   Employee number: 108
   Employee name: Claude Vige
*/

```

## 예제

이 예제에서는 파생 클래스에서 이름이 같은 다른 속성에 의해 숨겨진 기본 클래스의 속성에 액세스하는 방법을 보여 줍니다.

```

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
{
    static void Main()
    {
        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.

```
public new string Name
```

- `(Employee)` 캐스트는 기본 클래스의 숨겨진 속성에 액세스하는 데 사용됩니다.

```
((Employee)m1).Name = "Mary";
```

멤버를 숨기는 방법에 대한 자세한 내용은 [new 한정자](#)를 참조하세요.

## 예제

이 예제에서 두 클래스 `Cube` 및 `Square`는 추상 클래스 `Shape`를 구현하고 해당 `abstract Area` 속성을 재정의 합니다. 속성의 `override` 한정자를 사용합니다. 프로그램은 변을 입력으로 사용하고 사각형과 정육면체의 면적을 계산합니다. 또한 면적을 입력으로 사용하고 사각형 및 정육면체의 해당 변을 계산합니다.

```

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

- [C# 프로그래밍 가이드](#)
- [속성](#)
- [인터페이스 속성](#)
- [자동으로 구현된 속성](#)

# 인터페이스 속성(C# 프로그래밍 가이드)

2020-11-02 • 4 minutes to read • [Edit Online](#)

[interface](#)에 속성을 선언할 수 있습니다. 다음 예제에서는 인터페이스 속성 접근자를 선언합니다.

```
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. C# 8.0부터 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:

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

앞의 예제에서는 [명시적 인터페이스 구현](#)을 보여 주었습니다. 예를 들어 `Employee` 클래스가 두 인터페이스 `ICitizen` 및 `IEmployee`를 구현하고 두 인터페이스에 모두 `Name` 속성이 있으면 명시적 인터페이스 멤버 구현이 필요합니다. 즉, 다음과 같은 속성 선언이 있다고 가정합니다.

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

이 선언은 `IEmployee` 인터페이스의 `Name` 속성을 구현합니다. 또한 다음과 같은 선언이 있다고 가정합니다.

```
string ICitizen.Name
{
    get { return "Citizen Name"; }
    set { }
}
```

이 선언은 `ICitizen` 인터페이스의 `Name` 속성을 구현합니다.

```
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;
}
```

```
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);
```

210 Hazem Abolrous

## 샘플 출력

```
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)

2020-11-02 • 9 minutes to read • [Edit Online](#)

속성 또는 인덱서의 `get` 및 `set` 부분을 접근자라고 합니다. 기본적으로 이러한 접근자는 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. 이렇게 하려면 일반적으로 `set` 접근자의 접근성을 제한하는 동시에 `get` 접근자를 공개적으로 액세스할 수 있도록 유지해야 합니다. 예를 들어:

```
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.

## 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 재정의하는 코드에서 재정의된 접근자에 액세스할 수 있어야 합니다. 또한 속성/인덱서 및 접근자의 접근성이 재정의된 속성/인덱서 및 해당 접근자와 일치해야 합니다. 예를 들어:

```

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.

```

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 도메인](#)은 이 한정자에 의해 결정됩니다.

접근자의 액세스 한정자를 사용하지 않은 경우 접근자의 접근성 도메인은 속성 또는 인덱서의 접근성 수준에 의해 결정됩니다.

## 예제

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 예제에서는 `DerivedClass`, `Name` 속성의 `set` 접근자에 있는 `private`, `protected` 등의 제한적인 액세스 한정자가 접근자에 대한 액세스를 차단하고 할당을 시도할 때 오류를 생성함을 보여 줍니다.

```
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
*/

```

## 주석

`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

- [C# 프로그래밍 가이드](#)
- [속성](#)
- [인덱서](#)
- [액세스 한정자](#)

# 읽기/쓰기 속성 선언 및 사용 방법(C# 프로그래밍 가이드)

2021-02-18 • 5 minutes to read • [Edit Online](#)

속성은 개체 데이터에 대한 액세스가 보호, 제어, 확인되지 않을 위험 없이 공용 데이터 멤버의 편리함을 제공합니다. 이를 위해 기본 데이터 멤버의 값을 할당하고 검색하는 특수 메서드인 접근자가 사용됩니다. `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.

## 예제

```
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;
    }
}

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. 이 경우 모든 개체가 이러한 속성을 읽고 쓸 수 있습니다. 그러나 때로는 접근자 중 하나를 제외하는 것이 좋습니다. 예를 들어 `set` 접근자를 생략하면 속성은 읽기 전용이 됩니다.

```

public string Name
{
    get
    {
        return _name;
    }
    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 필드처럼 사용할 수 있습니다. 이 경우 다음 문과 같이 속성의 값을 가져오고 설정할 때 자연스러운 구문을 사용할 수 있습니다.

```

person.Name = "Joe";
person.Age = 99;

```

속성 `set` 메서드에 특수 `value` 변수를 사용할 수 있습니다. 이 변수에는 사용자가 지정한 값이 포함됩니다. 예를 들면 다음과 같습니다.

```

_name = value;

```

`Person` 개체의 `Age` 속성을 증가하기 위한 정리된 구문은 다음과 같습니다.

```

person.Age += 1;

```

개별 `set` 및 `get` 메서드를 사용하여 속성을 모델링한 경우 동등한 코드가 다음과 같이 표시될 수 있습니다.

```
person.SetAge(person.GetAge() + 1);
```

다음 예제에서는 `ToString` 메서드가 재정의되었습니다.

```
public override string ToString()
{
    return "Name = " + Name + ", Age = " + Age;
}
```

`ToString`이 프로그램에서 명시적으로 사용되지 않고 기본적으로 `WriteLine` 호출에 의해 호출됩니다.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [속성](#)
- [클래스 및 구조체](#)

# 자동으로 구현된 속성(C# 프로그래밍 가이드)

2020-11-02 • 4 minutes to read • [Edit Online](#)

C# 3.0 이상에서는 속성 접근자에 추가적인 논리가 필요하지 않을 경우 자동 구현 속성을 통해 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.

## 예제

다음 예제에서는 일부 자동 구현 속성이 있는 간단한 클래스를 보여 줍니다.

```
// This class is mutable. Its data can be modified from
// outside the class.
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.

C# 6 이상 버전에서는 필드와 유사하게 자동 구현 속성을 초기화할 수 있습니다.

```
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 집합(데이터)만 캡슐화하고 동작이 적거나 없는 작은 클래스나 구조체의 경우 set 접근자를 [private](#)로 선언하거나(소비자에 대한 변경 불가능) get 접근자만 선언하여(생성자를 제외한 모든 위치에서 변경 불가능) 개체를 변경 불가능으로 설정해야 합니다. 자세한 내용은 [자동으로 구현된 속성을 사용하여 간단한 클래스를 구현하는 방법](#)을 참조하세요.

## 참조

- [속성](#)
- [한정자](#)

# 자동으로 구현된 속성을 사용하여 간단한 클래스를 구현하는 방법(C# 프로그래밍 가이드)

2021-02-18 • 5 minutes to read • [Edit Online](#)

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.

다음 두 가지 방법으로 변경할 수 없는 속성을 만들 수 있습니다.

- `set` 접근자를 **비공개**로 선언할 수 있습니다. 속성은 형식 내에서만 설정할 수 있고 소비자는 변경할 수 없습니다.  
`private set` 접근자를 선언하는 경우 개체 이니셜라이저를 사용하여 속성을 초기화할 수 없습니다. 생성자나 팩터리 메서드를 사용해야 합니다.
- `get` 접근자만 선언하여 형식의 생성자를 제외한 어떤 위치에서도 속성을 변경할 수 없도록 만들 수 있습니다.

다음 예제는 `get` 접근자만 있는 속성이 `get` 및 `private` 집합이 있는 속성과 어떻게 다른지 보여 줍니다.

```
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
}
```

## 예제

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.

```
// 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";

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
    }
}

```

```
        Console.ReadKey();
    }

/* 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. 이 필드는 소스 코드에서 직접 액세스할 수 없습니다.

## 참조

- [속성](#)
- [struct](#)
- [개체 이니셜라이저 및 컬렉션 이니셜라이저](#)

# 메서드(C# 프로그래밍 가이드)

2021-02-18 • 25 minutes to read • [Edit Online](#)

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.

## NOTE

이 항목에서는 명명된 메서드에 대해 설명합니다. 익명 함수에 대한 자세한 내용은 [익명 함수](#)를 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.

## IMPORTANT

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.

```
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.

```

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:

```

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 형식](#)의 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 전달](#)을 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 전달하던

라도 변경 내용은 호출하는 메서드의 인수에 반영됩니다.

다음 예제와 같이 `class` 키워드를 사용하여 참조 형식을 만듭니다.

```
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.

```
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` 키워드를 사용하여 값을 반환할 수 있습니다. `return` 키워드에 이어 반환 형식과 일치하는 값을 포함하는 문은 메서드 호출자에 값을 반환합니다.

호출자에게 값으로 또는 C# 7.0부터 [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 반환 문은 메서드가 변수 이름 `estDistance`를 호출자에게 참조로 반환함을 나타냅니다.

```
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.

```
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.

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```
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.

```
ref int distance = plant
```

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.

```

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;
        }
    }
}

```

자세한 내용은 [return](#)을 참조하세요.

## 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.

### NOTE

비동기 메서드는 아직 완료되지 않은 첫 번째 대기된 개체를 검색할 때나 비동기 메서드의 끝에 도달할 때 종에서 먼저 발생하는 시점에 호출자에게 반환됩니다.

비동기 메서드의 반환 형식은 `Task<TResult>`, `Task` 또는 `void`일 수 있습니다. `void` 반환 형식은 기본적으로 `void` 반환 형식이 필요할 때 이벤트 처리기를 정의하는 데 사용합니다. `void`를 반환하는 비동기 메서드는 대기할 수 없고 `void`를 반환하는 메서드의 호출자는 메서드가 `throw`하는 예외를 `catch`할 수 없습니다.

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>`어야 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(); .
```

```

using System;
using System.Threading.Tasks;

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.

```

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 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>`, `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.

### 참조

- [C# 프로그래밍 가이드](#)
- [클래스 및 구조체](#)
- [액세스 한정자](#)
- [정적 클래스 및 정적 클래스 멤버](#)
- [상속](#)
- [추상/봉인된 클래스 및 클래스 멤버](#)
- [params](#)
- [return](#)
- [out](#)
- [ref](#)
- [매개 변수 전달](#)

# 로컬 함수(C# 프로그래밍 가이드)

2020-11-02 • 26 minutes to read • [Edit Online](#)

C# 7.0부터 C#에서는 로컬 함수를 지원합니다. 로컬 함수는 다른 멤버에 중첩된 형식의 private 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.

## NOTE

로컬 함수에서도 지원하는 기능을 램다 식으로 구현할 수 있는 경우도 있습니다. 비교를 보려면 [로컬 함수 및 램다 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.

```
<modifiers> <return-type> <method-name> <parameter-list>
```

로컬 함수에 다음 한정자를 사용할 수 있습니다.

- `async`
- `unsafe`
- `static` (C# 8.0 이상). 정적 로컬 함수는 지역 변수 또는 인스턴스 상태를 캡처할 수 없습니다.
- `extern` (C# 9.0 이상). 외부 로컬 함수는 `static`이어야 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.

```

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.

```

#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 특성](#)을 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. 메서드 반복기의 경우 반환된 시퀀스가 열거된 경우에만 예외가 나타나고 반복기를 검색할 때는 나타나지 않습니다. 비동기 메서드의 경우 비동기 메서드에서 throw된 예외는 반환된 작업을 대기할 때 관찰됩니다.

다음 예제에서는 지정한 범위의 훌수를 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.

```

using System;
using System.Collections.Generic;

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.

```

using System;
using System.Collections.Generic;

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

```

유사한 방식으로 비동기 작업과 함께 로컬 함수를 사용할 수 있습니다. 해당 작업이 대기되면 비동기 메서드에서 `throw`된 예외가 표시됩니다. 로컬 함수를 사용하면 코드가 빨리 실패하여 예외가 `throw`되는 동시에 관찰할 수 있습니다.

다음 예제에서는 `GetMultipleAsync`라는 비동기 메서드를 사용하여 지정된 시간(초) 동안 일시 종지하고 해당 시간(초)의 임의 배수인 값을 반환합니다. 최대 지연 시간은 5초입니다. 값이 5보다 크면 `ArgumentOutOfRangeException`이 발생합니다. 다음 예제와 같이 `GetMultipleAsync` 메서드에 값 6을 전달할 때 `throw`되는 예외는 작업이 대기된 경우에만 관찰됩니다.

```
using System;
using System.Threading.Tasks;

public class AsyncWithoutLocalExample
{
    public static async Task Main()
    {
        var t = GetMultipleAsync(6);
        Console.WriteLine("Got the task");

        var result = await t; // line 11
        Console.WriteLine($"The returned value is {result:N0}");
    }

    static async Task<int> GetMultipleAsync(int delayInSeconds)
    {
        if (delayInSeconds < 0 || delayInSeconds > 5)
            throw new ArgumentOutOfRangeException(nameof(delayInSeconds), "Delay cannot exceed 5 seconds.");

        await Task.Delay(delayInSeconds * 1000);
        return delayInSeconds * new Random().Next(2,10);
    }
}

// The example displays the output like this:
//
// Got the task
// Unhandled exception. System.ArgumentOutOfRangeException: Delay cannot exceed 5 seconds. (Parameter
// 'delayInSeconds')
//   at AsyncWithoutLocalExample.GetMultipleAsync(Int32 delayInSeconds) in AsyncWithoutLocal.cs:line 18
//   at AsyncWithoutLocalExample.Main() in AsyncWithoutLocal.cs:line 11
```

메서드 반복기를 사용하는 경우와 마찬가지로 이전 예제를 리팩터링하여 로컬 함수에 비동기 작업 코드를 추가 할 수 있습니다. 다음 예제 출력과 같이 `GetMultiple` 메서드를 호출하는 즉시 `ArgumentOutOfRangeException`이 `throw`됩니다.

```

using System;
using System.Threading.Tasks;

public class AsyncWithLocalExample
{
    public static async Task Main()
    {
        var t = GetMultiple(6); // line 8
        Console.WriteLine("Got the task");

        var result = await t;
        Console.WriteLine($"The returned value is {result:N0}");
    }

    static Task<int> GetMultiple(int delayInSeconds)
    {
        if (delayInSeconds < 0 || delayInSeconds > 5)
            throw new ArgumentOutOfRangeException(nameof(delayInSeconds), "Delay cannot exceed 5 seconds.");

        return GetValueAsync();

        async Task<int> GetValueAsync()
        {
            await Task.Delay(delayInSeconds * 1000);
            return delayInSeconds * new Random().Next(2,10);
        }
    }
}

// The example displays the output like this:
// 
// Unhandled exception. System.ArgumentOutOfRangeException: Delay cannot exceed 5 seconds. (Parameter
// 'delayInSeconds')
//   at AsyncWithLocalExample.GetMultiple(Int32 delayInSeconds) in AsyncWithLocal.cs:line 18
//   at AsyncWithLocalExample.Main() in AsyncWithLocal.cs:line 8

```

## 로컬 함수 및 람다 식

얼핏 보기에도 로컬 함수와 [람다 식](#)은 매우 유사합니다. 대부분의 경우 람다 식과 로컬 함수 사용 간 선택은 스타일 및 개인 기본 설정의 문제입니다. 그러나 하나 또는 다른 것을 사용할 수 있는 것에 알고 있어야 하는 실제 차이점이 있습니다.

계승 알고리즘의 로컬 함수 및 람다 식 구현 간의 차이점을 살펴보겠습니다. 로컬 함수를 사용하는 버전은 다음과 같습니다.

```

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.

```

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 함수를 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.

## 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 사용 되는 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.

```

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.

```
public 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 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. 이 기능은 C# 8.0 이상 버전에서 사용할 수 있습니다.

### NOTE

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.

```

public 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 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.

```

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 식에서 허용되지 않습니다. [컴파일러 오류 CS1621](#)을 참조하세요.

로컬 함수는 람다 식과 종복되는 것처럼 보일 수도 있지만, 실제로 다른 용도로 다르게 사용됩니다. 로컬 함수는 다른 메서드의 컨텍스트에서만 호출되는 함수를 작성하려는 경우에 더 효율적입니다.

## 참조

- [메서드](#)

# 참조 반환 및 참조 로컬

2020-05-20 • 18 minutes to read • [Edit Online](#)

C# 7.0부터 C#에서 참조 반환 값(ref return)을 지원합니다. 참조 반환 값을 사용하면 메서드가 값이 아니라 변수 참조를 호출자에게 다시 반환할 수 있습니다. 그러면 호출자는 반환된 변수를 마치 값이나 참조로 반환된 것처럼 처리하도록 선택할 수 있습니다. 호출자는 참조 로컬이라고 하는, 반환된 값에 대한 참조 자체인 새 변수를 만들 수 있습니다.

## 참조 반환 값이란?

대부분의 개발자는 호출된 메서드에 인수를 참조로 전달하는 데 익숙합니다. 호출된 메서드의 인수 목록에는 참조로 전달되는 변수가 포함됩니다. 호출자는 호출된 메서드에 의한 해당 값의 변경 내용을 관찰합니다. '참조 반환 값'은 메서드가 일부 변수에 대한 '참조'(또는 별칭)를 반환한다는 것을 의미합니다. 해당 변수의 범위는 메서드를 포함해야 합니다. 해당 변수의 수명은 메서드의 반환 이후로 연장되어야 합니다. 호출자가 메서드의 반환 값을 수정하면 메서드가 반환한 변수가 수정됩니다.

메서드가 '참조 반환 값'을 반환한다는 선언은 메서드가 변수에 별칭을 반환함을 나타냅니다. 설계 의도에 따라 호출 코드가 변수 수정을 비롯하여 별칭을 통해 해당 변수에 액세스할 수 있어야 합니다. 결과적으로 참조로 반환하는 메서드는 `void` 반환 형식을 사용할 수 없습니다.

메서드가 참조 반환 값으로 반환할 수 있는 식에는 몇 가지 제한 사항이 있습니다. 제한 사항은 다음과 같습니다.

- 반환 값의 수명은 메서드 실행 이후까지 연장되어야 합니다. 즉, 반환하는 메서드의 지역 변수일 수 없습니다. 클래스의 인스턴스 또는 정적 필드이거나 메서드에 전달된 인수일 수 있습니다. 지역 변수를 반환 하려고 하면 컴파일러 오류 CS8168, "obj'로컬은 참조 로컬이 아니므로 참조로 반환할 수 없습니다."가 생성됩니다.
- 반환 값은 리터럴 `null`일 수 없습니다. `null`을 반환하면 컴파일러 오류 CS8156, "식이 참조로 반환될 수 없으므로 이 컨텍스트에서 해당 식을 사용할 수 없습니다."가 생성됩니다.

참조 반환이 있는 메서드는 값이 현재 `null`(인스턴스화되지 않음) 값이거나 값 형식이 `nullable` 값 형식인 변수에 별칭을 반환할 수 있습니다.

- 반환 값은 상수, 열거형 멤버, 속성의 값 형식 반환 값 또는 `class`나 `struct`의 메서드일 수 없습니다. 이 규칙을 위반하면 컴파일러 오류 CS8156, "식이 참조로 반환될 수 없으므로 이 컨텍스트에서 해당 식을 사용할 수 없습니다."가 생성됩니다.

또한 참조 반환 값은 비동기 메서드에서 허용되지 않습니다. 비동기 메서드는 실행을 마치기 전에 반환할 수 있지만 반환 값을 여전히 알 수 없습니다.

## 참조 반환 값 정의

참조 반환 값을 반환하는 메서드는 다음 2가지 조건을 충족해야 합니다.

- 메서드 시그니처에는 반환 형식 앞에 `ref` 키워드가 포함됩니다.
- 메서드 본문의 각 `return` 문에는 반환된 인스턴스의 이름 앞에 `ref` 키워드가 포함됩니다.

다음 예제에서는 이러한 조건을 충족하면서 `p`라는 이름의 `Person` 개체에 대한 참조를 반환하는 메서드를 보여줍니다.

```
public ref Person GetContactInformation(string fname, string lname)
{
    // ...method implementation...
    return ref p;
}
```

## 참조 반환 값 사용

참조 반환 값은 호출된 메서드 범위 내 다른 변수에 대한 별칭입니다. 참조 반환의 사용은 다음과 같이 별칭이 있는 변수를 사용하는 것으로 해석할 수 있습니다.

- 해당 값을 할당할 경우 별칭이 있는 변수에 값을 할당하는 것입니다.
- 해당 값을 읽을 경우 별칭이 있는 변수의 값을 읽는 것입니다.
- '참조로' 반환하는 경우 동일한 변수에 대한 별칭을 반환하는 것입니다.
- 다른 메서드에 '참조로' 전달하는 경우 별칭이 있는 변수에 대한 참조를 전달하는 것입니다.
- [참조 로컬](#) 별칭을 만들 경우 동일한 변수에 대한 새 별칭을 만드는 것입니다.

## 참조 로컬

`GetContactInformation` 메서드가 다음과 같이 참조 반환으로 선언된다고 가정합니다.

```
public ref Person GetContactInformation(string fname, string lname)
```

값 형식 할당은 변수의 값을 읽고 다음과 같이 새 변수에 해당 값을 할당합니다.

```
Person p = contacts.GetContactInformation("Brandie", "Best");
```

이전의 할당은 `p`를 지역 변수로 선언합니다. 초기 값은 `GetContactInformation`으로 반환된 값을 읽은 것에서 복사됩니다. 향후 `p`에 대한 할당이 `GetContactInformation`으로 반환된 변수의 값을 변경하지 않습니다. `p` 변수는 더 이상 반환된 변수에 대한 별칭이 아닙니다.

'참조 로컬' 변수를 선언하여 원래 값에 대한 별칭을 복사합니다. 다음 할당에서 `p`는 `GetContactInformation`에서 반환된 변수에 대한 별칭입니다.

```
ref Person p = ref contacts.GetContactInformation("Brandie", "Best");
```

`p`가 해당 변수의 별칭이므로 이후 `p` 사용은 `GetContactInformation`에서 반환된 변수를 사용하는 것과 같습니다. 또한 `p`를 변경하면 `GetContactInformation`에서 반환된 변수도 변경됩니다.

`ref` 키워드는 지역 변수 선언 앞, '그리고' 메서드 호출 앞에 사용됩니다.

동일한 방법으로 참조로 값에 액세스할 수 있습니다. 경우에 따라 참조로 값에 액세스하면 비용이 많이 들 수 있는 복사 작업을 피함으로써 성능이 향상됩니다. 예를 들어, 다음 명령문은 값을 참조하는 데 사용되는 참조 로컬 값을 정의하는 방법을 보여줍니다.

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
```

`ref` 키워드는 지역 변수 선언 앞 '그리고' 두 번째 예의 값 앞에 사용됩니다. 두 예에서 변수 선언과 할당에 `ref` 키워드를 둘 다 포함하지 않으면 컴파일러 오류 CS8172, "값을 사용하여 참조 형식 변수를 초기화할 수 없습니다."가 생성됩니다.

C# 7.3 이전에 참조 로컬 변수는 초기화되기 전에 다른 스토리지 공간을 참조하도록 다시 할당될 수 없었습니다.

다. 해당 제한 사항이 제거되었습니다. 다음 예제에서는 재할당을 보여줍니다.

```
ref VeryLargeStruct refLocal = ref veryLargeStruct; // initialization
refLocal = ref anotherVeryLargeStruct; // reassigned, refLocal refers to different storage.
```

참조 로컬 변수는 선언될 때 초기화되어야 합니다.

## 참조 반환 및 참조 로컬: 예제

다음 예제에서는 정수 값의 배열을 저장하는 `NumberStore` 클래스를 정의합니다. `FindNumber` 메서드는 인수로 전달된 숫자보다 크거나 같은 첫 번째 숫자를 참조로 반환합니다. 인수보다 크거나 같은 숫자가 없으면 메서드는 인덱스 0의 숫자를 반환합니다.

```
using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        for (int ctr = 0; ctr < numbers.Length; ctr++)
        {
            if (numbers[ctr] >= target)
                return ref numbers[ctr];
        }
        return ref numbers[0];
    }

    public override string ToString() => string.Join(" ", numbers);
}
```

다음 예제에서는 `NumberStore.FindNumber` 메서드를 호출하여 16보다 크거나 같은 첫 번째 값을 검색합니다. 그런 다음 호출자가 메서드에서 반환된 값을 두 배로 만듭니다. 예제의 출력에서는 `NumberStore` 인스턴스의 배열 요소 값에 반영된 변경 내용을 보여줍니다.

```
var store = new NumberStore();
Console.WriteLine($"Original sequence: {store.ToString()}");
int number = 16;
ref var value = ref store.FindNumber(number);
value *= 2;
Console.WriteLine($"New sequence: {store.ToString()}");
// The example displays the following output:
//      Original sequence: 1 3 7 15 31 63 127 255 511 1023
//      New sequence: 1 3 7 15 62 63 127 255 511 1023
```

참조 반환 값이 지원되지 않을 경우 이러한 작업은 해당 값과 함께 배열 요소의 인덱스를 반환하여 수행됩니다. 그런 다음 호출자는 이 인덱스를 사용하여 별도 메서드 호출에서 값을 수정할 수 있습니다. 그러나 호출자가 인덱스를 수정하여 다른 배열 값에 액세스하고 수정할 수도 있습니다.

다음 예제에서는 C# 7.3 이후에 `FindNumber` 메서드를 다시 작성하여 참조 로컬 재할당을 사용하는 방법을 보여줍니다.

```
using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        ref int returnVal = ref numbers[0];
        var ctr = numbers.Length - 1;
        while ((ctr >= 0) && numbers[ctr] >= target)
        {
            returnVal = ref numbers[ctr];
            ctr--;
        }
        return ref returnVal;
    }

    public override string ToString() => string.Join(" ", numbers);
}
```

이 두 번째 버전은 검색된 수가 배열의 끝에 더 가까운 시나리오에서 더 긴 시퀀스를 사용하여 더욱 효율적입니다.

## 참고 항목

- [ref 키워드](#)
- [안전하고 효율적인 코드 작성](#)

# 매개 변수 전달(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

C#에서는 인수를 값 또는 참조로 매개 변수에 전달할 수 있습니다. 참조로 전달하면 함수 멤버, 메서드, 속성, 인덱서, 연산자 및 생성자가 매개 변수의 값을 변경하고 해당 변경 내용을 호출 환경에서 유지할 수 있습니다. 값 변경의 목적으로 매개 변수를 참조로 전달하려면 `ref` 또는 `out` 키워드를 사용합니다. 값을 변경하지 않고 복사 방지 목적으로 참조로 전달하려면 `in` 한정자를 사용합니다. 간단히 설명하기 위해 이 항목의 예제에서는 `ref` 키워드만 사용됩니다. `in`, `ref` 및 `out` 간의 차이점에 대한 자세한 내용은 [in, ref 및 out](#)을 참조하세요.

다음 예제에서는 값 및 참조 매개 변수 간의 차이점을 보여 줍니다.

```
class Program
{
    static void Main(string[] args)
    {
        int arg;

        // Passing by value.
        // The value of arg in Main is not changed.
        arg = 4;
        squareVal(arg);
        Console.WriteLine(arg);
        // Output: 4

        // Passing by reference.
        // The value of arg in Main is changed.
        arg = 4;
        squareRef(ref arg);
        Console.WriteLine(arg);
        // Output: 16
    }

    static void squareVal(int valParameter)
    {
        valParameter *= valParameter;
    }

    // Passing by reference
    static void squareRef(ref int refParameter)
    {
        refParameter *= refParameter;
    }
}
```

자세한 내용은 다음 항목을 참조하세요.

- [값 형식 매개 변수 전달](#)
- [참조 형식 매개 변수 전달](#)

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 [인수 목록](#)을 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스입니다.

## 참고 항목

- C# 프로그래밍 가이드
- 메서드

# 값 형식 매개 변수 전달(C# 프로그래밍 가이드)

2020-11-02 • 7 minutes to read • [Edit Online](#)

데이터에 대한 참조가 포함되어 있는 참조 형식 변수와는 달리, 값 형식 변수에는 해당 데이터가 직접 포함됩니다. 값 형식 변수를 메서드에 값으로 전달하는 것은 변수의 복사본을 메서드에 전달하는 것과 같습니다. 메서드 내에서 수행되는 매개 변수 변경 작업은 인수 변수에 저장된 원래 데이터에는 영향을 주지 않습니다. 호출된 메서드가 인수 값을 변경하도록 하려면 `ref` 또는 `out` 키워드를 사용하여 해당 메서드를 참조로 전달해야 합니다. `in` 키워드를 사용하여 값이 변경되지 않도록 보장하는 동안 복사를 방지하도록 참조로 값 매개 변수를 전달할 수도 있습니다. 간단한 설명을 위해 다음 예제에서는 `ref` 를 사용합니다.

## 값으로 값 형식 전달

다음 예제에서는 값 형식 매개 변수를 값으로 전달하는 방법을 보여 줍니다. `n` 변수가 `SquareIt` 메서드에 값으로 전달됩니다. 메서드 내에서 수행되는 변경 작업은 변수의 원래 값에는 영향을 주지 않습니다.

```
class PassingValByVal
{
    static void SquareIt(int x)
        // The parameter x is passed by value.
        // Changes to x will not affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(n); // Passing the variable by value.
        System.Console.WriteLine("The value after calling the method: {0}", n);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
   The value before calling the method: 5
   The value inside the method: 25
   The value after calling the method: 5
*/
```

`n` 변수는 값 형식이며 값 5를 데이터로 포함합니다. `SquareIt` 을 호출하면 `n` 의 내용이 `x` 매개 변수로 복사됩니다. 그러면 메서드 내에서 이 매개 변수의 값을 제곱합니다. 그러나 `Main` 에서는 `n` 의 값이 `SquareIt` 메서드를 호출한 후에도 호출 전과 동일합니다. 즉, 메서드 내에서 수행되는 변경은 지역 변수 `x` 에만 적용됩니다.

## 참조로 값 형식 전달

다음 예제는 인수가 `ref` 매개 변수로 전달된다는 점을 제외하면 이전 예제와 동일합니다. 메서드에서 `x` 가 변경되면 기본 인수 `n` 의 값도 변경됩니다.

```

class PassingValByRef
{
    static void SquareIt(ref int x)
    // The parameter x is passed by reference.
    // Changes to x will affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(ref n); // Passing the variable by reference.
        System.Console.WriteLine("The value after calling the method: {0}", n);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
The value before calling the method: 5
The value inside the method: 25
The value after calling the method: 25
*/

```

이 예제에서는 `n`의 값이 아니라 `n`에 대한 참조가 전달됩니다. `x` 매개 변수는 `int`가 아니며 `int`에 대한 참조 (이 예제에서는 `n`에 대한 참조)입니다. 그러므로 메서드 내에서 `x`를 제곱할 때 실제로는 `x` 가 참조하는 `n`을 제곱하게 됩니다.

## 값 형식 교환

인수의 값을 변경하는 일반적인 예로는 두 변수를 메서드에 전달하면 메서드가 변수의 내용을 교환하는 `swap` 메서드가 있습니다. 이때 인수는 `swap` 메서드에 참조로 전달해야 합니다. 이렇게 하지 않으면 메서드 내에서 매개 변수의 로컬 복사본이 교환되므로 호출하는 메서드에서는 변경이 수행되지 않습니다. 다음 예제에서는 정수 값을 바꿉니다.

```

static void SwapByRef(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}

```

`SwapByRef` 메서드를 호출할 때는 다음 예제와 같이 호출에 `ref` 키워드를 사용합니다.

```
static void Main()
{
    int i = 2, j = 3;
    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    SwapByRef (ref i, ref j);

    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}

/* Output:
   i = 2  j = 3
   i = 3  j = 2
*/
```

## 참조

- [C# 프로그래밍 가이드](#)
- [매개 변수 전달](#)
- [참조 형식 매개 변수 전달](#)

# 참조 형식 매개 변수 전달(C# 프로그래밍 가이드)

2020-11-02 • 8 minutes to read • [Edit Online](#)

참조 형식의 변수에는 해당 데이터가 직접 포함되지 않고 데이터에 대한 참조가 포함됩니다. 참조 형식 매개 변수를 값으로 전달하는 경우 클래스 멤버 값 등 참조된 개체에 속하는 데이터를 변경할 수 있습니다. 하지만 참조 자체의 값은 변경할 수 없습니다. 예를 들어 동일한 참조를 사용하여 새 개체에 대한 메모리를 할당하고 메서드 외부에 유지되도록 할 수 없습니다. 이렇게 하려면 `ref` 또는 `out` 키워드를 사용하여 매개 변수를 전달합니다. 간단한 설명을 위해 다음 예제에서는 `ref`를 사용합니다.

## 값으로 참조 형식 전달

다음 예제에서는 참조 형식 매개 변수 `arr` 을 `Change` 메서드에 값으로 전달하는 방법을 보여 줍니다. 매개 변수가 `arr`에 대한 참조이므로 배열 요소의 값을 변경할 수 있습니다. 그러나 다른 메모리 위치에 매개 변수를 다시 할당하려는 시도는 메서드 내부에서만 작동하고 원래 변수 `arr`에는 영향을 주지 않습니다.

```
class PassingRefByVal
{
    static void Change(int[] pArray)
    {
        pArray[0] = 888; // This change affects the original element.
        pArray = new int[5] {-3, -1, -2, -3, -4}; // This change is local.
        System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}", arr[0]);

        Change(arr);
        System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}", arr[0]);
    }
}

/* Output:
   Inside Main, before calling the method, the first element is: 1
   Inside the method, the first element is: -3
   Inside Main, after calling the method, the first element is: 888
*/
```

앞의 예제에서 참조 형식인 `arr` 배열은 `ref` 매개 변수 없이 메서드에 전달됩니다. 이 경우 `arr`을 가리키는 참조의 복사본이 메서드에 전달됩니다. 출력에서는 이 경우 메서드가 배열 요소의 내용을 1에서 888로 변경할 수 있음을 보여 줍니다. 그러나 `Change` 메서드 내에서 `new` 연산자를 사용하여 새 메모리 부분을 할당하면 `pArray` 변수가 새 배열을 참조합니다. 따라서 그 후의 변경 내용은 `Main` 내에서 생성된 원래 배열 `arr`에 영향을 주지 않습니다. 실제로 이 예제에서는 `Main` 내부와 `Change` 메서드 내부에 각각 하나씩, 두 개의 배열이 생성됩니다.

## 참조로 참조 형식 전달

다음 예제는 `ref` 키워드가 메서드 헤더 및 호출에 추가된다는 점을 제외하고 앞의 예제와 동일합니다. 메서드에서 발생하는 모든 변경 내용이 호출하는 프로그램의 원래 변수에 영향을 줍니다.

```

class PassingRefByRef
{
    static void Change(ref int[] pArray)
    {
        // Both of the following changes will affect the original variables:
        pArray[0] = 888;
        pArray = new int[5] {-3, -1, -2, -3, -4};
        System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}",
        arr[0]);

        Change(ref arr);
        System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}",
        arr[0]);
    }
}
/* Output:
   Inside Main, before calling the method, the first element is: 1
   Inside the method, the first element is: -3
   Inside Main, after calling the method, the first element is: -3
*/

```

메서드 내에서 발생하는 모든 변경 내용은 `Main`의 원래 배열에 영향을 줍니다. 실제로 원래 배열은 `new` 연산자를 사용하여 다시 할당됩니다. 따라서 `Change` 메서드를 호출한 후 `arr`에 대한 모든 참조는 `Change` 메서드에서 생성된 5개 요소의 배열을 가리킵니다.

## 두 문자열 교환

문자열 교환은 참조 형식 매개 변수를 참조로 전달하는 좋은 예입니다. 예제에서는 두 개의 문자열 `str1` 및 `str2`가 `Main`에서 초기화되고 `ref` 키워드로 수정되는 매개 변수로 `SwapStrings` 메서드에 전달됩니다. 두 문자열이 메서드 및 `Main` 내에서 교환됩니다.

```

class SwappingStrings
{
    static void SwapStrings(ref string s1, ref string s2)
        // The string parameter is passed by reference.
        // Any changes on parameters will affect the original variables.
    {
        string temp = s1;
        s1 = s2;
        s2 = temp;
        System.Console.WriteLine("Inside the method: {0} {1}", s1, s2);
    }

    static void Main()
    {
        string str1 = "John";
        string str2 = "Smith";
        System.Console.WriteLine("Inside Main, before swapping: {0} {1}", str1, str2);

        SwapStrings(ref str1, ref str2);    // Passing strings by reference
        System.Console.WriteLine("Inside Main, after swapping: {0} {1}", str1, str2);
    }
}

/* Output:
   Inside Main, before swapping: John Smith
   Inside the method: Smith John
   Inside Main, after swapping: Smith John
*/

```

이 예제에서는 호출하는 프로그램의 변수에 영향을 주기 위해 매개 변수를 참조로 전달해야 합니다. 메서드 헤더와 메서드 호출 모두에서 `ref` 키워드를 제거하는 경우 호출하는 프로그램에서는 아무것도 변경되지 않습니다.

문자열에 대한 자세한 내용은 [문자열](#)을 참조하세요.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [매개 변수 전달](#)
- [ref](#)
- [in](#)
- [out](#)
- [참조 형식](#)

# 메서드에 대한 구조체 전달과 클래스 참조 전달 간의 차이점을 이해하는 방법(C# 프로그래밍 가이드)

2021-02-18 • 6 minutes to read • [Edit Online](#)

다음 예제에서는 `struct`를 메서드에 전달하는 것과 `class` 인스턴스를 메서드에 전달하는 것이 어떻게 다른지를 보여 줍니다. 예제에서 두 인수(구조체와 클래스 인스턴스)는 모두 값으로 전달되며, 두 메서드 모두 인수의 한 필드 값을 변경합니다. 그러나 구조체를 전달할 때 전달되는 내용과 클래스 인스턴스를 전달할 때 전달되는 내용이 다르기 때문에 두 메서드의 결과는 서로 다릅니다.

구조체는 [값 형식\(C# 참조\)](#)이므로 메서드에 [구조체를 값으로 전달하는 경우](#) 메서드가 구조체 인수의 복사본을 받아서 작동합니다. 메서드가 호출 메서드의 원래 구조체에 액세스할 수 없으므로 어떤 방식으로든 변경할 수 없습니다. 메서드는 복사본만 변경할 수 있습니다.

클래스 인스턴스는 값 형식이 아니라 [참조 형식](#)입니다. 메서드에 [참조 형식을 값으로 전달하는 경우](#) 메서드가 클래스 인스턴스에 대한 참조의 복사본을 받습니다. 즉, 호출된 메서드는 인스턴스 주소의 복사본을 수신하고, 호출 메서드는 인스턴스의 원래 주소를 보유합니다. 호출 메서드의 클래스 인스턴스에 주소가 있고, 호출된 메서드의 매개 변수에 주소의 복사본이 있으며, 두 주소가 모두 동일한 개체를 참조합니다. 매개 변수에 주소의 복사본만 포함되므로 호출된 메서드는 호출 메서드의 클래스 인스턴스 주소를 변경할 수 없습니다. 그러나 호출된 메서드는 주소의 복사본을 사용하여 원래 주소와 주소의 복사본이 모두 참조하는 클래스 멤버에 액세스할 수 있습니다. 호출된 메서드가 클래스 멤버를 변경하는 경우 호출 메서드의 원래 클래스 인스턴스도 변경됩니다.

다음 예제의 출력에서 차이점을 보여 줍니다. `ClassTaker` 메서드가 매개 변수의 주소를 사용하여 클래스 인스턴스의 지정된 필드를 찾기 때문에 클래스 인스턴스의 `willIChange` 필드 값은 해당 메서드 호출에 의해 변경됩니다. 인수 값이 해당 주소의 복사본이 아니라 구조체 자체의 복사본이기 때문에 호출 메서드의 구조체

`willIChange` 필드는 `StructTaker` 메서드 호출에 의해 변경되지 않습니다. `StructTaker`는 복사본을 변경하고, `StructTaker` 호출이 완료되면 복사본이 손실됩니다.

## 예제

```

using System;

class TheClass
{
    public string willIChange;
}

struct TheStruct
{
    public string willIChange;
}

class TestClassAndStruct
{
    static void ClassTaker(TheClass c)
    {
        c.willIChange = "Changed";
    }

    static void StructTaker(TheStruct s)
    {
        s.willIChange = "Changed";
    }

    static void Main()
    {
        TheClass testClass = new TheClass();
        TheStruct testStruct = new TheStruct();

        testClass.willIChange = "Not Changed";
        testStruct.willIChange = "Not Changed";

        ClassTaker(testClass);
        StructTaker(testStruct);

        Console.WriteLine("Class field = {0}", testClass.willIChange);
        Console.WriteLine("Struct field = {0}", testStruct.willIChange);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   Class field = Changed
   Struct field = Not Changed
*/

```

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [클래스](#)
- [구조체 형식](#)
- [매개 변수 전달](#)

# 암시적 형식 지역 변수(C# 프로그래밍 가이드)

2020-11-02 • 13 minutes to read • [Edit Online](#)

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 배열](#)을 참조하세요.

다음 예제에서는 `var`을 사용하여 지역 변수를 선언하는 다양한 방법을 보여 줍니다.

```
// 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

```
for (var x = 1; x < 10; x++)
```

- `foreach` 초기화 문에서 사용

```
foreach (var item in list) {...}
```

- `using` 문에서 사용

```
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.

```
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.

```
private var bookTitles;
```

`bookTitles`는 `var` 형식의 클래스 필드입니다. 이 필드는 평가할 식이 없으므로 어떤 형식의 `bookTitles`가 될

지 컴파일러가 추론하는 것이 불가능합니다. 또한 필드에 식을 추가(로컬 변수에서처럼)하는 것으로는 부족합니다.

```
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 패러독스를 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 사용하는데 문제가 없습니다.

```
// 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\(Language-Integrated Query\)](#)
- `for`
- `foreach, in`
- `using` 문

# 쿼리 식에서 암시적으로 형식화된 지역 변수 및 배열을 사용하는 방법(C# 프로그래밍 가이드)

2021-02-18 • 4 minutes to read • [Edit Online](#)

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.

```
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);
    }
}
```

## 예제

다음 예제에서는 유사하지만 `var` 사용이 선택 사항인 상황에서 `var` 키워드를 사용합니다. `student.LastName`이 문자열이기 때문에 쿼리를 실행하면 문자열 시퀀스가 반환됩니다. 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.

```
// 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

- [C# 프로그래밍 가이드](#)
- [확장명 메서드](#)
- [LINQ\(Language-Integrated Query\)](#)
- [var](#)
- [C#의 LINQ](#)

# 확장명 메서드(C# 프로그래밍 가이드)

2020-11-02 • 24 minutes to read • [Edit Online](#)

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 [IEnumerable<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 식](#)을 참조하세요.

```
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 변수 앞에 `이` 한정자가 있습니다. 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.

```
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.

```
using ExtensionMethods;
```

또한 다음 구문을 사용하여 애플리케이션에서 확장 메서드를 호출할 수 있습니다.

```
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. 실제로 캡슐화의 원칙을 위반하지 않습니다. 확장명 메서드는 확장하는 형식의 `private` 변수에 액세스할 수 없습니다.

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.

```
using System.Linq;
```

`System.Core.dll`에 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. 일치 항목이 없으면 형식에 대해 정의된 확장 메서드를 검색하고 찾은 첫 번째 확장 메서드에 바인딩합니다. 다음 예제에서는 컴파일러가 바인딩 할 확장명 메서드 또는 인스턴스 메서드를 확인하는 방법을 보여 줍니다.

## 예제

다음 예제에서는 C# 컴파일러가 메서드 호출을 형식의 인스턴스 메서드 또는 확장명 메서드에 바인딩할 것인지 결정할 때 따르는 규칙을 보여 줍니다. 정적 클래스 `Extensions`는 `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 일치 하므로 호출되지 않습니다.

일치하는 시그니처를 가진 인스턴스 메서드를 찾을 수 없으면 컴파일러는 일치하는 확장명 메서드(있는 경우)에 바인딩 합니다.

```
// Define an interface named IMyInterface.
namespace DefineIMyInterface
{
    using System;

    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)
        {

```

```

static void Main()
{
    // 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 통신하는데 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.

```
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.File` 및 `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. C# 7.2부터 확장 메서드의 첫 번째 인수에 `ref` 한정자를 추가할 수 있습니다. `ref` 한정자를 추가하면 첫 번째 인수가 참조로 전달됩니다. 이렇게 하면 확장되는 구조체의 상태를 변경하는 확장 메서드를 작성할 수 있습니다.

## 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.

- 시그니처가 형식에 정의된 메서드와 동일한 확장 메서드는 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 사용해 서는 안 됩니다. 소스 코드를 소유하고 있는 라이브러리에 중요 기능을 추가하려는 경우 어셈블리 버전 관리를 위한 .NET 지침을 따르세요. 자세한 내용은 [어셈블리 버전 관리](#)를 참조하세요.

## 참고 항목

- C# 프로그래밍 가이드
- 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

# 사용자 지정 확장명 메서드 구현 및 호출 방법(C# 프로그래밍 가이드)

2021-02-18 • 5 minutes to read • [Edit Online](#)

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.

```

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 기준 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)

2021-02-18 • 2 minutes to read • [Edit Online](#)

확장 메서드를 사용하여 특정 열거형 형식과 관련된 기능을 추가할 수 있습니다.

## 예제

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.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Linq;

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 변

수의 현재 값을 반영합니다. 이는 확장 메서드가 정의된 정적 클래스에서 내부적으로 직접 호출됨을 보여 줍니다.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [확장명 메서드](#)

# 명명된 인수와 선택적 인수(C# 프로그래밍 가이드)

2021-02-18 • 18 minutes to read • [Edit Online](#)

C# 4에서는 명명된 인수와 선택적 인수가 도입되었습니다. 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 호출된 메서드의 매개 변수 목록에 있는 매개 변수의 순서와 일치시킬 필요가 없습니다. 각 인수에 대한 매개 변수를 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.

```
PrintOrderDetails("Gift Shop", 31, "Red Mug");
```

매개 변수의 순서를 기억하지 못하지만 해당 이름을 알고 있는 경우 임의의 순서로 인수를 보낼 수 있습니다.

```
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

```
PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
```

- C# 7.2로 시작하고 올바른 위치에 사용되는 한 유효합니다. 아래 예제에서 `orderNum` 매개 변수는 올바른 위치에 있지만 명시적으로 명시되지 않습니다.

```
PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");
```

잘못된 순서의 명명된 인수 다음에 오는 위치 인수는 유효하지 않습니다.

```
// This generates CS1738: Named argument specifications must appear after all fixed arguments have been
specified.
PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
```

## 예제

다음 코드는 몇 가지 추가 코드와 함께 이 섹션의 예제를 구현합니다.

```
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");      // C# 7.2 onwards
        PrintOrderDetails("Gift Shop", orderNum: 31, "Red Mug");                      // C# 7.2 onwards

        // 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}");
    }
}
```

## 선택적 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.

```
//anExample.ExampleMethod(3, ,4);
```

그러나 세 번째 매개 변수의 이름을 알고 있으면 명명된 인수를 사용하여 작업을 수행할 수 있습니다.

```
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])
```

#### NOTE

.NET [OptionalAttribute](#) 클래스를 사용하여 선택적 매개 변수를 선언할 수도 있습니다. `OptionalAttribute` 매개 변수는 기본값이 필요하지 않습니다.

## 예제

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.

```
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.

## COM 인터페이스

동적 객체에 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])
)

```

C# 3.0 및 이전 버전에서는 다음 예제와 같이 각 매개 변수에 대한 인수가 필요합니다.

```

// In C# 3.0 and earlier versions, you need to supply an argument for
// every parameter. The following call specifies a value for the first
// parameter, and sends a placeholder value for the other six. The
// default values are used for those parameters.
var excelApp = new Microsoft.Office.Interop.Excel.Application();
excelApp.Workbooks.Add();
excelApp.Visible = true;

var myFormat =
    Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFormatAccounting1;

excelApp.get_Range("A1", "B4").AutoFormat(myFormat, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing);

```

그러나 C# 4.0에서 도입된 명명된 인수와 선택적 인수를 사용하면 `AutoFormat` 호출을 훨씬 간소화할 수 있습니다. 명명된 인수와 선택적 인수를 사용하면 매개 변수의 기본값을 변경하지 않으려는 경우 선택적 매개 변수에 대한 인수를 생략할 수 있습니다. 다음 호출에서는 7개 매개 변수 중 하나에 대한 값만 지정되었습니다.

```

// The following code shows the same call to AutoFormat in C# 4.0. Only
// the argument for which you want to provide a specific value is listed.
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.

## Overload Resolution

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.

# Office 프로그래밍에 명명된 인수와 선택적 인수를 사용하는 방법(C# 프로그래밍 가이드)

2021-02-18 • 11 minutes to read • [Edit Online](#)

C# 4에서 도입된 명명된 인수와 선택적 인수는 C# 프로그래밍의 편의성, 유연성 및 가독성을 향상합니다. 또한 이러한 기능은 Microsoft Office 자동화 API와 같은 COM 인터페이스에 대한 액세스에 큰 도움이 됩니다.

다음 예제의 [ConvertToTable](#) 메서드에는 열과 행 수, 서식, 테두리, 글꼴, 색 등의 테이블 특성을 나타내는 매개 변수 16개가 있습니다. 대부분의 경우 모든 매개 변수에 대해 특정 값을 지정하지는 않으므로 16개 매개 변수는 모두 선택 사항입니다. 그러나 명명된 인수와 선택적 인수를 사용하지 않을 경우 각 매개 변수에 대해 값 또는 자리 표시자 값을 제공해야 합니다. 명명된 인수와 선택적 인수를 사용할 경우 프로젝트에 필요한 매개 변수의 값만 지정합니다.

이 절차를 완료하려면 컴퓨터에 Microsoft Office Word가 설치되어 있어야 합니다.

## NOTE

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.

## 새 콘솔 애플리케이션을 만들려면

- Visual Studio를 시작합니다.
- 파일 메뉴에서 **새로 만들기**를 가리킨 다음 **프로젝트**를 클릭합니다.
- 템플릿 범주 창에서 **Visual C#**을 확장한 다음 **Windows**를 클릭합니다.
- 템플릿 창의 맨 위에서 **.NET Framework 4**가 대상 프레임워크 상자에 표시되는지 확인합니다.
- 템플릿 창에서 **콘솔 애플리케이션**을 클릭합니다.
- 이름 필드에 프로젝트의 이름을 입력합니다.
- 확인 을 클릭합니다.

솔루션 탐색기 에 새 프로젝트가 표시됩니다.

## 참조를 추가하려면

- 솔루션 탐색기에서 프로젝트 이름을 마우스 오른쪽 단추로 클릭하고 **참조 추가**를 클릭합니다. 참조 추가 대화 상자가 나타납니다.
- .NET 페이지의 구성 요소 이름 목록에서 **Microsoft.Office.Interop.Word**를 선택합니다.
- 확인 을 클릭합니다.

## 필요한 using 지시문을 추가하려면

- 솔루션 탐색기에서 *Program.cs* 파일을 마우스 오른쪽 단추로 클릭하고 **코드 보기**를 클릭합니다.
- 다음 **using** 지시문을 코드 파일의 맨 위에 추가합니다.

```
using Word = Microsoft.Office.Interop.Word;
```

## Word 문서에 텍스트를 표시하려면

1. `Program.cs`의 `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.

```
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();
}
```

2. 메서드의 끝에 다음 코드를 추가하여 문서에서 텍스트를 표시할 위치 및 표시할 텍스트를 정의합니다.

```
// 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. . .");
```

## 애플리케이션을 실행하려면

1. 다음 문을 Main에 추가합니다.

```
DisplayInWord();
```

2. **Ctrl+F5**를 눌러 프로젝트를 실행합니다. 지정된 텍스트를 포함하는 Word 문서가 나타납니다.

## 텍스트를 표로 변경하려면

1. `ConvertToTable` 메서드를 사용하여 텍스트를 표로 둡습니다. 이 메서드에는 선택적 매개 변수 16개가 있습니다. IntelliSense는 다음 그림과 같이 선택적 매개 변수를 중괄호로 둡습니다.

```
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.

`DisplayInWord` 메서드의 끝에 다음 코드를 추가하여 간단한 표를 만듭니다. 인수는 `range`의 텍스트 문자열에 있는 쉼표가 표의 셀을 구분하도록 지정합니다.

```
// Convert to a simple table. The table will have a single row with  
// three columns.  
range.ConvertToTable(Separator: ",");
```

이전 버전의 C#에서 `ConvertToTable` 을 호출하려면 다음 코드와 같이 각 매개 변수에 대한 참조 인수가 필요합니다.

```
// Call to ConvertToTable in Visual C# 2008 or earlier. This code  
// is not part of the solution.  
var missing = Type.Missing;  
object separator = ",";  
range.ConvertToTable(ref separator, ref missing, ref missing,  
    ref missing, ref missing, ref missing,  
    ref missing, ref missing, ref missing,  
    ref missing, ref missing, ref missing,  
    ref missing);
```

2. **Ctrl+F5**를 눌러 프로젝트를 실행합니다.

## 다른 매개 변수로 실험하려면

1. 열 1개와 행 3개가 포함되도록 표를 변경하려면 `DisplayInWord`의 마지막 줄을 다음 문으로 바꾼 다음 **Ctrl+F5**를 입력합니다.

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);
```

2. 미리 정의된 표 형식을 지정하려면 `DisplayInWord`의 마지막 줄을 다음 문으로 바꾼 다음 **Ctrl+F5**를 입력합니다. 형식은 [WdTableFormat](#) 상수 중 하나일 수 있습니다.

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,  
    Format: Word.WdTableFormat.wdTableFormatElegant);
```

## 예제

다음 코드에는 전체 예제가 포함되어 있습니다.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
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);
        }
    }
}

```

## 참조

- 명명된 인수 및 선택적 인수

# 생성자(C# 프로그래밍 가이드)

2020-11-02 • 6 minutes to read • [Edit Online](#)

`class` 또는 `struct`를 만들 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 [생성자 사용 및 인스턴스 생성자](#)를 참조하세요.

## 매개 변수 없는 생성자

클래스에 대한 생성자를 제공하지 않으면 C#이 기본적으로 개체를 인스턴스화하고 멤버 변수를 [C# 형식의 기본값](#)에 나열된 기본값으로 설정하는 생성자를 새로 만듭니다. 구조체에 대한 생성자를 제공하지 않으면 C#이 [암시적 매개 변수 없는 생성자](#)를 사용하여 각 필드를 자동으로 기본값으로 초기화합니다. 자세한 내용 및 예제는 [인스턴스 생성자](#)를 참조하세요.

## 생성자 구문

생성자는 이름이 해당 형식의 이름과 동일한 메서드입니다. 해당 메서드 시그니처에는 메서드 이름과 매개 변수 목록만 포함되고 반환 형식은 포함되지 않습니다. 다음 예제에서는 `Person`이라는 클래스에 대한 생성자를 보여 줍니다.

```
public class Person
{
    private string last;
    private string first;

    public Person(string lastName, string firstName)
    {
        last = lastName;
        first = firstName;
    }

    // Remaining implementation of Person class.
}
```

생성을 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.

```
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.

```
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.

```
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 생성자](#)를 참조하세요.

## 섹션 내용

[생성자 사용](#)

[인스턴스 생성자](#)

[전용 생성자](#)

[정적 생성자](#)

[복사 생성자 작성 방법](#)

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [클래스 및 구조체](#)
- [종료자](#)
- [static](#)
- [이니셜라이저가 생성자와 반대 순서로 실행되는 이유는 무엇인가요? 1부](#)

# 생성자 사용(C# 프로그래밍 가이드)

2021-02-18 • 11 minutes to read • [Edit Online](#)

`class` 또는 `struct`가 만들어지면 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.

```
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. 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.

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E; //2.71828...
}
```

자세한 내용은 [전용 생성자](#)를 참조하세요.

`struct` 형식의 생성자는 `class` 생성자와 비슷하지만, `structs`는 컴파일러에서 자동으로 제공되므로 명시적 매개 변수 없는 생성자를 포함할 수 없습니다. 이 생성자는 `struct`의 각 필드를 [기본값](#)으로 초기화합니다. 그러나 이 매개 변수 없는 생성자는 `struct`가 `new`로 인스턴스화될 경우에만 호출됩니다. 예를 들어, 이 코드는 `Int32`에 매개 변수 없는 생성자를 사용하므로 정수가 초기화되었음을 확인할 수 있습니다.

```
int i = new int();
Console.WriteLine(i);
```

그러나 다음 코드는 `new` 를 사용하지 않으므로, 그리고 초기화되지 않은 개체를 사용하려고 하므로 컴파일러 오류를 일으킵니다.

```
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.

```
int a = 44; // Initialize the value type...
int b;
b = 33; // Or assign it before using it.
Console.WriteLine("{0}, {1}", a, b);
```

따라서 값 형식에 대해 매개 변수 없는 생성자를 호출할 필요가 없습니다.

클래스와 `structs` 모두 매개 변수를 사용하는 생성자를 정의할 수 있습니다. 매개 변수를 사용하는 생성자는 `new` 문 또는 `base` 문을 통해 호출해야 합니다. 클래스 및 `structs` 는 여러 생성자를 정의할 수 있으며, 매개 변수 없는 생성자를 정의하는 데에는 둘 다 필요하지 않습니다. 예를 들어:

```
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.

```
Employee e1 = new Employee(30000);
Employee e2 = new Employee(500, 52);
```

생성자는 `base` 키워드를 사용하여 기본 클래스의 생성자를 호출할 수 있습니다. 예를 들어:

```
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 클래스에서는 `base` 키워드를 사용해 매개 변수 없는 클래스 생성자를 명시적으로 호출하지 않으면, 기본 생성자(있는 경우)가 암시적으로 호출됩니다. 즉, 다음과 같은 생성자 선언은 실제로 동일합니다.

```
public Manager(int initData)
{
    //Add further instructions here.
}
```

```
public Manager(int initData)
    : base()
{
    //Add further instructions here.
}
```

기본 클래스가 매개 변수 없는 생성자를 제공하지 않으면 파생 클래스는 `base`를 사용해 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.

```
public Employee(int weeklySalary, int numberOfWeeks)
    : this(weeklySalary * numberOfWeeks)
{}
```

이전 예제에서 `this` 키워드를 사용하면 이 생성자가 호출됩니다.

```
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 방법을 정의합니다. 자세한 내용은 [액세스 한정자](#)를 참조하세요.

`static` 키워드를 사용하여 생성자를 정적으로 선언할 수 있습니다. 정적 생성자는 정적 필드에 액세스하기 직전에 자동으로 호출되며 일반적으로 정적 클래스 멤버를 초기화하는 데 사용됩니다. 자세한 내용은 [정적 생성자](#)를 참조하세요.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 [인스턴스 생성자](#) 및 [정적 생성자](#)를 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [클래스 및 구조체](#)

- 생성자
- 종료자

# 인스턴스 생성자(C# 프로그래밍 가이드)

2021-02-18 • 9 minutes to read • [Edit Online](#)

인스턴스 생성자는 `new` 식을 사용하여 `class`의 개체를 만들 때 인스턴스 멤버 변수를 만들고 초기화하는 데 사용됩니다. 정적 클래스 또는 비정적 클래스의 정적 변수를 초기화하려면 정적 생성자를 정의합니다. 자세한 내용은 [정적 생성자](#)를 참조하세요.

다음 예제에서는 인스턴스 생성자를 보여 줍니다.

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

    // constructor
    public Coords()
    {
        x = 0;
        y = 0;
    }
}
```

## NOTE

이해를 돋기 위해 이 클래스에는 `public` 필드가 포함되어 있습니다. `public` 필드를 사용하면 어디에 있든 프로그램 내의 모든 메서드가 확인되지 않고 개체의 내부 작업에 무제한 액세스할 수 있으므로 프로그래밍 시 권장되지 않습니다. 데이터 멤버는 일반적으로 `private`여야 하며, 클래스 메서드 및 속성을 통해서만 액세스해야 합니다.

`Coords` 클래스를 기반으로 하는 개체를 만들 때마다 이 인스턴스 생성자가 호출됩니다. 인수를 사용하지 않는 이러한 생성자를 [매개 변수 없는 생성자](#)라고 합니다. 그러나 추가 생성자를 제공하는 것이 유용한 경우가 많습니다. 예를 들어 데이터 멤버에 대한 초기 값을 지정할 수 있는 생성자를 `Coords` 클래스에 추가할 수 있습니다.

```
// A constructor with two arguments.
public Coords(int x, int y)
{
    this.x = x;
    this.y = y;
}
```

이렇게 하면 기본 또는 특정 초기 값으로 `Coords` 개체를 다음과 같이 만들 수 있습니다.

```
var p1 = new Coords();
var p2 = new Coords(5, 3);
```

클래스에 생성자가 없는 경우 매개 변수 없는 생성자가 자동으로 생성되고 개체 필드를 초기화하는 데 기본값이 사용됩니다. 예를 들어 `int`가 0으로 초기화됩니다. 형식 기본값에 대한 자세한 내용은 [C# 형식의 기본값](#)을 참조하세요. 따라서 `Coords` 클래스 매개 변수 없는 생성자는 모든 데이터 멤버를 0으로 초기화하기 때문에 클래스의 작동 방식을 변경하지 않고 완전히 제거할 수 있습니다. 여러 생성자를 사용하는 전체 예제는 이 항목의 뒷 부분에 있는 예제 1에서 제공되고, 자동으로 생성된 생성자의 예는 예제 2에서 제공됩니다.

인스턴스 생성자를 사용하여 기본 클래스의 인스턴스 생성자를 호출할 수도 있습니다. 클래스 생성자는 다음과 같이 이니셜라이저를 통해 기본 클래스의 생성자를 호출할 수 있습니다.

```

class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {
    }
}

```

이 예제에서 `circle` 클래스는 `Circle`이 파생되는 `Shape`에서 제공하는 생성자에 반경과 높이를 나타내는 값을 전달합니다. `Shape` 및 `Circle`을 사용하는 전체 예제는 이 항목에서 예제 3으로 나와 있습니다.

## 예제 1

다음 예제에서는 인수 없는 클래스 생성자와 두 개의 인수를 사용하는 클래스 생성자가 있는 클래스를 보여 줍니다.

```

class Coords
{
    public int x, y;

    // Default constructor.
    public Coords()
    {
        x = 0;
        y = 0;
    }

    // A constructor with two arguments.
    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    // Override the ToString method.
    public override string ToString()
    {
        return $"({x},{y})";
    }
}

class MainClass
{
    static void Main()
    {
        var p1 = new Coords();
        var p2 = new Coords(5, 3);

        // Display the results using the overriden ToString method.
        Console.WriteLine($"Coords #1 at {p1}");
        Console.WriteLine($"Coords #2 at {p2}");
        Console.ReadKey();
    }
}
/* Output:
Coords #1 at (0,0)
Coords #2 at (5,3)
*/

```

## 예제 2

이 예제에서 `Person` 클래스에는 생성자가 없으며, 매개 변수 없는 생성자가 자동으로 제공되고 필드는 해당 기

본값으로 초기화됩니다.

```
using System;

public class Person
{
    public int age;
    public string name;
}

class TestPerson
{
    static void Main()
    {
        var person = new Person();

        Console.WriteLine($"Name: {person.name}, Age: {person.age}");
        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Name: , Age: 0
```

`age`의 기본값은 `0`이고, `name`의 기본값은 `null`입니다.

### 예제 3

다음 예제에서는 기본 클래스 이니셜라이저를 사용하는 방법을 보여 줍니다. `Circle` 클래스는 제네릭 클래스 `Shape`에서 파생되고, `Cylinder` 클래스는 `Circle` 클래스에서 파생됩니다. 각 파생 클래스의 생성자는 해당 기본 클래스 이니셜라이저를 사용합니다.

```

using System;

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()
    {
        return pi * x * x;
    }
}

class Cylinder : Circle
{
    public Cylinder(double radius, double height)
        : base(radius)
    {
        y = height;
    }

    public override double Area()
    {
        return (2 * base.Area()) + (2 * pi * x * y);
    }
}

class TestShapes
{
    static void Main()
    {
        double radius = 2.5;
        double height = 3.0;

        Circle ring = new Circle(radius);
        Cylinder tube = new Cylinder(radius, height);

        Console.WriteLine("Area of the circle = {0:F2}", ring.Area());
        Console.WriteLine("Area of the cylinder = {0:F2}", tube.Area());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   Area of the circle = 19.63
   Area of the cylinder = 86.39
*/

```

기본 클래스 생성자 호출에 대한 추가 예제는 [virtual](#), [override](#), [base](#)를 참조하세요.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [클래스 및 구조체](#)
- [생성자](#)
- [종료자](#)
- [static](#)

# 전용 생성자(C# 프로그래밍 가이드)

2021-02-18 • 4 minutes to read • [Edit Online](#)

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. 예를 들어:

```
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 하는 데 사용됩니다. 클래스의 모든 메서드가 정적인 경우 전체 클래스를 정적 클래스로 만드는 것이 좋습니다. 자세한 내용은 [정적 클래스 및 정적 클래스 멤버](#)를 참조하세요.

## 예제

다음은 private 생성자를 사용하는 클래스의 예입니다.

```

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# 언어 사양

자세한 내용은 [C# 언어 사양의 전용 생성자](#)를 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [클래스 및 구조체](#)
- [생성자](#)
- [종료자](#)
- [private](#)
- [public](#)

# 정적 생성자(C# 프로그래밍 가이드)

2021-02-18 • 10 minutes to read • [Edit Online](#)

정적 생성자는 정적 데이터를 초기화하거나 한 번만 수행해야 하는 특정 작업을 수행하는 데 사용됩니다. 첫 번째 인스턴스가 만들어지거나 정적 멤버가 참조되기 전에 자동으로 호출됩니다.

```
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;
    }
}
```

## 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 첫 번째 인스턴스가 만들어지거나 정적 멤버가 참조되기 전에 자동으로 호출되어 [클래스](#)를 초기화합니다. 정적 생성자는 인스턴스 생성자보다 먼저 실행됩니다. 유형의 정적 생성자는 이벤트 또는 대리자에 할당된 정적 메서드가 호출될 때 호출되며 할당될 때는 호출되지 않습니다. 정적 필드 변수 이니셜라이저가 정적 생성자의 클래스에 있는 경우 정적 생성자가 실행되기 직전에, 클래스 선언에 나타나는 텍스트 순서대로 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, 프로그램을 실행 중인 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 암시적 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.

## NOTE

직접 액세스할 수 없더라도, 명시적 정적 생성자가 있을 경우 초기화 예외 문제 해결에 도움이 되도록 문서화해야 합니다.

## 사용법

- 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 제약 조건(형식 매개 변수 제약 조건)을 통해 컴파일 시간에 검사할 수 없는 형식 매개 변수에 런타임 검사를 편리하게 적용할 수 있습니다.

## 예제

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.

```
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 global start time {2}.",
            this.RouteNumber,
            elapsedTime.Milliseconds,
            globalStartTime.ToString());
    }
}
```

```

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](#) 섹션을 참조하세요.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [클래스 및 구조체](#)
- [생성자](#)
- [정적 클래스 및 정적 클래스 멤버](#)
- [종료자](#)
- [생성자 디자인 지침](#)
- [보안 경고 - CA2121: 정적 생성자는 private이어야 합니다.](#)

# 복사 생성자를 작성하는 방법(C# 프로그래밍 가이드)

2021-02-18 • 2 minutes to read • [Edit Online](#)

C#에서는 개체에 대한 복사 생성자를 제공하지 않지만 직접 작성할 수 있습니다.

## 예제

다음 예제에서 `Person` 클래스는 `Person` 인스턴스를 해당 인수로 사용하는 복사 생성자를 정의합니다. 인수의 속성 값이 `Person`의 새 인스턴스 속성에 할당됩니다. 코드에는 복사하려는 인스턴스의 `Name` 및 `Age` 속성을 클래스의 인스턴스 생성자에 보내는 대체 복사 생성자가 포함되어 있습니다.

```

using System;

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

```

- [ICloneable](#)
- [C# 프로그래밍 가이드](#)
- [클래스 및 구조체](#)
- [생성자](#)
- [종료자](#)

# 종료자(C# 프로그래밍 가이드)

2021-02-18 • 9 minutes to read • [Edit Online](#)

종료자(소멸자라고도 함)는 가비지 수집기에서 클래스 인스턴스를 수집할 때 필요한 최종 정리를 수행하는 데 사용됩니다.

## 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.

```
class Car
{
    ~Car() // finalizer
    {
        // cleanup statements...
    }
}
```

다음 예제와 같이 종료자를 식 본문 정의로 구현할 수도 있습니다.

```
using System;

public class Destroyer
{
    public override string ToString() => GetType().Name;

    ~Destroyer() => Console.WriteLine($"The {ToString()} destructor is executing.");
}
```

종료자는 개체의 기본 클래스에서 `Finalize`를 암시적으로 호출합니다. 따라서 종료자 호출은 다음 코드로 암시적으로 변환됩니다.

```
protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}
```

이 설계에서는 `Finalize` 메서드가 상속 체인의 모든 인스턴스에 대해 최다 파생에서 최소 파생까지 재귀적으

로 호출됩니다.

#### NOTE

빈 종료자는 사용할 수 없습니다. 클래스에 종료자가 포함되어 있으면 `Finalize` 큐에서 항목이 생성됩니다. 종료자를 호출하면 가비지 수집기가 호출되어 큐를 처리합니다. 종료자가 비어 있으면 성능이 불필요하게 저하됩니다.

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.

.NET Framework 애플리케이션에서(.NET Core 애플리케이션에서가 아닌) 프로그램이 종료될 때 종료자도 호출됩니다.

`Collect`을 호출하여 강제로 가비지를 수집할 수 있지만 대체로 성능 이슈가 발생할 수 있으므로 이 호출을 피해야 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
- `using` 문

## 예제

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 최다 파생 클래스의 인스턴스가 만들어집니다. 프로그램이 실행되면 세 클래스에 대한 종료자가 최다 파생부터 최소 파생까지 순서대로 자동으로 호출됩니다.

```

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.");
    }
}

class TestDestructors
{
    static void Main()
    {
        Third t = new Third();
    }
}
/* Output (to VS Output Window):
   Third's finalizer is called.
   Second's finalizer is called.
   First's finalizer is called.
*/

```

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 소멸자](#) 섹션을 참조하세요.

## 참조

- [IDisposable](#)
- [C# 프로그래밍 가이드](#)
- [생성자](#)
- [가비지 수집](#)

# 개체 및 컬렉션 이니셜라이저(C# 프로그래밍 가이드)

2020-11-02 • 17 minutes to read • [Edit Online](#)

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 속성 사용을 확인합니다. 자세한 내용은 [자동으로 구현된 속성](#)을 참조하세요.

```
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;
    }
}
```

```
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.

C# 6부터 객체 이니셜라이저는 필드 및 속성을 할당하는 것 외에 인덱서를 설정할 수도 있습니다. 기본적인 `Matrix` 클래스를 예로 들어 보겠습니다.

```
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.

```

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 `IndexersExample`에 적절한 인덱서가 있는 경우 이 모든 것이 유효합니다.

```

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.

```

public string name;
public double Size { set { ... }; }
public char this[int i] { set { ... }; }
public string this[char c, int i] { set { ... }; }

```

## 익명 형식의 객체 이니셜라이저

객체 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 객체 이니셜라이저를 사용하는 경우에만 초기화될 수 있습니다.

```

var pet = new { Age = 10, Name = "Fluffy" };

```

무명 형식을 사용하면 LINQ 쿼리 식의 `select` 절에서 원래 시퀀스의 객체를 값과 모양이 원본과 다를 수 있는 객체로 변환할 수 있습니다. 이 기능은 각 객체의 일부 정보만 시퀀스에 저장하려는 경우에 유용합니다. 다음 예제에서 제품 객체(`p`)는 많은 필드와 메서드를 포함하며, 제품 이름과 단위 가격이 들어 있는 객체 시퀀스만만들려 한다고 가정합니다.

```

var productInfos =
    from p in products
    select new { p.ProductName, p.UnitPrice };

```

이 쿼리를 실행하면 다음 예제와 같이 `productInfos` 문에서 액세스할 수 있는 객체 시퀀스가 `foreach` 변수에 포함됩니다.

```
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.

```
select new {p.ProductName, Price = p.UnitPrice};
```

## 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 단순한 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.

```
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.

```
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.

```
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.

```
var numbers = new Dictionary<int, string>
{
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};
```

이전 샘플에서는 `Item[TKey]`을 호출하여 값을 설정하는 코드를 생성합니다. C# 6 이전에는 다음 구문을 사용하여 사전 및 다른 연관 컨테이너를 초기화할 수 있었습니다. 괄호와 할당이 있는 인덱서 구문 대신 여러 값이 있는 개체를 사용합니다.

```
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.

## 예

다음 예제에서는 개체 및 컬렉션 이니셜라이저의 개념을 결합합니다.

```

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.");
    }
}

```

다음 예제에서는 [IEnumerable](#)을 구현하고 여러 매개 변수가 있는 `Add` 메서드를 포함하는 개체를 보여줍니다. 이는 `Add` 메서드의 시그니처에 해당하는 목록의 항목당 여러 요소가 있는 컬렉션 이니셜라이저를 사용합니다.

```

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.

```

public class DictionaryExample
{
    class RudimentaryMultiValuedDictionary<TKey, TValue> : IEnumerable<KeyValuePair<TKey, List<TValue>>>
    {
        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]

```

```

    public TDictionary<TKey, TValue> TryGetValue(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);
        }
    }
}

```

```

        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

- [C# 프로그래밍 가이드](#)
- [C#의 LINQ](#)
- [익명 형식](#)

# 개체 이니셜라이저를 사용하여 개체를 초기화하는 방법(C# 프로그래밍 가이드)

2021-02-18 • 5 minutes to read • [Edit Online](#)

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.

## 예제

다음 예제에서는 개체 이니셜라이저를 사용하여 새 `StudentName` 형식을 초기화하는 방법을 보여 줍니다. 이 예제에서는 `StudentName` 형식의 속성을 설정합니다.

```
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(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 야구 득점표를 할당할 수 있습니다.

```

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"]);
    }
}

```

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [개체 이니셜라이저 및 컬렉션 이니셜라이저](#)

# 컬렉션 이니셜라이저를 사용하여 사전을 초기화하는 방법(C# 프로그래밍 가이드)

2021-02-18 • 3 minutes to read • [Edit Online](#)

`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.

## 예제

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.

```
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 둑고, 바깥쪽 중괄호는 `students` `Dictionary< TKey, TValue >`에 추가할 키/값 쌍에 대한 이니셜라이저를 둑습니다. 마지막으로, 사전에 대한 전체 컬렉션 이니셜라이저가 중괄호로 둑어 있습니다. 두 번째 초기화에서 할당의 왼쪽은 키이고, 오른쪽은 값이며 `StudentName`에 개체 이니셜라이저를 사용합니다.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [개체 이니셜라이저 및 컬렉션 이니셜라이저](#)

# 중첩 형식(C# 프로그래밍 가이드)

2021-02-18 • 3 minutes to read • [Edit Online](#)

클래스, 구조체, 대리자 또는 인터페이스 내에서 정의된 형식을 중첩 형식이라고 합니다. 예

```
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.

- 구조체의 중첩 형식은 `public`, `internal` 또는 `private`일 수 있습니다.

다음 예제에서는 `Nested` 클래스를 `public`으로 설정합니다.

```
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:

```
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.

```
Container.Nested nest = new Container.Nested();
```

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [클래스 및 구조체](#)
- [액세스 한정자](#)
- [생성자](#)

# Partial 클래스 및 메서드(C# 프로그래밍 가이드)

2021-02-18 • 18 minutes to read • [Edit Online](#)

클래스, 구조체, 인터페이스 또는 메서드의 정의를 둘 이상의 소스 파일에 분할할 수 있습니다. 각 소스 파일에는 형식 또는 메서드 정의 섹션이 있으며 모든 부분은 애플리케이션이 컴파일될 때 결합됩니다.

## partial 클래스

클래스 정의를 분할하는 것이 바람직한 몇 가지 상황이 있습니다.

- 대규모 프로젝트에서 작업하는 경우 클래스를 개별 파일에 분산하면 여러 프로그래머가 동시에 클래스에 대해 작업할 수 있습니다.
- 자동으로 생성된 소스로 작업하는 경우 소스 파일을 다시 만들지 않고도 클래스에 코드를 추가할 수 있습니다. Visual Studio에서는 Windows Forms, 웹 서비스 래퍼 코드 등에 만들 때 이 방식을 사용합니다. Visual Studio에서 만든 파일을 수정하지 않고도 이러한 클래스를 사용하는 코드를 만들 수 있습니다.
- 클래스 정의를 분할하려면 다음과 같이 **partial** 키워드 한정자를 사용합니다.

```
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 **abstract**로 선언된 경우 전체 형식이 **abstract**로 간주됩니다. 부분이 **sealed**로 선언된 경우 전체 형식이 **sealed**로 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.

### NOTE

대리자 또는 열거형 선언에서는 **partial** 한정자를 사용할 수 없습니다.

다음 예제에서는 중첩된 대상 형식 자체는 부분이 아니어도 중첩된 형식이 부분일 수 있음을 보여 줍니다.

```

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.

```

[SerializableAttribute]
partial class Moon { }

[ObsoleteAttribute]
partial class Moon { }

```

이러한 선언은 다음 선언과 동일합니다.

```

[SerializableAttribute]
[ObsoleteAttribute]
class Moon { }

```

다음은 모든 부분 형식(Partial Type) 정의에서 병합됩니다.

- XML 주석
- 인터페이스
- 제네릭 형식 매개 변수 특성
- 클래스 특성
- 멤버

예를 들어 다음 선언을 살펴보세요.

```

partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }

```

이러한 선언은 다음 선언과 동일합니다.

```

class Earth : Planet, IRotate, IRevolve { }

```

#### 제한

partial 클래스 정의로 작업할 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.

```

public partial class A { }
//public class A { } // Error, must also be marked partial

```

- `partial` 한정자는 `class`, `struct` 또는 `interface` 키워드 바로 앞에만 올 수 있습니다.
- 다음 예제와 같이 부분 형식(Partial Type) 정의에 중첩된 부분 형식(Partial Type)을 사용할 수 있습니다.

```
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 있을 수 없습니다.
- 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(Partial Type) 정의에서 다음 키워드는 선택 사항이지만, 부분 형식(Partial Type) 정의에 있는 경우 동일한 형식의 다른 부분 정의에서 지정된 키워드와 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 1

### 설명

다음 예제에서는 `Coords` 클래스의 생성자 및 필드가 하나의 `partial` 클래스 정의에서 선언되고 `PrintCoords` 멤버가 다른 `partial` 클래스 정의에서 선언됩니다.

### 코드

```

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

```

## 예제 2

### 설명

다음 예제에서는 partial 구조체와 인터페이스도 개발할 수 있음을 보여 줍니다.

### 코드

```

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 구조체에는 부분 메서드(Partial Method)가 포함될 수 있습니다. 클래스의 한 부분에는 메서드의 시그니처가 포함되어 있습니다. 동일한 부분이나 다른 부분에서 선택적 구현을 정의할 수 있습니다. 구현을 제공하지 않으면 메서드와 모든 메서드 호출이 컴파일 시간에 제거됩니다.

부분 메서드(Partial Method)를 사용하면 클래스 중 한 부분의 구현자가 이벤트와 비슷하게 메서드를 정의할 수 있습니다. 클래스 중 다른 부분의 구현자는 메서드를 구현할지 여부를 결정할 수 있습니다. 메서드가 구현되지 않은 경우 컴파일러는 메서드 시그니처 및 모든 메서드 호출을 제거합니다. 호출의 인수 평가에서 발생하는 모든 결과를 포함하여 메서드 호출은 런타임에 영향을 주지 않습니다. 따라서 partial 클래스의 코드는 구현이 제공되지 않은 경우에도 부분 메서드(Partial Method)를 자유롭게 사용할 수 있습니다. 메서드가 호출되었지만 구현되지 않은 경우 컴파일 시간 또는 런타임 오류가 발생하지 않습니다.

부분 메서드(Partial Method)는 생성된 코드를 사용자 지정하는 방법으로 특히 유용합니다. 생성된 코드가 메서드를 호출할 수 있지만 개발자가 메서드를 구현할지 여부를 결정할 수 있도록 메서드 이름과 시그니처를 예약할 수 있습니다. partial 클래스와 마찬가지로, 부분 메서드(Partial Method)는 코드 생성기에 의해 생성된 코드와 개발자가 직접 만든 코드가 런타임 비용 없이 함께 작동할 수 있도록 합니다.

부분 메서드(Partial Method) 선언은 정의와 구현, 두 부분으로 이루어집니다. partial 클래스의 개별 부분이나 동일한 부분에 있을 수 있습니다. 구현 선언이 없는 경우 컴파일러는 정의하는 선언과 모든 메서드 호출을 최적화합니다.

```
// Definition in file1.cs
partial void onNameChanged();

// Implementation in file2.cs
partial void onNameChanged()
{
    // method body
}
```

- 부분 메서드(Partial Method) 선언은 상황별 키워드 `partial`로 시작해야 하고, 메서드에서 `void`를 반환해야 합니다.
- 부분 메서드(Partial Method)는 `in` 또는 `ref` 매개 변수를 사용할 수 있지만 `out` 매개 변수는 사용할 수 없습니다.
- 부분 메서드(Partial Method)는 암시적으로 `private`이므로 `가상`일 수 없습니다.
- 부분 메서드(Partial Method)는 본문의 존재 여부에 따라 정의하는지, 구현하는지가 결정되기 때문에 `extern`일 수 없습니다.
- 부분 메서드(Partial Method)는 `static` 및 `unsafe` 한정자를 사용할 수 없습니다.
- 부분 메서드(Partial Method)는 제네릭일 수 있습니다. 제약 조건은 정의하는 부분 메서드(Partial Method) 선언에 배치되며 필요에 따라 구현하는 선언에서 반복될 수 있습니다. 매개 변수 및 형식 매개 변수 이름이 정의하는 선언과 구현하는 선언에서 동일할 필요는 없습니다.
- 정의 및 구현된 부분 메서드(Partial Method)에 대한 대리자는 만들 수 있지만 정의만 된 부분 메서드(Partial Method)에 대한 대리자는 만들 수 없습니다.

## C# 언어 사양

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

## 참고 항목

- [C# 프로그래밍 가이드](#)

- 클래스
- 구조체 형식
- 인터페이스
- partial(형식)

# 익명 형식(C# 프로그래밍 가이드)

2020-11-02 • 10 minutes to read • [Edit Online](#)

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.

```
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. 따라서 쿼리에 작은 양의 데이터가 반환됩니다.

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

```
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);
}
```

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.

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

## 설명

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.

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. 익명 형식이나 익명 형식을 포함한 컬렉션을 메서드에 대한 인수로 전달하려면 매개 변수를 형식 개체로 선언하면 됩니다. 그러나 이렇게 하면 강력한 형식화를 사용하는 의미가 없습니다. 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.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [개체 이니셜라이저 및 컬렉션 이니셜라이저](#)
- [C#에서 LINQ 시작](#)
- [C#의 LINQ](#)

# 쿼리에서 요소 속성의 하위 집합을 반환하는 방법 (C# 프로그래밍 가이드)

2021-02-18 • 3 minutes to read • [Edit Online](#)

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.

```
select student.ID;
```

## 예제

다음 예제에서는 무명 형식을 사용하여 지정된 조건과 일치하는 각 소스 요소의 속성 하위 집합만 반환하는 방법을 보여 줍니다.

```
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.

```
select new { First = student.FirstName, Last = student.LastName };
```

앞의 예제에서 이 작업을 수행하는 경우 `Console.WriteLine` 문도 변경되어야 합니다.

```
Console.WriteLine(student.First + " " + student.Last);
```

## 코드 컴파일

이 코드를 실행하려면 System.Linq에 대한 `using` 지시문을 통해 클래스를 복사하여 C# 콘솔 애플리케이션 프로젝트에 붙여넣습니다.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [익명 형식](#)
- [C#의 LINQ](#)

# 인터페이스(C# 프로그래밍 가이드)

2020-11-02 • 14 minutes to read • [Edit Online](#)

인터페이스에는 비추상 [클래스](#) 또는 [구조체](#)에서 구현해야 하는 관련 기능 그룹에 대한 정의가 포함되어 있습니다. 인터페이스에서는 구현이 있어야 하는 `static` 메서드를 정의할 수 있습니다. C# 8.0부터 인터페이스는 구성을 위한 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 상속을 시뮬레이트하려는 경우 인터페이스를 사용해야 합니다.

다음 예제와 같이 `interface` 키워드를 사용하여 인터페이스를 정의합니다.

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

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

`IEquatable<T>` 인터페이스를 구현하는 모든 클래스나 구조체에는 인터페이스에서 지정한 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 구조체는 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 포함될 수 있습니다. 예제에 대한 링크는 [관련 단원](#)을 참조하세요. 인터페이스에는 인스턴스 필드, 인스턴스 생성자 또는 종료자가 포함될 수 없습니다. 인터페이스 멤버는 기본적으로 `public`입니다.

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

클래스 또는 구조체에서 인터페이스를 구현하는 경우 클래스 또는 구조체에서 인터페이스에서 선언하는 모든 멤버의 구현을 제공해야 하지만 기본 구현은 제공하지 않습니다. 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.

```

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\(C# 프로그래밍 가이드\)](#)을 참조하세요.

인터페이스는 하나 이상의 인터페이스에서 상속할 수 있습니다. 파생 인터페이스는 기본 인터페이스에서 멤버를 상속합니다. 파생 인터페이스를 구현하는 클래스는 파생 인터페이스의 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. 인터페이스에서 메서드의 기본 구현을 선언하는 경우 해당 인터페이스를 구현하는 모든 클래스가 해당 구현을 상속합니다. 인터페이스에 정의된 구현은 가상이며 구현하는 클래스가 해당 구현을 재정의할 수 있습니다.

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.

- 인터페이스는 일반적으로 추상 멤버만 갖는 추상 기본 클래스와 같습니다. 인터페이스를 구현하는 모든 클래스 또는 구조체는 모든 멤버를 구현해야 합니다. 필요에 따라 인터페이스에서 해당 멤버 일부 또는 모두의 기본 구현을 정의할 수 있습니다. 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.

## 관련 섹션

- [인터페이스 속성](#)
- [인터페이스의 인덱서](#)
- [인터페이스 이벤트를 구현하는 방법](#)
- [클래스 및 구조체](#)
- [상속](#)

- 인터페이스
- 메서드
- 다형성
- 추상/봉인된 클래스 및 클래스 멤버
- 속성
- 이벤트
- 인덱서

## 참조

- C# 프로그래밍 가이드
- 상속
- 식별자 이름

# 명시적 인터페이스 구현(C# 프로그래밍 가이드)

2020-11-02 • 6 minutes to read • [Edit Online](#)

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.

```
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.

```
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
```

두 인터페이스 멤버가 동일한 기능을 수행하지 않을 경우, 인터페이스들 중 하나 또는 둘 다 잘못 구현될 수 있습니다. 인터페이스를 통해서만 호출되고 해당 인터페이스에만 관련되는 클래스 멤버를 만드는 방식으로 인터페이스 멤버를 명시적으로 구현할 수 있습니다. 인터페이스 이름과 마침표를 사용하여 클래스 멤버의 이름을 지정하면 됩니다. 예를 들어:

```
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:

```
// Call the Paint methods from Main.

SampleClass obj = new SampleClass();
//obj.Paint(); // Compiler error.

IControl c = obj;
c.Paint(); // Calls IControl.Paint on SampleClass.

ISurface s = obj;
s.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:

```
interface ILeft
{
    int P { get; }
}
interface IRight
{
    int P();
}

class Middle : ILeft, IRight
{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}
```

C# 8.0부터 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.

```
public interface IControl
{
    void Paint() => Console.WriteLine("Default Paint method");
}
public class SampleClass : IControl
{
    // Paint() is inherited from IControl.
}
```

다음 샘플은 기본 구현을 호출합니다.

```
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](#)
- [클래스 및 구조체](#)
- [인터페이스](#)
- [상속](#)

# 인터페이스 멤버를 명시적으로 구현하는 방법(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

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.

## 예제

```

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
        // 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.

```
//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.

```
System.Console.WriteLine("Length: {0}", dimensions.GetLength());  
System.Console.WriteLine("Width: {0}", dimensions.GetWidth());
```

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [클래스 및 구조체](#)
- [인터페이스](#)
- [두 인터페이스의 멤버를 명시적으로 구현하는 방법](#)

# 두 인터페이스의 멤버를 명시적으로 구현하는 방법(C# 프로그래밍 가이드)

2020-11-02 • 3 minutes to read • [Edit Online](#)

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.

## 예제

```

// 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
*/

```

기본 측정값을 인치 단위로 설정하려면 일반적으로 Length 및 Width 메서드를 구현하고, IMetricDimensions 인터페이스에서 Length 및 Width 메서드를 명시적으로 구현합니다.

```
// 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.

```
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

- [C# 프로그래밍 가이드](#)
- [클래스 및 구조체](#)
- [인터페이스](#)
- [인터페이스 멤버를 명시적으로 구현하는 방법](#)

# 대리자(C# 프로그래밍 가이드)

2021-02-18 • 8 minutes to read • [Edit Online](#)

대리자는 특정 매개 변수 목록 및 반환 형식이 있는 메서드에 대한 참조를 나타내는 형식입니다. 대리자를 인스턴스화하면 모든 메서드가 있는 인스턴스를 호출되는 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.

```
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.

## NOTE

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 Expressions에 [C# 3.0 Cookbook, Third Edition: 250 개 이상의 솔루션에 대한 C# 3.0 프로그래머](#)
- 대리자 및 이벤트에 학습 [C# 3.0. 기본 사항 마스터 C# 3.0](#)

## 참조

- [Delegate](#)
- [C# 프로그래밍 가이드](#)
- [이벤트](#)

# 대리자 사용(C# 프로그래밍 가이드)

2020-11-02 • 14 minutes to read • [Edit Online](#)

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 `Del` 대리자를 선언합니다.

```
public delegate void Del(string message);
```

대리자 개체는 일반적으로 대리자가 래핑할 메서드의 이름을 제공하거나 [익명 함수](#)를 사용하여 생성합니다. 대리자를 인스턴스화하고 나면 대리자에 대한 메서드 호출이 대리자에 의해 해당 메서드로 전달됩니다. 호출자가 대리자에게 전달한 매개 변수가 메서드로 전달되며 메서드의 반환 값(있는 경우)이 대리자에 의해 호출자로 반환됩니다. 이 과정을 대리자 호출이라고 합니다. 인스턴스화된 대리자는 래핑된 메서드 자체인 것처럼 호출할 수 있습니다. 예를 들어:

```
// Create a method for a delegate.  
public static void DelegateMethod(string message)  
{  
    Console.WriteLine(message);  
}
```

```
// Instantiate the delegate.  
Del handler = DelegateMethod;  
  
// Call the delegate.  
handler("Hello World");
```

대리자 형식은 .NET의 `Delegate` 클래스에서 파생되며, [봉인](#)되어 있으므로 `Delegate`에서는 파생될 수 없으며 해당 클래스에서 사용자 지정 클래스를 파생할 수도 없습니다. 인스턴스화된 대리자는 개체이므로 매개 변수로 전달하거나 속성에 할당할 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 `Del` 형식을 매개 변수로 사용합니다.

```
public static void MethodWithCallback(int param1, int param2, Del callback)  
{  
    callback("The number is: " + (param1 + param2).ToString());  
}
```

위에서 작성된 대리자를 해당 메서드로 전달할 수 있습니다.

```
MethodWithCallback(1, 2, handler);
```

그러면 콘솔에 다음 출력이 표시됩니다.

```
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 대리자를 생성할 때 해당 대리자는 인스턴스와 메서드를 모두 참조합니다. 대리자는 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 대리자는 메서드만 참조합니다. 다음의 선언을 살펴보세요.

```
public class MethodClass
{
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

이제는 위에 나와 있는 정적 `DelegateMethod` 와 함께 3개 메서드를 `Del` 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. 예를 들어:

```
var obj = new MethodClass();
Del d1 = obj.Method1;
Del d2 = obj.Method2;
Del d3 = DelegateMethod;

//Both types of assignment are valid.
Del 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. 예를 들어:

```
//remove Method1
allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2
Del 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.

```
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. 예를 들면 다음과 같습니다.

```
delegate void Delegate1();
delegate void Delegate2();

static void method(Delegate1 d, Delegate2 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

- [C# 프로그래밍 가이드](#)
- [대리자](#)
- [대리자의 가변성 사용](#)
- [대리자의 가변성](#)
- [Func 및 Action 제네릭 대리자에 가변성 사용](#)
- [이벤트](#)

# 대리자 비교: 명명된 메서드 및 무명 메서드(C# 프로그래밍 가이드)

2020-11-02 • 4 minutes to read • [Edit Online](#)

대리자는 명명된 메서드에 연결할 수 있습니다. 명명된 메서드를 사용하여 대리자를 인스턴스화하면 메서드가 매개 변수로 전달됩니다. 예를 들면 다음과 같습니다.

```
// Declare a delegate.  
delegate void Del(int x);  
  
// Define a named method.  
void DoWork(int k) { /* ... */ }  
  
// Instantiate the delegate using the method as a parameter.  
Del 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. 자세한 내용은 [무명 함수](#)를 참조하세요.

## 설명

대리자 매개 변수로 전달하는 메서드에는 대리자 선언과 동일한 시그니처가 있어야 합니다.

대리자 인스턴스는 정적 또는 인스턴스 메서드를 캡슐화할 수 있습니다.

대리자는 `out` 매개 변수를 사용할 수 있지만, 멀티캐스트 이벤트 대리자의 경우 호출될 대리자를 알 수 없기 때문에 사용하지 않는 것이 좋습니다.

## 예제 1

다음은 대리자를 선언하고 사용하는 간단한 예제입니다. 대리자 `Del` 및 연결된 메서드 `MultiplyNumbers`에 동일한 시그니처가 있습니다.

```

// Declare a delegate
delegate void Del(int i, double j);

class MathClass
{
    static void Main()
    {
        MathClass m = new MathClass();

        // Delegate instantiation using "MultiplyNumbers"
        Del 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
*/

```

## 예제 2

다음 예제에서는 한 대리자가 정적 메서드와 인스턴스 메서드 모두에 매핑되며 각각의 특정 정보를 반환합니다.

```

// Declare a delegate
delegate void Del();

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:
        Del 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

- [C# 프로그래밍 가이드](#)
- [대리자](#)
- [대리자를 결합하는 방법\(멀티캐스트 대리자\)](#)
- [이벤트](#)

# 대리자를 결합하는 방법(멀티캐스트 대리자)(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

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.

## 예제

```

using System;

// Define a custom delegate that has a string parameter and returns void.
delegate void CustomDel(string s);

class TestClass
{
    // Define two methods that have the same signature as CustomDel.
    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.
        CustomDel 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;

        // Create the delegate object hiDel that references the
        // method Hello.
        hiDel = Hello;

        // Create 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](#)
- [C# 프로그래밍 가이드](#)
- [이벤트](#)

# 대리자를 선언, 인스턴스화, 사용하는 방법(C# 프로그래밍 가이드)

2021-02-18 • 9 minutes to read • [Edit Online](#)

C# 1.0 이상 버전에서는 다음 예제와 같이 대리자를 선언할 수 있습니다.

```
// 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}");  
}
```

```
// Create an instance of the delegate.  
Del del1 = new Del(Notify);
```

C# 2.0에서는 다음 예제와 같이 더욱 간단한 방법으로 이전 선언을 쓸 수 있습니다.

```
// C# 2.0 provides a simpler way to declare an instance of Del.  
Del del2 = Notify;
```

C# 2.0 이상 버전에서는 다음 예제와 같이 익명 메서드를 사용하여 `delegate`를 선언하고 초기화할 수도 있습니다.

```
// Instantiate Del by using an anonymous method.  
Del del3 = delegate(string name)  
{ Console.WriteLine($"Notification received for: {name}"); };
```

C# 3.0 이상에서는 다음 예제와 같이 람다식을 사용하여 대리자를 선언하고 인스턴스화할 수도 있습니다.

```
// Instantiate Del by using a lambda expression.  
Del del4 = name => { Console.WriteLine($"Notification received for: {name}"); };
```

자세한 내용은 [람다식](#)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

```

// 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.

```
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.

```
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.

```
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.

```
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](#)

# 배열(C# 프로그래밍 가이드)

2021-02-18 • 6 minutes to read • [Edit Online](#)

배열 데이터 구조에 형식이 동일한 변수를 여러 개 저장할 수 있습니다. 요소의 형식을 지정하여 배열을 선언합니다. 배열이 모든 형식의 요소를 저장하도록 하려는 경우 `object` 를 해당 형식으로 지정할 수 있습니다. C#의 통합 형식 시스템에서 모든 형식(사전 정의되거나 사용자 정의된 형식, 참조 형식, 값 형식)은 `Object`에서 직접 또는 간접적으로 상속합니다.

```
type[] arrayName;
```

## 예제

다음 예제에서는 단일 차원, 다차원 및 가변 배열을 만듭니다.

```
class TestArraysClass
{
    static void Main()
    {
        // Declare a single-dimensional array of 5 integers.
        int[] array1 = new int[5];

        // Declare and set array element values.
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax.
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

        // Declare a two dimensional array.
        int[,] multiDimensionalArray1 = new int[2, 3];

        // Declare and set array element values.
        int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

        // Declare a jagged array.
        int[][] jaggedArray = new int[6][];

        // Set the values of the first array in the jagged array structure.
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
    }
}
```

## 배열 개요

배열에는 다음과 같은 속성이 있습니다.

- 배열은 [단일 차원](#), [다차원](#) 또는 [가변](#)일 수 있습니다.
- 차원 수와 각 차원의 길이는 배열 인스턴스를 만들 때 설정됩니다. 이러한 값은 인스턴스의 수명 동안 변경할 수 없습니다.
- 숫자 배열 요소의 기본값은 0으로 설정되고, 참조 요소는 `null`로 설정됩니다.
- 가변 배열은 여러 배열로 구성되어 있기 때문에 해당 요소가 참조 형식이며, `null`로 초기화됩니다.
- 배열은 0으로 인덱싱됩니다. `n` 요소는 `0`부터 `n-1`로 인덱싱됩니다.
- 배열 요소 형식은 배열 형식을 비롯한 어떤 형식도 될 수 있습니다.
- 배열 형식은 [Array](#) 추상 기본 형식에서 파생된 [참조 형식](#)입니다. 이 형식은 [IEnumerable](#) 및

[IEnumerable<T>](#)을 구현하므로 C#의 모든 배열에서 `foreach` 반복을 사용할 수 있습니다.

### 개체 형식 배열

C#의 배열은 C 및 C++와 같이 인접한 메모리의 주소 지정 가능한 영역이 아니라 실제로 개체입니다. [Array](#)는 모든 배열 형식의 추상 기본 형식입니다. [Array](#)에 포함된 속성 및 다른 클래스 멤버를 사용할 수 있습니다. 이러한 예로 `Length` 속성을 사용하여 배열의 길이를 가져오는 경우가 있습니다. 다음 코드에서는 `numbers` 배열의 길이(5)를 `lengthOfNumbers`라는 변수에 할당합니다.

```
int[] numbers = { 1, 2, 3, 4, 5 };
int lengthOfNumbers = numbers.Length;
```

[Array](#) 클래스는 배열의 정렬, 검색 및 복사를 위한 다른 여러 유용한 메서드와 속성을 제공합니다. 다음 예제에서는 `Rank` 속성을 사용하여 배열의 차원 수를 표시합니다.

```
class TestArraysClass
{
    static void Main()
    {
        // Declare and initialize an array.
        int[,] theArray = new int[5, 10];
        System.Console.WriteLine("The array has {0} dimensions.", theArray.Rank);
    }
}
// Output: The array has 2 dimensions.
```

## 추가 정보

- [단일 차원 배열 사용 방법](#)
- [다차원 배열 사용 방법](#)
- [가변 배열 사용 방법](#)
- [배열에 foreach 사용](#)
- [인수로 배열 전달](#)
- [암시적으로 형식화된 배열](#)
- [C# 프로그래밍 가이드](#)
- [컬렉션](#)

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

# 1차원 배열(C# 프로그래밍 가이드)

2020-11-02 • 4 minutes to read • [Edit Online](#)

배열 요소 형식과 요소 수를 지정하는 `new` 연산자를 사용하여 1차원 배열을 만듭니다. 다음 예제에서는 정수가 5개인 배열을 선언합니다.

```
int[] array = new int[5];
```

이 배열에는 `array[0]` ~ `array[4]`의 요소가 포함되어 있습니다. 배열의 요소는 기본값, 즉 정수에 대해 요소 형식 `0`으로 초기화됩니다.

문자열 배열을 선언하는 다음 예제와 같이, 배열은 지정되는 모든 요소 형식을 저장할 수 있습니다.

```
string[] stringArray = new string[6];
```

## 배열 초기화

배열을 선언할 때 배열의 요소를 초기화할 수 있습니다. 길이 지정자는 초기화 목록의 요소 수에 따라 유추되므로 필요하지 않습니다. 예를 들어:

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

다음 코드는 각 배열 요소가 요일 이름으로 초기화되는 문자열 배열의 선언을 보여 줍니다.

```
string[] weekDays = new string[] { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

다음 코드와 같이 선언 시 배열을 초기화할 때 `new` 식과 배열 형식을 피할 수 있습니다. 이를 **암시적으로 형식화된 배열**이라고 합니다.

```
int[] array2 = { 1, 3, 5, 7, 9 };
string[] weekDays2 = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

배열 변수를 만들지 않고 선언할 수 있지만 이 변수에 새 배열을 할당할 때는 `new` 연산자를 사용해야 합니다. 예를 들어:

```
int[] array3;
array3 = new int[] { 1, 3, 5, 7, 9 }; // OK
//array3 = {1, 3, 5, 7, 9}; // Error
```

## 값 형식 및 참조 형식 배열

다음 배열 선언을 살펴보세요.

```
SomeType[] array4 = new SomeType[10];
```

이 문의 결과는 `SomeType`이 값 형식인지 또는 참조 형식인지에 따라 달라집니다. 값 형식인 경우 이 문은 각각

`SomeType` 형식인 10개 요소의 배열을 만듭니다. `SomeType`이 참조 형식인 경우 이 문은 각각 `null` 참조로 초기화된 10개 요소의 배열을 만듭니다. 두 인스턴스 모두에서 요소가 요소 형식에 대한 기본값으로 초기화됩니다. 값 형식과 참조 형식에 대한 자세한 내용은 [값 형식](#) 및 [참조 형식](#)을 참조하세요.

## 참조

- [Array](#)
- [배열](#)
- [다차원 배열](#)
- [가변 배열](#)

# 다차원 배열(C# 프로그래밍 가이드)

2020-11-02 • 3 minutes to read • [Edit Online](#)

배열에 둘 이상의 차원이 있을 수 있습니다. 예를 들어 다음 선언은 행 4개, 열 2개의 2차원 배열을 만듭니다.

```
int[,] array = new int[4, 2];
```

다음 선언은 세 가지 차원 4, 2, 3의 배열을 만듭니다.

```
int[, ,] array1 = new int[4, 2, 3];
```

## 배열 초기화

다음 예제와 같이 선언 시 배열을 초기화할 수 있습니다.

```

// Two-dimensional array.
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// The same array with dimensions specified.
int[,] array2Da = new int[4, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// A similar array with string elements.
string[,] array2Db = new string[3, 2] { { "one", "two" }, { "three", "four" },
                                         { "five", "six" } };

// Three-dimensional array.
int[, ,] array3D = new int[, ,] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                   { { 7, 8, 9 }, { 10, 11, 12 } } };
// The same array with dimensions specified.
int[, ,] array3Da = new int[2, 2, 3] { { { 1, 2, 3 }, { 4, 5, 6 } },
                                         { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
System.Console.WriteLine(array2D[0, 0]);
System.Console.WriteLine(array2D[0, 1]);
System.Console.WriteLine(array2D[1, 0]);
System.Console.WriteLine(array2D[1, 1]);
System.Console.WriteLine(array2D[3, 0]);
System.Console.WriteLine(array2Db[1, 0]);
System.Console.WriteLine(array3Da[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++)
{
    total *= array3D.GetLength(i);
}
System.Console.WriteLine("{0} equals {1}", allLength, total);

// Output:
// 1
// 2
// 3
// 4
// 7
// three
// 8
// 12
// 12 equals 12

```

순위를 지정하지 않고 배열을 초기화할 수도 있습니다.

```
int[,] array4 = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

초기화하지 않고 배열 변수를 선언하도록 선택할 경우 `new` 연산자를 사용하여 변수에 배열을 할당해야 합니다. `new` 사용은 다음 예제에서 확인할 수 있습니다.

```
int[,] array5;
array5 = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } }; // OK
//array5 = {{1,2}, {3,4}, {5,6}, {7,8}}; // Error
```

다음 예제에서는 특정 배열 요소에 값을 할당합니다.

```
array5[2, 1] = 25;
```

마찬가지로, 다음 예제에서는 특정 배열 요소의 값을 가져와 `elementValue` 변수에 할당합니다.

```
int elementValue = array5[2, 1];
```

다음 코드 예제에서는 가변 배열을 제외하고 배열 요소를 기본 값으로 초기화합니다.

```
int[,] array6 = new int[10, 10];
```

## 참조

- [C# 프로그래밍 가이드](#)
- [배열](#)
- [1차원 배열](#)
- [가변 배열](#)

# 가변 배열(C# 프로그래밍 가이드)

2021-02-18 • 6 minutes to read • [Edit Online](#)

가변 배열은 요소, 아마도 요소의 크기가 서로 다른 배열입니다. 가변 배열을 "배열의 배열"이라고도 합니다. 다음 예제에서는 가변 배열을 선언, 초기화 및 액세스하는 방법을 보여 줍니다.

다음은 각각 정수의 1차원 배열인 세 개의 요소를 포함하는 1차원 배열의 선언입니다.

```
int[][] jaggedArray = new int[3][];
```

`jaggedArray` 를 사용하려면 먼저 해당 요소를 초기화해야 합니다. 다음과 같이 요소를 초기화할 수 있습니다.

```
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

각 요소는 정수의 1차원 배열입니다. 첫 번째 요소는 5개 정수의 배열이고, 두 번째 요소는 4개 정수의 배열이고, 세 번째 요소는 2개 정수의 배열입니다.

이니셜라이저를 사용하여 배열 요소에 값을 채울 수도 있으며, 이 경우 배열 크기가 필요 없습니다. 예를 들어:

```
jaggedArray[0] = new int[] { 1, 3, 5, 7, 9 };
jaggedArray[1] = new int[] { 0, 2, 4, 6 };
jaggedArray[2] = new int[] { 11, 22 };
```

다음과 같이 선언 시 배열을 초기화할 수도 있습니다.

```
int[][] jaggedArray2 = new int[][]
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};
```

다음과 같은 약식 형태를 사용할 수 있습니다. 요소에 대한 기본 초기화가 없으므로 요소 초기화에서 `new` 연산자를 생략할 수 없습니다.

```
int[][] jaggedArray3 =
{
    new int[] { 1, 3, 5, 7, 9 },
    new int[] { 0, 2, 4, 6 },
    new int[] { 11, 22 }
};
```

가변 배열은 여러 배열로 구성되어 있기 때문에 해당 요소가 참조 형식이며, `null`로 초기화됩니다.

다음 예제처럼 개별 배열 요소에 액세스할 수 있습니다.

```
// Assign 77 to the second element ([1]) of the first array ([0]):  
jaggedArray3[0][1] = 77;  
  
// Assign 88 to the second element ([1]) of the third array ([2]):  
jaggedArray3[2][1] = 88;
```

가변 배열과 다차원 배열을 함께 사용할 수 있습니다. 다음은 서로 다른 크기의 세 가지 2차원 배열 요소를 포함하는 1차원 가변 배열의 선언과 초기화입니다. 자세한 내용은 [다차원 배열](#)을 참조하세요.

```
int[,] jaggedArray4 = new int[3][,]  
{  
    new int[,] { {1,3}, {5,7} },  
    new int[,] { {0,2}, {4,6}, {8,10} },  
    new int[,] { {11,22}, {99,88}, {0,9} }  
};
```

이 예제와 같이, 첫 번째 배열의 `[1,0]` 요소 값(`5`)을 표시하는 개별 요소에 액세스할 수 있습니다.

```
System.Console.WriteLine("{0}", jaggedArray4[0][1, 0]);
```

`Length` 메서드는 가변 배열에 포함된 배열 수를 반환합니다. 예를 들어 이전 배열을 선언했다고 가정합니다.

```
System.Console.WriteLine(jaggedArray4.Length);
```

이 경우 위 줄은 값 3을 반환합니다.

## 예제

이 예제에서는 요소 자체가 배열인 배열을 작성합니다. 각 배열 요소의 크기가 서로 다릅니다.

```

class ArrayTest
{
    static void Main()
    {
        // Declare the array of two elements.
        int[][] arr = new int[2][];

        // Initialize the elements.
        arr[0] = new int[5] { 1, 3, 5, 7, 9 };
        arr[1] = new int[4] { 2, 4, 6, 8 };

        // Display the array elements.
        for (int i = 0; i < arr.Length; i++)
        {
            System.Console.Write("Element({0}): ", i);

            for (int j = 0; j < arr[i].Length; j++)
            {
                System.Console.Write("{0}{1}", arr[i][j], j == (arr[i].Length - 1) ? "" : " ");
            }
            System.Console.WriteLine();
        }
        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
   Element(0): 1 3 5 7 9
   Element(1): 2 4 6 8
*/

```

## 참고 항목

- [Array](#)
- [C# 프로그래밍 가이드](#)
- [배열](#)
- [1차원 배열](#)
- [다차원 배열](#)

# 배열에 foreach 사용(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

`foreach` 문은 배열의 요소를 반복하는 단순하고 깔끔한 방법을 제공합니다.

1차원 배열의 경우 `foreach` 문은 인덱스 0으로 시작하고 인덱스 `Length - 1`로 끝나는 늘어나는 인덱스 순서로 요소를 처리합니다.

```
int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };
foreach (int i in numbers)
{
    System.Console.WriteLine("{0} ", i);
}
// Output: 4 5 6 1 2 3 -2 -1 0
```

다차원 배열의 경우 요소는 가장 오른쪽 차원의 인덱스가 먼저 증가한 이후, 다음 왼쪽 차원의 인덱스, 그다음 왼쪽 차원의 인덱스가 증가하는 방식으로 트래버스됩니다.

```
int[,] numbers2D = new int[3, 2] { { 9, 99 }, { 3, 33 }, { 5, 55 } };
// Or use the short form:
// int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.WriteLine("{0} ", i);
}
// Output: 9 99 3 33 5 55
```

그러나 다차원 배열에서 중첩 `for` 루프를 사용하면 배열 요소를 처리하는 순서를 더 강력하게 제어할 수 있습니다.

## 참고 항목

- [Array](#)
- [C# 프로그래밍 가이드](#)
- [배열](#)
- [1차원 배열](#)
- [다차원 배열](#)
- [가변 배열](#)

# 배열을 인수로 전달(C# 프로그래밍 가이드)

2020-11-02 • 5 minutes to read • [Edit Online](#)

배열을 메서드 매개 변수에 인수로 전달할 수 있습니다. 배열은 참조 형식이므로 메서드를 통해 요소 값을 변경할 수 있습니다.

## 1차원 배열을 인수로 전달

초기화된 1차원 배열을 메서드에 전달할 수 있습니다. 예를 들어 다음 문은 인쇄 메서드에 배열을 보냅니다.

```
int[] theArray = { 1, 3, 5, 7, 9 };
PrintArray(theArray);
```

다음 코드는 인쇄 메서드의 부분 구현을 보여 줍니다.

```
void PrintArray(int[] arr)
{
    // Method code.
}
```

다음 예제와 같이 한 단계로 새 배열을 초기화하고 전달할 수 있습니다.

```
PrintArray(new int[] { 1, 3, 5, 7, 9 });
```

### 예제

다음 예제에서는 문자열 배열이 초기화되고 문자열에 대한 `DisplayArray` 메서드에 인수로 전달됩니다. 메서드가 배열 요소를 표시합니다. 다음으로 `ChangeArray` 메서드는 배열 요소를 반대로 전환한 다음, `ChangeArrayElements` 메서드는 배열의 처음 세 요소를 수정합니다. 각 메서드가 반환된 후 `DisplayArray` 메서드는 배열을 값으로 전달해도 배열 요소가 변경되지 않는다는 것을 보여줍니다.

```

using System;

class ArrayExample
{
    static void DisplayArray(string[] arr) => Console.WriteLine(string.Join(" ", arr));

    // Change the array by reversing its elements.
    static void ChangeArray(string[] arr) => Array.Reverse(arr);

    static void ChangeArrayElements(string[] arr)
    {
        // Change the value of the first three array elements.
        arr[0] = "Mon";
        arr[1] = "Wed";
        arr[2] = "Fri";
    }

    static void Main()
    {
        // Declare and initialize an array.
        string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
        // Display the array elements.
        DisplayArray(weekDays);
        Console.WriteLine();

        // Reverse the array.
        ChangeArray(weekDays);
        // Display the array again to verify that it stays reversed.
        Console.WriteLine("Array weekDays after the call to ChangeArray:");
        DisplayArray(weekDays);
        Console.WriteLine();

        // Assign new values to individual array elements.
        ChangeArrayElements(weekDays);
        // Display the array again to verify that it has changed.
        Console.WriteLine("Array weekDays after the call to ChangeArrayElements:");
        DisplayArray(weekDays);
    }
}

// The example displays the following output:
//      Sun Mon Tue Wed Thu Fri Sat
//
//      Array weekDays after the call to ChangeArray:
//      Sat Fri Thu Wed Tue Mon Sun
//
//      Array weekDays after the call to ChangeArrayElements:
//      Mon Wed Fri Wed Tue Mon Sun

```

## 다차원 배열을 인수로 전달

초기화된 다차원 배열을 1차원 배열 전달과 동일한 방식으로 메서드에 전달합니다.

```

int[,] theArray = { { 1, 2 }, { 2, 3 }, { 3, 4 } };
Print2DArray(theArray);

```

다음 코드는 2차원 배열을 해당 인수로 허용하는 인쇄 메서드의 부분 선언을 보여 줍니다.

```

void Print2DArray(int[,] arr)
{
    // Method code.
}

```

다음 예제와 같이 한 단계로 새 배열을 초기화하고 전달할 수 있습니다.

```
Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
```

### 예제

다음 예제에서는 정수의 2차원 배열이 초기화되고 `Print2DArray` 메서드에 전달됩니다. 메서드가 배열 요소를 표시합니다.

```
class ArrayClass2D
{
    static void Print2DArray(int[,] arr)
    {
        // Display the array elements.
        for (int i = 0; i < arr.GetLength(0); i++)
        {
            for (int j = 0; j < arr.GetLength(1); j++)
            {
                System.Console.WriteLine("Element({0},{1})={2}", i, j, arr[i, j]);
            }
        }
    }
    static void Main()
    {
        // Pass the array as an argument.
        Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
   Element(0,0)=1
   Element(0,1)=2
   Element(1,0)=3
   Element(1,1)=4
   Element(2,0)=5
   Element(2,1)=6
   Element(3,0)=7
   Element(3,1)=8
*/
```

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [배열](#)
- [1차원 배열](#)
- [다차원 배열](#)
- [가변 배열](#)

# 암시적으로 형식화된 배열(C# 프로그래밍 가이드)

2020-11-02 • 3 minutes to read • [Edit Online](#)

배열 인스턴스의 형식이 배열 이니셜라이저에 지정된 요소에서 유추되는 암시적으로 형식화된 배열을 만들 수 있습니다. 암시적으로 형식화된 변수에 대한 규칙은 암시적으로 형식화된 배열에도 적용됩니다. 자세한 내용은 [암시적으로 형식화된 지역 변수](#)를 참조하세요.

암시적으로 형식화된 배열은 일반적으로 무명 형식, 개체 및 컬렉션 이니셜라이저와 함께 쿼리 식에 사용됩니다.

다음 예제에서는 암시적으로 형식화된 배열을 만드는 방법을 보여 줍니다.

```
class ImplicitlyTypedArraySample
{
    static void Main()
    {
        var a = new[] { 1, 10, 100, 1000 }; // int[]
        var b = new[] { "hello", null, "world" }; // string[]

        // single-dimension jagged array
        var c = new[]
        {
            new[]{1,2,3,4},
            new[]{5,6,7,8}
        };

        // jagged array of strings
        var d = new[]
        {
            new[]{"Luca", "Mads", "Luke", "Dinesh"},
            new[]{"Karen", "Suma", "Frances"}
        };
    }
}
```

앞의 예제에서 암시적으로 형식화된 배열의 경우 초기화 문의 왼쪽에 대괄호가 사용되지 않는 것을 확인합니다. 또한 가변 배열이 1차원 배열과 마찬가지로 `new []`를 사용하여 초기화됩니다.

## 개체 이니셜라이저의 암시적으로 형식화된 배열

배열이 포함된 무명 형식을 만드는 경우 해당 형식의 개체 이ни셜라이저에서 배열을 암시적으로 형식화해야 합니다. 다음 예제에서 `contacts`는 각각 `PhoneNumbers`라는 배열이 포함된 무명 형식의 암시적으로 형식화된 배열입니다. `var` 키워드는 개체 이니셜라이저 내부에서 사용되지 않았습니다.

```
var contacts = new[]
{
    new {
        Name = " Eugene Zabokritski",
        PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }
    },
    new {
        Name = " Hanying Feng",
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

## 참고 항목

- C# 프로그래밍 가이드
- 암시적 형식 지역 변수
- 배열
- 익명 형식
- 개체 이니셜라이저 및 컬렉션 이니셜라이저
- var
- C#의 LINQ

# 문자열(C# 프로그래밍 가이드)

2020-11-02 • 30 minutes to read • [Edit Online](#)

문자열은 값이 텍스트인 `String` 형식의 개체입니다. 내부적으로 텍스트는 `Char` 개체의 순차적 읽기 전용 컬렉션으로 저장됩니다. C# 문자열의 끝에 null 종료 문자가 없으므로 C# 문자열에는 포함된 null 문자('\'0')를 여러 개 사용할 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`은 동일하며 원하는 명명 규칙을 사용할 수 있습니다. `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.

```
// 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);
```

문자 배열이 포함된 문자열을 초기화할 경우를 제외하고는 문자열 개체를 만들기 위해 `new` 연산자를 사용하지 않습니다.

문자열 길이가 0인 새 `String` 개체를 만들려면 `Empty` 상수 값이 포함된 문자열을 초기화하세요. 빈 문자열을 문

자열 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.

## 문자열 개체의 불변성

문자열 개체는 **변경할 수 없습니다**. 즉, 생성된 후에는 바꿀 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.

```
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 주의해야 합니다. 문자열에 대한 참조를 만든 후 원래 문자열을 "수정"하더라도 참조는 문자열을 수정할 때 만든 새 개체가 아니라 원래 개체를 계속 가리킵니다. 이 동작은 다음 코드에서 볼 수 있습니다.

```
string s1 = "Hello ";
string s2 = s1;
s1 += "World";

System.Console.WriteLine(s2);
//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.

## 일반 및 축자 문자열 리터럴

다음 예제와 같이 C#에서 제공하는 이스케이프 문자를 포함해야 하는 경우 일반 문자열 리터럴을 사용합니다.

```
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 Aeolian Harp", by Samuel Taylor Coleridge
```

문자열 텍스트에 백슬래시 문자가 포함된 경우 가독성을 높이고 편의를 위해 축자 문자열을 사용합니다(예: 파일 경로). 축자 문자열에서는 문자열 텍스트의 일부로 줄 바꿈 문자가 유지되므로 여러 줄 문자열을 초기화하는데 사용할 수 있습니다. 축자 문자열 내에 따옴표를 포함하려면 큰따옴표를 사용하세요. 다음 예제에서는 몇 가

지 일반적인 축자 문자열에 대한 사용을 보여 줍니다.

```
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."
```

## 문자열 이스케이프 시퀀스

이스케이프 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
\u	유니코드 이스케이프 시퀀스(UTF-16)	\uHHHH (범위: 0000-FFFF; 예제: \u00E7 "ç" =)
\U	유니코드 이스케이프 시퀀스(UTF-32)	\U00HHHHHH (범위: 000000-10FFFF; 예: \U0001F47D = "⌚")
\x	길이가 변하는 경우를 제외하고 "\u"와 유사한 유니코드 이스케이프 시퀀스합 니다.	\xH[H][H][H] (범위: 0-FFFF; 예: \x00E7 나 \x0E7 또는 \xE7 "ç" =)

## WARNING

\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 \xA1 은 코드 포인트 U+00A1 인 ";"을 생성합니다. 그러나 다음 문자가 "A" 또는 "a"이면 이스케이프 시퀀스는 \xA1 로 해석되어 코드 포인트 U+0A1A인 █ 를 생성합니다. 이러한 경우 4자리 16진수(예: \x00A1 )를 모두 지정하면 잘못된 해석을 방지할 수 있습니다.

## NOTE

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" 는 조사식 창에 "c\\files.txt"로 나타납니다.

## 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 처음 접하는 경우 빠른 개요를 위해 [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.

```
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.
```

### 복합 형식 지정

`String.Format`은 중괄호 안에 자리 표시자를 활용하여 형식 문자열을 만듭니다. 이 예제는 위에서 사용한 문자열 보간 방법과 유사한 출력을 생성합니다.

```
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 [.NET 형식의 서식 지정](#)을 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. [Substring](#) 메서드와 마찬가지로, [Replace](#)는 실제로 새 문자열을 반환하며, 원래 문자열은 수정되지 않습니다. 자세한 내용은 [문자열 검색 방법](#) 및 [문자열 내용 수정 방법](#)을 참조하세요.

```
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.

```
string s5 = "Printing backwards";

for (int i = 0; i < s5.Length; i++)
{
    System.Console.Write(s5[s5.Length - i - 1]);
}
// Output: "sdrawkcab gnitnirP"
```

문자열에서 개별 문자를 수정해야 하는 기능이 [String](#) 메서드에서 제공되지 않는 경우에는 [StringBuilder](#) 개체를 사용하여 개별 문자를 "현재 위치"에서 수정한 후 새 문자열을 만들어 [StringBuilder](#) 메서드로 결과를 저장할 수 있습니다. 다음 예제에서는 특정 방식으로 원래 문자열을 수정한 다음 나중에 사용할 수 있도록 결과를 저장해야 한다고 가정합니다.

```
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. 빈 문자열은 유효한 [System.String](#) 개체이므로 빈 문자열에 대해 메서드를 호출할 수 있습니다. 빈 문자열은 다음과 같이 초기화됩니다.

```
string s = String.Empty;
```

반면, null 문자열은 [System.String](#) 개체의 인스턴스를 참조하지 않으므로 null 문자열에서 메서드를 호출하려고 하면 [NullReferenceException](#)이 발생합니다. 그러나 다른 문자열과 연결 및 비교 작업에서는 null 문자열을 사용할 수 있습니다. 다음 예제에는 null 문자열에 대한 참조로 예외가 발생하거나 발생하지 않는 몇 가지 경우가 나와 있습니다.

```
static void Main()
{
    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 생성을 위한 [StringBuilder](#) 사용

.NET에서 문자열 작업은 고도로 최적화되어 있으므로 대부분의 경우 성능에 크게 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](#) 문자열을 사용하면 개별 문자를 다시 할당할 수도 있지만 기본 제공 문자열 데이터 형식을 지원하지는 않습니다. 예를 들어 이 코드는 새 문자열을 만들지 않고 문자열의 콘텐츠를 변경합니다.

```
System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal pet");
sb[0] = 'C';
System.Console.WriteLine(sb.ToString());
System.Console.ReadLine();

//Outputs Cat: the ideal pet
```

이 예제에서 [StringBuilder](#) 개체는 숫자 형식 집합에서 문자열을 만드는 데 사용됩니다.

```

using System;
using System.Text;

namespace CSRefStrings
{
    class TestStringBuilder
    {
        static void Main()
        {
            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
            Console.WriteLine();
        }
    }
}

```

## 문자열, 확장 메서드 및 LINQ

`String` 형식이 `IEnumerable<T>`을 구현하므로 문자열에서 `Enumerable` 클래스에 정의된 확장 메서드를 사용할 수 있습니다. 시각적인 혼란을 방지하기 위해 `String` 형식의 경우 이러한 메서드가 IntelliSense에서 제외되지만, 제외되더라도 사용할 수는 있습니다. 문자열에 LINQ 쿼리 식을 사용할 수도 있습니다. 자세한 내용은 [LINQ 및 문자열](#)을 참조하세요.

## 관련 항목

항목	설명
<a href="#">문자열 내용 수정 방법</a>	문자열을 변환하고 문자열의 내용을 수정하는 기술을 보여 줍니다.
<a href="#">문자열 비교 방법</a>	문자열의 서수 및 문화권 비교를 수행하는 방법을 보여 줍니다.
<a href="#">여러 문자열 연결 방법</a>	여러 문자열을 하나로 조인하는 다양한 방법을 보여줍니다.
<a href="#">String.Split을 사용하여 문자열 구문 분석 방법</a>	<code>String.Split</code> 메서드를 사용하여 문자열을 구문 분석하는 방법을 보여주는 코드 예제가 포함되어 있습니다.
<a href="#">문자열 검색 방법</a>	문자열에서 특정 텍스트 또는 패턴에 대해 검색을 사용하는 방법을 설명합니다.
<a href="#">문자열이 숫자 값을 나타내는지 확인 방법</a>	문자열에 올바른 숫자 값이 있는지 여부를 확인할 수 있도록 문자열을 안전하게 구문 분석하는 방법을 보여 줍니다.
<a href="#">문자열 보간</a>	문자열의 서식을 지정하는 편리한 구문을 제공하는 문자열 보간 기능에 대해 설명합니다.

항목	설명
<a href="#">기본적인 문자열 작업</a>	<a href="#">System.String</a> 및 <a href="#">System.Text.StringBuilder</a> 메서드를 사용하여 기본적인 문자열 작업을 수행하는 항목에 대한 링크를 제공합니다.
<a href="#">.NET에서 문자열 구문 분석</a>	.NET 기본 형식의 문자열 표현을 해당 형식의 인스턴스로 변환하는 방법에 대해 설명합니다.
<a href="#">.NET에서 날짜 및 시간 문자열 구문 분석</a>	"01/24/2008"과 같은 문자열을 <a href="#">System.DateTime</a> 개체로 변환하는 방법을 보여 줍니다.
<a href="#">문자열 비교</a>	문자열을 비교하는 방법에 대한 정보가 포함되어 있으며, C# 및 Visual Basic의 예제를 제공합니다.
<a href="#">StringBuilder 클래스 사용</a>	<a href="#">StringBuilder</a> 클래스를 사용하여 동적 문자열 개체를 만들고 수정하는 방법을 설명합니다.
<a href="#">LINQ 및 문자열</a>	LINQ 쿼리를 사용하여 다양한 문자열 작업을 수행하는 방법에 대한 정보를 제공합니다.
<a href="#">C# 프로그래밍 가이드</a>	C#에서 프로그래밍 구문을 설명하는 항목에 대한 링크를 제공합니다.

# 문자열이 숫자 값을 나타내는지 확인하는 방법(C# 프로그래밍 가이드)

2020-11-02 • 4 minutes to read • [Edit Online](#)

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.

```
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.

## NOTE

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.

## 예제

다음 예제에서는 `long`, `byte` 및 `decimal` 값의 문자열 표현과 함께 `TryParse`를 사용하는 방법을 보여 줍니다.

```
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.

## 참조

- [바이트 배열을 int로 변환하는 방법](#)
- [문자열을 숫자로 변환하는 방법](#)
- [16진수 문자열과 숫자 형식 간에 변환하는 방법](#)
- [숫자 문자열 구문 분석](#)
- [형식 서식 지정](#)

# 인덱서(C# 프로그래밍 가이드)

2020-11-02 • 6 minutes to read • [Edit Online](#)

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 속성과 유사합니다.

다음 예제에서는 간단한 `get` 및 `set` 접근자 메서드를 사용해서 값을 할당하거나 검색하는 제네릭 클래스를 정의 합니다. `Program` 클래스는 이 클래스의 인스턴스를 만들어 문자열을 저장합니다.

```
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.
```

## NOTE

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.

```

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:

```

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 객체를 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 섹션

- [인덱서 사용](#)
- [인터페이스의 인덱서](#)
- [속성 및 인덱서 비교](#)
- [접근자 접근성 제한](#)

## C# 언어 사양

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

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [속성](#)

# 인덱서 사용(C# 프로그래밍 가이드)

2020-11-02 • 13 minutes to read • [Edit Online](#)

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.

```
// Indexer declaration
public int this[int index]
{
    // get and set accessors
}
```

## IMPORTANT

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.

인덱서 값은 변수로 분류되지 않으므로 인덱서 값을 `ref` 또는 `out` 매개 변수로 전달할 수 없습니다.

다른 언어에서 사용할 수 있는 이름을 인덱서에 제공하려면 다음 예제에 표시된 대로 `System.Runtime.CompilerServices.IndexerNameAttribute`를 사용합니다.

```
// 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, `tempRecord.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.

```
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.

```

using System;

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 적절한 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.

```

using System;

// 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

```

using System;

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.

```

using System;
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

```

using System;

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 확인할 수 있게 해주는 Length 속성을 제공합니다. 인덱서 자체 안에 오류 처리 코드를 넣을 수도 있습니다. 사용자를 위해 인덱서 접근자 내에서 throw하는 모든 예외를 문서화해

야 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)

2020-11-02 • 4 minutes to read • [Edit Online](#)

[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.

```
public interface ISomeInterface
{
    //...

    // Indexer declaration:
    string this[int index]
    {
        get;
        set;
    }
}
```

인덱서의 시그니처는 동일한 인터페이스에 선언된 다른 모든 인덱서의 시그니처와 달라야 합니다.

## 예제

다음 예제에서는 인터페이스 인덱서를 구현하는 방법을 보여 줍니다.

```
// 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.
        get => arr[index];
        set => arr[index] = value;
    }
}
```

```

IndexerClass test = new IndexerClass();
System.Random rand = new System.Random();
// 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. 예

```

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 `IEmployee`를 구현하고 두 인터페이스의 인덱서 시그니처가 같으면 명시적 인터페이스 멤버 구현이 필요합니다. 즉, 다음과 같은 인덱서 선언이 있다고 가정합니다.

```

string IEmployee.this[int index]
{
}

```

이 선언은 `IEmployee` 인터페이스의 인덱서를 구현합니다. 또한 다음과 같은 선언이 있다고 가정합니다.

```

string ICitizen.this[int index]
{
}

```

이 선언은 `ICitizen` 인터페이스의 인덱서를 구현합니다.

## 참조

- [C# 프로그래밍 가이드](#)
- [인덱서](#)
- [속성](#)
- [인터페이스](#)

# 속성 및 인덱서 비교(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

인덱서는 속성과 비슷합니다. 다음 표에 나와 있는 차이점을 제외하면 속성 접근자에 대해 정의된 모든 규칙이 인덱서 접근자에도 적용됩니다.

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

- [C# 프로그래밍 가이드](#)
- [인덱서](#)
- [속성](#)

# 이벤트(C# 프로그래밍 가이드)

2020-11-02 • 5 minutes to read • [Edit Online](#)

[클래스](#) 나 개체에서는 특정 상황이 발생할 때 이벤트를 통해 다른 클래스나 개체에 이를 알려줄 수 있습니다. 이벤트를 보내거나 [발생시키는](#) 클래스를 [게시자](#)라고 하며 이벤트를 받거나 [처리하는](#) 클래스를 [구독자](#)라고 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 수행할 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 사양](#)의 [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

대리자 및 이벤트에 학습 C# 3.0. 기본 사항 마스터 C# 3.0

## 참조

- [EventHandler](#)
- [C# 프로그래밍 가이드](#)
- [대리자](#)
- [Windows Forms에서 이벤트 처리기 만들기](#)

# 이벤트 구독 및 구독 취소 방법(C# 프로그래밍 가이드)

2020-11-02 • 9 minutes to read • [Edit Online](#)

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 IDE**를 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.

```
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.

```
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.

```
void HandleCustomEvent(object sender, CustomEventArgs a)
{
    // Do something useful here.
}
```

- 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.

```
publisher.RaiseCustomEvent += HandleCustomEvent;
```

앞의 구문은 C# 2.0의 새로운 기능입니다. `new` 키워드를 사용하여 명시적으로 캡슐화 대리자를 만들어야 한다는 점에서 C# 1.0 구문과 정확히 동일합니다.

```
publisher.RaiseCustomEvent += new CustomEventHandler(HandleCustomEvent);
```

람다 식을 사용하여 이벤트 처리기를 지정할 수도 있습니다.

```
public Form1()
{
    InitializeComponent();
    this.Click += (s,e) =>
    {
        MessageBox.Show(((MouseEventArgs)e).Location.ToString());
    };
}
```

무명 메서드를 사용하여 이벤트를 구독하려면

- 나중에 이벤트 구독을 취소할 필요가 없는 경우 더하기 대입 연산자(`+=`)를 사용하여 이벤트에 무명 메서드를 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.

```
publisher.RaiseCustomEvent += delegate(object o, CustomEventArgs e)
{
    string s = o.ToString() + " " + e.ToString();
    Console.WriteLine(s);
};
```

익명 함수를 사용하여 이벤트를 구독한 경우 쉽게 이벤트 구독을 취소할 수 없습니다. 이 시나리오에서 구독을 취소하려면 이벤트를 구독하고, 대리자 변수에 무명 메서드를 저장한 다음 이벤트에 대리자를 추가하는 코드로 돌아가야 합니다. 일반적으로 코드의 뒷부분에서 이벤트 구독을 취소해야 하는 경우 익명 함수를 사용하여 이벤트를 구독하지 않는 것이 좋습니다. 익명 함수에 대한 자세한 내용은 [익명 함수를 참조하세요](#).

## 구독 해제

이벤트가 발생할 때 이벤트 처리기가 호출되지 않도록 하려면 이벤트 구독을 취소합니다. 리소스 누수를 방지 하려면 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 있지만 하면 가비지 수집 시 구독자 개체가 삭제되지 않습니다.

이벤트 구독을 취소하려면

- 빼기 대입 연산자(`-=`)를 사용하여 이벤트 구독을 취소합니다.

```
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)

2020-11-02 • 7 minutes to read • [Edit Online](#)

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.

```
public delegate void EventHandler(object sender, EventArgs e);
```

## NOTE

.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 둘 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.

```
public class CustomEventArgs : EventArgs
{
    public CustomEventArgs(string message)
    {
        Message = message;
    }

    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.

```
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.

```
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.

```
public event CustomEventHandler RaiseCustomEvent;
```

- c. 제네릭 버전을 사용하는 경우에는 사용자 지정 대리자가 필요하지 않습니다. 대신, 게시 클래스에 서 이벤트 유형을 `EventHandler<CustomEventArgs>`로 지정하고 고유한 클래스 이름을 꺽쇠 괄호로 둘어 대체합니다.

```
public event EventHandler<CustomEventArgs> RaiseCustomEvent;
```

## 예제

다음 예제에서는 사용자 지정 `EventArgs` 클래스와 `EventHandler<TEventArgs>`를 이벤트 유형으로 사용하여 이전 단계를 보여 줍니다.

```
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)

2020-11-02 • 5 minutes to read • [Edit Online](#)

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.

## NOTE

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.

## 예제

```
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
*/

```

- C# 프로그래밍 가이드
- 이벤트
- 대리자
- 액세스 한정자
- Windows Forms에서 이벤트 처리기 만들기

# 인터페이스 이벤트를 구현하는 방법(C# 프로그래밍 가이드)

2020-11-02 • 5 minutes to read • [Edit Online](#)

인터페이스는 이벤트를 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.

```
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 받으며 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 세이프 참조를 캐스팅하여 수신할 `OnDraw` 이벤트를 결정합니다.

```
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)

2020-11-02 • 2 minutes to read • [Edit Online](#)

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.

## 예제

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.

```
event EventHandler IDrawingObject.OnDraw
{
    add
    {
        lock (objectLock)
        {
            PreDrawEvent += value;
        }
    }
    remove
    {
        lock (objectLock)
        {
            PreDrawEvent -= value;
        }
    }
}
```

## 참조

- [이벤트](#)
- [event](#)

# 제네릭(C# 프로그래밍 가이드)

2020-11-02 • 9 minutes to read • [Edit Online](#)

제네릭에서 .NET에 도입한 형식 매개 변수 개념은 클라이언트 코드에서 클래스 또는 메서드를 선언하고 인스턴스화할 때까지 하나 이상의 형식 지정을 지원하는 클래스 및 메서드를 디자인할 수 있도록 합니다. 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.

```
// 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 클래스 및 메서드는 제네릭이 아닌 클래스 및 메서드에서는 결합할 수 없는 방식으로 재사용성, 형식 안전성 및 효율성을 결합합니다. 제네릭은 컬렉션 및 해당 컬렉션에서 작동하는 메서드에서 가장 자주 사용됩니다. [System.Collections.Generic](#) 네임스페이스에는 몇 가지 제네릭 기반 컬렉션 클래스가 있습니다. [ArrayList](#)와 같은 제네릭이 아닌 컬렉션은 권장되지 않으며 호환성을 위해 유지 관리됩니다. 자세한 내용은 [.NET의 제네릭](#)을 참조하세요.

물론, 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 `List<T>` 클래스를 사용해야 합니다.) 형식 매개 변수 `T`는 일반적으로 구체적인 형식을 사용하여 목록에 저장된 항목의 형식을 나타내는 여러 위치에서 사용되며, 다음과 같은 방법으로 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>`가 `GenericList<int>`와 같이 구체적인 형식으로 인스턴스화되면 `T`가 나타날 때마다 `int`로 바뀝니다.

```

// 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 변경하기만 하면 다음 코드를 쉽게 수정하여 문자열이나 다른 모든 사용자 지정 형식 목록을 만들 수 있습니다.

```

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");
    }
}

```

## 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. 이러한 제네릭 컬렉션 클래스는 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.
- 제네릭 데이터 형식에 사용되는 형식에 대한 정보는 리플렉션을 사용하여 런타임 시 얻을 수 있습니다.

## 관련 단원

- [제네릭 형식 매개 변수](#)
- [형식 매개 변수에 대한 제약 조건](#)
- [제네릭 클래스](#)
- [제네릭 인터페이스](#)
- [제네릭 메서드](#)
- [제네릭 대리자](#)
- [C++ 템플릿과 C# 제네릭의 차이점](#)
- [제네릭 및 리플렉션](#)
- [런타임의 제네릭](#)

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)을 참조하세요.

## 참조

- [System.Collections.Generic](#)
- [C# 프로그래밍 가이드](#)
- [유형](#)
- [`<typeparam>`](#)
- [`<typeparamref>`](#)

- .NET의 제네릭

# 제네릭 형식 매개 변수(C# 프로그래밍 가이드)

2020-11-02 • 4 minutes to read • [Edit Online](#)

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 꺽쇠 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.

```
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.

## 형식 매개 변수 명명 지침

- 필수적 단일 문자 이름으로도 자체 설명이 가능하여 설명적인 이름을 굳이 사용할 필요가 없는 경우가 아니면 제네릭 형식 매개 변수 이름을 설명적인 이름으로 지정하세요.

```
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.

```
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.

```
public interface ISessionChannel<TSession>
{
    TSession Session { get; }
}
```

- 선택적 매개 변수 이름 안에 형식 매개 변수에 적용되는 제약 조건을 나타내 보세요. 예를 들어 `ISession` 으로 제한되는 매개 변수의 이름은 `TSession` 이 될 수 있습니다.

코드 분석 규칙 [CA1715](#)를 사용하여 형식 매개 변수의 이름이 적절하게 지정되었는지 확인할 수 있습니다.

## 참고 항목

- [System.Collections.Generic](#)
- [C# 프로그래밍 가이드](#)

- 제네릭
- C++ 템플릿과 C# 제네릭의 차이점

# 형식 매개 변수에 대한 제약 조건(C# 프로그래밍 가이드)

2021-02-18 • 27 minutes to read • [Edit Online](#)

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.

제약 조건	설명
<code>where T : struct</code>	형식 인수는 null을 허용하지 않는 값 형식이어야 합니다. Null 허용 값 형식에 대한 자세한 내용은 <a href="#">Null 허용 값 형식</a> 을 참조하세요. 모든 값 형식에 액세스할 수 있는 매개 변수가 없는 생성자가 있으므로, <code>struct</code> 제약 조건은 <code>new()</code> 제약 조건을 나타내고 <code>new()</code> 제약 조건과 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. C# 8.0 이상의 null 허용 컨텍스트에서 <code>T</code> 는 null을 허용하지 않는 참조 형식이어야 합니다.
<code>where T : class?</code>	형식 인수는 null을 허용하거나 null을 허용하지 않는 참조 형식이어야 합니다. 이 제약 조건은 모든 클래스, 인터페이스, 대리자 또는 배열 형식에도 적용됩니다.
<code>where T : notnull</code>	형식 인수는 nullable이 아닌 형식이어야 합니다. 인수는 C# 8.0 이상의 null을 허용하지 않는 참조 형식이거나 null을 허용하지 않는 값 형식일 수 있습니다.
<code>where T : unmanaged</code>	형식 인수는 nullable이 아닌 비관리형 형식이어야 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 public 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;base class name&gt;</code>	형식 인수가 지정된 기본 클래스이거나 지정된 기본 클래스에서 파생되어야 합니다. C# 8.0 이상의 null 허용 컨텍스트에서 <code>T</code> 는 지정된 기본 클래스에서 파생된 null을 허용하지 않는 참조 형식이어야 합니다.
<code>where T : &lt;base class name&gt;?</code>	형식 인수가 지정된 기본 클래스이거나 지정된 기본 클래스에서 파생되어야 합니다. C# 8.0 이상의 null 허용 컨텍스트에서 <code>T</code> 는 지정된 기본 클래스에서 파생된 null을 허용하거나 null을 허용하지 않는 형식일 수 있습니다.

제약 조건	설명
<code>where T : &lt;interface name&gt;</code>	형식 인수가 지정된 인터페이스이거나 지정된 인터페이스를 구현해야 합니다. 여러 인터페이스 제약 조건을 지정할 수 있습니다. 제약 인터페이스가 제네릭일 수도 있습니다. C# 8.0 이상의 null 허용 컨텍스트에서 <code>T</code> 는 지정된 인터페이스를 구현하는 null을 허용하지 않는 형식이어야 합니다.
<code>where T : &lt;interface name&gt;?</code>	형식 인수가 지정된 인터페이스이거나 지정된 인터페이스를 구현해야 합니다. 여러 인터페이스 제약 조건을 지정할 수 있습니다. 제약 인터페이스가 제네릭일 수도 있습니다. C# 8.0 이상의 null 허용 컨텍스트에서 <code>T</code> 는 null 허용 참조 형식, null을 허용하지 않는 참조 형식 또는 값 형식이어야 합니다. <code>T</code> 는 null 허용 값 형식이 아닐 수 있습니다.
<code>where T : U</code>	<code>T</code> 에 대해 제공되는 형식 인수는 <code>U</code> 에 대해 제공되는 인수 이거나 이 인수에서 파생되어야 합니다. null 허용 컨텍스트에서 <code>U</code> 가 null을 허용하지 않는 참조 형식인 경우 <code>T</code> 는 null을 허용하지 않는 참조 형식이어야 합니다. <code>U</code> 가 null 허용 참조 형식인 경우 <code>T</code> 는 null을 허용하거나 null을 허용하지 않는 참조 형식일 수 있습니다.

## 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 단순 할당 또는 [System.Object](#)에서 지원하지 않는 메서드 호출 이외의 작업을 제네릭 멤버에서 사용하는 경우 형식 매개 변수에 제약 조건을 적용해야 합니다. 예를 들어 기본 클래스 제약 조건은 이 형식의 개체나 이 형식에서 파생된 개체만 형식 인수로 사용된다고 컴파일러에 알립니다. 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 소개](#)에 있음)에 추가할 수 있는 기능을 보여 줍니다.

```

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.

```
class EmployeeList<T> where T : Employee, IEmployee, System.IComparable<T>, new()
{
    // ...
}
```

`where T : class` 제약 조건을 적용하는 경우 `==` 및 `!=` 연산자는 참조 ID만 테스트하고 값이 같은지 테스트하지 않으므로 형식 매개 변수에 사용하지 않도록 합니다. 이러한 연산자가 인수로 사용되는 형식에서 오버로드 되는 경우에도 이 동작이 발생합니다. 다음 코드는 이 내용을 보여 줍니다. `String` 클래스가 `==` 연산자를 오버로드하지만 출력이 `false`입니다.

```
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 일치 여부를 테스트해야 하는 경우에도 `where T : IEquatable<T>` 또는 `where T : IComparable<T>` 제약 조건을 적용하고 제네릭 클래스를 생성하는 데 사용할 모든 클래스에서 인터페이스를 구현하는 것이 좋습니다.

## 여러 매개 변수 제한

다음 예제와 같이 여러 매개 변수에 제약 조건을 적용하고, 단일 매개 변수에 여러 제약 조건을 적용할 수 있습니다.

```
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.

- `!=` 및 `==` 연산자는 구체적인 형식 인수가 이러한 연산자를 지원한다는 보장이 없기 때문에 사용할 수 없습니다.
- `System.Object` 로/에서 변환하거나 임의의 인터페이스 형식으로 명시적으로 변환할 수 있습니다.
- `null`과 비교할 수 있습니다. 바인딩되지 않은 매개 변수를 `null` 과 비교하는 경우 형식 인수가 값 형식이면 비교에서 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.

```
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 꺽쇠괄호 안에 선언해야 합니다.

```
//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

C# 8.0부터 null 허용 컨텍스트에서 `notnull` 제약 조건을 사용하여 형식 인수가 null을 허용하지 않는 값 형식 또는 null을 허용하지 않는 참조 형식이어야 하도록 지정할 수 있습니다. `notnull` 제약 조건은 `nullable enable` 컨텍스트에서만 사용할 수 있습니다. nullable 형식을 감지하지 않는 컨텍스트에서 `notnull` 제약 조건을 추가하는 경우 컴파일러가 경고를 생성합니다.

다른 제약 조건과 달리 형식 인수가 `notnull` 제약 조건을 위반하면 컴파일러는 해당 코드가 `nullable enable` 컨텍스트에서 컴파일될 때 경고를 생성합니다. 코드가 nullable 형식을 감지하지 않는 컨텍스트에서 컴파일된 경우에는 컴파일러가 경고나 오류를 생성하지 않습니다.

C# 8.0부터 null 허용 컨텍스트에서 `class` 제약 조건은 형식 인수가 null을 허용하지 않는 참조 형식이어야 하도록 지정합니다. null 허용 컨텍스트에서 형식 매개 변수가 null 허용 참조 형식이면 컴파일러가 경고를 생성합니다.

## 관리되지 않는 제약 조건

C# 7.3부터 `unmanaged` 제약 조건을 사용하여 형식 매개 변수가 nullable이 아닌 [비관리형 형식](#)이어야 함을 지정 할 수 있습니다. `unmanaged` 제약 조건을 사용하면 다음 예제와 같이 메모리 블록으로 조작할 수 있는 형식을 사용하도록 재사용 가능한 루틴을 작성할 수 있습니다.

```
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

C# 7.3부터 [System.Delegate](#) 또는 [System.MulticastDelegate](#)를 기본 클래스 제약 조건으로 사용할 수도 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.

```
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.

```
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

C# 7.3부터 [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 유효한 값을 모두 찾은 다음, 해당 값을 문자열 표현에 매핑하는 사전을 작성합니다.

```
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.

```
enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

```
var map = EnumNamedValues<Rainbow>();

foreach (var pair in map)
    Console.WriteLine($"{pair.Key}:\t{pair.Value}");
```

## 참조

- [System.Collections.Generic](#)
- [C# 프로그래밍 가이드](#)
- [제네릭 소개](#)
- [제네릭 클래스](#)
- [new 제약 조건](#)

# 제네릭 클래스(C# 프로그래밍 가이드)

2020-11-02 • 10 minutes to read • [Edit Online](#)

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 가능성성이 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 확인 할 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 만드는데 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>`, 경우 클라이언트 코드는 형식 인수를 지정하여 폐쇄형 생성 형식(`Node<int>`)을 만들어 클래스를 참조할 수 있습니다. 또는 제네릭 기본 클래스를 지정하는 경우와 같이 형식 매개 변수를 지정하지 않고 개방형 생성 형식(`Node<T>`)을 만들 수 있습니다. 제네릭 클래스는 구체적인 클래스, 폐쇄형 생성 클래스 또는 개방형 생성 기본 클래스에서 상속할 수 있습니다.

```

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.

```

//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.

```

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.

```

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.

```

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.

```
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](#)
- [C# 프로그래밍 가이드](#)
- [제네릭](#)
- [열거자의 상태 저장](#)
- [상속 퍼즐, 1부](#)

# 제네릭 인터페이스(C# 프로그래밍 가이드)

2020-11-02 • 8 minutes to read • [Edit Online](#)

제네릭 컬렉션 클래스에 대한 인터페이스 또는 컬렉션의 항목을 나타내는 제네릭 클래스에 대한 인터페이스를 정의하는 것이 대개 유용합니다. 값 형식에 대해 boxing 및 unboxing 연산을 하지 않으려면 제네릭 클래스에서 `IComparable` 대신 `IComparable<T>`과 같은 제네릭 인터페이스를 사용하는 것이 좋습니다..NET 클래스 라이브러리에는 `System.Collections.Generic` 네임스페이스의 컬렉션 클래스에 사용할 제네릭 인터페이스가 여러 개 정의되어 있습니다.

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. 이렇게 하면 `SortedList<T>`의 `BubbleSort` 메서드가 목록 요소에서 제네릭 `CompareTo` 메서드를 사용할 수 있게 됩니다. 이 예제에서 목록 요소는 `IComparable<Person>`을 구현하는 단순 클래스인 `Person`입니다.

```
//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 = new string[]
        {
            "Franscoise",
            "Bill",
            "Li",
            "Sandra",
            "Gunnar",
            "Alok",
            "Hiroyuki",
            "Maria",
            "Alessandro",
            "Raul"
        };

        int[] ages = new int[] { 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.

```

class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{
}

```

하나의 인터페이스는 다음과 같은 두 개 이상의 형식 매개 변수를 정의할 수 있습니다.

```

interface IDictionary<K, V>
{
}

```

클래스에 적용되는 상속 규칙은 인터페이스에도 적용됩니다.

```

interface IMonth<T> { }

interface IJanuary : IMonth<int> { } //No error
interface IFebuary<T> : IMonth<int> { } //No error
interface IMarch<T> : IMonth<T> { } //No error
//interface IApril<T> : IMonth<T, U> {} //Error

```

제네릭 인터페이스가 반공변성(Contravariance)인 경우, 제네릭 인터페이스는 제네릭이 아닌 인터페이스에서 상속할 수 있습니다. 즉, 형식 매개 변수만 반환 값으로 사용한다는 의미입니다. .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.

```

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.

```

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.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [제네릭 소개](#)
- [interface](#)
- [제네릭](#)

# 제네릭 메서드(C# 프로그래밍 가이드)

2020-11-02 • 5 minutes to read • [Edit Online](#)

제네릭 메서드는 다음과 같은 형식 매개 변수를 사용하여 선언된 메서드입니다.

```
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 보여 줍니다.

```
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.

```
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.

```
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.

```
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 인수와만 함께 사용할 수 있습니다.

```
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.

```
void DoWork() { }
void DoWork<T>() { }
void DoWork<T, U>() { }
```

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)을 참조하세요.

## 참조

- [System.Collections.Generic](#)
- [C# 프로그래밍 가이드](#)
- [제네릭 소개](#)
- [메서드](#)

# 제네릭 및 배열(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

C# 2.0 이상에서 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 throw 됩니다.

다음 코드 예제는 `IList<T>` 입력 매개 변수를 사용하는 단일 제네릭 메서드가 목록과 배열(여기에서는 정수 배열) 모두를 통해 반복하는 방법을 보여 줍니다.

```
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](#)
- [C# 프로그래밍 가이드](#)
- [제네릭](#)
- [배열](#)
- [제네릭](#)

# 제네릭 대리자(C# 프로그래밍 가이드)

2020-11-02 • 3 minutes to read • [Edit Online](#)

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.

```
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.

```
Del<int> m2 = Notify;
```

제네릭 클래스 내에 정의된 대리자는 클래스 메서드와 같은 방식으로 제네릭 클래스 형식 매개 변수를 사용할 수 있습니다.

```
class Stack<T>
{
    T[] items;
    int index;

    public delegate void StackDelegate(T[] items);
}
```

대리자를 참조하는 코드는 포함 클래스의 형식 인수를 다음과 같이 지정해야 합니다.

```
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.

```
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)
    {
        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](#)
- [C# 프로그래밍 가이드](#)
- [제네릭 소개](#)
- [제네릭 메서드](#)
- [제네릭 클래스](#)
- [제네릭 인터페이스](#)
- [대리자](#)
- [제네릭](#)

# C++ 템플릿과 C# 제네릭의 차이점(C# 프로그래밍 가이드)

2020-11-02 • 5 minutes to read • [Edit Online](#)

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 런타임에 C# 제네릭 형식 대체가 수행되어 인스턴스화된 개체에 대해 제네릭 형식 정보가 유지된다는 점입니다. 자세한 내용은 [런타임의 제네릭](#)을 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 <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)

2020-11-02 • 8 minutes to read • [Edit Online](#)

제네릭 형식 또는 메서드가 MSIL(Microsoft Intermediate Language)로 컴파일되면 자체적으로 형식 매개 변수를 갖는 것으로 식별하는 메타데이터가 포함됩니다. 제네릭 형식의 MSIL이 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 MSIL의 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.

```
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.

```
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 MSIL의 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 MSIL의 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.

```
class Customer { }
class Order { }
```

```
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.

```
Stack<Order> orders = new Stack<Order>();
```

값 형식과는 달리 `Order` 형식에 대한 또 다른 특수화된 버전의 `Stack<T>` 클래스는 만들어지지 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.

```
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 런타임에 리플렉션을 통해 쿼리할 수 있고 실제 형식과 형식 매개 변수를 모두 확인할 수 있습니다.

## 참고 항목

- [System.Collections.Generic](#)
- [C# 프로그래밍 가이드](#)
- [제네릭 소개](#)
- [제네릭](#)

# 제네릭 및 리플렉션(C# 프로그래밍 가이드)

2020-11-02 • 8 minutes to read • [Edit Online](#)

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.

.NET Framework 2.0에서는 제네릭 형식에 대한 런타임 정보를 사용할 수 있도록 [Type](#) 클래스에 여러 개의 새 멤버가 추가되었습니다. 이러한 메서드 및 속성을 사용하는 방법에 대한 자세한 내용은 이러한 클래스에 대한 설명서를 참조하세요. [System.Reflection.Emit](#) 네임스페이스에도 제네릭을 지원하는 새 멤버가 포함되어 있습니다. [방법: 리플렉션 내보내기를 사용하여 제네릭 형식 정의](#)를 참조하세요.

제네릭 리플렉션에 사용되는 용어의 고정 조건 목록은 [IsGenericType](#) 속성 설명을 참조하세요.

SYSTEM.ETYPE 멤버 이름	설명
<a href="#">IsGenericType</a>	형식이 제네릭이면 true를 반환합니다.
<a href="#">GetGenericArguments</a>	생성된 형식에 제공된 형식 인수 또는 제네릭 형식 정의의 형식 매개 변수를 나타내는 <a href="#">Type</a> 개체의 배열을 반환합니다.
<a href="#">GetGenericTypeDefinition</a>	현재 생성된 형식에 대한 기본 제네릭 형식 정의 반환합니다.
<a href="#">GetGenericParameterConstraints</a>	현재 제네릭 형식 매개 변수에 대한 제약 조건을 나타내는 <a href="#">Type</a> 개체의 배열을 반환합니다.
<a href="#">ContainsGenericParameters</a>	형식 또는 바깥쪽 형식이나 메서드에 제공되지 않은 특정 형식에 대한 형식 매개 변수가 포함된 경우 true를 반환합니다.
<a href="#">GenericParameterAttributes</a>	현재 제네릭 형식 매개 변수의 특수 제약 조건을 설명하는 <a href="#">GenericParameterAttributes</a> 플래그의 조합을 가져옵니다.
<a href="#">GenericParameterPosition</a>	<a href="#">Type</a> 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.
<a href="#">IsGenericParameter</a>	현재 <a href="#">Type</a> 이 제네릭 형식 또는 메서드 정의의 형식 매개 변수를 나타내는지를 나타내는 값을 가져옵니다.
<a href="#">IsGenericTypeDefinition</a>	현재 <a href="#">Type</a> 이 다른 제네릭 형식을 생성하는데 사용될 수 있는 제네릭 형식 정의를 나타내는지 여부를 가리키는 값을 가져옵니다. 형식이 제네릭 형식의 정의를 나타내는 경우 true를 반환합니다.
<a href="#">DeclaringMethod</a>	현재 제네릭 형식 매개 변수를 정의한 제네릭 메서드를 반환하거나 형식 매개 변수가 제네릭 메서드에 의해 정의되지 않은 경우 null을 반환합니다.

SYSTEM.ETYPE 멤버 이름	설명
MakeGenericType	형식 배열의 요소를 현재 제네릭 형식 정의의 형식 매개 변수로 대체하고 생성된 형식을 나타내는 Type 개체를 반환합니다.

또한 [MethodInfo](#) 클래스에 새 멤버는 제네릭 메서드에 대한 런타임 정보를 사용하도록 설정합니다. 제네릭 메서드를 반영하는데 사용되는 용어에 대한 고정 조건 목록은 [IsGenericMethod](#) 속성 설명을 참조하세요.

SYSTEM.REFLECTION.MEMBERINFO 멤버 이름	설명
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 <a href="#">MethodInfo</a> 가 제네릭 메서드의 정의를 나타내는 경우 true를 반환합니다.
MakeGenericMethod	현재 제네릭 메서드 정의의 형식 매개 변수를 형식 배열의 요소로 대체하고, 결과로 생성된 메서드를 나타내는 <a href="#">MethodInfo</a> 개체를 반환합니다.

## 참조

- [C# 프로그래밍 가이드](#)
- [제네릭](#)
- [리플렉션 및 제네릭 형식](#)
- [제네릭](#)

# 제네릭 및 특성(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

제네릭 형식에 특성을 적용하는 방법은 제네릭이 아닌 형식의 경우와 동일합니다. 특성 적용에 대한 자세한 내용은 [특성](#)을 참조하세요.

사용자 지정 특성은 형식 인수가 제공되지 않는 개방형 제네릭 형식 및 모든 형식 매개 변수에 인수를 제공하는 폐쇄형으로 생성된 제네릭 형식만 참조할 수 있습니다.

다음 예에서는 이러한 사용자 지정 특성을 사용합니다.

```
class CustomAttribute : System.Attribute
{
    public System.Object info;
}
```

특성은 개방형 제네릭 형식을 참조할 수 있습니다.

```
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.

```
public class GenericClass2<T, U> { }

[CustomAttribute(info = typeof(GenericClass2<, >))]
class ClassB { }
```

특성은 폐쇄형으로 생성된 제네릭 형식을 참조할 수 있습니다.

```
public class GenericClass3<T, U, V> { }

[CustomAttribute(info = typeof(GenericClass3<int, double, string>))]
class ClassC { }
```

제네릭 형식 매개 변수를 참조하는 특성은 컴파일 시간 오류를 발생시킵니다.

```
//[CustomAttribute(info = typeof(GenericClass3<int, T, string>))] //Error
class ClassD<T> { }
```

제네릭 형식은 [Attribute](#)에서 상속할 수 없습니다.

```
//public class CustomAtt<T> : System.Attribute {} //Error
```

런타임에 제네릭 형식 또는 형식 매개 변수에 대한 정보를 얻으려면 [System.Reflection](#)의 메서드를 사용합니다. 자세한 내용은 [제네릭 및 리플렉션](#)을 참조하세요.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [제네릭](#)
- [특성](#)

# 네임스페이스(C# 프로그래밍 가이드)

2021-02-18 • 3 minutes to read • [Edit Online](#)

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

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

`System`은 네임스페이스이고 `Console`은 해당 네임스페이스의 클래스입니다. 전체 키워드가 필요하지 않으므로 다음 예처럼 `using` 키워드를 사용할 수 있습니다.

```
using System;
```

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

자세한 내용은 [using 지시문](#)을 참조하세요.

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

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

네임스페이스 이름은 유효한 C# [식별자](#) 이름이어야 합니다.

## 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 사양](#)의 [네임스페이스](#) 섹션을 참조하세요.

## 참고 항목

- C# 프로그래밍 가이드
- 네임스페이스 사용
- My 네임스페이스를 사용하는 방법
- 식별자 이름
- using 지시문
- :: 연산자

# 네임스페이스 사용(C# 프로그래밍 가이드)

2020-11-02 • 10 minutes to read • [Edit Online](#)

네임스페이스는 C# 프로그램 내에서 두 가지 방법으로 많이 사용됩니다. 첫째,.NET 클래스는 네임스페이스를 사용하여 많은 클래스를 구성합니다. 둘째, 고유한 네임스페이스를 선언하면 대규모 프로그래밍 프로젝트에서 클래스 및 메서드 이름의 범위를 제어할 수 있습니다.

## 네임스페이스 액세스

대부분의 C# 애플리케이션은 `using` 지시문 섹션으로 시작합니다. 이 섹션에는 애플리케이션이 자주 사용하는 네임스페이스가 나열되어, 프로그래머가 내부에 포함된 메서드를 사용할 때마다 정규화된 이름을 지정할 필요가 없도록 합니다.

예를 들어 다음 줄을 포함할 수 있습니다.

```
using System;
```

프로그램의 시작 부분에서 프로그래머는 다음 코드를 사용할 수 있습니다.

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

위 코드를 아래 코드 대신 사용합니다.

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

## 네임스페이스 별칭

`using` 지시문을 사용하여 네임스페이스에 대한 별칭을 만들 수도 있습니다. [네임스페이스 별칭 한정자 ::](#)를 사용하여 별칭이 지정된 네임스페이스의 구성원에 액세스합니다. 다음 예제에서는 네임스페이스 별칭을 만들고 사용하는 방법을 보여 줍니다.

```
using generics = System.Collections.Generic;

namespace AliasExample
{
    class TestClass
    {
        static void Main()
        {
            generics::Dictionary<string, int> dict = new generics::Dictionary<string, int>()
            {
                ["A"] = 1,
                ["B"] = 2,
                ["C"] = 3
            };

            foreach (string name in dict.Keys)
            {
                System.Console.WriteLine($"{name} {dict[name]}");
            }
            // Output:
            // A 1
            // B 2
            // C 3
        }
    }
}
```

## 네임스페이스를 사용하여 범위 제어

`namespace` 키워드는 범위를 선언하는 데 사용됩니다. 프로젝트 내에서 범위를 만드는 기능은 코드 구성에 도움이 되며, 전역적으로 고유한 형식을 만들 수 있게 해줍니다. 다음 예제에서 `SampleClass`라는 클래스는 서로 중첩된 두 개의 네임스페이스에 정의되어 있습니다. `.` 토큰은 호출되는 메서드를 구분하는 데 사용됩니다.

```

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

    // Create a nested namespace, and define another class.
    namespace NestedNamespace
    {
        class SampleClass
        {
            public void SampleMethod()
            {
                System.Console.WriteLine(
                    "SampleMethod inside NestedNamespace");
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // Displays "SampleMethod inside SampleNamespace."
            SampleClass outer = new SampleClass();
            outer.SampleMethod();

            // Displays "SampleMethod inside SampleNamespace."
            SampleNamespace.SampleClass outer2 = new SampleNamespace.SampleClass();
            outer2.SampleMethod();

            // Displays "SampleMethod inside NestedNamespace."
            NestedNamespace.SampleClass inner = new NestedNamespace.SampleClass();
            inner.SampleMethod();
        }
    }
}

```

## 정규화된 이름

네임스페이스 및 형식에는 논리적 계층 구조를 나타내는 정규화된 이름으로 설명된 고유한 제목이 있습니다.  
예를 들어 `A.B` 문은 `A`는 네임스페이스 또는 형식의 이름이고 `B`는 그 안에 중첩됨을 암시합니다.

다음 예제에서는 중첩된 클래스 및 네임스페이스가 있습니다. 정규화된 이름이 각 엔터티 뒤에 주석으로 표시됩니다.

```

namespace N1      // N1
{
    class C1      // N1.C1
    {
        class C2  // N1.C1.C2
        {
        }
    }
    namespace N2  // N1.N2
    {
        class C2  // N1.N2.C2
        {
        }
    }
}

```

앞의 코드 세그먼트에서:

- **N1** 네임스페이스는 전역 네임스페이스의 멤버입니다. 정규화된 이름은 **N1**입니다.
- **N2** 네임스페이스는 **N1**의 멤버입니다. 정규화된 이름은 **N1.N2**입니다.
- **C1** 클래스는 **N1**의 멤버입니다. 정규화된 이름은 **N1.C1**입니다.
- 이 코드에서는 클래스 이름 **C2** 가 두 번 사용되었습니다. 그러나 정규화된 이름은 고유합니다. **C2** 의 첫 번째 인스턴스는 **C1** 내에서 선언되었으므로 정규화된 이름이 **N1.C1.C2**입니다. **C2** 의 두 번째 인스턴스는 **N2** 네임스페이스 내에서 선언되었으므로 정규화된 이름이 **N1.N2.C2**입니다.

앞의 코드 세그먼트를 사용하여 다음과 같이 **N1.N2** 네임스페이스에 새 클래스 멤버 **c3**를 추가할 수 있습니다.

```

namespace N1.N2
{
    class C3  // N1.N2.C3
    {
    }
}

```

일반적으로 **네임스페이스 별칭 한정자** `::` 을(를) 사용하여 네임스페이스 별칭을 참조하거나, `global::` 을(를) 사용하여 전역 네임스페이스를 참조하고 `.` 을(를) 사용하여 형식 또는 구성원을 한정합니다.

네임스페이스 대신 형식을 참조하는 별칭과 함께 `::` 을 사용하면 오류가 발생합니다. 다음은 그 예입니다.

```
using Alias = System.Console;
```

```

class TestClass
{
    static void Main()
    {
        // Error
        //Alias::WriteLine("Hi");

        // OK
        Alias.WriteLine("Hi");
    }
}

```

**global** 단어는 미리 정의된 별칭이 아니므로 `global.X` 에 특별한 의미가 없습니다. `::` 과 함께 사용하는 경우에만 특별한 의미가 있습니다.

`global::`은 항상 별칭이 아니라 전역 네임스페이스를 참조하기 때문에 `global`이란 별칭을 정의할 경우 컴파일러 경고 CS0440이 생성됩니다. 예를 들어 다음 줄에서는 경고가 생성됩니다.

```
using global = System.Collections; // Warning
```

별칭과 함께 `::`을 사용하는 것은 좋은 방법이며, 예기치 않은 추가 형식의 도입을 방지합니다. 예를 들어 다음 예제를 고려해보세요.

```
using Alias = System;
```

```
namespace Library
{
    public class C : Alias::Exception { }
}
```

이 예제는 작동하지만, `Alias`라는 형식을 이후에 도입할 경우 `Alias.`가 해당 형식에 대신 바인딩합니다. `Alias::Exception`을 사용하면 `Alias`가 네임스페이스 별칭으로 처리되며 형식으로 잘못 인식되지 않습니다.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [네임스페이스](#)
- [멤버 액세스 식](#)
- [:: 연산자](#)
- [extern alias](#)

# My 네임스페이스를 사용하는 방법(C# 프로그래밍 가이드)

2020-11-02 • 4 minutes to read • [Edit Online](#)

`Microsoft.VisualBasic.MyServices` 네임스페이스(Visual Basic의 `My`)는 여러 .NET 클래스에 대한 쉽고 직관적인 액세스를 제공하여 컴퓨터, 애플리케이션, 설정, 리소스 등을 조작하는 코드를 작성할 수 있게 해줍니다. 원래 Visual Basic에서 사용하도록 설계되었지만 `MyServices` 네임스페이스는 C# 애플리케이션에서 사용할 수 있습니다.

Visual Basic에서 `MyServices` 네임스페이스를 사용하는 방법에 대한 자세한 내용은 [My를 사용한 개발](#)을 참조하세요.

## 참조 추가

솔루션에서 `MyServices` 클래스를 사용하려면 먼저 Visual Basic 라이브러리에 대한 참조를 추가해야 합니다.

### Visual Basic 라이브러리에 대한 참조 추가

- 솔루션 탐색기에서 참조 노드를 마우스 오른쪽 단추로 클릭하고 참조 추가를 선택합니다.
- 참조 대화 상자가 나타나면 목록 아래로 스크롤한 다음 `Microsoft.VisualBasic.dll`을 선택합니다.

프로그램의 시작 부분에 있는 `using` 섹션에 다음 줄을 포함할 수도 있습니다.

```
using Microsoft.VisualBasic.Devices;
```

## 예제

이 예제에서는 `MyServices` 네임스페이스에 포함된 다양한 정적 메서드를 호출합니다. 이 코드를 컴파일하려면 `Microsoft.VisualBasic.dll`에 대한 참조를 프로젝트에 추가해야 합니다.

```

using System;
using Microsoft.VisualBasic.Devices;

class TestMyServices
{
    static void Main()
    {
        // Play a sound with the Audio class:
        Audio myAudio = new Audio();
        Console.WriteLine("Playing sound...");
        myAudio.Play(@"c:\WINDOWS\Media\chimes.wav");

        // Display time information with the Clock class:
        Clock myClock = new Clock();
        Console.Write("Current day of the week: ");
        Console.WriteLine(myClock.LocalTime.DayOfWeek);
        Console.Write("Current date and time: ");
        Console.WriteLine(myClock.LocalTime);

        // Display machine information with the Computer class:
        Computer myComputer = new Computer();
        Console.WriteLine("Computer name: " + myComputer.Name);

        if (myComputer.Network.IsAvailable)
        {
            Console.WriteLine("Computer is connected to network.");
        }
        else
        {
            Console.WriteLine("Computer is not connected to network.");
        }
    }
}

```

**MyServices** 네임스페이스의 모든 클래스를 C# 애플리케이션에서 호출할 수 있는 것은 아닙니다. 예를 들어 **FileSystemProxy** 클래스는 호환되지 않습니다. 이 특정한 경우에는 **FileSystem**의 일부이고 **VisualBasic.dll**에도 포함된 정적 메서드를 대신 사용할 수 있습니다. 예를 들어 이러한 메서드 중 하나를 사용하여 딕터리를 복제하는 방법은 다음과 같습니다.

```

// Duplicate a directory
Microsoft.VisualBasic.FileIO.FileSystem.CopyDirectory(
    @"C:\original_directory",
    @"C:\copy_of_original_directory");

```

## 참조

- [C# 프로그래밍 가이드](#)
- [네임스페이스](#)
- [네임스페이스 사용](#)

# 안전하지 않은 코드 및 포인터 - (C# 프로그래밍 가이드)

2021-02-18 • 3 minutes to read • [Edit Online](#)

형식 안전성 및 보안을 유지하기 위해 C#에서는 포인터 산술 연산을 기본적으로 지원하지 않습니다. 그러나 [unsafe](#) 키워드를 사용하여 포인터를 사용할 수 있는 안전하지 않은 컨텍스트를 정의할 수 있습니다. 포인터에 대한 자세한 내용은 [포인터 형식](#)을 참조하세요.

## NOTE

CLR(공용 언어 런타임)에서 안전하지 않은 코드를 확인할 수 없는 코드라고 합니다. C#에 안전하지 않은 코드가 반드시 위험한 것은 아니며, CLR에서 안전을 확인할 수 없는 코드입니다. 그러므로 CLR은 완전히 신뢰할 수 있는 어셈블리에 있는 경우 안전하지 않은 코드만 실행합니다. 안전하지 않은 코드를 사용하는 경우 코드로 인해 보안 위험이나 포인터 오류가 발생하지 않도록 확인해야 합니다.

## 안전하지 않은 코드 개요

안전하지 않은 코드에는 다음과 같은 속성이 있습니다.

- 메서드, 형식 및 코드 블록은 안전하지 않은 것으로 정의할 수 있습니다.
- 경우에 따라 안전하지 않은 코드는 배열 범위 검사를 제거하여 애플리케이션의 성능을 향상할 수 있습니다.
- 포인터가 필요한 네이티브 함수를 호출하는 경우 안전하지 않은 코드가 필요합니다.
- 안전하지 않은 코드를 사용하면 보안 및 안정성 위험이 발생합니다.
- 안전하지 않은 블록을 포함하는 코드는 [-안전하지 않은](#) 컴파일러 옵션을 사용하여 컴파일해야 합니다.

## 관련 단원

자세한 내용은 다음을 참조하세요.

- [포인터 형식](#)
- [고정 크기 버퍼](#)

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 [안전하지 않은 코드](#) 토픽을 참조하세요.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [unsafe](#)

# 고정 크기 버퍼(C# 프로그래밍 가이드)

2020-11-02 • 7 minutes to read • [Edit Online](#)

C#에서는 `fixed` 문을 사용하여 데이터 구조에 고정 크기 배열이 있는 버퍼를 만들 수 있습니다. 고정 크기 버퍼는 다른 언어 또는 플랫폼에서 데이터 원본과 interop하는 메서드를 작성하는 경우 유용합니다. 고정 배열은 일반 구조체 멤버에 허용되는 특성이나 한정자를 사용할 수 있습니다. 배열 형식이 `bool`, `byte`, `char`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint`, `ulong`, `float` 또는 `double`이어야 한다는 것이 유일한 제한 사항입니다.

```
private fixed char name[30];
```

## 설명

안전한 코드에서 배열을 포함하는 C# 구조체는 배열 요소를 포함하지 않습니다. 대신 구조체에 요소에 대한 참조가 포함되어 있습니다. `unsafe` 코드 블록에서 사용될 경우 `struct`에 고정 크기 배열을 포함할 수 있습니다.

`pathName` 이 참조이므로 다음 `struct`의 크기는 배열에 있는 요소의 수에 따라 달라지지 않습니다.

```
public struct PathArray
{
    public char[] pathName;
    private int reserved;
}
```

`struct`은 안전하지 않은 코드에 포함된 배열을 포함할 수 있습니다. 다음 예제에서 `fixedBuffer` 배열은 고정 크기입니다. `fixed` 명령문을 사용하여 첫 번째 요소에 대한 포인터를 설정합니다. 이 포인터를 통해 배열의 요소에 액세스합니다. `fixed` 명령문은 `fixedBuffer`의 인스턴스 필드를 메모리의 특정 위치에 고정합니다.

```

internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}

internal unsafe class Example
{
    public Buffer buffer = default;
}

private static void AccessEmbeddedArray()
{
    var example = new Example();

    unsafe
    {
        // Pin the buffer to a fixed location in memory.
        fixed (char* charPtr = example.buffer.fixedBuffer)
        {
            *charPtr = 'A';
        }
        // Access safely through the index:
        char c = example.buffer.fixedBuffer[0];
        Console.WriteLine(c);

        // Modify through the index:
        example.buffer.fixedBuffer[0] = 'B';
        Console.WriteLine(example.buffer.fixedBuffer[0]);
    }
}

```

128개 요소 `char` 배열의 크기는 256바이트입니다. 고정 크기 `char` 버퍼는 인코딩에 관계없이 항상 문자당 2바이트를 사용합니다. 이는 `char` 버퍼가 `CharSet = CharSet.Auto` 또는 `CharSet = CharSet.Ansi`를 사용하는 API 메서드 또는 구조체로 마샬링되는 경우에도 마찬가지입니다. 자세한 내용은 [CharSet](#)을 참조하세요.

앞의 예제에서는 C# 7.3부터 사용할 수 있으며 고정하지 않은 `fixed` 필드에 액세스하는 방법을 보여 줍니다.

또 다른 일반적인 고정 크기 배열은 `bool` 배열입니다. `bool` 배열의 요소 크기는 항상 1바이트입니다. `bool` 배열은 비트 배열이나 버퍼를 만드는 데 적합하지 않습니다.

고정 크기 버퍼는 잠재적으로 오버플로될 수 있는 관리되지 않는 배열을 형식에 포함하는 CLR(공용 언어 런타임)에 지시하는 `System.Runtime.CompilerServices.UnsafeValueTypeAttribute`으로 컴파일됩니다. 이는 CLR에서 버퍼 오버런 검색 기능을 자동으로 사용하도록 설정하는 `stackalloc`를 사용하여 만든 메모리와 비슷합니다. 이전 예제에서는 `unsafe struct`에 고정 크기 버퍼가 존재할 수 있는 방법을 보여줍니다.

```

internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}

```

`Buffer`에 대해 컴파일에서 생성된 C#은 다음과 같은 특성이 있습니다.

```
internal struct Buffer
{
    [StructLayout(LayoutKind.Sequential, Size = 256)]
    [CompilerGenerated]
    [UnsafeValueType]
    public struct <fixedBuffer>e__FixedBuffer
    {
        public char FixedElementField;
    }

    [FixedBuffer(typeof(char), 128)]
    public <fixedBuffer>e__FixedBuffer fixedBuffer;
}
```

고정 크기 버퍼는 다음과 같은 측면에서 일반 배열과 다릅니다.

- [안전하지 않은 컨텍스트](#)에서만 사용할 수 있습니다.
- 단지 구조체의 인스턴스 필드일 수 있습니다.
- 항상 벡터 또는 1차원 배열입니다.
- 선언에는 `fixed char id[8]` 와 같은 길이가 포함되어야 합니다. `fixed char id[]` 을 사용할 수 없습니다.

## 참조

- [C# 프로그래밍 가이드](#)
- [안전하지 않은 코드 및 포인터](#)
- [fixed 문](#)
- [상호 운용성](#)

# 포인터 형식(C# 프로그래밍 가이드)

2020-11-02 • 9 minutes to read • [Edit Online](#)

안전하지 않은 컨텍스트에서는 형식이 포인터 형식, 값 형식 또는 참조 형식이 될 수 있습니다. 포인터 형식 선언은 다음 형식 중 하나를 사용합니다.

```
type* identifier;  
void* identifier; //allowed but not recommended
```

포인터 형식에서 `*` 앞에 지정된 형식을 **참조 형식**이라고 합니다. **비관리형 형식**만 참조 형식일 수 있습니다.

포인터 형식은 **개체**에서 상속되지 않으며 포인터 형식과 `object`는 서로 변환되지 않습니다. 또한 boxing과 unboxing은 포인터를 지원하지 않습니다. 그러나 다른 포인터 형식 간의 변환 및 포인터 형식과 정수 형식 사이의 변환은 허용됩니다.

동일한 선언에서 여러 포인터를 선언하는 경우 별표(\*)는 기본 형식에만 함께 사용되고 각 포인터 이름의 접두사로는 사용되지 않습니다. 다음은 그 예입니다.

```
int* p1, p2, p3; // Ok  
int *p1, *p2, *p3; // Invalid in C#
```

개체 참조는 포인터가 해당 개체 참조를 가리키는 경우에도 가비지 수집될 수 있으므로 포인터는 참조나 참조가 들어 있는 **구조체**를 가리킬 수 없습니다. 가비지 수집기는 포인터 형식에서 개체를 가리키는지 여부를 추적하지 않습니다.

`myType*` 형식의 포인터 변수 값은 `myType` 형식의 변수 주소입니다. 다음은 포인터 형식 선언의 예제입니다.

예제	DESCRIPTION
<code>int* p</code>	<code>p</code> 는 정수에 대한 포인터입니다.
<code>int** p</code>	<code>p</code> 는 정수에 대한 포인터를 가리키는 포인터입니다.
<code>int*[] p</code>	<code>p</code> 는 정수에 대한 포인터의 1차원 배열입니다.
<code>char* p</code>	<code>p</code> 는 문자에 대한 포인터입니다.
<code>void* p</code>	<code>p</code> 는 알 수 없는 형식에 대한 포인터입니다.

포인터 간접 참조 연산자 `*`를 사용하면 포인터 변수가 가리키는 위치의 내용에 액세스할 수 있습니다. 예를 들어, 다음 선언을 참조하십시오.

```
int* myVariable;
```

여기서 `*myVariable` 식은 `int`에 포함된 주소에 있는 `myVariable` 변수를 가리킵니다.

**fixed 문** 및 **포인터 변환** 항목에 포인터에 대한 몇 가지 예제가 나와 있습니다. 다음 예제는 `unsafe` 키워드 및 `fixed` 문을 사용하고 정수 포인터를 증분하는 방법을 보여줍니다. 이 코드를 실행하려면 콘솔 애플리케이션의 주 함수에 붙여 넣습니다. 이러한 예제는 **-unsafe** 컴파일러 옵션 집합으로 컴파일되어야 합니다.

```

// Normal pointer to an object.
int[] a = new int[5] { 10, 20, 30, 40, 50 };
// Must be in unsafe code to use interior pointers.
unsafe
{
    // Must pin object on heap so that it doesn't move while using interior pointers.
    fixed (int* p = &a[0])
    {
        // p is pinned as well as object, so create another pointer to show incrementing it.
        int* p2 = p;
        Console.WriteLine(*p2);
        // Incrementing p2 bumps the pointer by four bytes due to its type ...
        p2 += 1;
        Console.WriteLine(*p2);
        p2 += 1;
        Console.WriteLine(*p2);
        Console.WriteLine("-----");
        Console.WriteLine(*p);
        // Dereferencing p and incrementing changes the value of a[0] ...
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}

Console.WriteLine("-----");
Console.WriteLine(a[0]);

/*
Output:
10
20
30
-----
10
11
12
-----
12
*/

```

`void*` 형식의 포인터에는 간접 참조 연산자를 적용할 수 없습니다. 그러나 캐스트를 사용하여 `void` 포인터를 다른 포인터 형식으로 변환하거나 반대로 변환할 수 있습니다.

포인터는 `null`일 수 있습니다. `null` 포인터에 간접 참조 연산자를 적용할 때 발생하는 동작은 구현에 따라 다릅니다.

메서드 사이에 포인터를 전달하면 정의되지 않은 동작이 발생할 수 있습니다. `in`, `out` 또는 `ref` 매개 변수를 통해, 또는 함수 결과로 지역 변수에 포인터를 반환하는 메서드를 고려합니다. `fixed` 블록에서 포인터가 설정되면 이 포인터가 가리키는 변수의 고정 상태가 해제될 수 있습니다.

다음 표에서는 안전하지 않은 컨텍스트에서 포인터에 대해 수행할 수 있는 연산자와 문을 보여 줍니다.

연산자/문	사용
<code>*</code>	포인터 간접 참조를 수행합니다.
<code>-&gt;</code>	포인터를 통해 구조체 멤버에 액세스합니다.
<code>[]</code>	포인터를 인덱싱합니다.

연산자/문	사용
<code>&amp;</code>	변수 주소를 가져옵니다.
<code>++</code> 및 <code>--</code>	포인터를 증가 및 감소시킵니다.
<code>+</code> 및 <code>-</code>	포인터 연산을 수행합니다.
<code>==</code> , <code>!=</code> , <code>&lt;</code> , <code>&gt;</code> , <code>&lt;=</code> 및 <code>&gt;=</code>	포인터를 비교합니다.
<code>stackalloc</code>	스택에 메모리를 할당합니다.
<code>fixed statement</code>	해당 주소를 찾을 수 있도록 임시로 변수를 고정합니다.

포인터에 관련 연산자에 대한 자세한 내용은 [포인터 관련 연산자를 참조하세요](#).

## C# 언어 사양

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

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [안전하지 않은 코드 및 포인터](#)
- [포인터 변환](#)
- [참조 형식](#)
- [값 형식](#)
- [unsafe](#)

# 포인터 변환(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

다음 표에서는 미리 정의된 암시적 포인터 변환을 보여 줍니다. 암시적 변환은 메서드 호출, 할당 문을 비롯한 대부분의 경우에서 발생할 수 있습니다.

## 암시적 포인터 변환

시작	대상
임의의 포인터 형식	void*
null	임의의 포인터 형식

명시적 포인터 변환은 캐스트 식을 통해 암시적 변환이 없는 변환을 수행하는 데 사용됩니다. 다음 표에는 이러한 변환이 나와 있습니다.

## 명시적 포인터 변환

시작	대상
임의의 포인터 형식	다른 포인터 형식
sbyte, byte, short, ushort, int, uint, long 또는 ulong	임의의 포인터 형식
임의의 포인터 형식	sbyte, byte, short, ushort, int, uint, long 또는 ulong

## 예제

다음 예제에서 `int`에 대한 포인터는 `byte`에 대한 포인터로 변환됩니다. 포인터는 변수의 최하위 주소 지정 바이트를 가리킵니다. `int` 크기(4바이트)까지 결과를 연속적으로 증가할 경우 변수의 나머지 바이트를 표시할 수 있습니다.

```
// compile with: -unsafe
```

```

class ClassConvert
{
    static void Main()
    {
        int number = 1024;

        unsafe
        {
            // Convert to byte:
            byte* p = (byte*)&number;

            System.Console.WriteLine("The 4 bytes of the integer:");

            // Display the 4 bytes of the int variable:
            for (int i = 0 ; i < sizeof(int) ; ++i)
            {
                System.Console.Write(" {0:X2}", *p);
                // Increment the pointer:
                p++;
            }
            System.Console.WriteLine();
            System.Console.WriteLine("The value of the integer: {0}", number);

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
}

/* Output:
   The 4 bytes of the integer: 00 04 00 00
   The value of the integer: 1024
*/

```

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [포인터 형식](#)
- [참조 형식](#)
- [값 형식](#)
- [unsafe](#)
- [fixed 문](#)
- [stackalloc](#)

# 포인터를 사용하여 바이트 배열을 복사하는 방법 (C# 프로그래밍 가이드)

2020-11-02 • 4 minutes to read • [Edit Online](#)

다음 예제에서는 포인터를 사용하여 배열 간에 바이트를 복사합니다.

이 예제에서는 `Copy` 메서드에서 포인터를 사용할 수 있도록 하는 `unsafe` 키워드를 사용합니다. `fixed` 문은 소스 및 대상 배열에 대한 포인터를 선언하는 데 사용됩니다. `fixed` 명령문은 소스 및 대상 배열의 위치가 메모리에 고정되므로 가비지 수집에 의해 이동되지 않습니다. `fixed` 블록이 완료되면 배열에 대한 메모리 블록이 고정 해제됩니다. 이 예제의 `Copy` 메서드는 `unsafe` 키워드를 사용하기 때문에 `-unsafe` 컴파일러 옵션으로 컴파일해야 합니다.

이 예제에서는 두 번째 관리되지 않는 포인터보다는 인덱스를 사용하여 두 배열의 요소에 액세스합니다.

`pSource` 및 `pTarget` 포인터의 선언은 배열을 고정합니다. 이 기능은 C# 7.3부터 사용할 수 있습니다.

## 예제

```
static unsafe void Copy(byte[] source, int sourceOffset, byte[] target,
    int targetOffset, int count)
{
    // If either array is not instantiated, you cannot complete the copy.
    if ((source == null) || (target == null))
    {
        throw new System.ArgumentException();
    }

    // If either offset, or the number of bytes to copy, is negative, you
    // cannot complete the copy.
    if ((sourceOffset < 0) || (targetOffset < 0) || (count < 0))
    {
        throw new System.ArgumentException();
    }

    // If the number of bytes from the offset to the end of the array is
    // less than the number of bytes you want to copy, you cannot complete
    // the copy.
    if ((source.Length - sourceOffset < count) ||
        (target.Length - targetOffset < count))
    {
        throw new System.ArgumentException();
    }

    // The following fixed statement pins the location of the source and
    // target objects in memory so that they will not be moved by garbage
    // collection.
    fixed (byte* pSource = source, pTarget = target)
    {
        // Copy the specified number of bytes from source to target.
        for (int i = 0; i < count; i++)
        {
            pTarget[targetOffset + i] = pSource[sourceOffset + i];
        }
    }
}

static void UnsafeCopyArrays()
{
    // Create two arrays of the same length.
```

```

int length = 100;
byte[] byteArray1 = new byte[length];
byte[] byteArray2 = new byte[length];

// Fill byteArray1 with 0 - 99.
for (int i = 0; i < length; ++i)
{
    byteArray1[i] = (byte)i;
}

// Display the first 10 elements in byteArray1.
System.Console.WriteLine("The first 10 elements of the original are:");
for (int i = 0; i < 10; ++i)
{
    System.Console.Write(byteArray1[i] + " ");
}
System.Console.WriteLine("\n");

// Copy the contents of byteArray1 to byteArray2.
Copy(byteArray1, 0, byteArray2, 0, length);

// Display the first 10 elements in the copy, byteArray2.
System.Console.WriteLine("The first 10 elements of the copy are:");
for (int i = 0; i < 10; ++i)
{
    System.Console.Write(byteArray2[i] + " ");
}
System.Console.WriteLine("\n");

// Copy the contents of the last 10 elements of byteArray1 to the
// beginning of byteArray2.
// The offset specifies where the copying begins in the source array.
int offset = length - 10;
Copy(byteArray1, offset, byteArray2, 0, length - offset);

// Display the first 10 elements in the copy, byteArray2.
System.Console.WriteLine("The first 10 elements of the copy are:");
for (int i = 0; i < 10; ++i)
{
    System.Console.Write(byteArray2[i] + " ");
}
System.Console.WriteLine("\n");
/* Output:
   The first 10 elements of the original are:
   0 1 2 3 4 5 6 7 8 9

   The first 10 elements of the copy are:
   0 1 2 3 4 5 6 7 8 9

   The first 10 elements of the copy are:
   90 91 92 93 94 95 96 97 98 99
*/
}
}

```

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [안전하지 않은 코드 및 포인터](#)
- [-unsafe\(C# 컴파일러 옵션\)](#)
- [가비지 수집](#)

# XML 문서 주석(C# 프로그래밍 가이드)

2020-11-02 • 3 minutes to read • [Edit Online](#)

C#에서는 주석이 참조하는 코드 블록 바로 앞의 소스 코드의 특별 주석 필드(세 개의 슬래시로 표시)에 XML 요소를 포함하여 코드에 대한 문서를 만들 수 있습니다. 예를 들면 다음과 같습니다.

```
/// <summary>
/// This class performs an important function.
/// </summary>
public class MyClass {}
```

-설명서 옵션을 사용하여 컴파일하는 경우 컴파일러는 소스 코드에서 모든 XML 태그를 검색하고 XML 문서 파일을 만듭니다. 컴파일러에서 생성한 파일을 기반으로 해서 최종 문서를 만들려면 사용자 지정 도구를 만들거나 [DocFX](#) 또는 [Sandcastle](#)과 같은 도구를 사용하면 됩니다.

XML 요소를 참조하려면(예를 들어, 함수가 XML 문서 주석에서 설명하려는 특정 XML 요소를 처리하는 경우) 표준 인용 메커니즘( < 및 > )을 사용할 수 있습니다. 코드 참조(`cref`) 요소에서 제네릭 식별자를 참조하려면 이스케이프 문자(예: `cref="List<T>"` ) 또는 괄호(`cref="List{T}"`)를 사용할 수 있습니다. 특별한 경우로, 컴파일러는 제네릭 식별자를 참조할 때 문서 주석을 더 쉽게 작성할 수 있도록 괄호를 꺾쇠 괄호로 구문 분석합니다.

## NOTE

XML 문서 주석은 메타데이터가 아닙니다. 이러한 주석은 컴파일된 어셈블리에 포함되지 않으므로 리플렉션을 통해 액세스할 수 없습니다.

## 단원 내용

- [문서 주석에 대한 권장 태그](#)
- [XML 파일 처리](#)
- [문서 태그에 대한 구분 기호](#)
- [XML 문서 기능을 사용하는 방법](#)

## 관련 단원

자세한 내용은 다음을 참조하세요.

- [-설명서\(문서 주석 처리\)](#)

## C# 언어 사양

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

## 참조

- [C# 프로그래밍 가이드](#)

# 문서 주석에 대한 권장 태그(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

C# 컴파일러는 코드의 문서 주석을 처리하고 `/doc` 명령줄 옵션에서 지정한 이름의 파일에 XML로 형식을 저장합니다. 컴파일러에서 생성한 파일을 기반으로 해서 최종 문서를 만들려면 사용자 지정 도구를 만들거나 DocFX 또는 Sandcastle과 같은 도구를 사용하면 됩니다.

태그는 형식, 형식 멤버 등의 코드 구문에서 처리됩니다.

## NOTE

네임스페이스에는 문서 주석을 적용할 수 없습니다.

컴파일러는 유효한 XML인 태그를 모두 처리합니다. 다음 태그는 사용자 문서에서 일반적으로 사용되는 기능을 제공합니다.

## Tags

<c>	<para>	<see>*	<value>
<code>	<param>*	<seealso>*	
<example>	<paramref>	<summary>	
<exception>*	<permission>*	<typeparam>*	
<include>*	<remarks>	<typeparamref>	
<list>	<inheritdoc>	<returns>	

(\* 컴파일러에서 구문을 확인함을 나타냅니다.)

문서 주석의 텍스트에 꺽쇠 괄호를 표시하려면 각각 `&lt;` 및 `&gt;` 인 `<` 및 `>`의 HTML 인코딩을 사용합니다. 이 인코딩은 다음 예제에서 확인할 수 있습니다.

```
/// <summary>
/// This property always returns a value &lt; 1.
/// </summary>
```

## 참조

- [C# 프로그래밍 가이드](#)
- [-doc\(C# 컴파일러 옵션\)](#)
- [XML 문서 주석](#)

# XML 파일 처리(C# 프로그래밍 가이드)

2020-11-02 • 13 minutes to read • [Edit Online](#)

컴파일러는 문서 생성을 위해 태그가 지정되는 코드의 각 구문에 대해 ID 문자열을 생성합니다. 코드에 태그를 지정하는 방법에 대한 자세한 내용은 [문서 주석에 대한 권장 태그](#)를 참조하세요. ID 문자열은 구문을 고유하게 식별합니다. XML 파일을 처리하는 프로그램은 ID 문자열을 사용하여 문서가 적용되는 해당 .NET 메타데이터 또는 리플렉션 항목을 식별할 수 있습니다.

## ID 문자열

XML 파일은 코드의 계층적 표현이 아닙니다. 각 요소에 대해 생성된 ID가 포함된 단순 목록입니다.

컴파일러는 ID 문자열을 생성할 때 다음 규칙을 관찰합니다.

- 문자열에 공백이 없음.
- 문자열의 첫 부분이 뒤에 콜론이 붙은 한 글자를 사용해 멤버의 종류를 식별합니다. 사용되는 멤버 유형은 다음과 같습니다.

문자	멤버 형식	참고
N	namespace	네임스페이스에 문서 주석을 추가할 수는 없지만 지원되는 경우 문서 주석에 대한 cref 참조를 만들 수는 있습니다.
T	type	클래스, 인터페이스, 구조체, 열거형 또는 대리자일 수 있습니다.
F	필드(field)	
P	속성(property)	인덱서 또는 기타 인덱싱된 속성을 포함합니다.
M	메서드	생성자 및 연산자와 같은 특수 메서드를 포함합니다.
E	event	
!	오류 문자열	문자열의 나머지 부분은 오류에 대한 정보를 제공합니다. C# 컴파일러는 확인할 수 없는 링크에 대해 오류 정보를 생성합니다.

- 문자열의 두 번째 부분은 네임스페이스의 루트부터 시작되는 항목의 정규화된 이름입니다. 항목의 이름과 바깥쪽 형식 및 네임스페이스는 마침표로 구분됩니다. 항목 자체의 이름에 마침표가 있으면 이러한 요소를 구분하는 마침표가 해시 기호('#')로 바뀝니다. 항목 이름에는 해시 기호가 직접적으로 포함되지 않는다고 가정합니다. 예를 들어 String 생성자의 정규화된 이름은 "System.String.#ctor"입니다.
- 속성 및 메서드의 경우 괄호로 묶은 매개 변수 목록이 뒤에 나옵니다. 매개 변수가 없으면 괄호도 없습니다. 매개 변수는 쉼표로 구분됩니다. 각 매개 변수의 인코딩은 .NET 시그니처에서 매개 변수가 인코딩되는 방식을 그대로 따릅니다.

- 기본 형식. 일반 형식(ELEMENT\_TYPE\_CLASS 또는 ELEMENT\_TYPE\_VALUETYPE)은 해당 형식의 정규화된 이름으로 표시됩니다.
- 내장 형식(예: ELEMENT\_TYPE\_I4, ELEMENT\_TYPE\_OBJECT, ELEMENT\_TYPE\_STRING, ELEMENT\_TYPE\_TYPEDBYREF 및 ELEMENT\_TYPE\_VOID)은 해당 전체 형식의 정규화된 이름으로 표시됩니다. 예를 들면 System.Int32 또는 System.TypedReference와 같습니다.
- ELEMENT\_TYPE\_PTR은 수정된 형식 뒤에 '\*'로 표시됩니다.
- ELEMENT\_TYPE\_BYREF는 수정된 형식 뒤에 '@'으로 표시됩니다.
- ELEMENT\_TYPE\_PINNED는 수정된 형식 뒤에 '^'로 표시됩니다. C# 컴파일러에서는 이 인수가 생성되지 않습니다.
- ELEMENT\_TYPE\_CMOD\_REQ는 수정된 형식 뒤에 '!' 및 한정자 클래스의 정규화된 이름으로 표시됩니다. C# 컴파일러에서는 이 인수가 생성되지 않습니다.
- ELEMENT\_TYPE\_CMOD\_OPT는 수정된 형식 뒤에 '!' 및 한정자 클래스의 정규화된 이름으로 표시됩니다.
- ELEMENT\_TYPE\_SZARRAY는 배열의 요소 형식 뒤에 "[]"로 표시됩니다.
- ELEMENT\_TYPE\_GENERICARRAY는 배열의 요소 형식 뒤에 "[?]"로 표시됩니다. C# 컴파일러에서는 이 인수가 생성되지 않습니다.
- ELEMENT\_TYPE\_ARRAY는 [lowerbound: size ,lowerbound: size ]로 표시됩니다. 여기서 쉼표 수는 순위 - 1에 해당하는 값이고, 각 차원의 하한과 크기(확인된 경우)는 10진수로 표시됩니다. 하한이나 크기가 지정되지 않은 경우에는 생략됩니다. 특정 차원의 하한과 크기를 생략하면 ':'도 생략됩니다. 예를 들어 하한이 1이고 크기가 지정되지 않은 2차원 배열은 [1;1:]로 표시됩니다.
- ELEMENT\_TYPE\_FNPTR은 "=FUNC: type (signature)"로 표시됩니다. 여기서 type은 반환 형식이고 signature는 메서드의 인수입니다. 인수가 없으면 괄호는 생략됩니다. C# 컴파일러에서는 이 인수가 생성되지 않습니다.

다음 시그니처 구성 요소는 오버로드된 메서드를 구분하는데 사용되지 않으므로 표시되지 않습니다.

- 호출 규칙
- 반환 형식
- ELEMENT\_TYPE\_SENTINEL
- 변환 연산자(`op_Implicit` 및 `op_Explicit`)에 한해 메서드의 반환 값은 뒤에 반환 형식이 붙은 '~'로 인코딩됩니다.
- 제네릭 형식의 경우에는 형식 이름 뒤에 억음 악센트 기호와 제네릭 형식 매개 변수의 수를 나타내는 숫자가 차례로 붙습니다. 예를 들어:

`<member name="T:SampleClass`2">`는 `public class SampleClass<T, U>`로 정의된 형식의 태그입니다.

제네릭 형식을 매개 변수로 사용하는 메서드의 경우 제네릭 형식 매개 변수는 '0,1과 같이 앞에 억음 악센트 기호가 붙은 숫자로 지정됩니다. 각 숫자는 해당 형식의 제네릭 매개 변수에 대한 0부터 시작되는 배열 표기법을 나타냅니다.

## 예

다음 예제에서는 클래스와 해당 멤버의 ID 문자열이 생성되는 방식을 보여 줍니다.

```
namespace N
{
    // ...
}
```

```
/// <summary>
/// Enter description here for class X.
/// ID string generated is "T:N.X".
/// </summary>
public unsafe class X
{
    /// <summary>
    /// Enter description here for the first constructor.
    /// ID string generated is "M:N.X.#ctor".
    /// </summary>
    public X() { }

    /// <summary>
    /// Enter description here for the second constructor.
    /// ID string generated is "M:N.X.#ctor(System.Int32)".
    /// </summary>
    /// <param name="i">Describe parameter.</param>
    public X(int i) { }

    /// <summary>
    /// Enter description here for field q.
    /// ID string generated is "F:N.X.q".
    /// </summary>
    public string q;

    /// <summary>
    /// Enter description for constant PI.
    /// ID string generated is "F:N.X.PI".
    /// </summary>
    public const double PI = 3.14;

    /// <summary>
    /// Enter description for method f.
    /// ID string generated is "M:N.X.f".
    /// </summary>
    /// <returns>Describe return value.</returns>
    public int f() { return 1; }

    /// <summary>
    /// Enter description for method bb.
    /// ID string generated is "M:N.X.bb(System.String,System.Int32@,System.Void*)".
    /// </summary>
    /// <param name="s">Describe parameter.</param>
    /// <param name="y">Describe parameter.</param>
    /// <param name="z">Describe parameter.</param>
    /// <returns>Describe return value.</returns>
    public int bb(string s, ref int y, void* z) { return 1; }

    /// <summary>
    /// Enter description for method gg.
    /// ID string generated is "M:N.X.gg(System.Int16[],System.Int32[0:,0:])".
    /// </summary>
    /// <param name="array1">Describe parameter.</param>
    /// <param name="array">Describe parameter.</param>
    /// <returns>Describe return value.</returns>
    public int gg(short[] array1, int[,] array) { return 0; }

    /// <summary>
    /// Enter description for operator.
    /// ID string generated is "M:N.X.op>Addition(N.X,N.X)".
    /// </summary>
    /// <param name="x">Describe parameter.</param>
    /// <param name="xx">Describe parameter.</param>
    /// <returns>Describe return value.</returns>
    public static X operator +(X x, X xx) { return x; }

    /// <summary>
    /// Enter description for property.
    /// ID string generated is "P:N.X.prop".
    /// </summary>
```

```

/// </summary>
public int prop { get { return 1; } set { } }

/// <summary>
/// Enter description for event.
/// ID string generated is "E:N.X.d".
/// </summary>
public event D d;

/// <summary>
/// Enter description for property.
/// ID string generated is "P:N.X.Item(System.String)".
/// </summary>
/// <param name="s">Describe parameter.</param>
/// <returns></returns>
public int this[string s] { get { return 1; } }

/// <summary>
/// Enter description for class Nested.
/// ID string generated is "T:N.X.Nested".
/// </summary>
public class Nested { }

/// <summary>
/// Enter description for delegate.
/// ID string generated is "T:N.X.D".
/// </summary>
/// <param name="i">Describe parameter.</param>
public delegate void D(int i);

/// <summary>
/// Enter description for operator.
/// ID string generated is "M:N.X.op_Explicit(N.X)~System.Int32".
/// </summary>
/// <param name="x">Describe parameter.</param>
/// <returns>Describe return value.</returns>
public static explicit operator int(X x) { return 1; }
}

}

```

## 참조

- [C# 프로그래밍 가이드](#)
- [-doc\(C# 컴파일러 옵션\)](#)
- [XML 문서 주석](#)

# 문서 태그에 대한 구분 기호(C# 프로그래밍 가이드)

2020-11-02 • 6 minutes to read • [Edit Online](#)

XML 문서 주석을 사용하려면 문서 주석이 시작되고 끝나는 위치를 컴파일러에 알리는 구분 기호가 필요합니다. XML 문서 태그에 다음과 같은 종류의 구분 기호를 사용할 수 있습니다.

- `///`

한 줄 구분 기호입니다. 문서 예제에 표시되고 C# 프로젝트 템플릿에 사용되는 형식입니다. 구분 기호 뒤에 공백 문자가 있으면 해당 문자는 XML 출력에 포함되지 않습니다.

## NOTE

코드 편집기에서 `///` 구분 기호를 입력한 후에는 Visual Studio IDE(통합 개발 환경)에서 자동으로 `<summary>` 및 `</summary>` 태그가 삽입되고 이 태그 내에서 커서가 이동합니다. 이 기능은 [옵션 대화 상자](#)에서 켜거나 끌 수 있습니다.

- `/** */`

여러 줄 구분 기호입니다.

`/** */` 구분 기호를 사용할 때 따라야 하는 몇 가지 서식 설정 규칙이 있습니다.

- `/**` 구분 기호가 포함된 줄에서 줄의 나머지 부분이 공백인 경우 해당 줄은 주석에 대해 처리되지 않습니다. `/**` 구분 기호 다음의 첫 번째 문자가 공백인 경우 해당 공백 문자는 무시되고 줄의 나머지 부분이 처리됩니다. 그러지 않으면 `/**` 구분 기호 다음 줄의 전체 텍스트가 주석의 일부로 처리됩니다.
- `*/` 구분 기호가 포함된 줄에서 `*/` 구분 기호까지 공백만 있으면 해당 줄은 무시됩니다. 그렇지 않으면 `*/` 구분 기호 이전 줄의 텍스트가 주석의 일부로 처리됩니다.
- `/**` 구분 기호로 시작하는 줄 다음 줄에 대해 컴파일러는 각 줄의 시작 부분에서 일반적인 패턴을 찾습니다. 패턴은 선택적 공백과 별표(`*`)로 구성되고 그다음에 더 많은 선택적 공백이 올 수 있습니다. 컴파일러가 `/**` 구분 기호나 `*/` 구분 기호로 시작되지 않는 각 줄의 시작 부분에서 일반적인 패턴을 찾으면 각 줄에 대해 해당 패턴을 무시합니다.

다음 예제에서는 이러한 규칙을 보여 줍니다.

- 다음 주석에서 유일하게 처리되는 부분은 `<summary>`로 시작하는 줄입니다. 세 가지 태그 서식이 동일한 주석을 생성합니다.

```
/** <summary>text</summary> */  
  
/**  
<summary>text</summary>  
*/  
  
/**  
* <summary>text</summary>  
*/
```

- 컴파일러는 두 번째 및 세 번째 줄의 시작 부분에서 "\*"의 일반적인 패턴을 식별합니다. 이 패턴은

출력에 포함되지 않습니다.

```
/**  
 * <summary>  
 * text </summary>*/
```

- 다음 주석에서는 세 번째 줄의 두 번째 문자가 별표가 아니기 때문에 컴파일러가 일반적인 패턴을 찾을 수 없습니다. 따라서 두 번째 및 세 번째 줄의 모든 텍스트가 주석의 일부로 처리됩니다.

```
/**  
 * <summary>  
 * text </summary>  
 */
```

- 다음 주석에서는 두 가지로 이유로 컴파일러가 패턴을 찾을 수 없습니다. 첫째, 별표 앞의 공백수가 일치하지 않습니다. 둘째, 다섯 번째 줄이 공백과 일치하지 않는 탭으로 시작합니다. 따라서 두 번째 줄부터 다섯 번째 줄까지 모든 텍스트가 주석의 일부로 처리됩니다.

```
/**  
 * <summary>  
 * text  
 * text2  
 * </summary>  
 */
```

## 참조

- [C# 프로그래밍 가이드](#)
- [XML 문서 주석](#)
- [-doc\(C# 컴파일러 옵션\)](#)

# XML 문서 기능을 사용하는 방법

2020-11-02 • 7 minutes to read • [Edit Online](#)

다음 샘플은 문서화된 형식에 대한 기본 개요를 제공합니다.

## 예제

```
// If compiling from the command line, compile with: -doc:YourFileName.xml

/// <summary>
/// Class level summary documentation goes here.
/// </summary>
/// <remarks>
/// Longer comments can be associated with a type or member through
/// the remarks tag.
/// </remarks>
public class TestClass : TestInterface
{
    /// <summary>
    /// Store for the Name property.
    /// </summary>
    private string _name = null;

    /// <summary>
    /// The class constructor.
    /// </summary>
    public TestClass()
    {
        // TODO: Add Constructor Logic here.
    }

    /// <summary>
    /// Name property.
    /// </summary>
    /// <value>
    /// A value tag is used to describe the property value.
    /// </value>
    public string Name
    {
        get
        {
            if (_name == null)
            {
                throw new System.Exception("Name is null");
            }
            return _name;
        }
    }

    /// <summary>
    /// Description for SomeMethod.
    /// </summary>
    /// <param name="s"> Parameter description for s goes here.</param>
    /// <seealso cref="System.String">
    /// You can use the cref attribute on any tag to reference a type or member
    /// and the compiler will check that the reference exists.
    /// </seealso>
    public void SomeMethod(string s)
    {
    }
}
```

```

///<summary>
/// Some other method.
///</summary>
///<returns>
/// Return values are described through the returns tag.
///</returns>
///<seealso cref="SomeMethod(string)">
/// Notice the use of the cref attribute to reference a specific method.
///</seealso>
public int SomeOtherMethod()
{
    return 0;
}

public int InterfaceMethod(int n)
{
    return n * n;
}

///<summary>
/// The entry point for the application.
///</summary>
///<param name="args">A list of command line arguments.</param>
static int Main(System.String[] args)
{
    // TODO: Add code to start application here.
    return 0;
}

///<summary>
/// Documentation that describes the interface goes here.
///</summary>
///<remarks>
/// Details about the interface go here.
///</remarks>
interface TestInterface
{
    ///<summary>
    /// Documentation that describes the method goes here.
    ///</summary>
    ///<param name="n">
    /// Parameter n requires an integer argument.
    ///</param>
    ///<returns>
    /// The method returns an integer.
    ///</returns>
    int InterfaceMethod(int n);
}

```

이 예제에서는 다음 내용과 같은 내용의 *.xm* 파일을 생성합니다.

```

<?xml version="1.0"?>
<doc>
    <assembly>
        <name>xmlesample</name>
    </assembly>
    <members>
        <member name="T:TestClass">
            <summary>
                Class level summary documentation goes here.
            </summary>
            <remarks>
                Longer comments can be associated with a type or member through
                the remarks tag.
            </remarks>
        </member>
    
```

```

<member name="F:TestClass._name">
    <summary>
        Store for the Name property.
    </summary>
</member>
<member name="M:TestClass.#ctor">
    <summary>
        The class constructor.
    </summary>
</member>
<member name="P:TestClass.Name">
    <summary>
        Name property.
    </summary>
    <value>
        A value tag is used to describe the property value.
    </value>
</member>
<member name="M:TestClass.SomeMethod(System.String)">
    <summary>
        Description for SomeMethod.
    </summary>
    <param name="s"> Parameter description for s goes here.</param>
    <seealso cref="T:System.String">
        You can use the cref attribute on any tag to reference a type or member
        and the compiler will check that the reference exists.
    </seealso>
</member>
<member name="M:TestClass.SomeOtherMethod">
    <summary>
        Some other method.
    </summary>
    <returns>
        Return values are described through the returns tag.
    </returns>
    <seealso cref="M:TestClass.SomeMethod(System.String)">
        Notice the use of the cref attribute to reference a specific method.
    </seealso>
</member>
<member name="M:TestClass.Main(System.String[])>
    <summary>
        The entry point for the application.
    </summary>
    <param name="args"> A list of command line arguments.</param>
</member>
<member name="T:TestInterface">
    <summary>
        Documentation that describes the interface goes here.
    </summary>
    <remarks>
        Details about the interface go here.
    </remarks>
</member>
<member name="M:TestInterface.InterfaceMethod(System.Int32)">
    <summary>
        Documentation that describes the method goes here.
    </summary>
    <param name="n">
        Parameter n requires an integer argument.
    </param>
    <returns>
        The method returns an integer.
    </returns>
</member>
</members>
</doc>

```

# 코드 컴파일

예제를 컴파일하려면 다음 명령을 입력합니다.

```
csc XMLsample.cs /doc:XMLsample.xml
```

이 명령으로 **TYPE** 명령을 사용하거나 브라우저에서 볼 수 있는 XML 파일 *XMLsample.xml*이 생성됩니다.

## 강력한 프로그래밍

XML 문서는 **///**로 시작합니다. 새 프로젝트를 만드는 경우 마법사에서 몇 개의 시작 **///** 줄을 자동으로 넣습니다. 이러한 주석의 처리에는 몇 가지 제한이 있습니다.

- 문서는 잘 구성된 XML이어야 합니다. XML이 잘 구성되지 않은 경우 경고가 생성되고, 문서 파일에 오류가 발생했다는 주석이 포함됩니다.
- 개별자는 각자 고유한 태그 집합을 만들 수 있습니다. [권장 태그 집합](#)이 있습니다. 권장 태그 중 일부는 특별한 의미가 있습니다.
  - <param>** 태그는 매개 변수를 설명하는 데 사용됩니다. 사용되는 경우 컴파일러는 매개 변수가 있고 모든 매개 변수가 문서에서 설명되었는지 확인합니다. 확인에 실패하면 컴파일러가 경고를 실행합니다.
  - cref** 특성을 태그에 연결하여 코드 요소를 참조할 수 있습니다. 컴파일러에서 이 코드 요소가 있음을 확인합니다. 확인에 실패하면 컴파일러가 경고를 실행합니다. 컴파일러는 **cref** 특성에 설명된 형식을 찾을 때 모든 **using** 문을 따릅니다.
  - <summary>** 태그는 Visual Studio 내의 IntelliSense에서 형식 또는 멤버에 대한 추가 정보를 표시하는 데 사용됩니다.

### NOTE

XML 파일은 형식 및 멤버에 대한 전체 정보를 제공하지 않습니다(예: 형식 정보가 포함되지 않음). 형식 또는 멤버에 대한 전체 정보를 가져오려면 실제 형식 또는 멤버에 대한 리플렉션과 함께 문서 파일을 사용하세요.

## 참조

- [C# 프로그래밍 가이드](#)
- [-doc\(C# 컴파일러 옵션\)](#)
- [XML 문서 주석](#)
- [DocFX 설명서 프로세서](#)
- [Sandcastle 설명서 프로세서](#)

# <c>(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

## 구문

```
<c>text</c>
```

## 매개 변수

- `text`

코드로 표시하려는 텍스트입니다.

## 설명

<c> 태그를 사용하면 설명 내의 텍스트를 코드로 표시해야 함을 지정할 수 있습니다. 여러 줄을 코드로 지정하려면 <code>를 사용합니다.

-doc로 컴파일하여 문서 주석을 파일로 처리합니다.

## 예제

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary><c>DoWork</c> is a method in the <c>TestClass</c> class.
    /// </summary>
    public static void DoWork(int Int1)
    {

        /// text for Main
        static void Main()
        {
        }
}
```

## 참조

- [C# 프로그래밍 가이드](#)
- [문서 주석에 대한 권장 태그](#)

# <code>(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

## 구문

```
<code>content</code>
```

## 매개 변수

- `content`

코드로 표시할 텍스트입니다.

## 설명

`<code>` 태그는 여러 코드 줄을 나타내는데 사용됩니다. 설명 내의 한 줄 텍스트가 코드로 표시되도록 지정하려면 `<c>`를 사용합니다.

`-doc`로 컴파일하여 문서 주석을 파일로 처리합니다.

## 예제

```
<code> 태그를 사용하는 방법에 대한 예제는 <example> 문서를 참조하세요.
```

## 참조

- [C# 프로그래밍 가이드](#)
- [문서 주석에 대한 권장 태그](#)

# cref 특성(C# 프로그래밍 가이드)

2020-11-02 • 3 minutes to read • [Edit Online](#)

XML 문서 태그의 `cref` 특성은 "코드 참조"를 의미합니다. 태그의 내부 텍스트를 형식, 메서드, 속성 등의 코드 요소로 지정합니다. DocFX 및 Sandcastle 등의 문서화 도구는 `cref` 특성을 사용하여 형식 또는 멤버가 문서화된 페이지에 대한 하이퍼링크를 자동으로 생성합니다.

## 예제

다음 예제에서는 `<see>` 태그에 사용된 `cref` 특성을 보여 줍니다.

```
// Save this file as CRefTest.cs
// Compile with: csc CRefTest.cs -doc:Results.xml

namespace TestNamespace
{
    /// <summary>
    /// TestClass contains several cref examples.
    /// </summary>
    public class TestClass
    {
        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass"/> constructor as a cref attribute.
        /// </summary>
        public TestClass()
        { }

        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass(int)"/> constructor as a cref
        attribute.
        /// </summary>
        public TestClass(int value)
        { }

        /// <summary>
        /// The GetZero method.
        /// </summary>
        /// <example>
        /// This sample shows how to call the <see cref="GetZero"/> method.
        /// </code>
        /// <class>TestClass</class>
        /// {
        ///     static int Main()
        ///     {
        ///         return GetZero();
        ///     }
        /// }
        /// </example>
        public static int GetZero()
        {
            return 0;
        }

        /// <summary>
        /// The GetGenericValue method.
        /// </summary>
        /// <remarks>
        /// This sample shows how to specify the <see cref="GetGenericValue"/> method as a cref attribute.
        /// </remarks>
    }
}
```

```
public static T GetGenericValue<T>(T para)
{
    return para;
}

/// <summary>
/// GenericClass.
/// </summary>
/// <remarks>
/// This example shows how to specify the <see cref="GenericClass{T}" /> type as a cref attribute.
/// </remarks>
class GenericClass<T>
{
    // Fields and members.
}

class Program
{
    static int Main()
    {
        return TestClass.GetZero();
    }
}
```

컴파일하면 프로그램이 다음 XML 파일을 생성합니다. 예를 들어 `GetZero` 메서드의 `cref` 특성은 컴파일러에서 `"M:TestNamespace.TestClass.GetZero"`로 변환되었습니다. "M:" 접두사는 "메서드"를 의미하며, DocFX 및 Sandcastle 등의 문서화 도구에서 인식되는 규칙입니다. 접두사의 전체 목록은 [XML 파일 처리](#)를 참조하세요.

```

<?xml version="1.0"?>
<doc>
    <assembly>
        <name>CRefTest</name>
    </assembly>
    <members>
        <member name="T:TestNamespace.TestClass">
            <summary>
                TestClass contains several cref examples.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.#ctor">
            <summary>
                This sample shows how to specify the <see cref="T:TestNamespace.TestClass"/> constructor as a cref attribute.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.#ctor(System.Int32)">
            <summary>
                This sample shows how to specify the <see cref="M:TestNamespace.TestClass.#ctor(System.Int32)"/> constructor as a cref attribute.
            </summary>
        </member>
        <member name="M:TestNamespace.TestClass.GetZero">
            <summary>
                The GetZero method.
            </summary>
            <example>
                This sample shows how to call the <see cref="M:TestNamespace.TestClass.GetZero"/> method.
                <code>
                    class TestClass
                    {
                        static int Main()
                        {
                            return GetZero();
                        }
                    }
                </code>
            </example>
        </member>
        <member name="M:TestNamespace.TestClass.GetGenericValue`1(``0)">
            <summary>
                The GetGenericValue method.
            </summary>
            <remarks>
                This sample shows how to specify the <see cref="M:TestNamespace.TestClass.GetGenericValue`1(``0)"/> method as a cref attribute.
            </remarks>
        </member>
        <member name="T:TestNamespace.GenericClass`1">
            <summary>
                GenericClass.
            </summary>
            <remarks>
                This example shows how to specify the <see cref="T:TestNamespace.GenericClass`1"/> type as a cref attribute.
            </remarks>
        </member>
    </members>
</doc>

```

## 참조

- [XML 문서 주석](#)
- [문서 주석에 대한 권장 태그](#)

# <example>(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

## 구문

```
<example>description</example>
```

## 매개 변수

- `description`

코드 샘플에 대한 설명입니다.

## 설명

<example> 태그를 사용하면 메서드 또는 기타 라이브러리 멤버를 사용하는 방법의 예제를 지정할 수 있습니다.  
여기에는 일반적으로 <code> 태그를 같이 사용합니다.

-doc로 컴파일하여 문서 주석을 파일로 처리합니다.

## 예제

```
// Save this file as CRefTest.cs
// Compile with: csc CRefTest.cs -doc:Results.xml

namespace TestNamespace
{
    /// <summary>
    /// TestClass contains several cref examples.
    /// </summary>
    public class TestClass
    {
        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass"/> constructor as a cref attribute.
        /// </summary>
        public TestClass()
        { }

        /// <summary>
        /// This sample shows how to specify the <see cref="TestClass(int)"/> constructor as a cref
        attribute.
        /// </summary>
        public TestClass(int value)
        { }

        /// <summary>
        /// The GetZero method.
        /// </summary>
        /// <example>
        /// This sample shows how to call the <see cref="GetZero"/> method.
        /// </example>
        /// <code>
        /// class TestClass
        /// {
        ///     static int Main()
        ///     {
        ///         return GetZero();
        ///     }
        /// }
```

```

    ////
    /// }
    /// </code>
    /// </example>
public static int GetZero()
{
    return 0;
}

/// <summary>
/// The GetGenericValue method.
/// </summary>
/// <remarks>
/// This sample shows how to specify the <see cref="GetGenericValue"/> method as a cref attribute.
/// </remarks>

public static T GetGenericValue<T>(T para)
{
    return para;
}

/// <summary>
/// GenericClass.
/// </summary>
/// <remarks>
/// This example shows how to specify the <see cref="GenericClass{T}"/> type as a cref attribute.
/// </remarks>
class GenericClass<T>
{
    // Fields and members.
}

class Program
{
    static int Main()
    {
        return TestClass.GetZero();
    }
}
}

```

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [문서 주석에 대한 권장 태그](#)

# <exception>(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

## 구문

```
<exception cref="member">description</exception>
```

## 매개 변수

- `cref = "member"`

현재 컴파일 환경에서 사용할 수 있는 예외에 대한 참조입니다. 컴파일러는 지정된 예외가 있으며 `member`를 출력 XML의 정식 요소 이름으로 변환하는지 확인합니다. `member`는 큰따옴표(" ")로 묶어야 합니다.

제네릭 형식을 참조하도록 `member` 형식을 지정하는 방법에 대한 자세한 내용은 [XML 파일 처리](#)를 참조하세요.

- `description`

예외에 대한 설명입니다.

## 설명

`<exception>` 태그를 사용하면 `throw`할 수 있는 예외를 지정할 수 있습니다. 이 태그는 메서드, 속성, 이벤트 및 인덱서에 대한 정의에 적용할 수 있습니다.

`-doc`로 컴파일하여 문서 주석을 파일로 처리합니다.

예외 처리에 대한 자세한 내용은 [예외 및 예외 처리](#)를 참조하세요.

## 예제

```
// compile with: -doc:DocFileName.xml

/// Comment for class
public class EClass : System.Exception
{
    // class definition...
}

/// Comment for class
class TestClass
{
    /// <exception cref="System.Exception">Thrown when...</exception>
    public void DoSomething()
    {
        try
        {
        }
        catch (EClass)
        {
        }
    }
}
```

## 참조

- [C# 프로그래밍 가이드](#)
- [문서 주석에 대한 권장 태그](#)

# <include>(C# 프로그래밍 가이드)

2020-11-02 • 4 minutes to read • [Edit Online](#)

## 구문

```
<include file='filename' path='tagpath[@name="id"]' />
```

## 매개 변수

- `filename`

문서가 포함된 XML 파일의 이름입니다. 파일 이름은 소스 코드 파일에 상대 경로로 한정될 수 있습니다. `filename`을 작은따옴표(' ')로 묁습니다.

- `tagpath`

`filename`의 태그 경로로, `name` 태그에 연결됩니다. 경로를 작은따옴표(' ')로 묁습니다.

- `name`

주석 앞에 오는 태그의 이름 지정자입니다. `name`에는 `id`가 있습니다.

- `id`

주석 앞에 오는 태그의 ID입니다. ID를 큰따옴표(" ")로 묁습니다.

## 설명

`<include>` 태그를 사용하면 소스 코드의 형식과 멤버를 설명하는 다른 파일의 주석을 참조할 수 있습니다. 소스 코드 파일에 직접 문서 주석을 포함하는 대신 사용되는 대안입니다. 별도 파일에 문서를 배치하면 소스 코드와 별도로 문서에 소스 제어를 적용할 수 있습니다. 한 사람은 소스 코드 파일을 체크 아웃하고 다른 사람은 문서 파일을 체크 아웃할 수 있습니다.

`<include>` 태그는 XML XPath 구문을 사용합니다. `<include>` 사용자 지정하는 방법은 XPath 설명서를 참조하세요.

## 예제

다중 파일 예제입니다. 다음은 `<include>` 태그를 사용하는 첫 번째 파일입니다.

```
// compile with: -doc:DocFileName.xml

/// <include file='xml_include_tag.doc' path='MyDocs/MyMembers[@name="test"]/*' />
class Test
{
    static void Main()
    {
    }
}

/// <include file='xml_include_tag.doc' path='MyDocs/MyMembers[@name="test2"]/*' />
class Test2
{
    public void Test()
    {
    }
}
```

두 번째 파일인 *xml\_include\_tag.doc*에는 다음과 같은 문서 주석이 포함되어 있습니다.

```
<MyDocs>

<MyMembers name="test">
<summary>
The summary for this type.
</summary>
</MyMembers>

<MyMembers name="test2">
<summary>
The summary for this other type.
</summary>
</MyMembers>

</MyDocs>
```

## 프로그램 출력

다음 명령줄을 사용하여 테스트 및 Test2 클래스를 컴파일하는 경우 다음 출력이 생성됩니다.

`-doc:DocFileName.xml`. Visual Studio에서 프로젝트 디자이너의 빌드 창에 XML 문서 주석 옵션을 지정합니다. C# 컴파일러는 `<include>` 태그를 발견할 경우 현재 소스 파일 대신 *xml\_include\_tag.doc*에서 문서 주석을 검색합니다. 그런 다음 컴파일러는 *DocFileName.xml*을 생성합니다. 이 파일은 [Sandcastle](#) 등의 문서 도구에서 최종 문서를 생성하는 데 사용하는 파일입니다.

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>xml_include_tag</name>
    </assembly>
    <members>
        <member name="T:Test">
            <summary>
The summary for this type.
            </summary>
        </member>
        <member name="T:Test2">
            <summary>
The summary for this other type.
            </summary>
        </member>
    </members>
</doc>
```

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [문서 주석에 대한 권장 태그](#)

# <inheritdoc>(C# 프로그래밍 가이드)

2021-02-18 • 2 minutes to read • [Edit Online](#)

## 구문

```
<inheritdoc/>
```

## InheritDoc

기본 클래스, 인터페이스 및 유사한 메서드에서 XML 주석을 상속합니다. 이렇게 하면 중복 XML 주석을 복사하여 붙여 넣을 필요가 없으며 XML 주석이 자동으로 동기화됩니다.

## 설명

기본 클래스 또는 인터페이스에 XML 주석을 추가하고 InheritDoc가 주석을 구현 클래스에 복사하도록 합니다.

동기 메서드에 XML 주석을 추가하고 InheritDoc가 동일한 메서드의 비동기 버전에 주석을 복사하도록 합니다.

특정 멤버에서 주석을 복사하려면 `cref` 특성을 사용하여 멤버를 지정할 수 있습니다.

## 예

```
// compile with: -doc:DocFileName.xml

/// <summary>
/// You may have some primary information about this class.
/// </summary>
public class MainClass
{
}

///<inheritdoc/>
public class TestClass: MainClass
{}
```

```
// compile with: -doc:DocFileName.xml

/// <summary>
/// You may have some primary information about this interface.
/// </summary>
public interface ITestInterface
{}

///<inheritdoc cref="ITestInterface"/>
public class TestClass : ITestInterface
{}
```

## 참고 항목

- C# 프로그래밍 가이드
- 문서 주석에 대한 권장 태그

# <list>(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

## 구문

```
<list type="bullet|number|table">
  <listheader>
    <term>term</term>
    <description>description</description>
  </listheader>
  <item>
    <term>term</term>
    <description>description</description>
  </item>
</list>
```

## 매개 변수

- `term`  
`description`에서 정의되는, 정의할 용어입니다.
- `description`  
글머리 기호 또는 번호 매기기 목록의 항목이나 `term`의 정의입니다.

## 설명

`<listheader>` 블록은 테이블 또는 정의 목록의 머리를 행을 정의하는 데 사용됩니다. 테이블을 정의할 때는 머리글에 용어 항목만 제공하면 됩니다.

목록의 각 항목은 `<item>` 블록을 사용하여 지정됩니다. 정의 목록을 만들 때는 `term`과 `description`을 모두 지정해야 합니다. 그러나 테이블, 글머리 기호 목록 또는 번호 매기기 목록의 경우 `description` 항목만 제공하면 됩니다.

목록 또는 테이블에 `<item>` 블록을 필요한 개수만큼 포함할 수 있습니다.

-doc로 컴파일하여 문서 주석을 파일로 처리합니다.

## 예제

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>Here is an example of a bulleted list:
    /// <list type="bullet">
    /// <item>
    /// <description>Item 1.</description>
    /// </item>
    /// <item>
    /// <description>Item 2.</description>
    /// </item>
    /// </list>
    /// </summary>
    static void Main()
    {
    }
}
```

## 참조

- [C# 프로그래밍 가이드](#)
- [문서 주석에 대한 권장 태그](#)

# <para>(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

## 구문

```
<para>content</para>
```

## 매개 변수

- `content`

단락의 텍스트입니다.

## 설명

<para> 태그는 <summary>, <remarks> 또는 <returns> 같은 태그 내에 사용되어 텍스트에 구조를 추가할 수 있게 합니다.

-doc로 컴파일하여 문서 주석을 파일로 처리합니다.

## 예제

```
<para> 사용 예제는 <summary> 를 참조하세요.
```

## 참조

- [C# 프로그래밍 가이드](#)
- [문서 주석에 대한 권장 태그](#)

# <param>(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

## 구문

```
<param name="name">description</param>
```

## 매개 변수

- `name`

메서드 매개 변수의 이름입니다. 이름을 큰따옴표(" ")로 둑습니다.

- `description`

매개 변수에 대한 설명입니다.

## 설명

`<param>` 태그는 메서드의 매개 변수 중 하나를 설명하기 위해 메서드 선언에 대한 주석에서 사용해야 합니다. 여러 매개 변수를 문서화하려면 여러 개의 `<param>` 태그를 사용합니다.

`<param>` 태그에 대한 텍스트는 IntelliSense, 개체 브라우저, 코드 주석 웹 보고서에 표시됩니다.

`-doc`로 컴파일하여 문서 주석을 파일로 처리합니다.

## 예제

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    // Single parameter.
    /// <param name="Int1">Used to indicate status.</param>
    public static void DoWork(int Int1)
    {
    }

    // Multiple parameters.
    /// <param name="Int1">Used to indicate status.</param>
    /// <param name="Float1">Used to specify context.</param>
    public static void DoWork(int Int1, float Float1)
    {
    }

    // text for Main
    static void Main()
    {
    }
}
```

## 참조

- C# 프로그래밍 가이드
- 문서 주석에 대한 권장 태그

# <paramref>(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

## 구문

```
<paramref name="name"/>
```

## 매개 변수

- `name`

참조할 매개 변수의 이름입니다. 이름을 큰따옴표(" ")로 둑습니다.

## 설명

`<paramref>` 태그를 사용하면 `<summary>` 또는 `<remarks>` 블록 등의 코드 주석에 포함된 단어가 매개 변수를 참조함을 나타낼 수 있습니다. XML 파일을 처리하여 굵게, 기울임꼴 글꼴 등의 고유한 방식으로 이 단어에 서식을 지정할 수 있습니다.

`-doc`로 컴파일하여 문서 주석을 파일로 처리합니다.

## 예제

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// The <paramref name="int1"/> parameter takes a number.
    /// </summary>
    public static void DoWork(int int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [문서 주석에 대한 권장 태그](#)

# <permission>(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

## 구문

```
<permission cref="member">description</permission>
```

## 매개 변수

- `cref = "member"`

현재 컴파일 환경에서 호출할 수 있는 멤버 또는 필드에 대한 참조입니다. 컴파일러는 지정된 코드 요소가 있으며 `member`를 출력 XML의 정식 요소 이름으로 변환하는지 확인합니다. `member`는 큰따옴표(" ")로 묶어야 합니다.

제네릭 형식에 대한 `cref` 참조를 만드는 방법에 대한 자세한 내용은 [cref 특성](#)을 참조하세요.

- `description`

멤버 액세스 권한에 대한 설명입니다.

## 설명

`<permission>` 태그를 사용하면 멤버 액세스 권한을 문서화할 수 있습니다. [PermissionSet](#) 클래스를 사용하면 멤버 액세스 권한을 지정할 수 있습니다.

`-doc`로 컴파일하여 문서 주석을 파일로 처리합니다.

## 예제

```
// compile with: -doc:DocFileName.xml

class TestClass
{
    /// <permission cref="System.Security.PermissionSet">Everyone can access this method.</permission>
    public static void Test()
    {
    }

    static void Main()
    {
    }
}
```

## 참조

- [C# 프로그래밍 가이드](#)
- [문서 주석에 대한 권장 태그](#)

# <remarks>(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

## 구문

```
<remarks>description</remarks>
```

## 매개 변수

- [Description](#)

멤버에 대한 설명입니다.

## 설명

<remarks> 태그는 형식에 대한 정보를 추가하여 <summary>로 지정된 정보를 보완하기 위해 사용됩니다. 이 정보는 개체 브라우저 창에 표시됩니다.

-doc로 컴파일하여 문서 주석을 파일로 처리합니다.

## 예제

```
// compile with: -doc:DocFileName.xml

/// <summary>
/// You may have some primary information about this class.
/// </summary>
/// <remarks>
/// You may have some additional information about this class.
/// </remarks>
public class TestClass
{
    /// text for Main
    static void Main()
    {
    }
}
```

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [문서 주석에 대한 권장 태그](#)

# <returns>(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

## 구문

```
<returns>description</returns>
```

## 매개 변수

- `description`

반환 값에 대한 설명입니다.

## 설명

`<returns>` 태그는 메서드 선언의 주석에서 반환 값을 설명하는 데 사용해야 합니다.

`-doc`로 컴파일하여 문서 주석을 파일로 처리합니다.

## 예제

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <returns>Returns zero.</returns>
    public static int GetZero()
    {
        return 0;
    }

    /// text for Main
    static void Main()
    {
    }
}
```

## 참조

- [C# 프로그래밍 가이드](#)
- [문서 주석에 대한 권장 태그](#)

# <see>(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

## 구문

```
<see cref="member"/>
```

## 매개 변수

- `cref = "member"`

현재 컴파일 환경에서 호출할 수 있는 멤버 또는 필드에 대한 참조입니다. 컴파일러는 지정된 코드 요소가 있으며 `member`를 출력 XML의 요소 이름에 전달하는지 확인합니다. 멤버는 큰따옴표(" ")로 묶으세요.

## 설명

`<see>` 태그를 사용하면 텍스트 내부에서 링크를 지정할 수 있습니다. 참조 섹션에 텍스트를 배치해야 한다고 지정하려면 `<seealso>`를 사용합니다. 코드 요소의 문서 페이지에 대한 내부 하이퍼링크를 만들려면 `cref` 특성을 사용합니다. 또한 `href`는 하이퍼링크로 작동하는 유효한 특성입니다.

`-doc`로 컴파일하여 문서 주석을 파일로 처리합니다.

다음 예제에서는 요약 섹션의 `<see>` 태그를 보여 줍니다.

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// <para>Here's how you could make a second paragraph in a description. <see
    cref="System.Console.WriteLine(System.String)"> for information about output statements.</para>
    /// <seealso cref="TestClass.Main"/>
    /// </summary>
    public static void DoWork(int Int1)
    {

    /// text for Main
    static void Main()
    {
    }
}
```

## 참조

- [C# 프로그래밍 가이드](#)
- [문서 주석에 대한 권장 태그](#)

# <seealso>(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

## 구문

```
<seealso cref="member"/>
```

## 매개 변수

- `cref = "member"`

현재 컴파일 환경에서 호출할 수 있는 멤버 또는 필드에 대한 참조입니다. 컴파일러는 지정된 코드 요소가 있으며 `member`를 출력 XML의 요소 이름에 전달하는지 확인합니다. `member`는 큰따옴표(" ")로 둘어야 합니다.

제네릭 형식에 대한 cref 참조를 만드는 방법에 대한 자세한 내용은 [cref 특성](#)을 참조하세요.

## 설명

`<seealso>` 태그를 사용하면 참고 항목 섹션에 표시할 텍스트를 지정할 수 있습니다. 텍스트 내에서 링크를 지정하려면 `<see>`를 사용합니다.

`-doc`로 컴파일하여 문서 주석을 파일로 처리합니다.

## 예제

<seealso> 사용 예제는 [<summary>](#)를 참조하세요.

## 참조

- [C# 프로그래밍 가이드](#)
- [문서 주석에 대한 권장 태그](#)

# <summary>(C# 프로그래밍 가이드)

2020-11-02 • 3 minutes to read • [Edit Online](#)

## 구문

```
<summary>description</summary>
```

## 매개 변수

- `description`

개체에 대한 요약입니다.

## 설명

<summary> 태그를 사용하여 형식 또는 형식 멤버를 설명해야 합니다. <remarks>를 사용하여 형식 설명에 보충 정보를 추가할 수 있습니다. `cref 특성`을 사용하면 DocFX 및 Sandcastle 등의 문서화 도구를 통해 코드 요소의 문서화 페이지에 대한 내부 하이퍼링크를 만들 수 있습니다.

<summary> 태그의 텍스트는 IntelliSense의 형식에 대한 유일한 정보 소스이며 개체 브라우저 창에도 표시됩니다.

-doc로 컴파일하여 문서 주석을 파일로 처리합니다. 컴파일러에서 생성한 파일을 기반으로 해서 최종 문서를 만들려면 사용자 지정 도구를 만들거나 DocFX 또는 Sandcastle과 같은 도구를 사용하면 됩니다.

## 예제

```
// compile with: -doc:DocFileName.xml

/// text for class TestClass
public class TestClass
{
    /// <summary>DoWork is a method in the TestClass class.
    /// <para>Here's how you could make a second paragraph in a description. <see
    cref="System.Console.WriteLine(System.String)"/> for information about output statements.</para>
    /// <seealso cref="TestClass.Main"/>
    /// </summary>
    public static void DoWork(int Int1)
    {
    }

    /// text for Main
    static void Main()
    {
    }
}
```

앞의 예제에서는 다음 XML 파일을 생성합니다.

```

<?xml version="1.0"?>
<doc>
  <assembly>
    <name>YourNamespace</name>
  </assembly>
  <members>
    <member name="T:TestClass">
      text for class TestClass
    </member>
    <member name="M:TestClass.DoWork(System.Int32)">
      <summary>DoWork is a method in the TestClass class.
      <para>Here's how you could make a second paragraph in a description. <see
      cref="M:System.Console.WriteLine(System.String)"/> for information about output statements.</para>
    </summary>
    <seealso cref="M:TestClass.Main"/>
    </member>
    <member name="M:TestClass.Main">
      text for Main
    </member>
  </members>
</doc>

```

## 예제

다음 예제에서는 제네릭 형식에 대한 `cref` 참조를 만드는 방법을 보여 줍니다.

```

// compile with: -doc:DocFileName.xml

// the following cref shows how to specify the reference, such that,
// the compiler will resolve the reference
/// <summary cref="C{T}">
/// </summary>
class A { }

// the following cref shows another way to specify the reference,
// such that, the compiler will resolve the reference
// <summary cref="C &lt; T &gt;">

// the following cref shows how to hard-code the reference
/// <summary cref="T:C`1">
/// </summary>
class B { }

/// <summary cref="A">
/// </summary>
/// <typeparam name="T"></typeparam>
class C<T> { }

class Program
{
  static void Main() { }
}

```

앞의 예제에서는 다음 XML 파일을 생성합니다.

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>CRefTest</name>
  </assembly>
  <members>
    <member name="T:A">
      <summary cref="T:C`1">
      </summary>
    </member>
    <member name="T:B">
      <summary cref="T:C`1">
      </summary>
    </member>
    <member name="T:C`1">
      <summary cref="T:A">
      </summary>
      <typeparam name="T"></typeparam>
    </member>
  </members>
</doc>
```

## 참조

- [C# 프로그래밍 가이드](#)
- [문서 주석에 대한 권장 태그](#)

# <typeparam>(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

## 구문

```
<typeparam name="name">description</typeparam>
```

## 매개 변수

- `name`

형식 매개 변수의 이름입니다. 이름을 큰따옴표(" ")로 묶습니다.

- `description`

형식 매개 변수에 대한 설명입니다.

## 설명

<typeparam> 태그는 제네릭 형식 또는 메서드 선언의 주석에서 형식 매개 변수를 설명하는 데 사용해야 합니다. 제네릭 형식 또는 메서드의 각 형식 매개 변수에 대한 태그를 추가합니다.

자세한 내용은 [제네릭](#)을 참조하세요.

<typeparam> 태그에 대한 텍스트는 IntelliSense와 [개체 브라우저 창](#)의 코드 주석 웹 보고서에 표시됩니다.

-doc로 컴파일하여 문서 주석을 파일로 처리합니다.

## 예제

```
// compile with: -doc:DocFileName.xml

/// comment for class
public class TestClass
{
    /// <summary>
    /// Creates a new array of arbitrary type <typeparamref name="T"/>
    /// </summary>
    /// <typeparam name="T">The element type of the array</typeparam>
    public static T[] mkArray<T>(int n)
    {
        return new T[n];
    }
}
```

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [문서 주석에 대한 권장 태그](#)

# <typeparamref>(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

## 구문

```
<typeparamref name="name"/>
```

## 매개 변수

- `name`

형식 매개 변수의 이름입니다. 이름을 큰따옴표(" ")로 묶습니다.

## 설명

제네릭 형식 및 메서드의 형식 매개 변수에 대한 자세한 내용은 [제네릭](#)을 참조하세요.

이 태그를 사용하면 문서 파일의 소비자가 기울임꼴 등 다른 고유한 방식으로 단어의 서식을 지정할 수 있습니다.

[-doc](#)로 컴파일하여 문서 주석을 파일로 처리합니다.

## 예제

```
// compile with: -doc:DocFileName.xml

/// comment for class
public class TestClass
{
    /// <summary>
    /// Creates a new array of arbitrary type <typeparamref name="T"/>
    /// </summary>
    /// <typeparam name="T">The element type of the array</typeparam>
    public static T[] mkArray<T>(int n)
    {
        return new T[n];
    }
}
```

## 참조

- [C# 프로그래밍 가이드](#)
- [문서 주석에 대한 권장 태그](#)

# <value>(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

## 구문

```
<value>property-description</value>
```

## 매개 변수

- `property-description`

속성에 대한 설명입니다.

## 설명

`<value>` 태그를 사용하면 속성이 나타내는 값을 설명할 수 있습니다. Visual Studio .NET 개발 환경에서 코드 마법사를 통해 속성을 추가하면 새 속성에 대해 `<summary>` 태그가 추가됩니다. 그런 다음, `<value>` 태그를 수동으로 추가하여 속성이 나타내는 값을 설명해야 합니다.

`-doc`로 컴파일하여 문서 주석을 파일로 처리합니다.

## 예제

```
// compile with: -doc:DocFileName.xml

/// text for class Employee
public class Employee
{
    private string _name;

    /// <summary>The Name property represents the employee's name.</summary>
    /// <value>The Name property gets/sets the value of the string field, _name.</value>

    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }
}

/// text for class MainClass
public class MainClass
{
    /// text for Main
    static void Main()
    {
    }
}
```

## 참조

- [C# 프로그래밍 가이드](#)
- [문서 주석에 대한 권장 태그](#)

# 예외 및 예외 처리(C# 프로그래밍 가이드)

2021-02-18 • 7 minutes to read • [Edit Online](#)

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 찾은 다음 이러한 `catch` 블록을 먼저 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.

```
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 catch하지 마세요. `System.Exception` 을 catch하는 경우 `catch` 블록 끝에서 `throw` 키워드를 사용하여 다시 throw하세요.
- `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 throw되더라도 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 [예외](#)를 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

## 참조

- [SystemException](#)
- [C# 키워드](#)
- [throw](#)
- [try-catch](#)
- [try-finally](#)
- [try-catch-finally](#)
- [예외](#)

# 예외 사용(C# 프로그래밍 가이드)

2021-02-18 • 8 minutes to read • [Edit Online](#)

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. 예를 들어:

```
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. 예를 들어:

```
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 호출되는 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 호출되더라도 무시됩니다. catch 블록은 항상 가장 구체적인 것(또는 가장 많이 파생된 것)에서 가장 덜 구체적인 것 순으로 정렬해야 합니다. 예를 들어:

```
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 정리하거나, 런타임 시 가비지 수집기를 기다리지 않은 채 외부 리소스(예: 그래픽 핸들, 데이터베이스 연결 또는 파일 스트림)를 해제하여 개체를 마무리할 수 있습니다. 예를 들어:

```

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(0xFF);
    }
    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 호출되는 `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.

# 예외 처리(C# 프로그래밍 가이드)

2021-02-18 • 11 minutes to read • [Edit Online](#)

`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 `throw`되는지 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.

```
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.
}
```

```
try
{
    // Code to try goes here.
}
finally
{
    // Code to execute after the try block goes here.
}
```

```
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 필터`라고 합니다. 예외 형식은 `Exception`에서 파생되어야 합니다. 일반적으로 `try` 블록에서 `throw`될 수 있는 모든 예외를 처리하는 방법을 알고 있거나 `catch` 블록의 끝에 `throw` 문을 포함한 경우가 아니라면 `Exception`을 예외 필터로 지정하지 마세요.

다른 예외 클래스를 사용하는 여러 `catch` 블록을 함께 연결할 수 있습니다. `catch` 블록은 코드의 위에서 아래

로 계산되지만 throw되는 각 예외에 대해 하나의 `catch` 블록만 실행됩니다. throw된 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 catch합니다.

- 예외가 throw되는 이유를 충분히 이해하고 있으면 `FileNotFoundException` 개체를 catch할 때 새 파일 이름을 입력하라는 메시지를 사용자에게 표시하는 경우처럼 구체적인 복구를 구현할 수 있습니다.
- 보다 구체적인 새 예외를 생성하고 throw할 수 있습니다.

```
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 부분적으로 처리하려고 합니다. 다음 예제에서 `catch` 블록은 예외를 다시 throw하기 전에 오류 로그에 항목을 추가하는 데 사용됩니다.

```
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. 이렇게 하면 해당 조건이 true인 경우에만 특정 `catch` 절이 일치합니다. 다음 예제에서는 두 `catch` 절에서 같은 예외 클래스를 사용하지만 추가 조건을 확인하여 다른 오류 메시지를 생성합니다.

```
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.

```
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. 자세한 내용은 [using 문](#)을 참조하세요.

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.

```
FileStream? file = null;
FileInfo fileinfo = new System.IO.FileInfo("./file.txt");
try
{
    file = fileinfo.OpenWrite();
    file.WriteByte(0xF);
}
finally
{
    // Check for null because OpenWrite might have failed.
    file?.Close();
}
```

## C# 언어 사양

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

## 참조

- C# 참조
- try-catch
- try-finally
- try-catch-finally
- using 문

# 예외 만들기 및 Throw(C# 프로그래밍 가이드)

2021-02-18 • 9 minutes to read • [Edit Online](#)

예외는 프로그램을 실행하는 동안 오류가 발생했음을 나타내는 데 사용됩니다. 오류를 설명하는 예외 개체가 만들어지고 `throw` 키워드를 통해 `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.

```
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.

```
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.

```
static int GetValueFromArray(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException ex)
    {
        throw new ArgumentException("Index is out of range", nameof(index), ex);
    }
}
```

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 개체는 throw 문의 지점에서 CLR(공용 언어 런타임)에 의해 자동으로 만들어지므로 해당 예외는 스택 추적이 시작되는 지점에서 throw되어야 합니다.

모든 예외에 이름이 Message인 속성이 포함되어 있습니다. 예외의 이유를 설명하려면 이 문자열을 설정해야 합니다. 보안이 중요한 정보는 메시지 텍스트에 넣으면 안 됩니다. Message 외에 ArgumentException에는 예외를 throw한 인수의 이름으로 설정해야 하는 ParamName 속성이 포함되어 있습니다. 속성 setter에서는 ParamName을 value로 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.IndexOutOfRangeException를 throw하지 마세요.
- 릴리스 모드가 아닌 디버그 모드에서 throw될 수 있는 예외를 만들지 마세요. 개발 단계에서 런타임 오류를 식별 하려면 대신 디버그 어설션을 사용하세요.

## 예외 클래스 정의

프로그램에서는 System 네임스페이스의 미리 정의된 예외 클래스를 throw하거나(이전에 언급한 위치 제외) Exception에서 파생시켜 자체 예외 클래스를 만들 수 있습니다. 파생 클래스는 매개 변수 없는 생성자, 메시지 속성을 설정하는 생성자, Message 및 InnerException 속성을 설정하는 생성자 두 개를 포함하여 네 개 이상의 생성자를 정의해야 합니다. 네 번째 생성자는 예외를 serialize하는 데 사용됩니다. 새 예외 클래스는 serialize할 수 있어야 합니다. 예를 들어:

```
[Serializable]
public class InvalidDepartmentException : Exception
{
    public InvalidDepartmentException() : base() { }
    public InvalidDepartmentException(string message) : base(message) { }
    public InvalidDepartmentException(string message, Exception inner) : base(message, inner) { }

    // A constructor is needed for serialization when an
    // exception propagates from a remoting server to the client.
    protected InvalidDepartmentException(System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context) : base(info, context) { }
}
```

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

# 컴파일러 생성 예외(C# 프로그래밍 가이드)

2021-02-18 • 3 minutes to read • [Edit Online](#)

기본 작업이 실패하면 .NET 런타임에 의해 자동으로 일부 예외가 throw됩니다. 이러한 예외와 관련 오류 조건이 다음 표에 나와 있습니다.

예외	설명
<a href="#">ArithmeticException</a>	<a href="#">DivideByZeroException</a> , <a href="#">OverflowException</a> 등의 산술 연산 중에 발생하는 예외에 대한 기본 클래스입니다.
<a href="#">ArrayTypeMismatchException</a>	제공된 요소의 실제 형식이 배열의 실제 형식과 호환되지 않아 배열이 요소를 저장할 수 없는 경우 throw됩니다.
<a href="#">DivideByZeroException</a>	정수 값을 0으로 나누려고 시도할 경우 throw됩니다.
<a href="#">IndexOutOfRangeException</a>	인덱스가 0보다 작거나 배열 경계를 벗어날 때 배열을 인덱싱하려고 시도할 경우 throw됩니다.
<a href="#">InvalidCastException</a>	기본 형식에서 인터페이스 또는 파생 형식으로의 명시적 변환이 런타임에 실패할 경우 throw됩니다.
<a href="#">NullReferenceException</a>	값이 <code>null</code> 인 개체를 참조하려고 시도할 경우 throw됩니다.
<a href="#">OutOfMemoryException</a>	<code>new</code> 연산자를 사용한 메모리 할당 시도가 실패할 경우 throw됩니다. 이 예외는 공용 언어 런타임에 사용할 수 있는 메모리가 모두 사용되었음을 나타냅니다.
<a href="#">OverflowException</a>	<code>checked</code> 컨텍스트의 산술 연산이 오버플로될 경우 throw됩니다.
<a href="#">StackOverflowException</a>	보류 종인 메서드 호출이 너무 많아 실행 스택이 모두 사용될 경우 throw됩니다. 대개 매우 깊은 재귀나 무한 재귀를 나타냅니다.
<a href="#">TypeInitializationException</a>	정적 생성자가 예외를 throw하고 이 예외를 catch할 수 있는 호환되는 <code>catch</code> 절이 없는 경우 throw됩니다.

## 참조

- [try-catch](#)
- [try-finally](#)
- [try-catch-finally](#)

# try-catch를 사용하여 예외를 처리하는 방법(C# 프로그래밍 가이드)

2021-02-18 • 2 minutes to read • [Edit Online](#)

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.

## 예제

이 예제에서 `IndexOutOfRangeException`은 가장 적합한 예외가 아닙니다. 호출자가 전달한 `index` 인수로 인해 오류가 발생하기 때문에 `ArgumentOutOfRangeException`이 메서드에 더 적합합니다.

```
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 ArgumentOutOfRangeException("index parameter is out of range.", e);
    }
}
```

## 설명

예외가 발생하는 코드는 try 블록으로 묶여 있습니다. `IndexOutOfRangeException`이 발생할 경우 처리하기 위해 catch 문이 바로 뒤에 추가됩니다. catch 블록은 `IndexOutOfRangeException`을 처리하고 더 적합한 `ArgumentOutOfRangeException` 예외를 대신 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.

# finally를 사용하여 정리 코드를 실행하는 방법(C# 프로그래밍 가이드)

2021-02-18 • 3 minutes to read • [Edit Online](#)

`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.

```
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.

```
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` 개체가 `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

throw되는 경우에도 `finally` 블록 사용이 가비지 수집을 기다리는 것보다 더 낫습니다.

## 참고 항목

- [using 문](#)
- [try-catch](#)
- [try-finally](#)
- [try-catch-finally](#)

# CLS 규격이 아닌 예외를 catch하는 방법

2020-11-02 • 4 minutes to read • [Edit Online](#)

C++/CLI를 포함한 일부 .NET 언어에서는 개체가 [Exception](#)에서 파생되지 않은 예외를 throw할 수 있습니다. 이러한 예외를 *CLS 규격이 아닌 예외* 또는 *예외가 아닌 항목*이라고 합니다. C#에서는 CLS 규격이 아닌 예외를 throw할 수 없지만 다음 두 가지 방법으로 해당 예외를 catch할 수 있습니다.

- `catch (RuntimeWrappedException e)` 블록 내에서.

기본적으로 Visual C# 어셈블리는 CLS 규격이 아닌 예외를 래핑된 예외로 catch합니다.

`RuntimeWrappedException.WrappedException` 속성을 통해 액세스할 수 있는 원래 예외에 액세스해야 할 경우 이 메서드를 사용합니다. 이 항목의 뒤에 나오는 절차에서는 이 방식으로 예외를 catch하는 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.

## 예제

다음 예제에서는 C++/CLI로 작성된 클래스 라이브러리에서 throw된 CLS 규격이 아닌 예외를 catch하는 방법을 보여 줍니다. 이 예제에서 C# 클라이언트 코드는 throw되는 예외 형식이 `System.String`이라는 것을 사전에 알고 있습니다. 코드에서 해당 형식에 액세스할 수 있으면 `RuntimeWrappedException.WrappedException` 속성을 다시 원래 형식으로 캐스팅할 수 있습니다.

```
// 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);
    }
}
```

## 참조

- [RuntimeWrappedException](#)

- 예외 및 예외 처리

# 파일 시스템 및 레지스트리(C# 프로그래밍 가이드)

2020-11-02 • 3 minutes to read • [Edit Online](#)

다음 문서들은 C# 및 .NET를 사용하여 파일, 폴더 및 레지스트리에서 다양한 기본 작업을 수행하는 방법을 보여 줍니다.

## 단원 내용

제목	설명
<a href="#">디렉터리 트리를 반복하는 방법</a>	디렉터리 트리를 수동으로 반복하는 방법을 보여 줍니다.
<a href="#">파일, 폴더 및 드라이브에 대한 정보를 가져오는 방법</a>	파일, 폴더 및 드라이브에 대한 생성 시간 및 크기와 같은 정보를 검색하는 방법을 보여 줍니다.
<a href="#">파일 또는 폴더를 만드는 방법</a>	새 파일 또는 폴더를 만드는 방법을 보여 줍니다.
<a href="#">파일 및 폴더를 복사, 삭제, 이동하는 방법(C# 프로그래밍 가이드)</a>	파일 및 폴더를 복사, 삭제 및 이동하는 방법을 보여 줍니다.
<a href="#">파일 작업에 진행률 대화 상자를 제공하는 방법</a>	특정 파일 작업에 대한 표준 Windows 진행률 대화 상자를 표시하는 방법을 보여 줍니다.
<a href="#">텍스트 파일에 쓰는 방법</a>	텍스트 파일에 쓰는 방법을 보여 줍니다.
<a href="#">텍스트 파일에서 읽는 방법</a>	텍스트 파일에서 읽는 방법을 보여 줍니다.
<a href="#">텍스트 파일을 한 번에 한 줄씩 읽는 방법</a>	한 번에 하나씩 파일에서 텍스트를 검색하는 방법을 보여 줍니다.
<a href="#">레지스트리에 키를 만드는 방법</a>	시스템 레지스트리에 키를 작성하는 방법을 보여 줍니다.

## 관련 단원

- [파일 및 스트림 I/O](#)
- [파일 및 폴더를 복사, 삭제, 이동하는 방법\(C# 프로그래밍 가이드\)](#)
- [C# 프로그래밍 가이드](#)
- [System.IO](#)

# 디렉터리 트리를 반복하는 방법(C# 프로그래밍 가이드)

2020-11-02 • 14 minutes to read • [Edit Online](#)

"디렉터리 트리 반복" 구는 지정된 루트 폴더 아래의 임의 깊이까지 중첩된 각 하위 디렉터리에 있는 각 파일에 대한 액세스를 의미합니다. 반드시 각 파일을 열 필요는 없습니다. 단순히 파일 또는 하위 디렉터리의 이름을 `string`으로 검색하거나, `System.IO.FileInfo` 또는 `System.IO.DirectoryInfo` 개체의 형태로 추가 정보를 검색할 수 있습니다.

## NOTE

Windows에서 "디렉터리" 및 "폴더" 용어는 같은 의미로 사용됩니다. 대부분의 설명서와 사용자 인터페이스 텍스트는 "폴더"라는 용어를 사용하지만 .NET 클래스 라이브러리는 "디렉터리"라는 용어를 사용합니다.

지정된 루트 아래의 모든 디렉터리에 대해 확실히 액세스 권한이 있는 가장 간단한 경우에는 `System.IO.SearchOption.AllDirectories` 플래그를 사용할 수 있습니다. 이 플래그는 지정된 패턴과 일치하는 모든 중첩된 하위 디렉터리를 반환합니다. 다음 예제에서는 이 플래그를 사용하는 방법을 보여 줍니다.

```
root.GetDirectories("*.*", System.IO.SearchOption.AllDirectories);
```

이 방식의 취약성은 지정된 루트 아래의 하위 디렉터리 중 하나에서 `DirectoryNotFoundException` 또는 `UnauthorizedAccessException`이 발생할 경우 전체 메서드가 실패하고 디렉터리가 반환되지 않습니다. `GetFiles` 메서드를 사용하는 경우도 마찬가지입니다. 특정 하위 폴더에서 이러한 예외를 처리해야 하는 경우 다음 예제와 같이 디렉터리 트리를 수동으로 탐색해야 합니다.

디렉터리 트리를 수동으로 탐색하는 경우 하위 디렉터리를 먼저 처리하거나(전위 순회), 파일을 먼저 처리(후위 순회)할 수 있습니다. 전위 순회를 수행하는 경우 해당 폴더 자체에 바로 있는 파일을 반복하기 전에 현재 폴더 아래의 전체 트리를 탐색합니다. 이 문서의 뒷부분에 나오는 예제에서는 후위 순회를 수행하지만 전위 순회를 수행하도록 쉽게 수정할 수 있습니다.

또 다른 옵션은 재귀 또는 스택 기반 탐색을 사용하는지 여부입니다. 이 문서의 뒷부분에 나오는 예제에서는 두 가지 방식을 모두 보여 줍니다.

파일 및 폴더에 대해 다양한 작업을 수행해야 하는 경우 단일 대리자를 통해 호출할 수 있는 별도 함수로 작업을 리팩터링하여 이러한 예제를 모듈화할 수 있습니다.

## NOTE

NTFS 파일 시스템에는 재분석 지점이 연결 지점, 기호 링크 및 하드 링크 형태로 포함될 수 있습니다. `GetFiles`, `GetDirectories` 등의 .NET 메서드는 재분석 지점 아래의 하위 디렉터리를 반환하지 않습니다. 이 동작은 두 재분석 지점이 서로를 가리키는 경우 무한 루프로 전환되는 위험으로부터 보호합니다. 일반적으로 재분석 지점을 다룰 때는 파일이 실수로 수정되거나 삭제되지 않도록 특별히 주의해야 합니다. 재분석 지점을 정밀하게 제어해야 하는 경우 플랫폼 호출 또는 네이티브 코드를 사용하여 적절한 Win32 파일 시스템 메서드를 직접 호출합니다.

## 예제

다음 예제에서는 재귀를 사용하여 디렉터리 트리를 탐색하는 방법을 보여 줍니다. 재귀 방식은 세련된 방식이긴 하지만 디렉터리 트리가 크고 깊이 중첩된 경우 스택 오버플로 예외가 발생할 가능성이 있습니다.

처리되는 특정 예외와 각 파일 또는 폴더에서 수행되는 특정 작업은 예로만 제공됩니다. 특정 요구 사항에 맞게 이 코드를 수정해야 합니다. 자세한 내용은 코드 주석을 참조하세요.

```
public class RecursiveFileSearch
{
    static System.Collections.Specialized.StringCollection log = new
System.Collections.Specialized.StringCollection();

    static void Main()
    {
        // Start with drives if you have to search the entire computer.
        string[] drives = System.Environment.GetLogicalDrives();

        foreach (string dr in drives)
        {
            System.IO.DriveInfo di = new System.IO.DriveInfo(dr);

            // Here we skip the drive if it is not ready to be read. This
            // is not necessarily the appropriate action in all scenarios.
            if (!di.IsReady)
            {
                Console.WriteLine("The drive {0} could not be read", di.Name);
                continue;
            }
            System.IO.DirectoryInfo rootDir = di.RootDirectory;
            WalkDirectoryTree(rootDir);
        }

        // Write out all the files that could not be processed.
        Console.WriteLine("Files with restricted access:");
        foreach (string s in log)
        {
            Console.WriteLine(s);
        }
        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key");
        Console.ReadKey();
    }

    static void WalkDirectoryTree(System.IO.DirectoryInfo root)
    {
        System.IO.FileInfo[] files = null;
        System.IO.DirectoryInfo[] subDirs = null;

        // First, process all the files directly under this folder
        try
        {
            files = root.GetFiles("*.*");
        }
        // This is thrown if even one of the files requires permissions greater
        // than the application provides.
        catch (UnauthorizedAccessException e)
        {
            // This code just writes out the message and continues to recurse.
            // You may decide to do something different here. For example, you
            // can try to elevate your privileges and access the file again.
            log.Add(e.Message);
        }

        catch (System.IO.DirectoryNotFoundException e)
        {
            Console.WriteLine(e.Message);
        }

        if (files != null)
        {
            foreach (System.IO.FileInfo fi in files)
            {

```

```

    {
        // In this example, we only access the existing FileInfo object. If we
        // want to open, delete or modify the file, then
        // a try-catch block is required here to handle the case
        // where the file has been deleted since the call to TraverseTree().
        Console.WriteLine(fi.FullName);
    }

    // Now find all the subdirectories under this directory.
    subDirs = root.GetDirectories();

    foreach (System.IO.DirectoryInfo dirInfo in subDirs)
    {
        // Recursive call for each subdirectory.
        WalkDirectoryTree(dirInfo);
    }
}
}
}

```

## 예제

다음 예제에서는 재귀를 사용하지 않고 디렉터리 트리의 파일 및 폴더를 반복하는 방법을 보여 줍니다. 이 기술은 LIFO(후입선출) 스택인 제네릭 `Stack<T>` 컬렉션 형식을 사용합니다.

처리되는 특정 예외와 각 파일 또는 폴더에서 수행되는 특정 작업은 예로만 제공됩니다. 특정 요구 사항에 맞게 이 코드를 수정해야 합니다. 자세한 내용은 코드 주석을 참조하세요.

```

public class StackBasedIteration
{
    static void Main(string[] args)
    {
        // Specify the starting folder on the command line, or in
        // Visual Studio in the Project > Properties > Debug pane.
        TraverseTree(args[0]);

        Console.WriteLine("Press any key");
        Console.ReadKey();
    }

    public static void TraverseTree(string root)
    {
        // Data structure to hold names of subfolders to be
        // examined for files.
        Stack<string> dirs = new Stack<string>(20);

        if (!System.IO.Directory.Exists(root))
        {
            throw new ArgumentException();
        }
        dirs.Push(root);

        while (dirs.Count > 0)
        {
            string currentDir = dirs.Pop();
            string[] subDirs;
            try
            {
                subDirs = System.IO.Directory.GetDirectories(currentDir);
            }
            // An UnauthorizedAccessException exception will be thrown if we do not have
            // discovery permission on a folder or file. It may or may not be acceptable
            // to ignore the exception and continue enumerating the remaining files and
            // folders. It is also possible (but unlikely) that a DirectoryNotFoundException
            // will be raised. This will happen if currentDir has been deleted by
            // another application or thread after our call to Directory.Exists. The
        }
    }
}

```

```

        // choice of which exceptions to catch depends entirely on the specific task
        // you are intending to perform and also on how much you know with certainty
        // about the systems on which this code will run.
        catch (UnauthorizedAccessException e)
        {
            Console.WriteLine(e.Message);
            continue;
        }
        catch (System.IO.DirectoryNotFoundException e)
        {
            Console.WriteLine(e.Message);
            continue;
        }

        string[] files = null;
        try
        {
            files = System.IO.Directory.GetFiles(currentDir);
        }

        catch (UnauthorizedAccessException e)
        {
            Console.WriteLine(e.Message);
            continue;
        }

        catch (System.IO.DirectoryNotFoundException e)
        {
            Console.WriteLine(e.Message);
            continue;
        }
        // Perform the required action on each file here.
        // Modify this block to perform your required task.
        foreach (string file in files)
        {
            try
            {
                // Perform whatever action is required in your scenario.
                System.IO.FileInfo fi = new System.IO.FileInfo(file);
                Console.WriteLine("{0}: {1}, {2}", fi.Name, fi.Length, fi.CreationTime);
            }
            catch (System.IO.FileNotFoundException e)
            {
                // If file was deleted by a separate application
                // or thread since the call to TraverseTree()
                // then just continue.
                Console.WriteLine(e.Message);
                continue;
            }
        }

        // Push the subdirectories onto the stack for traversal.
        // This could also be done before handing the files.
        foreach (string str in subDirs)
            dirs.Push(str);
    }
}

```

일반적으로 모든 폴더를 테스트하여 애플리케이션에 폴더를 열 수 있는 권한이 있는지 확인하려면 너무 많은 시간이 걸립니다. 따라서 코드 예제에서는 해당 작업 부분을 `try/catch` 블록으로 묶습니다. 폴더에 대한 액세스가 거부될 경우 사용 권한을 높인 후 다시 액세스를 시도하도록 `catch` 블록을 수정할 수 있습니다. 일반적으로 애플리케이션을 알 수 없는 상태로 유지하지 않고 처리할 수 있는 예외만 `catch`합니다.

디렉터리 트리의 내용을 메모리 내 또는 디스크에 저장해야 하는 경우 최상의 옵션은 각 파일의 `FullName` 속성(

`string` 형식)만 저장하는 것입니다. 그런 다음 이 문자열을 사용하여 필요에 따라 [FileInfo](#) 또는 [DirectoryInfo](#) 개체를 새로 만들거나, 추가 처리가 필요한 파일을 열 수 있습니다.

## 강력한 프로그래밍

강력한 파일 반복 코드는 파일 시스템의 여러 복잡성을 고려해야 합니다. Windows 파일 시스템에 대한 자세한 내용은 [NTFS 개요](#)를 참조하세요.

## 참조

- [System.IO](#)
- [LINQ 및 파일 딕렉터리](#)
- [파일 시스템 및 레지스트리\(C# 프로그래밍 가이드\)](#)

# 파일, 폴더 및 드라이브에 대한 정보를 가져오는 방법(C# 프로그래밍 가이드)

2020-11-02 • 5 minutes to read • [Edit Online](#)

.NET에서 다음 클래스를 사용하여 파일 시스템 정보에 액세스할 수 있습니다.

- [System.IO.FileInfo](#)
- [System.IO.DirectoryInfo](#)
- [System.IO.DriveInfo](#)
- [System.IO.Directory](#)
- [System.IO.File](#)

[FileInfo](#) 및 [DirectoryInfo](#) 클래스는 파일 또는 디렉터리를 나타내며, NTFS 파일 시스템에서 지원되는 많은 파일 특성을 노출하는 속성을 포함합니다. 또한 파일 및 폴더를 열고, 닫고, 이동, 삭제하기 위한 메서드도 포함합니다. 파일, 폴더 또는 드라이브의 이름을 나타내는 문자열을 생성자에 전달하여 이러한 클래스의 인스턴스를 만들 수 있습니다.

```
System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
```

[DirectoryInfo.GetDirectories](#), [DirectoryInfo.GetFiles](#), [DriveInfo.RootDirectory](#) 호출을 사용하여 파일, 폴더 또는 드라이브의 이름을 가져올 수도 있습니다.

[System.IO.Directory](#) 및 [System.IO.File](#) 클래스는 디렉터리와 파일에 대한 정보를 가져오기 위한 정적 메서드를 제공합니다.

## 예제

다음 예제에서는 파일 및 폴더에 대한 정보에 액세스하는 다양한 방법을 보여 줍니다.

```
class FileSysInfo
{
    static void Main()
    {
        // You can also use System.Environment.GetLogicalDrives to
        // obtain names of all logical drives on the computer.
        System.IO.DriveInfo di = new System.IO.DriveInfo(@"C:\");
        Console.WriteLine(di.TotalFreeSpace);
        Console.WriteLine(di.VolumeLabel);

        // Get the root directory and print out some information about it.
        System.IO.DirectoryInfo dirInfo = di.RootDirectory;
        Console.WriteLine(dirInfo.Attributes.ToString());

        // Get the files in the directory and print out some information about them.
        System.IO.FileInfo[] fileNames = dirInfo.GetFiles("*.*");

        foreach (System.IO.FileInfo fi in fileNames)
        {
            Console.WriteLine("{0}: {1}: {2}", fi.Name, fi.LastAccessTime, fi.Length);
        }

        // Get the subdirectories directly that is under the root.
    }
}
```

```

// See "How to: Iterate Through a Directory Tree" for an example of how to
// iterate through an entire tree.
System.IO.DirectoryInfo[] dirInfos = dirInfo.GetDirectories(".*");

foreach (System.IO.DirectoryInfo d in dirInfos)
{
    Console.WriteLine(d.Name);
}

// The Directory and File classes provide several static methods
// for accessing files and directories.

// Get the current application directory.
string currentDirName = System.IO.Directory.GetCurrentDirectory();
Console.WriteLine(currentDirName);

// Get an array of file names as strings rather than FileInfo objects.
// Use this method when storage space is an issue, and when you might
// hold on to the file name reference for a while before you try to access
// the file.
string[] files = System.IO.Directory.GetFiles(currentDirName, "*.txt");

foreach (string s in files)
{
    // Create the FileInfo object only when needed to ensure
    // the information is as current as possible.
    System.IO.FileInfo fi = null;
    try
    {
        fi = new System.IO.FileInfo(s);
    }
    catch (System.IO.FileNotFoundException e)
    {
        // To inform the user and continue is
        // sufficient for this demonstration.
        // Your application may require different behavior.
        Console.WriteLine(e.Message);
        continue;
    }
    Console.WriteLine("{0} : {1}", fi.Name, fi.Directory);
}

// Change the directory. In this case, first check to see
// whether it already exists, and create it if it does not.
// If this is not appropriate for your application, you can
// handle the System.IO.IOException that will be raised if the
// directory cannot be found.
if (!System.IO.Directory.Exists(@"C:\Users\Public\TestFolder\")) {
    System.IO.Directory.CreateDirectory(@"C:\Users\Public\TestFolder\");

    System.IO.Directory.SetCurrentDirectory(@"C:\Users\Public\TestFolder\");

    currentDirName = System.IO.Directory.GetCurrentDirectory();
    Console.WriteLine(currentDirName);

    // Keep the console window open in debug mode.
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
}

```

## 강력한 프로그래밍

사용자 지정 경로 문자열을 처리하는 경우 다음 조건에 대한 예외도 처리해야 합니다.

- 파일 이름 형식이 잘못된 경우. 예를 들어 잘못된 문자를 포함하거나 공백만 포함합니다.
- 파일 이름이 null인 경우
- 파일 이름이 시스템에 정의된 최대 길이보다 긴 경우
- 파일 이름에 콜론(:)이 포함된 경우

애플리케이션에 지정된 파일을 읽을 수 있는 권한이 없는 경우 `Exists` 메서드는 경로가 있는지 여부에 관계없이 `false`를 반환합니다. 이 메서드는 예외를 throw하지 않습니다.

## 참조

- [System.IO](#)
- [C# 프로그래밍 가이드](#)
- [파일 시스템 및 레지스트리\(C# 프로그래밍 가이드\)](#)

# 파일 또는 폴더를 만드는 방법(C# 프로그래밍 가이드)

2020-11-02 • 6 minutes to read • [Edit Online](#)

프로그래밍 방식으로 컴퓨터에 폴더를 만들고, 하위 폴더를 만들고, 하위 폴더에 파일을 만들고, 파일에 데이터를 쓸 수 있습니다.

## 예제

```
public class CreateFileOrFolder
{
    static void Main()
    {
        // Specify a name for your top-level folder.
        string folderName = @"c:\Top-Level Folder";

        // To create a string that specifies the path to a subfolder under your
        // top-level folder, add a name for the subfolder to folderName.
        string pathString = System.IO.Path.Combine(folderName, "SubFolder");

        // You can write out the path name directly instead of using the Combine
        // method. Combine just makes the process easier.
        string pathString2 = @"c:\Top-Level Folder\SubFolder2";

        // You can extend the depth of your path if you want to.
        //pathString = System.IO.Path.Combine(pathString, "SubSubFolder");

        // Create the subfolder. You can verify in File Explorer that you have this
        // structure in the C: drive.
        //    Local Disk (C:)
        //        Top-Level Folder
        //            SubFolder
        System.IO.Directory.CreateDirectory(pathString);

        // Create a file name for the file you want to create.
        string fileName = System.IO.Path.GetRandomFileName();

        // This example uses a random string for the name, but you also can specify
        // a particular name.
        //string fileName = "MyNewFile.txt";

        // Use Combine again to add the file name to the path.
        pathString = System.IO.Path.Combine(pathString, fileName);

        // Verify the path that you have constructed.
        Console.WriteLine("Path to my file: {0}\n", pathString);

        // Check that the file doesn't already exist. If it doesn't exist, create
        // the file and write integers 0 - 99 to it.
        // DANGER: System.IO.File.Create will overwrite the file if it already exists.
        // This could happen even with random file names, although it is unlikely.
        if (!System.IO.File.Exists(pathString))
        {
            using (System.IO.FileStream fs = System.IO.File.Create(pathString))
            {
                for (byte i = 0; i < 100; i++)
                {
                    fs.WriteByte(i);
                }
            }
        }
    }
}
```

```

        }

    else
    {
        Console.WriteLine("File \\"{0}" already exists.", fileName);
        return;
    }

    // Read and display the data from your file.
    try
    {
        byte[] readBuffer = System.IO.File.ReadAllBytes(pathString);
        foreach (byte b in readBuffer)
        {
            Console.Write(b + " ");
        }
        Console.WriteLine();
    }
    catch (System.IO.IOException e)
    {
        Console.WriteLine(e.Message);
    }

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}

// Sample output:

// Path to my file: c:\Top-Level Folder\SubFolder\ttxvause.vv0

//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 8
//3 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
}

```

폴더가 이미 있으면 [CreateDirectory](#)는 아무 작업도 수행하지 않으며 예외가 throw되지 않습니다. 그러나 [File.Create](#)는 기존 파일을 새 파일로 바꿉니다. 이 예제에서는 `if - else` 문을 사용하여 기존 파일이 바뀌지 않도록 합니다.

예제에서 다음을 변경하여 특정 이름의 파일이 이미 있는지 여부에 따라 다른 결과를 지정할 수 있습니다. 파일이 없는 경우 코드에서 파일을 만듭니다. 파일이 있는 경우 코드에서 해당 파일에 데이터를 추가합니다.

- 임의가 아닌 파일 이름을 지정합니다.

```

// Comment out the following line.
//string fileName = System.IO.Path.GetRandomFileName();

// Replace that line with the following assignment.
string fileName = "MyNewFile.txt";

```

- 다음 코드를 사용하여 `if - else` 문을 `using` 문으로 바꿉니다.

```

using (System.IO.FileStream fs = new System.IO.FileStream(pathString, FileMode.Append))
{
    for (byte i = 0; i < 100; i++)
    {
        fs.WriteByte(i);
    }
}

```

예제를 여러 번 실행하여 매번 파일에 데이터가 추가되는지 확인합니다.

시도할 수 있는 추가  `FileMode` 값은  `FileMode`을 참조하세요.

다음 조건에서 예외가 발생합니다.

- 폴더 이름 형식이 잘못된 경우. 예를 들어 잘못된 문자를 포함하거나 공백만으로 이루어져 있습니다 ( `ArgumentException` 클래스).  `Path` 클래스를 사용하여 유효한 경로 이름을 만듭니다.
- 만들 폴더의 부모 폴더가 읽기 전용입니다( `IOException` 클래스).
- 폴더 이름이 `null`입니다( `ArgumentNullException` 클래스).
- 폴더 이름이 너무 깁니다( `PathTooLongException` 클래스).
- 폴더 이름이 콜론(":")뿐입니다( `PathTooLongException` 클래스).

## .NET 보안

부분 신뢰 상황에서는  `SecurityException` 클래스의 인스턴스가  `throw`될 수 있습니다.

폴더를 만들 수 있는 권한이 없는 경우 이 예제에서는  `UnauthorizedAccessException` 클래스의 인스턴스가  `throw`됩니다.

## 참조

- [System.IO](#)
- [C# 프로그래밍 가이드](#)
- [파일 시스템 및 레지스트리\(C# 프로그래밍 가이드\)](#)

# 파일 및 폴더를 복사, 삭제, 이동하는 방법(C# 프로그래밍 가이드)

2020-11-02 • 5 minutes to read • [Edit Online](#)

다음 예제에서는 [System.IO](#) 네임스페이스의 [System.IO.File](#), [System.IO.Directory](#), [System.IO.FileInfo](#), [System.IO.DirectoryInfo](#) 클래스를 사용하여 파일과 폴더를 동기 방식으로 복사, 이동 및 삭제하는 방법을 보여 줍니다. 이러한 예제는 진행률 표시줄이나 다른 사용자 인터페이스를 제공하지 않습니다. 표준 진행률 대화 상자를 제공하려는 경우 [파일 작업에 대한 진행률 대화 상자를 제공하는 방법](#)을 참조하세요.

[System.IO.FileSystemWatcher](#)를 사용하여 여러 파일에 대해 작업할 때 진행률을 계산할 수 있는 이벤트를 제공할 수 있습니다. 또 다른 방법은 플랫폼 호출을 사용하여 Windows Shell에서 적절한 파일 관련 메서드를 호출하는 것입니다. 이러한 파일 작업을 비동기적으로 수행하는 방법에 대한 자세한 내용은 [비동기 파일 I/O](#)를 참조하세요.

## 예제

다음 예제에서는 파일 및 디렉터리를 복사하는 방법을 보여 줍니다.

```

// Simple synchronous file copy operations with no user interface.
// To run this sample, first create the following directories and files:
// C:\Users\Public\TestFolder
// C:\Users\Public\TestFolder\test.txt
// C:\Users\Public\TestFolder\SubDir\test.txt
public class SimpleFileCopy
{
    static void Main()
    {
        string fileName = "test.txt";
        string sourcePath = @"C:\Users\Public\TestFolder";
        string targetPath = @"C:\Users\Public\TestFolder\SubDir";

        // Use Path class to manipulate file and directory paths.
        string sourceFile = System.IO.Path.Combine(sourcePath, fileName);
        string destFile = System.IO.Path.Combine(targetPath, fileName);

        // To copy a folder's contents to a new location:
        // Create a new target folder.
        // If the directory already exists, this method does not create a new directory.
        System.IO.Directory.CreateDirectory(targetPath);

        // To copy a file to another location and
        // overwrite the destination file if it already exists.
        System.IO.File.Copy(sourceFile, destFile, true);

        // To copy all the files in one directory to another directory.
        // Get the files in the source folder. (To recursively iterate through
        // all subfolders under the current directory, see
        // "How to: Iterate Through a Directory Tree.")
        // Note: Check for target path was performed previously
        //       in this code example.
        if (System.IO.Directory.Exists(sourcePath))
        {
            string[] files = System.IO.Directory.GetFiles(sourcePath);

            // Copy the files and overwrite destination files if they already exist.
            foreach (string s in files)
            {
                // Use static Path methods to extract only the file name from the path.
                fileName = System.IO.Path.GetFileName(s);
                destFile = System.IO.Path.Combine(targetPath, fileName);
                System.IO.File.Copy(s, destFile, true);
            }
        }
        else
        {
            Console.WriteLine("Source path does not exist!");
        }

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

```

## 예제

다음 예제에서는 파일 및 디렉터리를 이동하는 방법을 보여 줍니다.

```

// Simple synchronous file move operations with no user interface.
public class SimpleFileMove
{
    static void Main()
    {
        string sourceFile = @"C:\Users\Public\public\test.txt";
        string destinationFile = @"C:\Users\Public\private\test.txt";

        // To move a file or folder to a new location:
        System.IO.File.Move(sourceFile, destinationFile);

        // To move an entire directory. To programmatically modify or combine
        // path strings, use the System.IO.Path class.
        System.IO.Directory.Move(@"C:\Users\Public\public\test\", @"C:\Users\Public\private");
    }
}

```

## 예제

다음 예제에서는 파일 및 디렉터리를 삭제하는 방법을 보여 줍니다.

```

// Simple synchronous file deletion operations with no user interface.
// To run this sample, create the following files on your drive:
// C:\Users\Public\DeleteTest\test1.txt
// C:\Users\Public\DeleteTest\test2.txt
// C:\Users\Public\DeleteTest\SubDir\test2.txt

public class SimpleFileDelete
{
    static void Main()
    {
        // Delete a file by using File class static method...
        if(System.IO.File.Exists(@"C:\Users\Public\DeleteTest\test.txt"))
        {
            // Use a try block to catch IOExceptions, to
            // handle the case of the file already being
            // opened by another process.
            try
            {
                System.IO.File.Delete(@"C:\Users\Public\DeleteTest\test.txt");
            }
            catch (System.IO.IOException e)
            {
                Console.WriteLine(e.Message);
                return;
            }
        }

        // ...or by using FileInfo instance method.
        System.IO.FileInfo fi = new System.IO.FileInfo(@"C:\Users\Public\DeleteTest\test2.txt");
        try
        {
            fi.Delete();
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }

        // Delete a directory. Must be writable or empty.
        try
        {
            System.IO.Directory.Delete(@"C:\Users\Public\DeleteTest");
        }
        catch (System.IO.IOException e)
        {

```

```

    {
        Console.WriteLine(e.Message);
    }
    // Delete a directory and all subdirectories with Directory static method...
    if(System.IO.Directory.Exists(@"C:\Users\Public\DeleteTest"))
    {
        try
        {
            System.IO.Directory.Delete(@"C:\Users\Public\DeleteTest", true);
        }

        catch (System.IO.IOException e)
        {
            Console.WriteLine(e.Message);
        }
    }

    // ...or with DirectoryInfo instance method.
    System.IO.DirectoryInfo di = new System.IO.DirectoryInfo(@"C:\Users\Public\public");
    // Delete this dir and all subdirs.
    try
    {
        di.Delete(true);
    }
    catch (System.IO.IOException e)
    {
        Console.WriteLine(e.Message);
    }
}
}

```

## 참고 항목

- [System.IO](#)
- [C# 프로그래밍 가이드](#)
- [파일 시스템 및 레지스트리\(C# 프로그래밍 가이드\)](#)
- [파일 작업에 진행률 대화 상자를 제공하는 방법](#)
- [파일 및 스트림 I/O](#)
- [공통적인 I/O 작업](#)

# 파일 작업에 대한 진행률 대화 상자를 제공하는 방법(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

Microsoft.VisualBasic 네임스페이스의 `CopyFile(String, String, UIOption)` 메서드를 사용하는 경우 Windows에서 파일 작업 진행률을 보여 주는 표준 대화 상자를 제공할 수 있습니다.

## NOTE

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.

**Visual Studio**에서 참조를 추가하려면

1. 메뉴 모음에서 **프로젝트, 참조 추가**를 선택합니다.  
참조 관리자 대화 상자가 나타납니다.
2. 어셈블리 영역에서 **프레임워크**가 선택되지 않은 경우 선택합니다.
3. 이름 목록에서 **Microsoft.VisualBasic** 확인란을 선택한 다음 **확인** 단추를 선택하여 대화 상자를 닫습니다.

## 예제

다음 코드는 `sourcePath`로 지정된 디렉터리를 `destinationPath`로 지정된 디렉터리에 복사합니다. 또한 이 코드는 작업이 완료되기까지 남은 예상 시간을 보여 주는 표준 대화 상자를 제공합니다.

```
// The following using directive requires a project reference to Microsoft.VisualBasic.
using Microsoft.VisualBasic.FileIO;

class FileProgress
{
    static void Main()
    {
        // Specify the path to a folder that you want to copy. If the folder is small,
        // you won't have time to see the progress dialog box.
        string sourcePath = @"C:\Windows\symbols\";
        // Choose a destination for the copied files.
        string destinationPath = @"C:\TestFolder";

        FileSystem.CopyDirectory(sourcePath, destinationPath,
            UIOption.AllDialogs);
    }
}
```

## 참조

- [파일 시스템 및 레지스트리\(C# 프로그래밍 가이드\)](#)

# 텍스트 파일에 쓰는 방법(C# 프로그래밍 가이드)

2021-02-18 • 7 minutes to read • [Edit Online](#)

이 문서에는 파일에 텍스트를 쓰는 다양한 방법을 보여 주는 몇 가지 예제가 있습니다. 처음 두 예제에서는 `System.IO.File` 클래스의 정적 편의 메서드를 사용하여 `IEnumerable<string>`의 각 요소와 `string`을 텍스트 파일에 씁니다. 세 번째 예제에서는 파일에 쓸 때 각 줄을 개별적으로 처리해야 하는 경우 파일에 텍스트를 추가하는 방법을 보여 줍니다. 처음 세 예제에서는 파일의 모든 기존 콘텐츠를 덮어씁니다. 마지막 예제에서는 기존 파일에 텍스트를 추가하는 방법을 보여 줍니다.

이 예에서는 파일에 모든 문자열 리터럴을 작성합니다. 파일에 작성된 텍스트의 서식을 지정하려면 `Format` 메서드 또는 C# `문자열 보간` 기능을 사용합니다.

## 파일에 문자열 컬렉션 쓰기

```
using System.IO;
using System.Threading.Tasks;

class WriteAllLines
{
    public static async Task ExampleAsync()
    {
        string[] lines =
        {
            "First line", "Second line", "Third line"
        };

        await File.WriteAllLinesAsync("WriteLines.txt", lines);
    }
}
```

이전 소스 코드 예제는 다음과 같습니다.

- 세 개 값이 있는 문자열 배열을 인스턴스화합니다.
- 다음과 같은 `File.WriteAllLinesAsync` 호출을 기다립니다.
  - 파일 이름 `WriteLines.txt`를 비동기적으로 만듭니다. 파일이 이미 있으면 덮어씁니다.
  - 지정된 줄을 파일에 씁니다.
  - 파일을 닫아 필요에 따라 자동으로 플러시하고 삭제합니다.

## 파일에 하나의 문자열 쓰기

```

using System.IO;
using System.Threading.Tasks;

class WriteAllText
{
    public static async Task ExampleAsync()
    {
        string text =
            "A class is the most powerful data type in C#. Like a structure, " +
            "a class defines the data and behavior of the data type. ";

        await File.WriteAllTextAsync("WriteText.txt", text);
    }
}

```

이전 소스 코드 예제는 다음과 같습니다.

- 할당된 문자열 리터럴이 제공된 문자열을 인스턴스화합니다.
- 다음과 같은 `File.WriteAllTextAsync` 호출을 기다립니다.
  - 파일 이름 `WriteText.txt`를 비동기적으로 만듭니다. 파일이 이미 있으면 덮어씁니다.
  - 지정된 텍스트를 파일에 씁니다.
  - 파일을 닫아 필요에 따라 자동으로 플러시하고 삭제합니다.

## 파일에 배열에서 선택한 문자열 쓰기

```

using System.IO;
using System.Threading.Tasks;

class StreamWriterOne
{
    public static async Task ExampleAsync()
    {
        string[] lines = { "First line", "Second line", "Third line" };
        using StreamWriter file = new("WriteLines2.txt");

        foreach (string line in lines)
        {
            if (!line.Contains("Second"))
            {
                await file.WriteLineAsync(line);
            }
        }
    }
}

```

이전 소스 코드 예제는 다음과 같습니다.

- 세 개 값이 있는 문자열 배열을 인스턴스화합니다.
- `WriteLines2.txt`의 파일 경로를 `using 선언`으로 사용하여 `StreamWriter`를 인스턴스화합니다.
- 모든 줄을 반복합니다.
- 줄에 `"Second"` 가 포함되지 않은 경우 해당 줄을 파일에 쓰는 `StreamWriter.WriteLineAsync(String)` 호출을 조건에 따라 기다립니다.

## 기존 파일에 텍스트 추가

```
using System.IO;
using System.Threading.Tasks;

class StreamWriterTwo
{
    public static async Task ExampleAsync()
    {
        using StreamWriter file = new("WriteLines2.txt", append: true);
        await file.WriteLineAsync("Fourth line");
    }
}
```

이전 소스 코드 예제는 다음과 같습니다.

- 세 개 값이 있는 문자열 배열을 인스턴스화합니다.
- *WriteLines2.txt*의 파일 경로를 `using` 선언으로 사용하여 `StreamWriter`를 인스턴스화하고 추가할 `true`를 전달합니다.
- 문자열을 파일에 추가된 줄로 쓰는 `StreamWriter.WriteLineAsync(String)` 호출을 기다립니다.

## 예외

다음 조건에서 예외가 발생합니다.

- `InvalidOperationException`: 파일이 있지만 읽기 전용인 경우
- `PathTooLongException`: 경로 이름이 너무 긴 경우
- `IOException`: 디스크가 꽉 찬 경우

파일 시스템을 사용하는 경우 예외를 발생시킬 수 있는 추가 조건이 있습니다. 방어적으로 프로그래밍하는 것이 좋습니다.

## 참조

- [C# 프로그래밍 가이드](#)
- [파일 시스템 및 레지스트리\(C# 프로그래밍 가이드\)](#)
- [샘플: 애플리케이션 스토리지에 컬렉션 저장](#)

# 텍스트 파일에서 읽는 방법(C# 프로그래밍 가이드)

2020-11-02 • 3 minutes to read • [Edit Online](#)

이 예제에서는 `System.IO.File` 클래스의 정적 메서드 `ReadAllText` 및 `ReadAllLines`을 사용하여 텍스트 파일의 내용을 읽습니다.

`StreamReader`을 사용하는 예제는 [텍스트 파일을 한 번에 한 줄씩 읽는 방법](#)을 참조하세요.

## NOTE

이 예제에 사용되는 파일은 [텍스트 파일에 쓰는 방법](#) 항목에서 생성되었습니다.

## 예제

```
class ReadFromFile
{
    static void Main()
    {
        // The files used in this example are created in the topic
        // How to: Write to a Text File. You can change the path and
        // file name to substitute text files of your own.

        // Example #1
        // Read the file as one string.
        string text = System.IO.File.ReadAllText(@"C:\Users\Public\TestFolder\WriteText.txt");

        // Display the file contents to the console. Variable text is a string.
        System.Console.WriteLine("Contents of WriteText.txt = {0}", text);

        // Example #2
        // Read each line of the file into a string array. Each element
        // of the array is one line of the file.
        string[] lines = System.IO.File.ReadAllLines(@"C:\Users\Public\TestFolder\WriteLines2.txt");

        // Display the file contents by using a foreach loop.
        System.Console.WriteLine("Contents of WriteLines2.txt = ");
        foreach (string line in lines)
        {
            // Use a tab to indent each line of the file.
            Console.WriteLine("\t" + line);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

## 코드 컴파일

코드를 복사하고 C# 콘솔 애플리케이션에 붙여넣습니다.

[텍스트 파일에 쓰는 방법](#)의 텍스트 파일을 사용하지 않는 경우 `ReadAllText` 및 `ReadAllLines`에 대한 인수를 사용자 컴퓨터의 적절한 경로 및 파일 이름으로 바꿉니다.

# 강력한 프로그래밍

다음 조건에서 예외가 발생합니다.

- 파일이 없거나 지정된 위치에 없는 경우. 경로와 파일 이름의 철자를 확인합니다.

## .NET 보안

파일 이름을 사용하여 파일 내용을 확인하지 마세요. 예를 들어 `myFile.cs` 파일이 C# 소스 파일이 아닐 수도 있습니다.

## 참조

- [System.IO](#)
- [C# 프로그래밍 가이드](#)
- [파일 시스템 및 레지스트리\(C# 프로그래밍 가이드\)](#)

# 텍스트 파일을 한 번에 한 줄씩 읽는 방법(C# 프로그래밍 가이드)

2020-11-02 • 2 minutes to read • [Edit Online](#)

이 예제에서는 `StreamReader` 클래스의 `ReadLine` 메서드를 사용하여 텍스트 파일 내용을 한 번에 한 줄씩 문자열로 읽어옵니다. 각 텍스트 줄은 `line` 문자열에 저장되고 화면에 표시됩니다.

## 예제

```
int counter = 0;
string line;

// Read the file and display it line by line.
System.IO.StreamReader file =
    new System.IO.StreamReader(@"c:\test.txt");
while((line = file.ReadLine()) != null)
{
    System.Console.WriteLine(line);
    counter++;
}

file.Close();
System.Console.WriteLine("There were {0} lines.", counter);
// Suspend the screen.
System.Console.ReadLine();
```

## 코드 컴파일

코드를 복사하고 콘솔 애플리케이션의 `Main` 메서드에 붙여넣습니다.

"c:\test.txt" 를 실제 파일 이름으로 바꿉니다.

## 강력한 프로그래밍

다음 조건에서 예외가 발생합니다.

- 파일이 없을 수 있는 경우

## .NET 보안

파일 이름을 바탕으로 파일 내용을 판단하면 안 됩니다. 예를 들어 `myFile.cs` 파일이 C# 소스 파일이 아닐 수도 있습니다.

## 참조

- [System.IO](#)
- [C# 프로그래밍 가이드](#)
- [파일 시스템 및 레지스트리\(C# 프로그래밍 가이드\)](#)

# 레지스트리에 키를 만드는 방법(C# 프로그래밍 가이드)

2020-11-02 • 4 minutes to read • [Edit Online](#)

이 예제에서는 현재 사용자의 레지스트리, "Names" 키 아래에 "Name" 및 "Isabella" 값 쌍을 추가합니다.

## 예제

```
Microsoft.Win32.RegistryKey key;
key = Microsoft.Win32.Registry.CurrentUser.CreateSubKey("Names");
key.SetValue("Name", "Isabella");
key.Close();
```

## 코드 컴파일

- 코드를 복사하고 콘솔 애플리케이션의 `Main` 메서드에 붙여넣습니다.
- `Names` 매개 변수를 레지스트리의 HKEY\_CURRENT\_USER 노드 바로 아래에 있는 키 이름으로 바꿉니다.
- `Name` 매개 변수를 Names 노드 바로 아래에 있는 값 이름으로 바꿉니다.

## 강력한 프로그래밍

레지스트리 구조를 검사하여 키에 적합한 위치를 찾습니다. 예를 들어 현재 사용자의 소프트웨어 키를 열고 회사 이름으로 키를 만들 수 있습니다. 그런 다음 회사 키에 레지스트리 값을 추가합니다.

다음 조건에서 예외가 발생할 수 있습니다.

- 키 이름이 null인 경우
- 사용자에게 레지스트리 키를 만들 수 있는 권한이 없는 경우
- 키 이름이 255자 제한을 초과하는 경우
- 키가 닫힌 경우
- 레지스트리 키가 읽기 전용인 경우

## .NET 보안

로컬 컴퓨터(`Microsoft.Win32.Registry.LocalMachine`)보다 사용자 폴더(`Microsoft.Win32.Registry.CurrentUser`)에 데이터를 쓰는 것이 더 안전합니다.

레지스트리 값을 만들 때 해당 값이 이미 있는 경우 수행할 작업을 결정해야 합니다. 다른 악성 프로세스에서 값을 이미 만들고 액세스했을 수도 있습니다. 레지스트리 값에 데이터를 넣으면 다른 프로세스에서 해당 데이터를 사용할 수 있습니다. 이를 방지하려면 `Overload:Microsoft.Win32.RegistryKey.GetValue` 메서드를 재정의합니다. 키가 아직 없는 경우 null이 반환됩니다.

레지스트리 키가 ACL(액세스 제어 목록)로 보호된 경우에도 암호 등을 레지스트리에 일반 텍스트로 저장하는 것은 안전하지 않습니다.

## 참조

- [System.IO](#)
- [C# 프로그래밍 가이드](#)
- [파일 시스템 및 레지스트리\(C# 프로그래밍 가이드\)](#)
- [C#을 사용하여 레지스트리에서 읽기, 쓰기 및 삭제](#)

# 상호 운용성(C# 프로그래밍 가이드)

2020-11-02 • 3 minutes to read • [Edit Online](#)

상호 운용성은 비관리 코드에 대한 기존 투자를 보존하고 활용할 수 있도록 합니다. CLR(공용 언어 런타임)의 제어 하에서 실행되는 코드를 관리 코드라고 하고, CLR 외부에서 실행되는 코드를 비관리 코드라고 합니다. 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.

## 섹션 내용

### [상호 운용성 개요](#)

C# 관리 코드와 비관리 코드 간에 상호 운용되도록 하는 방법을 설명합니다.

### [C# 기능을 사용하여 Office interop 개체에 액세스하는 방법](#)

Office 프로그래밍을 용이하게 하도록 Visual C#에 도입된 기능에 대해 설명합니다.

### [COM interop 프로그래밍에서 인덱싱된 속성을 사용하는 방법](#)

인덱싱된 속성을 사용하여 매개 변수가 있는 COM 속성에 액세스하는 방법을 설명합니다.

### [플랫폼 호출을 사용하여 WAV 파일을 재생하는 방법](#)

플랫폼 호출 서비스를 사용하여 Windows 운영 체제에서 .wav 사운드 파일을 재생하는 방법을 설명합니다.

### [연습: Office 프로그래밍](#)

Excel 통합 문서와 통합 문서에 대한 링크를 포함하는 Word 문서를 만드는 방법을 보여 줍니다.

### [COM 클래스 예제](#)

C# 클래스를 COM 개체로 노출하는 방법을 보여 줍니다.

## C# 언어 사양

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

## 참고 항목

- [Marshal.ReleaseComObject](#)
- [C# 프로그래밍 가이드](#)
- [비관리 코드와의 상호 운용](#)
- [연습: Office 프로그래밍](#)

# 상호 운용성 개요(C# 프로그래밍 가이드)

2020-11-02 • 9 minutes to read • [Edit Online](#)

이 항목에서는 C# 관리 코드와 비관리 코드 간의 상호 운용성을 사용하도록 설정하는 방법을 설명합니다.

## 플랫폼 호출

플랫폼 호출은 관리 코드가 Microsoft Windows API의 함수와 같이 DLL(동적 연결 라이브러리)에서 구현된 관리되지 않는 함수를 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.

### NOTE

CLR(공용 언어 런타임)은 시스템 리소스에 대한 액세스를 관리합니다. CLR 외부에 있는 비관리 코드를 호출하면 이 보안 메커니즘을 우회하므로 보안 위협이 제기됩니다. 예를 들어 비관리 코드는 CLR 보안 메커니즘을 우회하여 비관리 코드의 리소스를 직접 호출할 수 있습니다. 자세한 내용은 [.NET의 보안](#)을 참조하세요.

## C++ Interop

C# 또는 다른 .NET 언어로 작성된 코드에서 사용할 수 있도록 IJW(It Just Works)라고도 하는 C++ interop를 사용하여 네이티브 C++ 클래스를 래핑할 수 있습니다. 이렇게 하려면 C++ 코드를 작성하여 네이티브 DLL 또는 COM 구성 요소를 래핑합니다. 다른 .NET 언어와 달리 Visual C++에는 관리 코드와 비관리 코드가 동일한 애플리케이션 및 동일한 파일에 있을 수 있도록 하는 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. RCW에서 정의된 클래스의 인스턴스를 만듭니다. 그러면 COM 개체의 인스턴스가 생성됩니다.
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.

Visual C# 프로젝트 속성을 수정하여 어셈블리 COM이 표시되도록 설정할 수 있습니다. 자세한 내용은 [어셈블리 정보 대화 상자](#)를 참조하세요.

2. COM 형식 라이브러리를 생성하고 COM 사용을 위해 등록합니다.

COM interop에 대해 C# 어셈블리를 자동으로 등록하도록 Visual C# 프로젝트 속성을 수정할 수 있습니다. Visual Studio에서는 [Regasm.exe\(어셈블리 등록 도구\)](#)를 사용하며, 관리되는 어셈블리를 입력으로 사용하는 `/t:lb` 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.

## 참조

- [Interop 성능 향상](#)
- [Introduction to Interoperability between COM and .NET](#)(COM과 .NET 간 상호 운용성 소개)
- [Visual Basic의 COM Interop 소개](#)
- [관리 코드와 비관리 코드 간의 마샬링](#)
- [비관리 코드와의 상호 운용](#)
- [C# 프로그래밍 가이드](#)

# Office interop 개체에 액세스하는 방법(C# 프로그래밍 가이드)

2020-11-02 • 27 minutes to read • [Edit Online](#)

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 항목에서는 새 기능을 사용하여 Microsoft Office Excel 워크시트를 만들고 표시하는 코드를 작성합니다. 그런 다음 Excel 워크시트에 연결된 아이콘이 들어 있는 Office Word 문서를 추가하는 코드를 작성합니다.

이 연습을 완료하려면 Microsoft Office Excel 2007 및 Microsoft Office Word 2007 이상 버전이 컴퓨터에 설치되어 있어야 합니다.

## NOTE

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.

## 새 콘솔 애플리케이션을 만들려면

1. Visual Studio를 시작합니다.
2. 파일 메뉴에서 새로 만들기를 가리킨 다음 프로젝트를 클릭합니다. 새 프로젝트 대화 상자가 나타납니다.
3. 설치된 템플릿 창에서 **Visual C#** 을 확장한 다음 **Windows**를 클릭합니다.
4. 새 프로젝트 대화 상자 위쪽에서 **.NET Framework 4** 이상 버전이 대상 프레임워크로 선택되어 있는지 확인합니다.
5. 템플릿 창에서 콘솔 애플리케이션을 클릭합니다.
6. 이름 필드에 프로젝트의 이름을 입력합니다.
7. 확인을 클릭합니다.

솔루션 탐색기에 새 프로젝트가 표시됩니다.

## 참조를 추가하려면

1. 솔루션 탐색기에서 프로젝트 이름을 마우스 오른쪽 단추로 클릭하고 참조 추가를 클릭합니다. 참조 추가 대화 상자가 나타납니다.
2. 어셈블리 페이지의 구성 요소 이름 목록에서 **Microsoft.Office.Interop.Word**를 선택하고 Ctrl 키를 누른 상태로 **Microsoft.Office.Interop.Excel**을 선택합니다. 이러한 어셈블리가 보이지 않으면 어셈블리가 설치되어 있으며 표시되는지를 확인해야 할 수 있습니다. [방법: Office Primary Interop Assemblies](#)를 설치합니다.
3. 확인을 클릭합니다.

## 필요한 using 지시문을 추가하려면

- 솔루션 탐색기에서 `Program.cs` 파일을 마우스 오른쪽 단추로 클릭하고 코드 보기 를 클릭합니다.
- 다음 `using` 지시문을 코드 파일의 맨 위에 추가합니다.

```
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

## 은행 계좌 목록을 만들려면

- 다음 클래스 정의를 `Program.cs`의 `Program` 클래스 아래에 붙여 넣습니다.

```
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.

```
// 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

- 다음 메서드를 `Program` 클래스에 추가하여 Excel 워크시트를 설정합니다.

`Add` 메서드에는 특정 템플릿을 지정하기 위한 선택적 매개 변수가 있습니다. C# 4에서 도입된 선택적 매개 변수를 사용하면 매개 변수의 기본값을 사용하려는 경우 해당 매개 변수의 인수를 생략할 수 있습니다. 다음 코드에서는 인수가 전송되지 않으므로 `Add`는 기본 템플릿을 사용하며 새 통합 문서를 만듭니다. 이전 버전의 C#에서 이와 동일한 문을 사용하려면 자리 표시자 인수인 `ExcelApp.Workbooks.Add(Type.Missing)`를 사용해야 했습니다.

```

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;
}

```

2. `DisplayInExcel` 끝에 다음 코드를 추가합니다. 이 코드는 워크시트 첫 번째 행의 처음 두 열에 값을 삽입합니다.

```

// Establish column headings in cells A1 and B1.
workSheet.Cells[1, "A"] = "ID Number";
workSheet.Cells[1, "B"] = "Current Balance";

```

3. `DisplayInExcel` 끝에 다음 코드를 추가합니다. `foreach` 루프는 계좌 목록의 정보를 워크시트에서 연속 행의 처음 두 열에 삽입합니다.

```

var row = 1;
foreach (var acct in accounts)
{
    row++;
    workSheet.Cells[row, "A"] = acct.ID;
    workSheet.Cells[row, "B"] = acct.Balance;
}

```

4. 열 너비를 콘텐츠에 맞게 조정하려면 `DisplayInExcel` 끝에 다음 코드를 추가합니다.

```

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.

```

((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();

```

C# 4 이상 버전에서는 [-link](#) 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

1. `Main` 끝에 다음 줄을 추가합니다.

```
// Display the list in an Excel spreadsheet.  
DisplayInExcel(bankAccounts);
```

## 2. Ctrl+F5를 누릅니다.

두 계좌의 데이터가 포함된 Excel 워크시트가 표시됩니다.

## Word 문서를 추가하려면

1. C# 4 이상 버전에서 Office 프로그래밍을 향상하는 추가 방법을 설명하기 위해, 다음 코드에서는 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 C# 4에 도입된 다른 두 가지 새 기능이 통합됩니다. 이러한 기능은 참조 매개 변수가 포함된 COM 메서드 호출을 간소화합니다. 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). C# 4.0 이상 버전에서는 기본 값을 사용하려는 경우 모든 매개 변수 또는 원하는 매개 변수의 인수를 생략할 수 있습니다. C# 3.0 이하 버전에서는 각 매개 변수에 대해 인수를 제공해야 하며 매개 변수가 참조 매개 변수이므로 인수는 변수여야 합니다.

`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. C# 4.0 이상 버전에서는 이 두 매개 변수에 대해 명명된 인수를 사용하고 나머지 인수는 생략할 수 있습니다. 이러한 매개 변수는 참조 매개 변수이지만 `ref` 키워드를 사용하거나 인수로 보낼 변수를 만들 필요가 없으며, 값을 직접 보낼 수 있습니다. C# 3.0 이하 버전에서는 각 참조 매개 변수에 대해 가변 인수를 공급해야 합니다.

```
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);  
}
```

C# 3.0 이하 버전의 언어에서는 다음과 같은 더 복잡한 코드를 사용해야 합니다.

```

static void CreateIconInWordDoc2008()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four parameters, all of which are optional.
    // In Visual C# 2008 and earlier versions, an argument has to be sent
    // for every parameter. Because the parameters are reference
    // parameters of type object, you have to create an object variable
    // for the arguments that represents 'no value'.

    object useDefaultValue = Type.Missing;

    wordApp.Documents.Add(ref useDefaultValue, ref useDefaultValue,
        ref useDefaultValue, ref useDefaultValue);

    // PasteSpecial has seven reference parameters, all of which are
    // optional. In this example, only two of the parameters require
    // specified values, but in Visual C# 2008 an argument must be sent
    // for each parameter. Because the parameters are reference parameters,
    // you have to construct variables for the arguments.
    object link = true;
    object displayAsIcon = true;

    wordApp.Selection.PasteSpecial( ref useDefaultValue,
                                    ref link,
                                    ref useDefaultValue,
                                    ref displayAsIcon,
                                    ref useDefaultValue,
                                    ref useDefaultValue,
                                    ref useDefaultValue);
}

```

2. `Main` 끝에 다음 문을 추가합니다.

```

// Create a Word document that contains an icon that links to
// the spreadsheet.
CreateIconInWordDoc();

```

3. `DisplayInExcel` 끝에 다음 문을 추가합니다. `Copy` 메서드는 클립보드에 워크시트를 추가합니다.

```

// 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();

```

4. Ctrl+F5를 누릅니다.

아이콘이 포함된 Word 문서가 나타납니다. 아이콘을 두 번 클릭하여 워크시트를 포그라운드로 가져옵니다.

## Interop 형식 포함 속성을 설정하려면

1. 런타임에 PIA(주 interop 어셈블리)를 사용하지 않아도 되는 COM 형식을 호출할 때는 코드를 추가로 개선할 수 있습니다. PIA에 대한 종속성을 제거하면 버전을 독립적으로 실행할 수 있으며 보다 쉽게 배포할 수 있습니다. PIA를 사용하지 않는 프로그래밍의 장점에 대한 자세한 내용은 [연습: 관리되는 어셈블리의 형식 포함](#)을 참조하세요.

또한 COM 메서드에서 사용해야 하며 반환되는 형식은 `dynamic` 가 아닌 `Object` 형식을 사용하여 표시할 수 있으므로 프로그램이 더욱 쉬워집니다. `dynamic` 형식이 포함된 변수는 런타임까지 평가되지 않으므

로 명시적 캐스팅을 수행할 필요가 없습니다. 자세한 내용은 [dynamic 형식 사용](#)을 참조하세요.

C# 4에서는 PIA를 사용하는 대신 형식 정보를 포함하는 것이 기본 동작입니다. 이러한 기본 동작으로 인해 명시적 캐스팅이 필요하지 않으므로 위의 여러 예제가 간소화됩니다. 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.

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

2. 형식 정보를 포함하는 대신 기본값을 변경하여 PIA를 사용하려면 솔루션 탐색기에서 참조 노드를 확장하고 **Microsoft.Office.Interop.Excel** 또는 **Microsoft.Office.Interop.Word**를 선택합니다.
3. 속성 창이 보이지 않으면 F4키를 누릅니다.
4. 속성 목록에서 **Interop** 형식 포함을 찾은 다음 해당 값을 **False**로 변경합니다. 마찬가지로 명령 프롬프트에서 **-link** 대신 **-reference** 컴파일러 옵션을 사용하여 컴파일할 수 있습니다.

## 표에 서식을 더 추가하려면

1. `AutoFit`에서 `DisplayInExcel`에 대한 두 호출을 다음 문으로 바꿉니다.

```
// 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 제공할 수도 있고 모든 매개 변수에 인수를 제공하지 않을 수도 있습니다. 위의 문에서는 `Format` 매개 변수 하나에만 인수가 제공됩니다. `Format`은 매개 변수 목록의 첫 번째 매개 변수이므로 매개 변수 이름은 지정하지 않아도 됩니다. 그러나 다음 코드에 나와 있는 것처럼 매개 변수 이름을 포함하면 문을 더 쉽게 이해할 수 있습니다.

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(Format:
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

2. 결과를 보려면 CTRL+F5를 누릅니다. 기타 서식은 **XlRangeAutoFormat** 열거형에 나열됩니다.
3. 1단계의 문을 다음 코드와 비교합니다. 이 코드는 C# 3.0 이하 버전에 필요한 인수를 보여 줍니다.

```
// The AutoFormat method has seven optional value parameters. The
// following call specifies a value for the first parameter, and uses
// the default values for the other six.

// Call to AutoFormat in Visual C# 2008. This code is not part of the
// current solution.
excelApp.get_Range("A1", "B4").AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormatTable3,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing,
    Type.Missing);
```

## 예제

다음 코드에서는 전체 예제를 보여 줍니다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeProgramminWalkthruComplete
{
    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; }
}
}

```

## 참조

- [Type.Missing](#)
- [dynamic](#)
- [dynamic 형식 사용](#)
- [명명된 인수 및 선택적 인수](#)
- [Office 프로그래밍에 명명된 인수와 선택적 인수 사용 방법](#)

# COM interop 프로그래밍에서 인덱싱된 속성을 사용하는 방법(C# 프로그래밍 가이드)

2020-11-02 • 5 minutes to read • [Edit Online](#)

인덱싱된 속성은 매개 변수가 있는 COM 속성이 C# 프로그래밍에서 사용되는 방식을 개선합니다. 인덱싱된 속성은 Visual C#의 다른 기능(예: 명명된 인수 및 선택적 인수), 새 형식(dynamic), 포함된 형식 정보와 함께 작동하여 Microsoft Office 프로그래밍을 개선합니다.

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.

```
// 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.

```
// Visual C# 2010.  
var excelApp = new Excel.Application();  
// . . .  
Excel.Range targetRange = excelApp.Range["A1"];
```

## NOTE

또한 앞의 예제에서는 선택적 인수 기능을 사용하므로 `Type.Missing`을 생략할 수 있습니다.

마찬가지로, C# 3.0 이하에서 `Range` 객체의 `value` 속성 값을 설정하려면 두 개의 인수가 필요합니다. 하나는 범위 값의 형식을 지정하는 선택적 매개 변수의 인수를 제공합니다. 다른 하나는 `value` 속성의 값을 제공합니다. 다음 예제에서는 이러한 기법을 보여 줍니다. 둘 다 A1 셀의 값을 `Name`으로 설정합니다.

```
// Visual C# 2008.  
targetRange.set_Value(Type.Missing, "Name");  
// Or  
targetRange.Value2 = "Name";
```

인덱싱된 속성을 사용하면 대신 다음 코드와 같이 작성할 수 있습니다.

```
// Visual C# 2010.  
targetRange.Value = "Name";
```

인덱싱된 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.

```
// 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();
            //CSharp2008();
        }

        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](#)
- [Office 프로그래밍에 명명된 인수와 선택적 인수 사용 방법](#)
- [C# 기능을 사용하여 Office interop 개체에 액세스하는 방법](#)
- [연습: Office 프로그래밍](#)

# 플랫폼 호출을 사용하여 WAV 파일을 재생하는 방법(C# 프로그래밍 가이드)

2020-11-02 • 4 minutes to read • [Edit Online](#)

다음 C# 코드 예제에서는 플랫폼 호출 서비스를 사용하여 Windows 운영 체제에서 WAV 사운드 파일을 재생하는 방법을 설명합니다.

## 예제

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 열기를 클릭하여 플랫폼 호출을 통해 웨이브 파일을 재생합니다. 텍스트 상자에 선택한 파일의 전체 경로가 표시됩니다.

파일 열기 대화 상자가 필터 설정을 통해 .wav 확장명을 가진 파일만 표시하도록 필터링됩니다.

```
dialog1.Filter = "Wav Files (*.wav)|*.wav";
```

```

using System.Windows.Forms;
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 = 0x00002000,
            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:\";
            dialog1.Filter = "Wav Files (*.wav)|*.wav";
            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.
        }
    }
}

```

## 코드 컴파일

1. Visual Studio에서 새 C# Windows Forms 애플리케이션 프로젝트를 만들고 이름을 **WinSound**로 지정합니다.
2. 위의 코드를 복사하여 *Form1.cs* 파일 내용에 붙여넣습니다.

3. 다음 코드를 복사하여 *Form1.Designer.cs* 파일의 `InitializeComponent()` 메서드에서 기존 코드 뒤에 붙여 넣습니다.

```
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();
```

4. 코드를 컴파일하고 실행합니다.

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [상호 운용성 개요](#)
- [플랫폼 호출 자세히 보기](#)
- [플랫폼 호출을 사용하여 데이터 마샬링](#)

# 연습: Office 프로그래밍(C# 및 Visual Basic)

2020-11-02 • 30 minutes to read • [Edit Online](#)

Visual Studio에서는 Microsoft Office 프로그래밍을 개선하는 C# 및 Visual Basic의 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. Visual Basic의 기능으로는 자동 구현 속성, 람다 식의 문, 컬렉션 이니셜라이저 등이 있습니다.

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.

## 사전 요구 사항

이 연습을 완료하려면 Microsoft Office Excel 및 Microsoft Office Word가 컴퓨터에 설치되어 있어야 합니다.

### NOTE

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.

### Excel 추가 기능 애플리케이션을 설치하려면

1. Visual Studio를 시작합니다.
2. 파일 메뉴에서 새로 만들기를 가리킨 다음 프로젝트를 클릭합니다.
3. 설치된 템플릿 창에서 **Visual Basic** 또는 **Visual C#** 을 확장하고 **Office**를 확장한 다음 Office 제품의 버전 연도를 클릭합니다.
4. 템플릿 창에서 **Excel <version>** 추가 기능을 클릭합니다.
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 클릭합니다.
8. 솔루션 탐색기에 새 프로젝트가 표시됩니다.

### 참조를 추가하려면

1. 솔루션 탐색기에서 프로젝트 이름을 마우스 오른쪽 단추로 클릭하고 참조 추가를 클릭합니다. 참조 추가 대화 상자가 나타납니다.
2. 어셈블리 템의 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>.0.0.0`을 선택합니다. 이러한 어셈블리가 보이지 않

으면 어셈블리가 설치되어 있으며 표시되는지를 확인해야 할 수 있습니다([방법: Office 주 Interop 어셈블리 설치](#) 참조).

### 3. 확인을 클릭합니다.

필요한 **Import** 문 또는 **using** 문을 추가하려면

- 솔루션 탐색기에서 **ThisAddIn.vb** 또는 **ThisAddIn.cs** 파일을 마우스 오른쪽 단추로 클릭한 다음 코드 보기 를 클릭합니다.

- 다음 **Imports** 문(Visual Basic) 또는 **using** 지시문(C#)이 없으면 코드 파일 맨 위에 추가합니다.

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

```
Imports Microsoft.Office.Interop
```

은행 계좌 목록을 만들려면

- 솔루션 탐색기에서 프로젝트 이름을 마우스 오른쪽 단추로 클릭하고 추가를 클릭한 다음 클래스를 클릭합니다. Visual Basic을 사용하는 경우 Account.vb로, C#을 사용하는 경우에는 Account.cs로 클래스의 이름을 지정합니다. 추가를 클릭합니다.
- Account** 클래스 정의를 다음 코드로 바꿉니다. 클래스 정의는 자동으로 구현된 속성을 사용합니다. 자세한 내용은 [자동으로 구현된 속성](#)을 참조하세요.

```
class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

```
Public Class Account
    Property ID As Integer = -1
    Property Balance As Double
End Class
```

- 계좌 2개가 포함된 **bankAccounts** 목록을 만들려면 **ThisAddIn.vb** 또는 **ThisAddIn.cs**의 **ThisAddIn\_Startup** 메서드에 다음 코드를 추가합니다. 목록 선언은 컬렉션 이니셜라이저를 사용합니다. 자세한 내용은 [컬렉션 이니셜라이저](#)를 참조하세요.

```
var bankAccounts = new List<Account>
{
    new Account
    {
        ID = 345,
        Balance = 541.27
    },
    new Account
    {
        ID = 123,
        Balance = -127.44
    }
};
```

```

Dim bankAccounts As New List(Of Account) From {
    New Account With {
        .ID = 345,
        .Balance = 541.27
    },
    New Account With {
        .ID = 123,
        .Balance = -127.44
    }
}

```

데이터를 **Excel**로 내보내려면

- 같은 파일에서 `ThisAddIn` 클래스에 다음 메서드를 추가합니다. 이 메서드는 Excel 통합 문서를 설정하고 데이터를 해당 통합 문서로 내보냅니다.

```

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();
}

```

```

Sub DisplayInExcel(ByVal accounts As IEnumerable(Of Account),
                   ByVal DisplayAction As Action(Of Account, Excel.Range))

    With Me.Application
        ' Add a new Excel workbook.
        .Workbooks.Add()
        .Visible = True
        .Range("A1").Value = "ID"
        .Range("B1").Value = "Balance"
        .Range("A2").Select()

        For Each ac In accounts
            DisplayAction(ac, .ActiveCell)
            .ActiveCell.Offset(1, 0).Select()
        Next

        ' Copy the results to the Clipboard.
        .Range("A1:B3").Copy()
    End With
End Sub

```

이 메서드에는 두 가지 새로운 C# 기능이 사용됩니다. Visual Basic에서는 이 두 기능이 모두 이미 포함되어 있습니다.

- Add** 메서드에는 특정 템플릿을 지정하기 위한 **선택적 매개 변수**가 있습니다. C# 4에서 도입된 선택적 매개 변수를 사용하면 매개 변수의 기본값을 사용하려는 경우 해당 매개 변수의 인수를 생략

할 수 있습니다. 앞의 예제에서는 인수가 전송되지 않으므로 `Add`는 기본 템플릿을 사용하며 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.

```
// Visual C# 2010 provides indexed properties for COM programming.  
excelApp.Range["A1"].Value = "ID";  
excelApp.ActiveCell.Offset[1, 0].Select();
```

이전 버전의 언어에서는 다음과 같은 특수 구문을 사용해야 했습니다.

```
// In Visual C# 2008, you cannot access the Range, Offset, and Value  
// properties directly.  
excelApp.get_Range("A1").Value2 = "ID";  
excelApp.ActiveCell.get_Offset(1, 0).Select();
```

인덱싱된 속성을 직접 만들 수는 없습니다. 이 기능은 기존 인덱싱된 속성의 사용만을 지원합니다.

자세한 내용은 [COM interop 프로그래밍에서 인덱싱된 속성을 사용하는 방법](#)을 참조하세요.

2. 열 너비를 콘텐츠에 맞게 조정하려면 `DisplayInExcel` 끝에 다음 코드를 추가합니다.

```
excelApp.Columns[1].AutoFit();  
excelApp.Columns[2].AutoFit();
```

```
' Add the following two lines at the end of the With statement.  
.Columns(1).AutoFit()  
.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. [-link](#) 컴파일러 옵션을 통해 어셈블리를 참조할 때 **Interop 형식 포함**을 기본값인 `True` 또는 그와 동일한 값으로 설정하면 이 작업이 자동으로 수행됩니다. `dynamic` 형식을 사용하면 Visual Basic의 기존 기능인 런타임에 바인딩을 사용할 수 있으며, C# 3.0 이후 버전 언어에서 필요했던 명시적 캐스팅을 사용할 필요가 없습니다.

예를 들어 `excelApp.Columns[1]`는 `Object`를 반환하고, `AutoFit`는 Excel 범위 메서드입니다. `dynamic`을 사용하지 않는 경우에는 `excelApp.Columns[1]` 메서드를 호출하기 전에 `Range`에서 반환하는 개체를 `AutoFit` 인스턴스로 캐스팅해야 합니다.

```
// Casting is required in Visual C# 2008.  
((Excel.Range)excelApp.Columns[1]).AutoFit();  
  
// Casting is not required in Visual C# 2010.  
excelApp.Columns[1].AutoFit();
```

Interop 형식 포함에 대한 자세한 내용은 이 항목 뒷부분의 "PIA 참조를 찾으려면" 및 "PIA 종속성을 복원

하려면" 절차를 참조하세요. `dynamic`에 대한 자세한 내용은 [dynamic](#) 또는 [dynamic 형식 사용](#)을 참조하세요.

### DisplayInExcel를 호출하려면

1. `ThisAddIn_StartUp` 메서드의 끝에 다음 코드를 추가합니다. `DisplayInExcel` 호출에는 두 개의 인수가 포함됩니다. 첫 번째 인수는 처리할 계좌 목록의 이름이고, 두 번째 인수는 데이터 처리 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.

```
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;
    }
});
```

```
DisplayInExcel(bankAccounts,
    Sub(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 Then
            cell.Interior.Color = RGB(255, 0, 0)
            cell.Offset(0, 1).Interior.Color = RGB(255, 0, 0)
        End If
    End Sub)
```

2. F5 키를 눌러 프로그램을 실행합니다. 그러면 계좌의 데이터가 포함된 Excel 워크시트가 표시됩니다.

### Word 문서를 추가하려면

1. `ThisAddIn_StartUp` 메서드 끝에 다음 코드를 추가하여 Excel 통합 문서에 대한 링크가 포함된 Word 문서를 만듭니다.

```
var wordApp = new Word.Application();
wordApp.Visible = true;
wordApp.Documents.Add();
wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
```

```
Dim wordApp As New Word.Application
wordApp.Visible = True
wordApp.Documents.Add()
wordApp.Selection.PasteSpecial(Link:=True, DisplayAsIcon:=True)
```

이 코드는 COM 프로그래밍에서 `ref` 키워드를 생략하는 기능, 명명된 인수, 선택적 인수 등 다양한 C#의 새로운 기능을 보여 줍니다. Visual Basic에는 이러한 기능이 이미 포함되어 있습니다. `PasteSpecial` 메서드에는 모두 선택적 참조 매개 변수로 정의되는 7개의 매개 변수가 있습니다. 명명된 인수와 선택적 인수를 사용하면 액세스 할 매개 변수를 이름으로 지정하고 해당 매개 변수에만 인수를 보낼 수 있습니다. 이 예제에서는 클립보드에 통합 문서 링크를 만들어야 하고(`Link` 매개 변수) Word 문서에 링크를 아이콘

으로 표시하도록(`DisplayAsIcon` 매개 변수) 지정하는 인수를 전송합니다. Visual C#에서는 이러한 인수의 `ref` 키워드를 생략할 수도 있습니다.

#### 애플리케이션을 실행하려면

1. 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 프로젝트를 정리하려면

1. Visual Studio의 빌드 메뉴에서 솔루션 정리를 클릭합니다. 이렇게 하지 않으면 컴퓨터에서 Excel을 열 때마다 추가 기능이 실행됩니다.

#### PIA 참조를 찾으려면

1. 애플리케이션을 다시 실행하되 솔루션 정리를 클릭하지는 않습니다.
2. 시작을 선택합니다. **Microsoft Visual Studio <version>** 을 찾은 다음 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 > 열기를 선택합니다. **Visual Studio <version>** 을 두 번 클릭한 다음 프로젝트를 두 번 클릭합니다. 프로젝트 폴더를 열고 프로젝트 이름.dll에서 bin/Debug 폴더를 확인한 후 프로젝트 이름.dll을 두 번 클릭합니다. 새 창에 프로젝트 특성과 기타 모듈 및 어셈블리에 대한 참조가 표시됩니다. 어셈블리에는 `Microsoft.Office.Interop.Excel` 및 `Microsoft.Office.Interop.Word` 네임스페이스가 포함되어 있습니다. 기본적으로 Visual Studio에서 컴파일하는 필요한 형식을 참조된 PIA에서 어셈블리로 가져옵니다.

자세한 내용은 [방법: 어셈블리 콘텐츠 보기](#)를 참조하세요.

5. MANIFEST 아이콘을 두 번 클릭합니다. 프로젝트가 참조하는 항목이 들어 있는 어셈블리 목록이 포함된 창이 표시됩니다. `Microsoft.Office.Interop.Excel` 및 `Microsoft.Office.Interop.Word` 는 목록에 포함되어 있지 않습니다. 프로젝트에 필요한 형식을 어셈블리로 가져왔기 때문에 PIA에 대한 참조는 필요하지 않으므로 배포를 손쉽게 수행할 수 있습니다. PIA는 사용자의 컴퓨터에 없어도 되며 애플리케이션에서 특정 PIA 버전을 배포할 필요가 없으므로 필요한 PIA가 모든 버전에 있다면 여러 Office 버전에서 사용하도록 애플리케이션을 설계할 수 있습니다.

PIA를 배포할 필요가 없으므로 이전 버전을 비롯한 여러 Office 버전에서 사용 가능한 애플리케이션을 고급 시나리오에서 만들 수 있습니다. 그러나 사용 중인 Office 버전에서 제공되지 않는 API를 코드에서 사용하지 않는 경우에만 이러한 방식이 작동합니다. 특정 API가 이전 버전에서 제공되었는지 여부를 항상 명확하게 파악할 수 있는 것은 아니므로 이전 버전의 Office는 사용하지 않는 것이 좋습니다.

#### NOTE

Office 2003 이전 버전에서는 PIA가 게시되지 않았습니다. 그러므로 Office 2002 이하 버전에 대해 interop 어셈블리를 생성하려면 COM 참조를 가져와야 합니다.

6. 매니페스트 창과 어셈블리 창을 닫습니다.

#### PIA 종속성을 복원하려면

1. 솔루션 탐색기에서 모든 파일 표시 단추를 클릭합니다. References 폴더를 선택한 다음 `Microsoft.Office.Interop.Excel`을 선택합니다. F4 키를 눌러 속성 창을 표시합니다.
2. 속성 창에서 **Interop 형식 포함 속성**을 **True**에서 **False**로 변경합니다.
3. `Microsoft.Office.Interop.Word`에 대해 이 절차의 1-2단계를 반복합니다.

4. C#에서 `Autofit` 메서드 끝의 `DisplayInExcel`에 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.Excel`이 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.Excel`이 모두 목록에 있음을 확인할 수 있습니다. 애플리케이션에서 Excel 및 Word PIA를 참조하며 **Interop 형식 포함** 속성이 **False**로 설정되어 있으므로 두 어셈블리가 모두 최종 사용자의 컴퓨터에 있어야 합니다.
8. Visual Studio의 **빌드** 메뉴에서 **솔루션 정리**를 클릭하여 완성된 프로젝트를 정리합니다.

## 참조

- [자동으로 구현된 속성\(Visual Basic\)](#)
- [자동으로 구현된 속성\(C#\)](#)
- [컬렉션 이니셜라이저](#)
- [개체 이니셜라이저 및 컬렉션 이니셜라이저](#)
- [선택적 매개 변수](#)
- [위치 및 이름으로 인수 전달](#)
- [명명된 인수 및 선택적 인수](#)
- [초기 바인딩 및 런타임에 바인딩](#)
- [dynamic](#)
- [dynamic 형식 사용](#)
- [람다 식\(Visual Basic\)](#)
- [람다 식\(C#\)](#)
- [COM interop 프로그래밍에서 인덱싱된 속성을 사용하는 방법](#)
- [연습: Visual Studio에서 Microsoft Office 어셈블리의 형식 정보 포함](#)
- [연습: 관리되는 어셈블리의 형식 포함](#)
- [연습: Excel용 첫 VSTO 추가 기능 만들기](#)
- [COM Interop](#)
- [상호 운용성](#)

# COM 클래스 예제(C# 프로그래밍 가이드)

2020-11-02 • 4 minutes to read • [Edit Online](#)

다음은 COM 개체로 노출되는 클래스의 예제입니다. 이 코드를 .cs 파일에 배치하고 프로젝트에 추가한 후 **COM Interop** 등록 속성을 **True**로 설정합니다. 자세한 내용은 [방법: COM Interop에 대한 구성 요소 등록을 참조하세요.](#)

Visual 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 인터페이스에 선언되지 않은 클래스의 다른 **public** 멤버는 COM에 표시되지 않지만 다른 .NET 개체에 표시됩니다.

속성 및 메서드를 COM에 노출하려면 클래스 인터페이스에서 선언하고 **DispId** 특성을 사용하여 표시한 후 클래스에서 구현해야 합니다. 멤버가 인터페이스에서 선언되는 순서는 COM vtable에 사용되는 순서입니다.

클래스에서 이벤트를 노출하려면 이벤트 인터페이스에서 선언하고 **DispId** 특성을 사용하여 표시해야 합니다. 클래스는 이 인터페이스를 구현하지 않아야 합니다.

클래스는 클래스 인터페이스를 구현합니다. 둘 이상의 인터페이스를 구현할 수 있지만 첫 번째 구현이 기본 클래스 인터페이스가 됩니다. 여기에서 COM에 노출된 메서드 및 속성을 구현합니다. 이러한 메서드 및 속성은 **public**으로 표시되어야 하며 클래스 인터페이스의 선언과 일치해야 합니다. 또한 여기에서 클래스에 의해 발생된 이벤트를 선언합니다. 이러한 이벤트는 **public**으로 표시되어야 하며 이벤트 인터페이스의 선언과 일치해야 합니다.

## 예제

```
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
    {
    }
}
```

## 참고 항목

- [C# 프로그래밍 가이드](#)
- [상호 운용성](#)
- [프로젝트 디자이너, 빌드 페이지\(C#\)](#)

# C# 참고자료

2021-02-18 • 10 minutes to read • [Edit Online](#)

이 섹션에서는 C# 키워드, 연산자, 특수 문자, 전처리기 지시문, 컴파일러 옵션 및 컴파일러 오류와 경고에 대한 참조 자료를 제공합니다.

## 단원 내용

### [C# 키워드](#)

C# 키워드 및 구문에 대한 정보 링크를 제공합니다.

### [C# 연산자](#)

C# 연산자 및 구문에 대한 정보 링크를 제공합니다.

### [C# 특수 문자](#)

C#의 특수한 상황에 맞는 문자 및 용도에 대한 정보로 연결되는 링크를 제공합니다.

### [C# 전처리기 지시문](#)

C# 소스 코드에 포함할 컴파일러 명령에 대한 정보 링크를 제공합니다.

### [C# 컴파일러 옵션](#)

컴파일러 옵션 및 사용 방법에 대한 정보를 포함합니다.

### [C# 컴파일러 오류](#)

C# 컴파일러 오류 및 경고의 원인과 해결 방법을 보여 주는 코드 조각을 포함합니다.

### [C# 언어 사양](#)

C# 6.0 언어 사양. C# 6.0 언어에 대한 초안 제안입니다. 이 문서는 ECMA C# 표준 위원회와의 협력을 통해 구체화될 예정입니다. 버전 5.0은 2017년 12월에 [표준 ECMA-334 다섯 번째 버전](#) 문서로 릴리스되었습니다.

6.0 이후 C# 버전에서 구현된 기능은 언어 사양 제안에 나타납니다. 이 문서는 이러한 새 기능을 추가하기 위해 언어 사양에 대한 델타를 설명합니다. 이는 초안 제안 양식입니다. 이러한 사양은 구체화되어 공식적인 검토를 위해 ECMA 표준 위원회에 제출되며, 이후 버전의 C# 표준으로 통합됩니다.

### [C# 7.0 사양 제안](#)

C# 7.0에서 여러 가지 새로운 기능이 구현되었습니다. 새로운 기능에는 패턴 일치, 로컬 함수, 변수 선언, throw 식, 이진 리터럴 및 숫자 구분 기호가 포함됩니다. 이 풀더에는 각 기능에 대한 사양이 있습니다.

### [C# 7.1 사양 제안](#)

C# 7.1에 새로운 기능이 추가되었습니다. 먼저 `Task` 또는 `Task<int>`를 반환하는 `Main` 메서드를 작성할 수 있습니다. 이렇게 하면 `async` 한정자를 `Main`에 추가할 수 있습니다. `default` 식은 형식을 유추할 수 있는 위치에 형식 없이 사용할 수 있습니다. 또한 튜플 멤버 이름을 유추할 수 있습니다. 마지막으로, 패턴 일치는 제네릭과 함께 사용할 수 있습니다.

### [C# 7.2 사양 제안](#)

C# 7.2에 여러 작은 기능을 추가했습니다. `in` 키워드를 사용하여 읽기 전용 참조로 인수를 전달할 수 있습니다. `Span` 및 관련 유형에 대한 컴파일 시간 안정성을 지원하기 위한 여러 가지 하위 수준 변경 내용이 있습니다. 일부 상황에서는 나중에 인수가 위치할 때 명명된 인수를 사용할 수 있습니다. `private protected` 액세스 한정자를 사용하면 호출자가 동일한 어셈블리에 구현된 파생 형식으로 제한되도록 지정할 수 있습니다. `:<` 연산자는 변수에 대한 참조로 확인할 수 있습니다. 선행 숫자 구분 기호를 사용하여 16진수 및 이진 숫자를 포맷할 수도 있습니다.

### [C# 7.3 사양 제안](#)

C# 7.3은 몇 가지 작은 업데이트를 포함하는 또 다른 포인트 릴리스입니다. 제네릭 형식 매개 변수에 대해 새 제약 조건을 사용할 수 있습니다. 다른 변경 내용은 `stackalloc` 할당 사용을 포함하여 `fixed` 필드로 작업하는 것을 더 쉽게 만듭니다. `ref` 키워드로 선언된 로컬 변수는 새 스토리지를 참조하도록 다시 할당할 수 있습니다. 컴파일러에서 생성된 지원 필드를 대상으로 하는 자동 구현 속성에 특성을 배치할 수 있습니다. 식 변수는 이니

설라이저에서 사용할 수 있습니다. 튜플은 같음(또는 같지 않음)을 비교할 수 있습니다. 오버로드 확인에도 일부 개선이 있습니다.

#### C# 8.0 사양 제안

C# 8.0은 .NET Core 3.0에서 사용할 수 있습니다. 이 기능에는 nullable 참조 형식, 재귀 패턴 일치, 기본 인터페이스 메서드, 비동기 스트림, 범위 및 인덱스, 선언을 사용한 패턴, null 병합 할당 및 읽기 전용 인스턴스 멤버가 포함됩니다.

#### C# 9.0 사양 제안

C# 9.0은 .NET 5.0에서 사용할 수 있습니다. 해당 기능으로는 레코드, 최상위 문, 패턴 일치 향상, init 전용 setter, 대상으로 형식화된 새로운 식, 모듈 이니셜라이저, 부분 메서드 확장, 정적 익명 함수, 대상으로 형식화된 조건식, 공변 반환 형식, foreach 루프의 확장 GetEnumerator, 람다 무시 항목 매개 변수, 로컬 함수의 특성, 원시 크기 정수, 함수 포인터, localsinit 플래그 내보내기 표시 안 함, 무제한 형식 매개 변수 주석 등이 있습니다.

## 관련 단원

### C#용 Visual Studio 개발 환경 사용

IDE 및 편집기를 설명하는 개념 및 작업 항목의 링크를 제공합니다.

### C# 프로그래밍 가이드

C# 프로그래밍 언어를 사용하는 방법에 대한 정보를 포함합니다.

# C# 언어 버전 관리

2021-02-18 • 13 minutes to read • [Edit Online](#)

최신 C# 컴파일러는 프로젝트의 대상 프레임워크를 기반으로 기본 언어 버전을 결정합니다. Visual Studio는 이 값을 변경하는 UI를 제공하지 않지만, `csproj` 파일을 편집하여 값을 변경할 수 있습니다. 기본값이 이렇게 선택되면 항상 대상 프레임워크와 호환되는 최신 언어 버전을 사용할 수 있게 됩니다. 프로젝트의 대상과 호환되는 최신 언어 기능을 이용할 수 있다는 이점이 있습니다. 또한, 기본값이 이렇게 선택되면 대상 프레임워크에서 사용할 수 없는 형식이나 런타임 동작이 필요한 언어를 사용하지 않을 수 있습니다. 기본값보다 최신의 언어 버전을 선택할 경우 컴파일 시간 및 런타임 오류를 진단하기 어려워질 수 있습니다.

이 문서의 규칙은 Visual Studio 2019 또는 .NET SDK와 함께 제공되는 컴파일러에 적용됩니다. Visual Studio 2017 설치 또는 이전 .NET Core SDK 버전의 일부인 C# 컴파일러는 기본적으로 C# 7.0을 대상으로 합니다.

C# 8.0은 .NET Core 3.x 이상 버전에서만 지원됩니다. 대부분의 최신 기능에는 .NET Core 3.x에서 도입된 라이브러리 및 런타임 기능이 필요합니다.

- [기본 인터페이스 구현](#)에는 .NET Core 3.0 CLR의 새로운 기능이 필요합니다.
- [비동기 스트림](#)에는 새로운 형식인 `System.IAsyncDisposable`, `System.Collections.Generic.IAsyncEnumerable<T>` 및 `System.Collections.Generic.IAsyncEnumerator<T>`이 필요합니다.
- [인덱스 및 범위](#)에는 새로운 형식인 `System.Index` 및 `System.Range`이 필요합니다.
- [Null 허용 참조 형식](#)은 더 효과적인 경고를 제공하기 위해 몇 가지 [특성](#)을 사용하는데, 이러한 특성은 .NET Core 3.0에서 추가되었습니다. 다른 대상 프레임워크는 이러한 특성으로 주석이 추가되지 않았습니다. 즉, null 허용 경고가 잠재적인 문제를 정확하게 반영하지 못할 수 있습니다.

C# 9.0은 .NET 5 이상 버전에서만 지원됩니다.

## 기본값

컴파일러는 다음 규칙에 따라 기본값을 결정합니다.

대상 프레임워크	버전	C# 언어 버전 기본값
.NET	5.x	C# 9.0
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3
.NET Framework	모두	C# 7.3

프로젝트가 해당 미리 보기 언어 버전이 있는 미리 보기 프레임워크를 대상으로 하는 경우 사용되는 언어 버전은 미리 보기 언어 버전입니다. 따라서 릴리스된 .NET Core 버전을 대상으로 하는 프로젝트에 영향을 주지 않으면서 모든 환경에서 해당 미리 보기의 최신 기능을 사용할 수 있습니다.

## IMPORTANT

Visual Studio 2017은 만든 모든 프로젝트 파일에 `<LangVersion>latest</LangVersion>` 항목을 추가했습니다. 즉, C# 7.0이 추가됐습니다. 하지만 Visual Studio 2019로 업그레이드하면 대상 프레임워크에 관계없이 릴리스된 최신 버전이 추가됩니다. 이러한 프로젝트는 이제 기본 동작을 재정의합니다. 프로젝트 파일을 편집하여 해당 노드를 제거해야 합니다. 그러면 프로젝트에서 대상 프레임워크에 권장되는 컴파일러 버전이 사용됩니다.

## 기본값 재정의

C# 버전을 명시적으로 지정해야 하는 경우 다음과 같은 여러 가지 방법으로 수행할 수 있습니다.

- [프로젝트 파일](#)을 수동으로 편집합니다.
- 하위 디렉터리에 있는 여러 [프로젝트의](#) 언어 버전을 설정합니다.
- `-langversion` 컴파일러 옵션을 구성합니다.

## TIP

현재 사용 중인 언어 버전을 확인하려면 코드에 `#error version` (대/소문자 구분)을 입력합니다. 이렇게 하면 컴파일러가 사용 중인 컴파일러 버전 및 현재 선택된 언어 버전을 포함하는 메시지가 있는 진단 CS8304를 생성합니다.

### 프로젝트 파일 편집

프로젝트 파일에서 언어 버전을 설정할 수 있습니다. 예를 들어 미리 보기 기능에 명시적으로 액세스하려는 경우 다음과 같은 요소를 추가합니다.

```
<PropertyGroup>
  <LangVersion>preview</LangVersion>
</PropertyGroup>
```

값 `preview`는 컴파일러에서 지원하는 사용 가능한 최신 미리 보기 C# 언어를 사용합니다.

### 여러 프로젝트 구성

여러 프로젝트를 구성하려면 `<LangVersion>` 요소를 포함하는 `Directory.Build.props` 파일을 만들 수 있습니다. 일반적으로 솔루션 디렉터리에서 이 작업을 수행합니다. 솔루션 디렉터리의 `Directory.Build.props` 파일에 다음을 추가합니다.

```
<Project>
  <PropertyGroup>
    <LangVersion>preview</LangVersion>
  </PropertyGroup>
</Project>
```

해당 파일을 포함하는 디렉터리의 모든 하위 디렉터리에 있는 빌드는 미리 보기 C# 버전을 사용합니다. 자세한 내용은 [빌드 사용자 지정](#)에 대한 문서를 참조하세요.

## C# 언어 버전 참조

모든 표는 현재 C# 언어 버전을 보여줍니다. 컴파일러가 오래된 것이라면 일부 값을 파악하지 못할 수 있습니다. 최신 .NET SDK를 설치하는 경우 나열된 모든 항목에 액세스할 수 있습니다.

값	의미
latest	Visual Studio 2019에서 지원하는 최신 언어 버전입니다.
current	Visual Studio 2019에서 지원하는 최신 언어 버전입니다.
preview	Visual Studio 2019에서 지원하는 미리 보기 언어 버전입니다.
version 8.0	C# 8.0 언어 버전입니다.
version 7.3	C# 7.3 언어 버전입니다.
version 7.2	C# 7.2 언어 버전입니다.
version 7.1	C# 7.1 언어 버전입니다.
version 7.0	C# 7.0 언어 버전입니다.
version 6.0	C# 6.0 언어 버전입니다.
version 5.0	C# 5.0 언어 버전입니다.
version 4.6	C# 4.6 언어 버전입니다.
version 4.5	C# 4.5 언어 버전입니다.
version 4.0	C# 4.0 언어 버전입니다.
version 3.5	C# 3.5 언어 버전입니다.
version 3.0	C# 3.0 언어 버전입니다.
version 2.0	C# 2.0 언어 버전입니다.
version 1.0	C# 1.0 언어 버전입니다.

값	의미
<code>preview</code>	컴파일러가 최신 미리 보기 버전의 유효한 언어 구문을 모두 허용합니다.
<code>latest</code>	컴파일러가 최신 릴리스 버전(부 버전 포함)의 구문을 허용합니다.
<code>latestMajor</code> ( <code>default</code> )	컴파일러가 최신 릴리스 주 버전의 구문을 허용합니다.
<code>9.0</code>	컴파일러는 C# 9.0 이하에 포함된 구문만 허용합니다.
<code>8.0</code>	컴파일러는 C# 8.0 이하에 포함된 구문만 허용합니다.
<code>7.3</code>	컴파일러는 C# 7.3 이하에 포함된 구문만 허용합니다.
<code>7.2</code>	컴파일러는 C# 7.2 이하에 포함된 구문만 허용합니다.
<code>7.1</code>	컴파일러는 C# 7.1 이하에 포함된 구문만 허용합니다.
<code>7</code>	컴파일러는 C# 7.0 이하에 포함된 구문만 허용합니다.
<code>6</code>	컴파일러는 C# 6.0 이하에 포함된 구문만 허용합니다.
<code>5</code>	컴파일러는 C# 5.0 이하에 포함된 구문만 허용합니다.
<code>4</code>	컴파일러는 C# 4.0 이하에 포함된 구문만 허용합니다.
<code>3</code>	컴파일러는 C# 3.0 이하에 포함된 구문만 허용합니다.
<code>ISO-2</code> (또는 <code>2</code> )	컴파일러는 ISO/IEC 23270:2006 C#(2.0)에 포함된 구문만 허용합니다.
<code>ISO-1</code> (또는 <code>1</code> )	컴파일러는 ISO/IEC 23270:2003 C#(1.0/1.2)에 포함된 구문만 허용합니다.

**TIP**

Visual Studio용 개발자 명령 프롬프트를 열고 다음 명령을 실행하여 컴퓨터에서 사용할 수 있는 언어 버전의 목록을 확인합니다.

```
csc -langversion:?
```

이와 같이 `-langversion` compile 옵션을 질문하면 다음과 같은 내용이 출력됩니다.

```
Supported language versions:  
default  
1  
2  
3  
4  
5  
6  
7.0  
7.1  
7.2  
7.3  
8.0  
9.0 (default)  
latestmajor  
preview  
latest
```

# 값 형식(C# 참조)

2021-02-18 • 8 minutes to read • [Edit Online](#)

값 형식 및 참조 형식은 C# 형식의 두 가지 주요 범주입니다. 값 형식의 변수에는 형식의 인스턴스가 포함되어 있습니다. 이는 형식 인스턴스에 대한 참조를 포함하는 참조 형식의 변수와 다릅니다. 기본적으로 변수 값은 [할당](#) 시에, 인수를 메서드에 전달할 때, 그리고 메서드 결과를 반환할 때 복사됩니다. 값 형식 변수의 경우 해당 형식 인스턴스가 복사됩니다. 다음 예제에서는 해당 동작을 보여줍니다.

```
using System;

public struct MutablePoint
{
    public int X;
    public int Y;

    public MutablePoint(int x, int y) => (X, Y) = (x, y);

    public override string ToString() => $"({X}, {Y})";
}

public class Program
{
    public static void Main()
    {
        var p1 = new MutablePoint(1, 2);
        var p2 = p1;
        p2.Y = 200;
        Console.WriteLine($"{nameof(p1)} after {nameof(p2)} is modified: {p1}");
        Console.WriteLine($"{nameof(p2)}: {p2}");

        MutateAndDisplay(p2);
        Console.WriteLine($"{nameof(p2)} after passing to a method: {p2}");
    }

    private static void MutateAndDisplay(MutablePoint p)
    {
        p.X = 100;
        Console.WriteLine($"Point mutated in a method: {p}");
    }
}

// Expected output:
// p1 after p2 is modified: (1, 2)
// p2: (1, 200)
// Point mutated in a method: (100, 200)
// p2 after passing to a method: (1, 200)
```

앞의 예제와 같이 값 형식 변수에 대한 작업은 변수에 저장된 값 형식의 인스턴스에만 영향을 줍니다.

값 형식이 참조 형식의 데이터 구성원을 포함하는 경우에는 값 형식 인스턴스가 복사될 때 참조 형식의 인스턴스에 대한 참조만 복사됩니다. 복사 및 원래 값 형식 인스턴스는 모두 동일한 참조 형식 인스턴스에 액세스할 수 있습니다. 다음 예제에서는 해당 동작을 보여줍니다.

```

using System;
using System.Collections.Generic;

public struct TaggedInteger
{
    public int Number;
    private List<string> tags;

    public TaggedInteger(int n)
    {
        Number = n;
        tags = new List<string>();
    }

    public void AddTag(string tag) => tags.Add(tag);

    public override string ToString() => $"{Number} [{string.Join(", ", tags)}]";
}

public class Program
{
    public static void Main()
    {
        var n1 = new TaggedInteger(0);
        n1.AddTag("A");
        Console.WriteLine(n1); // output: 0 [A]

        var n2 = n1;
        n2.Number = 7;
        n2.AddTag("B");

        Console.WriteLine(n1); // output: 0 [A, B]
        Console.WriteLine(n2); // output: 7 [A, B]
    }
}

```

#### NOTE

코드를 오류가 덜 발생하고 더 강력하게 만들려면 변경할 수 없는 값 형식을 정의 및 사용합니다. 이 문서에서는 데모용으로만 변경 가능한 값 형식을 사용합니다.

## 값 형식 및 형식 제약 조건의 종류

값 형식은 다음 두 가지 중 하나일 수 있습니다.

- 데이터 및 관련 기능을 캡슐화하는 [구조 형식](#)
- 명명된 상수 집합으로 정의되고 선택 사항 또는 선택 사항의 조합을 나타내는 [열거 형식](#)

기본 값 형식 `T`의 모든 값과 추가 `Null` 값을 나타내는 [값 형식](#) `T?` Null 허용 값 형식인 경우를 제외하고는 값 형식의 변수에 `null`을 할당할 수 없습니다.

`struct` 제약 조건을 사용하여 형식 매개 변수가 `null`을 허용하지 않는 값 형식이라고 지정할 수 있습니다. 구조체 형식과 열거형 형식 모두 `struct` 제약 조건을 충족합니다. C# 7.3부터 기본 클래스 제약 조건([열거형 제약 조건](#))이라고 함)에서 `System.Enum`을 사용하여 형식 매개 변수가 열거형 형식이라고 지정할 수 있습니다.

## 기본 제공 값 형식

C#은 단순 형식이라고도 하는 다음과 같은 기본 제공 값 형식을 제공합니다.

- [정수 숫자 형식](#)

- [부동 소수점 숫자 형식](#)
- [부울 값을 나타내는 bool](#)
- [유니코드 UTF-16 문자를 나타내는 문자](#)

모든 단순 형식은 구조체 형식이며, 특정 추가 작업을 허용한다는 점에서 다른 구조체 형식과 다릅니다.

- 리터럴을 사용하여 단순 형식의 값을 제공할 수 있습니다. 예를 들어 `'A'`는 `char` 형식의 리터럴이고, `2001`은 `int` 형식의 리터럴입니다.
  - `const` 키워드를 사용하여 단순 형식의 상수를 선언할 수 있습니다. 다른 구조체 형식의 상수는 포함할 수 없습니다.
  - 해당 피연산자가 모두 단순 형식의 상수인 상수 식은 컴파일 시간에 계산됩니다.
- C# 7.0부터는 C#에서 [값 튜플](#)을 지원합니다. 값 튜플은 단순 형식이 아닌 값 형식입니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 다음 섹션을 참조하세요.

- [값 형식](#)
- [단순 형식](#)
- [변수](#)

## 참고 항목

- [C# 참조](#)
- [System.ValueType](#)
- [참조 형식](#)

# 정수 숫자 형식(C# 참조)

2020-05-02 • 8 minutes to read • [Edit Online](#)

정수 숫자 형식은 정수를 표현합니다. 모든 정수 숫자 형식은 [값 형식](#)입니다. 이것은 [기본 형식](#)이기도 하며, [리터럴](#)로 초기화할 수 있습니다. 모든 정수 숫자 형식은 [산술](#), [비트 논리](#), [비교](#) 및 [같음](#) 연산자를 지원합니다.

## 정수 형식의 특성

C#은 다음과 같은 미리 정의된 정수 형식을 지원합니다.

C# 형식/키워드	범위	SIZE	.NET 형식
<code>sbyte</code>	-128 ~ 127	부호 있는 8비트 정수	<a href="#">System.SByte</a>
<code>byte</code>	0 ~ 255	부호 없는 8비트 정수	<a href="#">System.Byte</a>
<code>short</code>	-32,768 ~ 32,767	부호 있는 16비트 정수	<a href="#">System.Int16</a>
<code>ushort</code>	0 ~ 65,535	부호 없는 16비트 정수	<a href="#">System.UInt16</a>
<code>int</code>	-2,147,483,648 ~ 2,147,483,647	부호 있는 32비트 정수	<a href="#">System.Int32</a>
<code>uint</code>	0 ~ 4,294,967,295	부호 없는 32비트 정수	<a href="#">System.UInt32</a>
<code>long</code>	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807	부호 있는 64비트 정수	<a href="#">System.Int64</a>
<code>ulong</code>	0 ~ 18,446,744,073,709,551,615	부호 없는 64비트 정수	<a href="#">System.UInt64</a>

이전 표에서 맨 왼쪽 열의 각 C# 형식 키워드는 해당하는 .NET 형식의 별칭입니다. 서로 교환하여 사용할 수 있습니다. 예를 들어 다음 선언은 동일한 형식의 변수를 선언합니다.

```
int a = 123;
System.Int32 b = 123;
```

각 정수 형식의 기본값은 `0`입니다. 각 정수 형식에는 해당 형식의 최솟값과 최댓값을 제공하는 `MinValue` 및 `MaxValue` 상수가 있습니다.

[System.Numerics.BigInteger](#) 구조체를 사용하여 상한 또는 하한이 없는 부호 있는 정수를 나타냅니다.

## 정수 리터럴

정수 리터럴은 다음 형식일 수 있습니다.

- **10진**: 접두사가 없음
- **16진수**: `0x` 또는 `0X` 접두사 사용
- **0/진**: `0b` 또는 `0B` 접두사 사용(C# 7.0 및 이후 버전에서 가능)

다음 코드에서는 각 예제를 보여 줍니다.

```
var decimalLiteral = 42;
var hexLiteral = 0x2A;
var binaryLiteral = 0b_0010_1010;
```

앞의 예제에서는 C# 7.0부터 지원되는 숫자 구분 기호인 `_`를 사용하는 방법도 보여 줍니다. 모든 종류의 숫자 리터럴에서 숫자 구분 기호를 사용할 수 있습니다.

정수 리터럴의 형식은 접미사로 다음과 같이 결정됩니다.

- 리터럴에 접미사가 없는 경우 해당 형식은 값이 표현될 수 있는 `int`, `uint`, `long`, `ulong` 형식 중 첫 번째 형식입니다.
- 리터럴에 접미사가 `u` 또는 `U`인 경우 해당 형식은 값이 표현될 수 있는 `uint`, `ulong` 형식 중 첫 번째 형식입니다.
- 리터럴에 접미사가 `L` 또는 `l`인 경우 해당 형식은 값이 표현될 수 있는 `long`, `ulong` 형식 중 첫 번째 형식입니다.

#### NOTE

소문자 `l`을 접미사로 사용할 수 있습니다. 그러나 이렇게 하면 문자 `l` 및 숫자 `1`을 혼동하기 쉬우므로 컴파일러 경고가 생성됩니다. 쉽게 구별할 수 있도록 `L`을 사용합니다.

- 리터럴의 접미사가 `UL`, `Ul`, `uL`, `ul`, `LU`, `Lu`, `lu` 또는 `lu`이면 해당 형식은 `ulong`입니다.

정수 리터럴로 표시되는 값이 `UInt64.MaxValue`를 초과하면 컴파일 오류 [CS1021](#)이 발생합니다.

결정된 정수 리터럴 형식이 `int`이고 리터럴이 나타내는 값이 대상 형식의 범위 내에 있는 경우, 해당 값이 암시적으로 `sbyte`, `byte`, `short`, `ushort`, `uint` 또는 `ulong`으로 변환될 수 있습니다.

```
byte a = 17;
byte b = 300; // CS0031: Constant value '300' cannot be converted to a 'byte'
```

앞의 예제에서 볼 수 있듯이 리터럴 값이 대상 형식의 범위 내에 있지 않으면 컴파일러 오류 [CS0031](#)이 발생합니다.

캐스트를 사용하여 정수 리터럴로 표시되는 값을 결정된 리터럴 형식 이외의 형식으로 변환할 수도 있습니다.

```
var signedByte = (sbyte)42;
var longVariable = (long)42;
```

## 변환

모든 정수 형식을 다른 정수 형식으로 변환할 수 있습니다. 대상 형식이 소스 형식의 모든 값을 저장할 수 있는 경우 변환은 암시적입니다. 저장할 수 없는 경우에는 [캐스트 식](#)을 사용하여 명시적 변환을 수행해야 합니다. 자세한 내용은 [기본 제공 숫자 변환](#)을 참조하세요.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 다음 섹션을 참조하세요.

- [정수 형식](#)
- [정수 리터럴](#)

## 참조

- [C# 참조](#)
- [값 형식](#)
- [부동 소수점 형식](#)
- [표준 숫자 형식 문자열](#)
- [.NET의 숫자](#)

# 부동 소수점 숫자 형식(C# 참조)

2021-02-18 • 11 minutes to read • [Edit Online](#)

부동 소수점 숫자 형식은 실수를 나타냅니다. 모든 부동 소수점 숫자 형식은 [값 형식](#)입니다. 이것은 [기본 형식](#)이기도 하며, [리터럴](#)로 초기화할 수 있습니다. 모든 부동 소수점 숫자 형식은 [산술](#), [비교](#) 및 [같음](#) 연산자를 지원합니다.

## 부동 소수점 형식의 특성

C#은 다음과 같은 미리 정의된 부동 소수점 형식을 지원합니다.

C# 형식/키워드	근사 범위	전체 자릿수	SIZE	.NET 형식
<code>float</code>	$\pm 1.5 \times 10^{-45} \sim \pm 3.4 \times 10^{38}$	~6-9개 자릿수	4바이트	<a href="#">System.Single</a>
<code>double</code>	$\pm 5.0 \times 10^{-324} \sim \pm 1.7 \times 10^{308}$	~15-17개 자릿수	8바이트	<a href="#">System.Double</a>
<code>decimal</code>	$\pm 1.0 \times 10^{-28} \sim \pm 7.9228 \times 10^{28}$	28-29개의 자릿수	16바이트	<a href="#">System.Decimal</a>

이전 표에서 맨 왼쪽 열의 각 C# 형식 키워드는 해당하는 .NET 형식의 별칭입니다. 서로 교환하여 사용할 수 있습니다. 예를 들어 다음 선언은 동일한 형식의 변수를 선언합니다.

```
double a = 12.3;  
System.Double b = 12.3;
```

각 부동 소수점 형식의 기본값은 `0`입니다. 각 부동 소수점 형식에는 해당 형식의 최소 및 최대 유한값을 제공하는 `MinValue` 및 `.MaxValue` 상수가 있습니다. 또한 `float` 및 `double` 형식은 숫자가 아닌 무한 값을 나타내는 상수를 제공합니다. 예를 들어 `double` 형식은 [Double.NaN](#), [Double.NegativeInfinity](#) 및 [Double.PositiveInfinity](#)와 같은 상수를 제공합니다.

필요한 전체 자릿수를 소수점 이하 자릿수에 따라 결정하는 경우에는 `decimal` 형식이 적합합니다. 이러한 숫자는 일반적으로 재무 애플리케이션에서 통화 금액(예: \$1.00), 이자율(예: 2.625%) 등에 사용됩니다. 소수점 한 자리 숫자인 경우에도 `decimal` 형식에서 더 정확하게 처리됩니다. 예를 들어 0.1은 `decimal` 인스턴스로 정확하게 표현될 수 있지만 0.1을 정확히 표현하는 `double` 또는 `float` 인스턴스는 없습니다. 10진 데이터에 `double` 또는 `float`를 사용하는 경우 숫자 형식의 차이로 인해 산술 계산에서 예기치 않은 반올림 오류가 발생할 수 있습니다. 성능 최적화가 정확성을 보장하는 것보다 중요한 경우 `decimal` 대신 `double`을 사용할 수 있습니다. 그러나 성능 차이는 계산 집약적인 애플리케이션을 제외한 모든 애플리케이션에서 알지 못합니다. `decimal`을 사용하지 않는 또 다른 가능한 이유는 스토리지 요구 사항을 최소화하는 것입니다. 예를 들어 [ML.NET](#)은 `float`를 사용합니다. 규모가 매우 큰 데이터 세트에서는 4바이트와 16바이트의 차이가 증폭되기 때문입니다. 자세한 내용은 [System.Decimal](#)을 참조하세요.

식에서 [정수](#) 형식과 `float` 및 `double` 형식을 혼합할 수 있습니다. 이 경우 정수 형식이 암시적으로 부동 소수점 형식 중 하나로 변환되며, 필요한 경우 `float` 형식이 암시적으로 `double`로 변환됩니다. 이 식은 다음과 같이 계산됩니다.

- 식에 `double` 형식이 있는 경우 식은 관계형 및 같음 비교에서 `double` 또는 `bool`로 계산됩니다.
- 식에 `double` 형식이 없는 경우 식은 관계형 및 같음 비교에서 `float` 또는 `bool`로 계산됩니다.

식에서 정수 형식과 `decimal` 형식을 혼합할 수도 있습니다. 이 경우 정수 형식은 암시적으로 `decimal` 형식으로 변환되고 식은 관계형 및 같음 비교에서 `decimal` 또는 `bool`로 계산됩니다.

식에서 `decimal` 형식을 `float` 및 `double` 형식과 혼합할 수 없습니다. 이 경우 산술, 비교 또는 같음 연산을 수행 하려면 다음 예제와 같이 명시적으로 피연산자를 `decimal` 형식으로 변환하거나 반대로 변환해야 합니다.

```
double a = 1.0;
decimal b = 2.1m;
Console.WriteLine(a + (double)b);
Console.WriteLine((decimal)a + b);
```

표준 숫자 서식 문자열 또는 사용자 지정 숫자 서식 문자열을 사용하여 부동 소수점 값의 형식을 지정할 수 있습니다.

## real 리터럴

real 리터럴의 형식은 접미사로 다음과 같이 결정됩니다.

- 접미사가 없거나 `d` 또는 `D` 접미사가 있는 리터럴은 `double` 형식입니다.
- `f` 또는 `F` 접미사가 있는 리터럴은 `float` 형식입니다.
- `m` 또는 `M` 접미사가 있는 리터럴은 `decimal` 형식입니다.

다음 코드에서는 각 예제를 보여 줍니다.

```
double d = 3D;
d = 4d;
d = 3.934_001;

float f = 3_000.5F;
f = 5.4f;

decimal myMoney = 3_000.5m;
myMoney = 400.75M;
```

앞의 예제에서는 C# 7.0부터 지원되는 숫자 구분 기호인 `_`를 사용하는 방법도 보여 줍니다. 모든 종류의 숫자 리터럴에서 숫자 구분 기호를 사용할 수 있습니다.

또한 다음 예제와 같이 과학적 표기법을 사용하여 real 리터럴의 지수 부분을 지정할 수도 있습니다.

```
double d = 0.42e2;
Console.WriteLine(d); // output 42

float f = 134.45E-2f;
Console.WriteLine(f); // output: 1.3445

decimal m = 1.5E6m;
Console.WriteLine(m); // output: 1500000
```

## 변환

부동 소수점 숫자 형식 간의 암시적 변환은 `float`에서 `double`로의 암시적 변환 하나뿐입니다. 그러나 [명시적 캐스트](#)를 사용하여 부동 소수점 형식을 다른 부동 소수점 형식으로 변환할 수 있습니다. 자세한 내용은 [기본 제공 숫자 변환](#)을 참조하세요.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 다음 섹션을 참조하세요.

- [부동 소수점 형식](#)
- [10진 형식](#)
- [real 리터럴](#)

## 참조

- [C# 참조](#)
- [값 형식](#)
- [정수 형식](#)
- [표준 숫자 형식 문자열](#)
- [.NET의 숫자](#)
- [System.Numerics.Complex](#)

# 기본 제공 숫자 변환(C# 참조)

2020-11-02 • 11 minutes to read • [Edit Online](#)

C#에서는 정수 및 부동 소수점 숫자 형식 집합을 제공합니다. 두 숫자 형식 간에는 암시적 또는 명시적 변환이 있습니다. 명시적 변환을 수행하려면 [캐스트 식](#) 사용해야 합니다.

## 암시적 숫자 변환

다음 표에서는 기본 제공 숫자 형식 간의 미리 정의된 암시적 숫자 변환을 보여 줍니다.

시작	대상
sbyte	short, int, long, float, double 또는 decimal
byte	short, ushort, int, uint, long, ulong, float, double 또는 decimal
short	int, long, float, double 또는 decimal
ushort	int, uint, long, ulong, float, double 또는 decimal
int	long, float, double 또는 decimal
uint	long, ulong, float, double 또는 decimal
long	float, double 또는 decimal
ulong	float, double 또는 decimal
float	double

### NOTE

int, uint, long 또는 ulong에서 float로 암시적 변환하거나 long 또는 ulong에서 double로 암시적 변환하는 경우 정밀도가 손실될 수도 있지만, 자릿수 손실은 없습니다. 다른 암시적 숫자 변환 시에는 정보 손실이 없습니다.

다음 사항에도 유의하세요.

- 정수 숫자 형식을 부동 소수점 숫자 형식으로 암시적으로 변환할 수 있습니다.
- byte 및 sbyte 형식으로의 암시적 변환은 없습니다. double 및 decimal 형식에서 암시적 변환은 없습니다.
- decimal 형식과 float 또는 double 형식 간의 암시적 변환은 없습니다.
- 대상 형식의 범위 내에 있는 경우 int 형식의 상수식 값(예: 정수 리터럴로 표시된 값)을 sbyte, byte, short, ushort, uint 또는 ulong으로 암시적으로 변환할 수 있습니다.

```
byte a = 13;
byte b = 300; // CS0031: Constant value '300' cannot be converted to a 'byte'
```

위 예제에서 볼 수 있듯이, 상수 값이 대상 형식의 범위 내에 있지 않으면 컴파일러 오류 [CS0031](#)이 발생합니다.

## 명시적 숫자 변환

다음 표에서는 [암시적 변환](#)이 없는 기본 제공 숫자 형식 간의 미리 정의된 명시적 변환을 보여 줍니다.

시작	대상
sbyte	<code>byte</code> , <code>ushort</code> , <code>uint</code> 또는 <code>ulong</code>
byte	<code>sbyte</code>
short	<code>sbyte</code> , <code>byte</code> , <code>ushort</code> , <code>uint</code> 또는 <code>ulong</code>
ushort	<code>sbyte</code> , <code>byte</code> 또는 <code>short</code>
int	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>uint</code> 또는 <code>ulong</code>
uint	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> 또는 <code>int</code>
long	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> 또는 <code>ulong</code>
ulong	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> 또는 <code>long</code>
float	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> 또는 <code>decimal</code>
double	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>float</code> 또는 <code>decimal</code>
decimal	<code>sbyte</code> , <code>byte</code> , <code>short</code> , <code>ushort</code> , <code>int</code> , <code>uint</code> , <code>long</code> , <code>ulong</code> , <code>float</code> 또는 <code>double</code>

### NOTE

명시적 숫자 변환으로 인해 데이터가 손실되거나 예외(일반적으로 [OverflowException](#))가 throw될 수 있습니다.

다음 사항에도 유의하세요.

- 정수 형식의 값을 다른 정수 형식으로 변환하면 결과는 오버플로 [검사 컨텍스트](#)에 따라 달라집니다. 확인된 컨텍스트에서 소스 값이 대상 형식의 범위 내에 있으면 변환이 성공합니다. 그렇지 않으면 [OverflowException](#)이 throw됩니다. 확인되지 않은 컨텍스트에서 변환은 항상 성공하고 다음과 같이 진행됩니다.
  - 소스 형식이 대상 형식보다 큰 경우 소스 값은 가장 중요한 비트인 해당 "extra"를 삭제함으로써 잘립니다. 그런 다음, 결과는 대상 형식의 값으로 처리됩니다.

- 소스 형식이 대상 형식보다 작은 경우 소스 값이 대상 형식과 크기가 같도록 부호 확장 또는 0 확장 됩니다. 부호 확장은 소스 형식이 서명된 경우 사용되며, 소스 형식이 서명되지 않은 경우 0 확장이 사용됩니다. 그런 다음, 결과는 대상 형식의 값으로 처리됩니다.
- 소스 형식이 대상 형식과 동일한 크기인 경우 소스 값은 대상 형식의 값으로 처리됩니다.
- `decimal` 값을 정수 형식으로 변환하면, 이 값은 0을 향한 방향으로 가장 가까운 정수값으로 반올림됩니다. 결과 정수 값이 대상 형식 범위에서 벗어났을 경우 `OverflowException`이 throw됩니다.
- `double` 또는 `float` 값을 정수 형식으로 변환하면, 이 값은 0을 향한 방향으로 가장 가까운 정수값으로 반올림됩니다. 결과 정수 값이 대상 형식 범위에서 벗어났을 경우 결과는 오버플로 `검사 컨텍스트`에 따라 달라집니다. `Checked` 컨텍스트에서는 `OverflowException`이 throw됩니다. 반면 `Unchecked` 컨텍스트에서 결과는 지정되지 않은 대상 형식 값이 됩니다.
- `double` 을 `float` 로 변환할 경우 `double` 값을 가장 가까운 `float` 값으로 반올림합니다. `double` 값이 너무 크거나 작아서 `float` 형식에 맞지 않는 경우 결과 값은 0 또는 무한대가 됩니다.
- `float` 또는 `double` 을 `decimal`로 변환할 경우 소수 값을 `decimal` 표현으로 변환한 다음 필요한 경우 가장 가까운 소수점 이하 28번째 자리로 반올림합니다. 소수 값에 따라 다음 결과 중 하나가 발생할 수 있습니다.
  - 소수 값이 너무 작아서 `decimal`로 나타낼 수 없을 경우 결과 값은 0이 됩니다.
  - 소수 값이 NaN(숫자가 아님), 무한대 또는 너무 커서 `decimal`로 나타낼 수 없을 경우 `OverflowException`을 throw합니다.
- `decimal` 을 `float` 또는 `double`로 변환하는 경우 소수 값이 각각 가장 가까운 `float` 또는 `double` 값으로 반올림됩니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 다음 섹션을 참조하세요.

- [암시적 숫자 변환](#)
- [명시적 숫자 변환](#)

## 참조

- [C# 참조](#)
- [캐스팅 및 형식 변환](#)

# bool(C# 참조)

2020-11-02 • 3 minutes to read • [Edit Online](#)

`bool` 형식 키워드는 부울 값(`true` 또는 `false`)을 나타내는 .NET `System.Boolean` 구조체 형식의 별칭입니다.

`bool` 형식의 값을 사용하여 논리 연산을 수행하려면 [부울 논리](#) 연산자를 사용합니다. `bool` 형식은 [비교](#) 및 [같음](#) 연산자의 결과 형식입니다. `bool` 식은 `if`, `do`, `while` 및 `for` 문과 [조건부 연산자](#) `?:`에서 제어하는 조건식입니다.

`bool` 형식의 기본값은 `false`입니다.

## 리터럴

`true` 및 `false` 리터럴을 사용하여 `bool` 변수를 초기화하거나 `bool` 값을 전달할 수 있습니다.

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked"); // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked"); // output: Not checked
```

## 값이 세 개인 부울 논리

예를 들어 값이 세 개인 논리를 지원해야 하는 경우(예: 값이 세 개인 부울 형식을 지원하는 데이터베이스에서 작업하는 경우) nullable `bool?` 형식을 사용합니다. `bool?` 피연산자의 경우 미리 정의된 `&` 및 `|` 연산자는 값이 세 개인 논리를 지원합니다. 자세한 내용은 [부울 논리 연산자](#) 문서의 [Nullable 부울 논리 연산자](#) 섹션을 참조하세요.

nullable 값 형식에 대한 자세한 내용은 [Nullable 값 형식](#)을 참조하세요.

## 변환

C#은 `bool` 형식을 포함하는 두 개의 변환만 제공합니다. 여기에는 해당하는 nullable `bool?` 형식으로의 암시적 변환과 `bool?` 형식에서의 명시적 변환이 있습니다. 그러나 .NET에서는 `bool` 형식으로 변환하거나 해당 형식에서 변환하는 데 사용할 수 있는 추가 메서드를 제공합니다. 자세한 내용은 `System.Boolean` API 참조 페이지의 [부울 값 사이의 변환](#) 섹션을 참조하세요.

## C# 언어 사양

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

## 참고 항목

- [C# 참조](#)
- [값 형식](#)
- [true 및 false 연산자](#)

# char(C# 참조)

2020-11-02 • 4 minutes to read • [Edit Online](#)

`char` 형식 키워드는 유니코드 UTF-16 문자를 나타내는 .NET `System.Char` 구조체 형식의 별칭입니다.

형식	범위	SIZE	.NET 형식
<code>char</code>	U+0000~U+FFFF	16비트	<code>System.Char</code>

`char` 형식의 기본값은 `\0` (U + 0000)입니다.

`char` 형식은 [비교](#), [같음](#), [증가](#) 및 [감소](#) 연산자를 지원합니다. 또한 `char` 피연산자의 경우 [산술](#) 및 [비트 논리](#) 연산자는 해당 문자 코드에 대한 연산을 수행하고 `int` 형식의 결과를 생성합니다.

`string` 형식은 텍스트를 `char` 값의 시퀀스로 나타냅니다.

## 리터럴

`char` 값을 다음 형식으로 지정할 수 있습니다.

- 문자 리터럴.
- 유니코드 이스케이프 시퀀스입니다. 이는 문자 코드의 네 개 기호를 사용하는 16진수 표현이 뒤에 표시되는 `\u`입니다.
- 16진수 이스케이프 시퀀스입니다. 이는 문자 코드의 16진수 표현이 뒤에 표시되는 `\x`입니다.

```
var chars = new[]
{
    'j',
    '\u006A',
    '\x006A',
    (char)106,
};
Console.WriteLine(string.Join(" ", chars)); // output: j j j j
```

앞의 예제에서 볼 수 있듯이, 문자 코드의 값을 해당하는 `char` 값으로 캐스팅할 수도 있습니다.

### NOTE

유니코드 이스케이프 시퀀스의 경우, 네 개의 16진수를 모두 지정해야 합니다. 즉, `\u006A` 은(는) 유효한 이스케이프 시퀀스이지만, `\u06A` 및 `\u6A` 은(는) 유효하지 않습니다.

16진수 이스케이프 시퀀스의 경우, 앞에 오는 0을 생략할 수 있습니다. 즉, `\x006A`, `\x06A` 및 `\x6A` 이스케이프 시퀀스가 유효하며 동일한 문자에 해당합니다.

## 변환

`char` 형식은 `ushort`, `int`, `uint`, `long`, `ulong` 등의 [정수](#) 형식으로 암시적으로 변환할 수 있습니다. `float`, `double`, `decimal` 등의 기본 제공 [부동 소수점](#) 숫자 형식으로 암시적으로 변환할 수도 있습니다. `sbyte`, `byte` 및 `short` 정수 형식으로 명시적으로 변환할 수 있습니다.

다른 형식에서 `char` 형식으로의 암시적 변환은 없습니다. 그러나 [정수](#) 또는 [부동 소수점](#) 숫자 형식을 `char`로

명시적으로 변환할 수 있습니다.

## C# 언어 사양

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

## 참고 항목

- [C# 참조](#)
- [값 형식](#)
- [문자열](#)
- [System.Text.Rune](#)
- [.NET의 문자 인코딩](#)

# 열거형(C# 참조)

2021-02-18 • 8 minutes to read • [Edit Online](#)

열거형(또는 열거형 형식)은 기본 정수 숫자 형식의 명명된 상수 집합에 의해 정의되는 값 형식입니다. 열거형을 정의하려면 `enum` 키워드를 정의하고 열거형 멤버의 이름을 지정합니다.

```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```

기본적으로 열거형 멤버의 연결된 상수 값은 `int` 형식입니다. 즉, 0으로 시작하고 정의 텍스트 순서에 따라 1씩 증가합니다. 다른 정수 숫자 형식을 열거형 형식의 기본 형식으로 명시적으로 지정할 수 있습니다. 다음 예제와 같이 연결된 상수 값을 명시적으로 지정할 수도 있습니다.

```
enum ErrorCode : ushort
{
    None = 0,
    Unknown = 1,
    ConnectionLost = 100,
    OutlierReading = 200
}
```

열거형 형식의 정의 내에서 메서드를 정의할 수 없습니다. 열거형 형식에 기능을 추가하려면 확장 메서드를 만듭니다.

0에 해당 열거형 멤버가 없는 경우에도 열거형 형식 `E`의 기본값은 식 `(E)0`에 의해 생성되는 값입니다.

열거형 형식을 사용하여 상호 배타적인 값의 집합에서의 선택 또는 선택의 조합을 나타낼 수 있습니다. 선택의 조합을 나타내려면 열거형 형식을 비트 플래그로 정의합니다.

## 비트 플래그로서 열거형 형식

열거형 형식으로 선택의 조합을 나타내려면 개별 선택이 비트 필드가 되도록 해당 선택의 열거형 멤버를 정의합니다. 즉, 이러한 멤버의 연결된 값은 제곱이어야 합니다. 그런 다음 비트 논리 연산자 `|` 또는 `&`를 사용하여 각각 선택을 조합하거나 선택의 조합을 교차할 수 있습니다. 열거형 형식이 비트 필드를 선언한다고 표시하면 `Flags` 특성을 적용합니다. 다음 예제와 같이 열거형 형식의 정의에 몇 가지 일반적인 조합을 포함할 수도 있습니다.

```

[Flags]
public enum Days
{
    None      = 0b_0000_0000, // 0
    Monday    = 0b_0000_0001, // 1
    Tuesday   = 0b_0000_0010, // 2
    Wednesday = 0b_0000_0100, // 4
    Thursday  = 0b_0000_1000, // 8
    Friday    = 0b_0001_0000, // 16
    Saturday  = 0b_0010_0000, // 32
    Sunday    = 0b_0100_0000, // 64
    Weekend   = Saturday | Sunday
}

public class FlagsEnumExample
{
    public static void Main()
    {
        Days meetingDays = Days.Monday | Days.Wednesday | Days.Friday;
        Console.WriteLine(meetingDays);
        // Output:
        // Monday, Wednesday, Friday

        Days workingFromHomeDays = Days.Thursday | Days.Friday;
        Console.WriteLine($"Join a meeting by phone on {meetingDays & workingFromHomeDays}");
        // Output:
        // Join a meeting by phone on Friday

        bool isMeetingOnTuesday = (meetingDays & Days.Tuesday) == Days.Tuesday;
        Console.WriteLine($"Is there a meeting on Tuesday: {isMeetingOnTuesday}");
        // Output:
        // Is there a meeting on Tuesday: False

        var a = (Days)37;
        Console.WriteLine(a);
        // Output:
        // Monday, Wednesday, Saturday
    }
}

```

자세한 내용과 예제는 [System.FlagsAttribute API 참조 페이지](#) 및 [System.Enum API 참조 페이지](#)의 **비독점적 멤버** 및 **Flags 특성** 섹션을 참조하세요.

## System.Enum 형식 및 열거형 제약 조건

[System.Enum](#) 형식은 모든 열거형 형식의 추상적 기본 클래스입니다. 이 형식은 열거형 형식 및 그 값에 대한 정보를 가져오는 여러 메서드를 제공합니다. 자세한 내용과 예제는 [System.Enum API 참조 페이지](#)를 참조하세요.

C# 7.3부터 기본 클래스 제약 조건([열거형 제약 조건](#)이라고 함)에서 [System.Enum](#)을 사용하여 형식 매개 변수가 열거형 형식이라고 지정할 수 있습니다. 또한 열거형 형식은 형식 매개 변수가 null을 허용하지 않는 값 형식이라고 지정하는 데 사용되는 [struct](#) 제약 조건을 충족합니다.

### 변환

모든 열거형 형식에는 열거형 형식과 기본 정수 형식 간의 명시적 변환이 있습니다. 열거형 값을 기본 형식에 [캐스트](#)하는 경우, 그 결과는 열거형 멤버의 연결된 정수 값입니다.

```

public enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}

public class EnumConversionExample
{
    public static void Main()
    {
        Season a = Season.Autumn;
        Console.WriteLine($"Integral value of {a} is {(int)a}"); // output: Integral value of Autumn is 2

        var b = (Season)1;
        Console.WriteLine(b); // output: Summer

        var c = (Season)4;
        Console.WriteLine(c); // output: 4
    }
}

```

열거형 형식에 연결된 특정 값이 포함되어 있는지 확인하려면 [Enum.IsDefined](#) 메서드를 사용합니다.

모든 열거형 형식에는 [System.Enum](#) 형식의 [boxing](#) 및 [unboxing](#) 변환이 있습니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 다음 섹션을 참조하세요.

- [열거형](#)
- [열거형 값 및 작업](#)
- [열거형 논리 연산자](#)
- [열거형 비교 연산자](#)
- [명시적 열거형 변환](#)
- [암시적 열거형 변환](#)

## 참조

- [C# 참조](#)
- [열거형 형식 문자열](#)
- [디자인 지침 - 열거형 디자인](#)
- [디자인 지침 - 열거형 명명 규칙](#)
- [switch 문](#)

# 구조체 형식(C# 참조)

2021-02-18 • 21 minutes to read • [Edit Online](#)

'구조체 형식'은 데이터와 관련 기능을 캡슐화할 수 있는 값 형식입니다. 구조체 형식은 `struct` 키워드를 사용하여 정의합니다.

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() => $"({X}, {Y})";
}
```

구조체 형식은 '값 의미 체계'를 갖습니다. 즉, 구조체 형식의 변수는 해당 형식의 인스턴스를 포함합니다. 기본적으로 변수 값은 할당 시에, 인수를 메서드에 전달할 때, 그리고 메서드 결과를 반환할 때 복사됩니다. 구조체 형식 변수의 경우, 해당 형식의 인스턴스가 복사됩니다. 자세한 내용은 [값 형식](#)을 참조하세요.

구조체 형식은 일반적으로 동작을 거의 제공하지 않거나 전혀 제공하지 않는 작은 데이터 중심 형식을 설계하는 데 사용합니다. 예를 들어, .NET에서는 구조체 형식을 사용하여 숫자([정수와 실수](#), [부울 값](#), [유니코드 문자](#), [시간 인스턴스](#))를 표현합니다. 형식의 동작이 중요한 경우에는 [클래스](#)를 정의하는 것이 좋습니다. 클래스 형식은 '참조 의미 체계'를 갖습니다. 즉, 클래스 형식의 변수는 인스턴스 자체가 아닌 해당 형식의 인스턴스에 대한 참조를 포함합니다.

구조체 형식에는 값 의미 체계가 있기 때문에 변경할 수 없는 구조체 형식을 정의하는 것이 좋습니다.

## readonly 구조체

C# 7.2부터 `readonly` 한정자를 사용하여 구조체 형식을 변경할 수 없도록 선언합니다. `readonly` 구조체의 모든 데이터 멤버는 다음과 같이 읽기 전용이어야 합니다.

- 모든 필드 선언에는 `readonly` 한정자가 있어야 합니다.
- 자동 구현된 속성을 포함하여 모든 속성은 읽기 전용이어야 합니다. C# 9.0 이상에서는 속성에 `init 접근자`가 있을 수 있습니다.

이렇게 하면 `readonly` 구조체의 멤버가 구조체의 상태를 수정하지 않습니다. C# 8.0 이상에서는 생성자를 제외한 다른 인스턴스 멤버는 암시적으로 `readonly`임을 의미합니다.

### NOTE

`readonly` 구조체에서 변경 가능한 참조 형식의 데이터 멤버는 여전히 자체 상태를 변경할 수 있습니다. 예를 들어 `List<T>` 인스턴스를 바꿀 수는 없지만 새 요소를 추가할 수는 있습니다.

다음 코드는 C# 9.0 이상에서 사용할 수 있는 `init` 전용 속성 setter를 사용하여 `readonly` 구조체를 정의합니다.

```

public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; init; }
    public double Y { get; init; }

    public override string ToString() => $"({X}, {Y})";
}

```

## readonly 인스턴스 멤버

C# 8.0부터 `readonly` 한정자를 사용하여 인스턴스 멤버가 구조체의 상태를 수정하지 않도록 선언할 수도 있습니다. 전체 구조체 형식을 `readonly`로 선언할 수 없는 경우 `readonly` 한정자를 사용하여 구조체의 상태를 수정하지 않는 인스턴스 멤버를 표시합니다.

`readonly` 인스턴스 멤버 내에서 구조체의 인스턴스 필드에 할당할 수 없습니다. 그러나 `readonly` 멤버는 `readonly` 가 아닌 멤버를 호출할 수 있습니다. 이 경우 컴파일러는 구조체 인스턴스의 복사본을 만들고 해당 복사본에서 `readonly` 가 아닌 멤버를 호출합니다. 따라서 원래 구조체 인스턴스는 수정되지 않습니다.

일반적으로 다음 종류의 인스턴스 멤버에 `readonly` 한정자를 적용합니다.

- 메서드:

```

public readonly double Sum()
{
    return X + Y;
}

```

`System.Object`에 선언된 메서드를 재정의하는 메서드에 `readonly` 한정자를 적용할 수도 있습니다.

```

public readonly override string ToString() => $"({X}, {Y})";

```

- 속성 및 인덱서:

```

private int counter;
public int Counter
{
    readonly get => counter;
    set => counter = value;
}

```

속성 또는 인덱서의 두 접근자에 모두 `readonly` 한정자를 적용해야 하는 경우 속성 또는 인덱서의 선언에 해당 한정자를 적용합니다.

### NOTE

컴파일러는 속성 선언에 `readonly` 한정자가 있는지 여부와 관계없이 자동 구현 속성의 `get` 접근자를 `readonly`로 선언합니다.

C# 9.0 이상에서는 `init` 접근자를 사용하여 속성 또는 인덱서에 `readonly` 한정자를 적용할 수 있습니다.

```

public readonly double X { get; init; }

```

구조체 형식의 정적 멤버에는 `readonly` 한정자를 적용할 수 없습니다.

컴파일러는 성능 최적화를 위해 `readonly` 한정자를 사용할 수 있습니다. 자세한 내용은 [안전하고 효율적인 C# 코드 작성](#)을 참조하세요.

## 구조체 형식 설계의 제한 사항

구조체 형식을 설계할 때는 [클래스](#) 형식과 같은 기능을 사용할 수 있으나, 다음과 같은 예외가 적용됩니다.

- 매개 변수가 없는 생성자를 선언할 수 없습니다. 모든 구조체 형식은 이미 해당 형식의 [기본값](#)을 생성하는 매개 변수 없는 암시적 생성자를 제공합니다.
- 인스턴스 필드 또는 속성은 선언과 동시에 초기화할 수 없습니다. 단, `static` 또는 `const` 필드 또는 정적 속성은 선언과 동시에 초기화할 수 있습니다.
- 구조체 형식의 생성자는 해당 형식의 모든 인스턴스 필드를 초기화해야 합니다.
- 구조체 형식은 다른 클래스 또는 구조체 형식에서 상속할 수 없으며, 클래스의 기본이 될 수 없습니다. 단, 구조체 형식은 [인터페이스](#)를 구현할 수 있습니다.
- 구조체 형식 내에서 [종료자](#)를 선언할 수 없습니다.

## 구조체 형식의 인스턴스화

C#에서는 선언된 변수를 사용하려면 먼저 초기화해야 합니다. 구조체 형식 변수는 `null` 일 수 없으므로([null 허용 값 형식](#)의 변수인 경우 제외), 해당 형식의 인스턴스를 인스턴스화해야 합니다. 이 작업은 몇 가지 방법으로 수행할 수 있습니다.

일반적으로, 구조체 형식은 `new` 연산자를 사용하여 적절한 생성자를 호출함으로써 인스턴스화합니다. 모든 구조체 형식은 하나 이상의 생성자를 갖습니다. 이는 해당 형식의 [기본값](#)을 생성하는 매개 변수 없는 암시적 생성자입니다. [기본값 식](#)을 사용하여 형식의 기본값을 생성할 수도 있습니다.

구조체 형식의 모든 인스턴스 필드가 액세스 가능한 경우, `new` 연산자 없이 인스턴스화할 수도 있습니다. 이 경우 인스턴스를 처음 사용하기 전에 모든 인스턴스 필드를 초기화해야 합니다. 다음 예제에서는 해당 작업을 수행하는 방법을 보여줍니다.

```
public static class StructWithoutNew
{
    public struct Coords
    {
        public double x;
        public double y;
    }

    public static void Main()
    {
        Coords p;
        p.x = 3;
        p.y = 4;
        Console.WriteLine($"{p.x}, {p.y}"); // output: (3, 4)
    }
}
```

[기본 제공 값 형식](#)의 경우, 해당 리터럴을 사용하여 해당 형식의 값을 지정합니다.

## 참조를 통해 구조체 형식 변수 전달

구조체 형식 변수를 메서드에 인수로 전달하거나 메서드에서 구조체 형식 값을 반환할 경우, 구조체 형식의 인스턴스 전체가 복사됩니다. 이로 인해 구조체 형식이 많이 사용되는 고성능 시나리오에서 코드의 성능이 저하될 수 있습니다. 구조체 형식 변수를 참조를 통해 전달하면 값이 복사되지 않도록 할 수 있습니다. 참조를 통해 인수를 전달해야 한다는 사실을 나타내려면 `ref`, `out` 또는 `in` 메서드 매개 변수 한정자를 사용합니다. 참조를 통해 메서드 결과를 반환하려면 `ref returns`를 사용합니다. 자세한 내용은 [안전하고 효율적인 C# 코드 작성](#)을 참조하세요.

## ref 구조체

C# 7.2부터 구조체 형식 선언에 `ref` 한정자를 사용할 수 있습니다. `ref` 구조체 형식의 인스턴스는 스택에 할당되며 관리되는 힙으로 이스케이프할 수 없습니다. 이를 위해 컴파일러는 다음과 같이 `ref` 구조체 형식의 사용을 제한합니다.

- `ref` 구조체는 배열의 요소 형식일 수 없습니다.
- `ref` 구조체는 클래스의 필드 또는 비 `ref` 구조체의 선언된 형식일 수 없습니다.
- `ref` 구조체는 인터페이스를 구현할 수 없습니다.
- `ref` 구조체는 `System.ValueType` 또는 `System.Object`에 boxing할 수 없습니다.
- `ref` 구조체는 형식 인수일 수 없습니다.
- **람다 식** 또는 **로컬 함수**에서 `ref` 구조체 변수를 캡처할 수 없습니다.
- `async` 메서드에서는 `ref` 구조체 변수를 사용할 수 없습니다. 그러나 동기 메서드에서는 `ref` 구조체 변수(예: `Task` 또는 `Task<TResult>`를 반환하는 변수)를 사용할 수 있습니다.
- **반복기**에서는 `ref` 구조체 변수를 사용할 수 없습니다.

일반적으로 `ref` 구조체 형식의 데이터 멤버도 포함하는 형식이 필요한 경우 `ref` 구조체 형식을 정의합니다.

```
public ref struct CustomRef
{
    public bool IsValid;
    public Span<int> Inputs;
    public Span<int> Outputs;
}
```

`ref` 구조체를 `readonly`로 선언하려면 형식 선언에서 `readonly` 및 `ref` 한정자를 결합합니다(`readonly` 한정자는 `ref` 한정자 앞에 와야함).

```
public readonly ref struct ConversionRequest
{
    public ConversionRequest(double rate, ReadOnlySpan<double> values)
    {
        Rate = rate;
        Values = values;
    }

    public double Rate { get; }
    public ReadOnlySpan<double> Values { get; }
}
```

.NET에서 `ref` 구조체의 예는 `System.Span<T>` 및 `System.ReadOnlySpan<T>`입니다.

## 구조체 제약 조건

`struct` 제약 조건의 `struct` 키워드를 사용하여 형식 매개 변수가 null을 허용하지 않는 값 형식이라고 지정할 수도 있습니다. 구조체 형식과 열거형 형식 모두 `struct` 제약 조건을 충족합니다.

## 변환

모든 구조체 형식(`ref` 구조체 형식 제외)에는 `System.ValueType` 및 `System.Object` 형식의 `boxing` 및 `unboxing` 변환이 있습니다. 구조체 형식과 구조체 형식이 구현하는 인터페이스 간에도 `boxing` 및 `unboxing` 변환이 있습니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 구조체 섹션](#)을 참조하세요.

C# 7.2 이상에 도입된 기능에 대한 자세한 내용은 다음 기능 제안 노트를 참조하세요.

- 읽기 전용 구조체
- 읽기 전용 인스턴스 멤버
- ref 유사 형식에 대한 컴파일 시간 안전성

## 참조

- C# 참조
- 디자인 지침 - 클래스와 구조체 간의 선택
- 디자인 지침-구조체 디자인
- 클래스 및 구조체

# 튜플 형식(C# 참조)

2020-11-02 • 17 minutes to read • [Edit Online](#)

C# 7.0 이상에서 사용할 수 있는 '튜플' 기능은 간단한 데이터 구조로 여러 데이터 요소를 그룹화하는 간결한 구문을 제공합니다. 다음 예제에서는 튜플 변수를 선언하고 초기화하며 관련 데이터 멤버에 액세스하는 방법을 보여 줍니다.

```
(double, int) t1 = (4.5, 3);
Console.WriteLine($"Tuple with elements {t1.Item1} and {t1.Item2}.");
// Output:
// Tuple with elements 4.5 and 3.

(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
// Output:
// Sum of 3 elements is 4.5.
```

앞의 예제와 같이 튜플 형식을 정의하려면 모든 관련 데이터 멤버의 형식과 필요한 경우 [필드 이름](#)을 지정합니다. 튜플 형식으로 메서드를 정의할 수는 없지만 다음 예제와 같이 .NET에서 제공하는 메서드를 사용할 수 있습니다.

```
(double, int) t = (4.5, 3);
Console.WriteLine(t.ToString());
Console.WriteLine($"Hash code of {t} is {t.GetHashCode()}.");
// Output:
// (4.5, 3)
// Hash code of (4.5, 3) is 718460086.
```

C# 7.3부터 튜플 형식은 [같은 연산자](#) `==` 및 `!=`을 지원합니다. 자세한 내용은 [튜플 같은](#) 섹션을 참조하세요.

튜플 형식은 [값 형식](#)이며 튜플 요소는 공용 필드입니다. 이에 따라 튜플은 '변경 가능한' 값 형식으로 설정됩니다.

## NOTE

튜플 기능을 사용하려면 .NET Core 및 .NET Framework 4.7 이상에서 사용할 수 있는 [System.ValueTuple](#) 형식 및 관련 제네릭 형식(예: [System.ValueTuple<T1,T2>](#))이 필요합니다. .NET Framework 4.6.2 이하를 대상으로 하는 프로젝트에서 튜플을 사용하려면 프로젝트에 NuGet 패키지 [System.ValueTuple](#)을 추가합니다.

임의의 많은 요소를 포함하는 튜플을 정의할 수 있습니다.

```
var t =
(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);
Console.WriteLine(t.Item26); // output: 26
```

## 튜플 사용 사례

튜플의 가장 일반적인 사용 사례 중 하나는 메서드 반환 형식입니다. 즉, [out](#) 메서드 매개 변수를 정의하는 대신 다음 예제와 같이 메서드 결과를 튜플 반환 형식으로 그룹화할 수 있습니다.

```

var xs = new[] { 4, 7, 9 };
var limits = FindMinMax(xs);
Console.WriteLine($"Limits of [{string.Join(" ", xs)}] are {limits.min} and {limits.max}");
// Output:
// Limits of [4 7 9] are 4 and 9

var ys = new[] { -9, 0, 67, 100 };
var (minimum, maximum) = FindMinMax(ys);
Console.WriteLine($"Limits of [{string.Join(" ", ys)}] are {minimum} and {maximum}");
// Output:
// Limits of [-9 0 67 100] are -9 and 100

(int min, int max) FindMinMax(int[] input)
{
    if (input is null || input.Length == 0)
    {
        throw new ArgumentException("Cannot find minimum and maximum of a null or empty array.");
    }

    var min = int.MaxValue;
    var max = int.MinValue;
    foreach (var i in input)
    {
        if (i < min)
        {
            min = i;
        }
        if (i > max)
        {
            max = i;
        }
    }
    return (min, max);
}

```

앞의 예제와 같이 반환된 튜플 인스턴스를 직접 사용하거나 개별 변수로 [분해](#) 할 수 있습니다.

**익명 형식** 대신 튜플 형식을 사용할 수도 있습니다(예: LINQ 쿼리에서). 자세한 내용은 [무명 형식과 튜플 형식 중에서 선택](#)을 참조하세요.

일반적으로 튜플을 사용하여 관련 데이터 요소를 느슨하게 그룹화합니다. 이 방법은 대개 프라이빗 및 내부 유 틸리티 메서드 내에서 유용합니다. 공용 API의 경우 [클래스](#) 또는 [구조체](#) 형식을 정의하는 것이 좋습니다.

## 튜플 필드 이름

다음 예제와 같이 튜플 초기화 식이나 튜플 형식 정의에 튜플 필드의 이름을 명시적으로 지정할 수 있습니다.

```

var t = (Sum: 4.5, Count: 3);
Console.WriteLine($"Sum of {t.Count} elements is {t.Sum}.");

(double Sum, int Count) d = (4.5, 3);
Console.WriteLine($"Sum of {d.Count} elements is {d.Sum}.");

```

C# 7.1부터는 필드 이름을 지정하지 않으면 다음 예제와 같이 튜플 초기화 식의 해당 변수 이름에서 이름이 유 추될 수 있습니다.

```

var sum = 4.5;
var count = 3;
var t = (sum, count);
Console.WriteLine($"Sum of {t.count} elements is {t.sum}.");

```

이를 튜플 프로젝션 이니셜라이저라고 합니다. 다음과 같은 경우에 변수 이름은 튜플 필드 이름으로 프로젝션되지 않습니다.

- 후보 이름이 튜플 형식의 멤버 이름인 경우(예: `Item3`, `ToString` 또는 `Rest`)
- 후보 이름이 명시적이든 암시적이든 다른 튜플 필드 이름과 중복되는 경우

이 경우 명시적으로 필드 이름을 지정하거나 기본 이름으로 필드에 액세스합니다.

튜플 필드의 기본 이름은 `Item1`, `Item2`, `Item3` 등입니다. 다음 예제와 같이 필드 이름이 명시적으로 지정되거나 유추되는 경우에도 언제든지 필드의 기본 이름을 사용할 수 있습니다.

```
var a = 1;
var t = (a, b: 2, 3);
Console.WriteLine($"The 1st element is {t.Item1} (same as {t.a}).");
Console.WriteLine($"The 2nd element is {t.Item2} (same as {t.b}).");
Console.WriteLine($"The 3rd element is {t.Item3}.");
// Output:
// The 1st element is 1 (same as 1).
// The 2nd element is 2 (same as 2).
// The 3rd element is 3.
```

[튜플 할당](#) 및 [튜플 같음 비교](#)에서는 필드 이름을 고려하지 않습니다.

컴파일 시간에 컴파일러는 기본값이 아닌 필드 이름을 해당하는 기본 이름으로 바꿉니다. 따라서 명시적으로 지정되거나 유추된 필드 이름을 런타임에 사용할 수 없습니다.

## 튜플 할당 및 분해

C#은 다음 두 조건을 모두 충족하는 튜플 형식 간에 할당을 지원합니다.

- 두 튜플 형식의 요소 수가 동일함
- 각 튜플 위치에서 오른쪽 튜플 요소의 형식이 해당하는 왼쪽 튜플 요소의 형식과 동일하거나 해당 형식으로 암시적으로 변환 가능함

튜플 요소 값은 튜플 요소의 순서에 따라 할당됩니다. 다음 예제와 같이 튜플 필드의 이름은 무시되고 할당되지 않습니다.

```
(int, double) t1 = (17, 3.14);
(double First, double Second) t2 = (0.0, 1.0);
t2 = t1;
Console.WriteLine($"{nameof(t2)}: {t2.First} and {t2.Second}");
// Output:
// t2: 17 and 3.14

(double A, double B) t3 = (2.0, 3.0);
t3 = t2;
Console.WriteLine($"{nameof(t3)}: {t3.A} and {t3.B}");
// Output:
// t3: 17 and 3.14
```

= 대입 연산자를 사용하여 튜플 인스턴스를 개별 변수로 '분해'할 수도 있습니다. 다음 중 한 가지 방법으로 해당 작업을 수행할 수 있습니다.

- 괄호 안에 각 변수의 형식을 명시적으로 선언합니다.

```

var t = ("post office", 3.6);
(string destination, double distance) = t;
Console.WriteLine($"Distance to {destination} is {distance} kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.

```

- 괄호 밖에서 `var` 키워드를 사용하여 형식화된 변수를 암시적으로 선언하며 컴파일러가 해당 형식을 추하도록 합니다.

```

var t = ("post office", 3.6);
var (destination, distance) = t;
Console.WriteLine($"Distance to {destination} is {distance} kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.

```

- 기존 변수를 사용합니다.

```

var destination = string.Empty;
var distance = 0.0;

var t = ("post office", 3.6);
(destination, distance) = t;
Console.WriteLine($"Distance to {destination} is {distance} kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.

```

튜플 및 기타 형식을 분해하는 방법에 관한 자세한 내용은 [튜플 및 기타 형식 분해](#)를 참조하세요.

## 튜플 같음

C# 7.3부터 튜플 형식에서는 `==` 및 `!=` 연산자를 지원합니다. 해당 연산자는 튜플 요소 순서에 따라 왼쪽 피연산자의 멤버를 오른쪽 피연산자의 해당 멤버와 비교합니다.

```

(int a, byte b) left = (5, 10);
(long a, int b) right = (5, 10);
Console.WriteLine(left == right); // output: True
Console.WriteLine(left != right); // output: False

var t1 = (A: 5, B: 10);
var t2 = (B: 5, A: 10);
Console.WriteLine(t1 == t2); // output: True
Console.WriteLine(t1 != t2); // output: False

```

앞의 예제와 같이 `==` 및 `!=` 연산에는 튜플 필드 이름이 고려되지 않습니다.

다음 조건이 둘 다 충족되면 두 튜플을 비교할 수 있습니다.

- 두 튜플의 요소 수가 동일합니다. 예를 들어 `t1` 및 `t2`의 요소 수가 다른 경우 `t1 != t2`는 컴파일되지 않습니다.
- 각 튜플 위치에서 왼쪽 및 오른쪽 튜플 피연산자의 해당 요소는 `==` 및 `!=` 연산자와 비교할 수 있습니다. 예를 들어 `1`은 `(1, 2)`와 비교할 수 없기 때문에 `(1, (2, 3)) == ((1, 2), 3)`은 컴파일되지 않습니다.

`==` 및 `!=` 연산자는 단락(short-circuiting) 방식으로 튜플을 비교합니다. 즉, 같지 않은 요소 쌍을 충족하거나 튜플의 끝에 도달하는 즉시 연산이 중지됩니다. 그러나 다음 예제와 같이 비교하기 전에 '모든' 튜플 요소가 평가됩니다.

```

Console.WriteLine(Display(1), Display(2)) == (Display(3), Display(4));

int Display(int s)
{
    Console.WriteLine(s);
    return s;
}
// Output:
// 1
// 2
// 3
// 4
// False

```

## 출력 매개 변수 튜플

일반적으로 `out` 매개 변수를 포함하는 메서드는 튜플을 반환하는 메서드로 리팩터링합니다. 그러나 `out` 매개 변수가 튜플 형식일 수 있는 경우가 있습니다. 다음 예제에서는 튜플을 `out` 매개 변수로 사용하는 방법을 보여줍니다.

```

var limitsLookup = new Dictionary<int, (int Min, int Max)>()
{
    [2] = (4, 10),
    [4] = (10, 20),
    [6] = (0, 23)
};

if (limitsLookup.TryGetValue(4, out (int Min, int Max) limits))
{
    Console.WriteLine($"Found limits: min is {limits.Min}, max is {limits.Max}");
}
// Output:
// Found limits: min is 10, max is 20

```

## 튜플과 `System.Tuple` 비교

`System.ValueTuple` 형식으로 지원되는 C# 튜플은 `System.Tuple` 형식으로 표현되는 튜플과 다릅니다. 주요 차이점은 다음과 같습니다.

- `ValueTuple` 형식은 값 형식입니다. `Tuple` 형식은 참조 형식입니다.
- `ValueTuple` 형식은 변경할 수 있습니다. `Tuple` 형식은 변경할 수 없습니다.
- `ValueTuple` 형식의 데이터 멤버는 필드입니다. `Tuple` 형식의 데이터 멤버는 속성입니다.

## C# 언어 사양

자세한 내용은 다음 기능 제안 노트를 참조하세요.

- 튜플 이름 유추(즉, 튜플 프로젝션 이니셜라이저)
- 튜플 형식에서 `==` 및 `!=` 지원

## 참조

- C# 참조
- 값 형식
- 무명 형식과 튜플 형식 중에서 선택
- `System.ValueTuple`

# Null 허용 값 형식(C# 참조)

2020-11-02 • 18 minutes to read • [Edit Online](#)

Null 허용 값 형식 `T?`는 기본 값 형식 `T`의 모든 값과 추가 `null` 값을 나타냅니다. 예를 들어 `bool?` 변수에는 다음 세 가지 값 중 하나를 할당할 수 있습니다. `true`, `false`, `null`. 기본 값 형식 `T`는 null 허용 값 형식 자체일 수 없습니다.

## NOTE

C# 8.0에서는 nullable 참조 형식 기능을 소개합니다. 자세한 내용은 [nullable 참조 형식](#)을 참조하세요. Null 허용 값 형식은 C# 2부터 사용할 수 있습니다.

Null 허용 값 형식은 제네릭 `System.Nullable<T>` 구조체의 인스턴스입니다. 서로 교환 가능한 형식인 `Nullable<T>` 또는 `T?` 중 하나에서 기본 값 형식 `T`의 null 허용 값 형식을 참조할 수 있습니다.

기본 값 형식의 정의되지 않은 값을 표시해야 하는 경우 일반적으로 null 허용 값 형식을 사용합니다. 예를 들어 부울(`bool`) 변수는 `true` 또는 `false`만 가능합니다. 그러나 일부 애플리케이션에서는 변수 값이 정의되지 않았거나 누락될 수 있습니다. 예를 들어 데이터베이스 필드는 `true` 또는 `false`를 포함하거나 아무 값도 없을 수 있습니다(즉 `NULL`). 이러한 시나리오에서 `bool?` 형식을 사용할 수 있습니다.

## 선언 및 할당

값 형식이 암시적으로 해당 null 허용 값 형식으로 변환될 수 있기 때문에 기본 값 형식에서와 마찬가지로 null 허용 값 형식의 변수에 값을 할당할 수 있습니다. `null` 값을 할당할 수도 있습니다. 예를 들어:

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// An array of a nullable value type:
int?[] arr = new int?[10];
```

Null 허용 값 형식의 기본값은 `null`을 나타냅니다. 즉, `Nullable<T>.HasValue` 속성이 `false`를 반환하는 인스턴스입니다.

## Null 허용 값 형식의 인스턴스 검사

C# 7.0부터 [is 형식 패턴 포함 연산자](#)를 사용하여 null 허용 값 형식의 인스턴스에서 `null` 여부를 검사하고 기본 형식의 값을 검색할 수 있습니다.

```

int? a = 42;
if (a is int valueOfA)
{
    Console.WriteLine($"a is {valueOfA}");
}
else
{
    Console.WriteLine("a does not have a value");
}
// Output:
// a is 42

```

항상 다음 읽기 전용 속성을 사용하여 null 허용 값 형식 변수의 값을 검사하고 가져올 수 있습니다.

- `Nullable<T>.HasValue`은 null 허용 값 형식의 인스턴스에 해당 기본 형식의 값이 있는지 여부를 나타냅니다.
- `Nullable<T>.Value`은 `.HasValue`가 `true`인 경우 기본 형식의 값을 가져옵니다. `.HasValue`가 `false`인 경우 `Value` 속성은 `InvalidOperationException`를 throw합니다.

다음 예제는 `.HasValue` 속성을 사용하여 표시하기 전에 변수가 값을 포함하는지 여부를 테스트합니다.

```

int? b = 10;
if (b.HasValue)
{
    Console.WriteLine($"b is {b.Value}");
}
else
{
    Console.WriteLine("b does not have a value");
}
// Output:
// b is 10

```

다음 예제와 같이 `.HasValue` 속성을 사용하는 대신 null 허용 값 형식 변수를 `null`과 비교할 수도 있습니다.

```

int? c = 7;
if (c != null)
{
    Console.WriteLine($"c is {c.Value}");
}
else
{
    Console.WriteLine("c does not have a value");
}
// Output:
// c is 7

```

## nullable 값 형식에서 기본 형식으로 변환

Null 허용 값 형식의 값을 `null`을 허용하지 않는 값 형식 변수에 할당하려는 경우 `null` 대신 할당할 값을 지정해야 할 수 있습니다. `null 병합 연산자 ??`를 사용하여 이 작업을 수행할 수 있습니다 (`Nullable<T>.GetValueOrDefault(T)` 메서드를 동일한 용도로 사용할 수도 있음).

```

int? a = 28;
int b = a ?? -1;
Console.WriteLine($"b is {b}"); // output: b is 28

int? c = null;
int d = c ?? -1;
Console.WriteLine($"d is {d}"); // output: d is -1

```

`null` 대신 기본 값 형식의 [기본값](#)을 사용하려면 `Nullable<T>.GetValueOrDefault()` 메서드를 사용합니다.

다음 예제와 같이 `null` 허용 값 형식을 `null`을 허용하지 않는 형식으로 명시적으로 캐스트할 수도 있습니다.

```

int? n = null;

//int m1 = n;    // Doesn't compile
int n2 = (int)n; // Compiles, but throws an exception if n is null

```

런타임 시 nullable 값 형식의 값이 `null`인 경우 명시적 캐스트는 [InvalidOperationException](#)를 throw합니다.

`null`을 허용하지 않는 값 형식 `T`는 해당 `null` 허용 값 형식 `T?`로 암시적으로 변환될 수 있습니다.

## 리프트 연산자

미리 정의된 단항 및 이항 [연산자](#) 또는 값 형식 `T`에서 지원하는 오버로드된 연산자는 해당 `null` 허용 값 형식 `T?`에서도 지원합니다. 리프트 연산자라고도 하는 이러한 연산자는 하나 또는 두 개의 피연산자가 `null`인 경우 `null` 값을 생성하고, 그렇지 않으면 연산자는 포함된 피연산자 값을 사용하여 결과를 계산합니다. 예를 들어:

```

int? a = 10;
int? b = null;
int? c = 10;

a++;      // a is 11
a = a * c; // a is 110
a = a + b; // a is null

```

### NOTE

| 형식의 경우 미리 정의된 `bool?` 및 `&` 연산자는 이 섹션에서 설명된 규칙을 따르지 않습니다. 연산자 평가의 결과는 피연산자 중 하나가 `null`인 경우에도 Null이 아닐 수 있습니다. 자세한 내용은 [부울 논리 연산자](#) 문서의 [Nullable 부울 논리 연산자](#) 섹션을 참조하세요.

**비교 연산자** `<`, `>`, `<=` 및 `>=`의 경우 피연산자 중 하나 또는 둘 모두가 `null`이면 결과는 `false`입니다. 그렇지 않으면 포함된 피연산자 값을 비교합니다. 특정 비교(예: `<=`)에서는 `false`를 반환하고 그 반대의 비교(`>`)에서는 `true`를 반환한다고 가정하지 마십시오. 다음 예제에서는 10이

- `null` 보다 크지도, 같지도 않고
- `null` 보다 작지도 않음을 보여줍니다.

```

int? a = 10;
Console.WriteLine($"{a} >= null is {a >= null}");
Console.WriteLine($"{a} < null is {a < null}");
Console.WriteLine($"{a} == null is {a == null}");
// Output:
// 10 >= null is False
// 10 < null is False
// 10 == null is False

int? b = null;
int? c = null;
Console.WriteLine($"null >= null is {b >= c}");
Console.WriteLine($"null == null is {b == c}");
// Output:
// null >= null is False
// null == null is True

```

같은 연산자 `==`의 경우 두 피연산자 모두 `null`이면 결과는 `true`이고 피연산자 중 하나만 `null`이면 결과는 `false`입니다. 그렇지 않으면 포함된 피연산자 값을 비교합니다.

같지 않은 연산자 `!=`의 경우 두 피연산자 모두 `null`이면 결과는 `false`이고 피연산자 중 하나만 `null`이면 결과는 `true`입니다. 그렇지 않으면 포함된 피연산자 값을 비교합니다.

두 값 형식 사이에 [사용자 정의 변환](#)이 있는 경우 해당 `null` 허용 값 형식 사이에도 같은 변환을 사용할 수 있습니다.

## boxing 및 unboxing

`Null` 허용 값 형식 `T?`의 인스턴스는 다음과 같이 `box`됩니다.

- `HasValue`가 `false`를 반환하는 경우 `Null` 참조가 생성됩니다.
- `HasValue`가 `true`를 반환하는 경우 `Nullable<T>`의 인스턴스가 아닌 기본 값 형식 `T`의 인스턴스가 `box`됩니다.

다음 예제와 같이 값 형식 `T`의 boxed 값을 해당 `null` 허용 값 형식 `T?`로 `unbox`할 수 있습니다.

```

int a = 41;
object aBoxed = a;
int? aNullable = (int?)aBoxed;
Console.WriteLine($"Value of aNullable: {aNullable}");

object aNullableBoxed = aNullable;
if (aNullableBoxed is int valueOfA)
{
    Console.WriteLine($"aNullableBoxed is boxed int: {valueOfA}");
}
// Output:
// Value of aNullable: 41
// aNullableBoxed is boxed int: 41

```

## Null 허용 값 형식 식별 방법

다음 예제는 `System.Type` 인스턴스가 구성된 `null` 허용 값 형식(지정된 형식 매개 변수 `T`가 포함된 `System.Nullable<T>` 형식)을 나타내는지 확인하는 방법을 보여 줍니다.

```

Console.WriteLine($"int? is {(IsNullable(typeof(int?)) ? "nullable" : "non nullable")} value type");
Console.WriteLine($"int is {(IsNullable(typeof(int)) ? "nullable" : "non-nullable")} value type");

bool IsNullable(Type type) => Nullable.GetUnderlyingType(type) != null;

// Output:
// int? is nullable value type
// int is non-nullable value type

```

예제에서와 같이 `typeof` 연산자를 사용하여 `System.Type` 인스턴스를 만듭니다.

인스턴스가 있는지 nullable 값 형식인지 여부를 확인하려는 경우 위의 코드로 테스트되도록 `Type` 인스턴스를 가져오는 데 `Object.GetType` 메서드를 사용하지 마세요. nullable 값 형식의 인스턴스에서 `Object.GetType` 메서드를 호출하는 경우 인스턴스는 `Object`로 `boxing`됩니다. Null 허용 값 형식의 Null이 아닌 인스턴스의 `boxing`은 기본 형식 값의 `boxing`과 동일하며, `GetType`는 null 허용 값 형식의 기본 형식을 나타내는 `Type` 인스턴스를 반환합니다.

```

int? a = 17;
Type typeOfA = a.GetType();
Console.WriteLine(typeOfA.FullName);
// Output:
// System.Int32

```

그리고 인스턴스가 null 허용 값 형식인지 여부를 확인하는 데 `is` 연산자를 사용하지 마세요. 다음 예제에서와 같이 null 허용 값 형식 인스턴스와 `is` 연산자를 사용하는 해당 형식 인스턴스를 구분할 수 없습니다.

```

int? a = 14;
if (a is int)
{
    Console.WriteLine("int? instance is compatible with int");
}

int b = 17;
if (b is int?)
{
    Console.WriteLine("int instance is compatible with int?");
}
// Output:
// int? instance is compatible with int
// int instance is compatible with int?

```

다음 예제에서 제공된 코드를 사용하여 인스턴스가 nullable 값 형식인지 여부를 확인할 수 있습니다.

```

int? a = 14;
Console.WriteLine(IsOfNullableType(a)); // output: True

int b = 17;
Console.WriteLine(IsOfNullableType(b)); // output: False

bool IsOfNullableType<T>(T o)
{
    var type = typeof(T);
    return Nullable.GetUnderlyingType(type) != null;
}

```

#### NOTE

이 섹션에서 설명하는 방법은 nullable 참조 형식의 경우에는 적용되지 않습니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 다음 섹션을 참조하세요.

- [Nullable 형식](#)
- [리프트 연산자](#)
- [암시적 null 허용 전환](#)
- [명시적 null 허용 전환](#)
- [리프트 변환 연산자](#)

## 참조

- [C# 참조](#)
- ['리프트'란 정확히 어떤 의미입니까?](#)
- [System.Nullable<T>](#)
- [System.Nullable](#)
- [Nullable.GetUnderlyingType](#)
- [nullable 참조 형식](#)

# 참조 형식(C# 참조)

2021-02-18 • 2 minutes to read • [Edit Online](#)

C# 형식은 참조 형식과 값 형식 두 가지가 있습니다. 참조 형식의 변수에는 데이터(객체)에 대한 참조가 저장되며, 값 형식의 변수에는 해당 데이터가 직접 포함됩니다. 참조 형식에서는 두 가지 변수가 같은 객체를 참조할 수 있으므로 한 변수에 대한 작업이 다른 변수에서 참조하는 객체에 영향을 미칠 수 있습니다. 값 형식에서는 각 변수에 데이터의 자체 사본이 들어 있으며 한 변수의 작업이 다른 변수에 영향을 미칠 수 없습니다(in, ref 및 out 매개 변수 제외, [in](#), [ref](#) 및 [out](#) 매개 변수 한정자 참조).

다음 키워드는 참조 형식을 선언하는 데 사용됩니다.

- [class](#)
- [interface](#)
- [delegate](#)
- [record](#)

C#는 다음과 같은 기본 참조 형식도 제공합니다.

- [dynamic](#)
- [object](#)
- [string](#)

## 참고 항목

- [C# 참조](#)
- [C# 키워드](#)
- [포인터 형식](#)
- [값 형식](#)

# 기본 제공 참조 형식(C# 참조)

2020-11-02 • 17 minutes to read • [Edit Online](#)

C#에는 여러 가지 기본 제공 참조 형식이 있습니다. .NET 라이브러리의 형식에 대한 동의어인 키워드 또는 연산자를 가지고 있습니다.

## 개체 유형

`object` 형식은 .NET에서 `System.Object`의 별칭입니다. C#의 통합 형식 시스템에서 모든 형식(사전 정의되거나 사용자 정의된 형식, 참조 형식, 값 형식)은 `System.Object`에서 직접 또는 간접적으로 상속합니다. `object` 형식의 변수에 모든 형식의 값을 할당할 수 있습니다. 모든 `object` 변수는 리터럴 `null`을 사용하여 기본값으로 할당할 수 있습니다. 값 형식의 변수가 개체로 변환된 경우 *boxed*라고 합니다. 형식 `object`의 변수가 값 형식으로 변환된 경우 *unboxed*라고 합니다. 자세한 내용은 [boxing 및 unboxing](#)을 참조하세요.

## 문자열 유형

`string` 형식은 0자 이상의 유니코드 문자 시퀀스를 나타냅니다. `string`은 .NET에서 `System.String`의 별칭입니다.

`string`은 참조 형식이지만 [같은 연산자](#) `==` 및 `!=`는 참조가 아니라 `string` 개체의 값을 비교하도록 정의됩니다. 이 때문에 좀 더 직관적으로 문자열이 같은지 테스트할 수 있습니다. 예:

```
string a = "hello";
string b = "h";
// Append to contents of 'b'
b += "ello";
Console.WriteLine(a == b);
Console.WriteLine(object.ReferenceEquals(a, b));
```

이 코드는 "True"를 표시한 다음 "False"를 표시하며, 이는 문자열의 내용이 동일하지만 `a`와 `b`는 동일한 문자열 인스턴스를 가리키지 않기 때문입니다.

+ [연산자](#)는 문자열을 연결합니다.

```
string a = "good " + "morning";
```

이 경우 "good morning"이 포함된 문자열 개체가 생성됩니다.

문자열은 [변경할 수 없습니다](#). 구문상 가능한 것처럼 보여도 개체를 만든 후에는 문자열 개체의 내용을 변경할 수 없습니다. 예를 들어 이 코드를 작성하면 컴파일러는 실제로 새 문자 시퀀스를 보유할 새 문자열 개체를 만들고, 새 개체가 `b`에 할당됩니다. `b`에 할당된 메모리(문자열 "h"가 포함된 경우)는 가비지 수집에 적합합니다.

```
string b = "h";
b += "ello";
```

`[]` [연산자](#)는 문자열의 개별 문자에 대한 읽기 전용 액세스에 사용할 수 있습니다. 유효한 인덱스는 `0`에서 시작되고 문자열의 길이보다 작아야 합니다.

```
string str = "test";
char x = str[2]; // x = 's';
```

비슷한 방식으로 `[]` 연산자도 문자열의 각 문자를 반복하는 데 사용할 수 있습니다.

```
string str = "test";

for (int i = 0; i < str.Length; i++)
{
    Console.Write(str[i] + " ");
}
// Output: t e s t
```

문자열 리터럴은 `string` 형식이며, quoted 및 `@`-quoted의 두 가지 형식으로 작성될 수 있습니다. 따옴표가 있는 문자열 리터럴은 큰따옴표(")로 묶여 있습니다.

```
"good morning" // a string literal
```

문자열 리터럴에는 모든 문자 리터럴이 포함될 수 있습니다. 이스케이프 시퀀스가 포함됩니다. 다음 예제에서는 이스케이프 시퀀스 `\\"`를 백슬래시에 사용하고, `\u0066`을 f에 사용하고, `\n`을 줄 바꿈에 사용합니다.

```
string a = "\\u0066\\n F";
Console.WriteLine(a);
// Output:
// \f
// F
```

#### NOTE

이스케이프 코드 `\udddd` (여기서 `ddd`는 4자리 숫자)는 유니코드 문자 U+`ddd`를 나타냅니다. 8자리 유니코드 이스케이프 코드 `\Uddddddddd`도 인식됩니다.

축자 문자열 리터럴은 `@`로 시작하며 큰따옴표로 묶여 있습니다. 예:

```
@"good morning" // a string literal
```

축자 문자열의 장점은 이스케이프 시퀀스가 처리되지 않으므로, 정규화된 Windows 파일 이름 등을 쉽게 쓸 수 있다는 것입니다.

```
@"c:\\Docs\\Source\\a.txt" // rather than "c:\\\\Docs\\\\Source\\\\a.txt"
```

`@`-quoted 문자열에 큰따옴표를 포함하려면 큰따옴표로 묶습니다.

```
@"""Ahoy!"" cried the captain." // "Ahoy!" cried the captain.
```

## 대리자 형식

`delegate` 형식의 선언은 메서드 시그니처와 유사합니다. 반환 값이 있으며 모든 형식의 매개 변수를 개수에 관계없이 사용할 수 있습니다.

```
public delegate void MessageDelegate(string message);
public delegate int AnotherDelegate(MyType m, long num);
```

.NET에서 `System.Action` 및 `System.Func` 유형은 많은 일반 대리자에 대한 일반적인 정의를 제공합니다. 새 사용자 지정 대리자 형식을 정의할 필요가 없습니다. 대신 제공된 제네릭 형식의 인스턴스화를 만들 수 있습니다.

`delegate`는 명명된 메서드나 무명 메서드를 캡슐화하는 데 사용할 수 있는 참조 형식입니다. 대리자는 C++의 함수 포인터와 비슷하지만 형식 안전성과 보안성을 제공한다는 점이 다릅니다. 대리자 적용에 대해서는 [대리자](#) 및 [제네릭 대리자](#)를 참조하세요. 대리자는 [이벤트](#)의 기반이 됩니다. 대리자는 명명된 메서드나 무명 메서드와

연결하여 인스턴스화할 수 있습니다.

대리자는 호환되는 반환 형식 및 입력 매개 변수가 있는 메서드나 람다 식을 사용하여 인스턴스화해야 합니다. 메서드 시그니처에서 허용되는 가변성 수준에 대한 자세한 내용은 [대리자의 가변성](#)을 참조하세요. 무명 메서드에서 사용하기 위해 메서드에 연결할 대리자와 코드를 함께 선언합니다.

## 동적 형식

`dynamic` 유형은 변수 및 해당 멤버에 대한 참조 사용이 컴파일 파일 형식 검사를バイ패스함을 나타냅니다. 대신, 이러한 작업은 런타임에 확인됩니다. `dynamic` 형식은 Office Automation API와 같은 COM API, IronPython 라이브러리 등의 동적 API 및 HTML DOM(문서 개체 모델)에 대한 액세스를 간소화합니다.

`dynamic` 형식은 대부분의 상황에서 `object` 형식처럼 동작합니다. 특히 null이 아닌 모든 식은 `dynamic` 형식으로 변환될 수 있습니다. `dynamic` 유형의 식을 포함하는 작업은 컴파일러에서 확인되거나 형식이 검사되지 않았다는 점에서 `dynamic` 유형은 `object`와 다릅니다. 컴파일러는 작업에 대한 정보를 패키지지하며, 나중에 해당 정보는 런타임에 작업을 평가하는 데 사용됩니다. 이 과정에서 `dynamic` 형식의 변수는 `object` 형식의 변수로 컴파일됩니다. 따라서 `dynamic` 형식은 컴파일 시간에만 존재하고 런타임에는 존재하지 않습니다.

다음 예제에서는 `dynamic` 형식의 변수와 `object` 형식의 변수를 비교합니다. 컴파일 시간에 각 변수의 형식을 확인하려면 `WriteLine` 문의 `dyn` 또는 `obj` 위에 마우스 포인터를 놓습니다. IntelliSense를 사용할 수 있는 편집기로 다음 코드를 복사합니다. IntelliSense는 `dyn`에 대해 `dynamic`을 표시하고 `obj`에 대해 `object`를 표시합니다.

```
class Program
{
    static void Main(string[] args)
    {
        dynamic dyn = 1;
        object obj = 1;

        // Rest the mouse pointer over dyn and obj to see their
        // types at compile time.
        System.Console.WriteLine(dyn.GetType());
        System.Console.WriteLine(obj.GetType());
    }
}
```

`WriteLine` 문은 `dyn` 및 `obj`의 런타임 형식을 표시합니다. 이 시점에는 둘 다 동일한 형식인 정수입니다. 다음 출력이 생성됩니다.

```
System.Int32
System.Int32
```

컴파일 시간에 `dyn` 및 `obj` 간의 차이를 보려면 앞의 예제에서 선언과 `WriteLine` 문 사이에 다음 두 줄을 추가합니다.

```
dyn = dyn + 3;
obj = obj + 3;
```

`obj + 3` 식에 정수와 개체를 추가하려는 시도와 관련해서 컴파일러 오류가 보고됩니다. 하지만 `dyn + 3`에 대한 오류는 보고되지 않습니다. `dyn`의 형식이 `dynamic`이기 때문에 `dyn`을 포함하는 식은 컴파일 시간에 확인되지 않습니다.

다음 예제에서는 여러 선언에 `dynamic`을 사용합니다. 또한 `Main` 메서드는 컴파일 시간 형식 검사를 런타임 형식 검사와 비교합니다.

```

using System;

namespace DynamicExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            ExampleClass ec = new ExampleClass();
            Console.WriteLine(ec.exampleMethod(10));
            Console.WriteLine(ec.exampleMethod("value"));

            // The following line causes a compiler error because exampleMethod
            // takes only one argument.
            //Console.WriteLine(ec.exampleMethod(10, 4));

            dynamic dynamic_ec = new ExampleClass();
            Console.WriteLine(dynamic_ec.exampleMethod(10));

            // Because dynamic_ec is dynamic, the following call to exampleMethod
            // with two arguments does not produce an error at compile time.
            // However, it does cause a run-time error.
            //Console.WriteLine(dynamic_ec.exampleMethod(10, 4));
        }
    }

    class ExampleClass
    {
        static dynamic field;
        dynamic prop { get; set; }

        public dynamic exampleMethod(dynamic d)
        {
            dynamic local = "Local variable";
            int two = 2;

            if (d is int)
            {
                return local;
            }
            else
            {
                return two;
            }
        }
    }
}

// Results:
// Local variable
// 2
// Local variable

```

## 참고 항목

- [C# 참조](#)
- [C# 키워드](#)
- [이벤트](#)
- [dynamic 형식 사용](#)
- [문자열 사용에 대한 모범 사례](#)
- [기본적인 문자열 작업](#)
- [새 문자열 만들기](#)
- [형식 테스트 및 캐스트 연산자](#)
- [패턴 일치, as 및 is 연산자를 사용하여 안전하게 캐스트하는 방법](#)
- [연습: 동적 개체 만들기 및 사용](#)
- [System.Object](#)

- [System.String](#)
- [System.Dynamic.DynamicObject](#)

# class(C# 참조)

2020-11-02 • 6 minutes to read • [Edit Online](#)

클래스는 다음 예제와 같이 `class` 키워드를 사용하여 선언됩니다.

```
class TestClass
{
    // Methods, properties, fields, events, delegates
    // and nested classes go here.
}
```

## 설명

C#에서는 단일 상속만 허용됩니다. 즉, 한 클래스는 하나의 기본 클래스에서만 구현을 상속할 수 있습니다. 그러나 한 클래스는 두 개 이상의 인터페이스를 구현할 수 있습니다. 다음 표에는 클래스 상속 및 인터페이스 구현에 대한 예제가 나와 있습니다.

상속	예제
없음	<code>class ClassA { }</code>
Single	<code>class DerivedClass : BaseClass { }</code>
없음. 두 개의 인터페이스 구현	<code>class ImplClass : IFace1, IFace2 { }</code>
단일. 하나의 인터페이스 구현	<code>class ImplDerivedClass : BaseClass, IFace1 { }</code>

다른 클래스 내에 중첩되는 것이 아니라 네임스페이스 내에서 직접 선언되는 클래스는 `public` 또는 `internal`일 수 있습니다. 기본적으로 클래스는 `internal`입니다.

중첩 클래스를 포함한 클래스 멤버는 `public`, `protected` `internal`, `protected`, `internal`, `private` 또는 `private protected`일 수 있습니다. 기본적으로 멤버는 `private`입니다.

자세한 내용은 [액세스 한정자](#)를 참조하세요.

형식 매개 변수가 포함된 제네릭 클래스를 선언할 수 있습니다. 자세한 내용은 [제네릭 클래스](#)를 참조하세요.

클래스에는 다음 멤버의 선언이 포함될 수 있습니다.

- [생성자](#)
- [상수](#)
- [필드](#)
- [종료자](#)
- [메서드](#)
- [속성](#)
- [인덱서](#)
- [연산자](#)
- [이벤트](#)
- [대리자](#)

- 클래스
- 인터페이스
- 구조체 형식
- 열거형 형식

## 예제

다음 예제에서는 클래스 필드, 생성자 및 메서드를 선언하는 방법을 보여 줍니다. 또한 개체 인스턴스화 및 인스턴스 데이터 출력을 보여 줍니다. 이 예제에서는 두 개의 클래스가 선언됩니다. 첫 번째 클래스인 `Child`는 `private` 필드 2개(`name` 및 `age`), `public` 생성자 2개, `public` 메서드 1개를 포함합니다. 두 번째 클래스인 `StringTest`는 `Main`을 포함하는 데 사용됩니다.

```

class Child
{
    private int age;
    private string name;

    // Default constructor:
    public Child()
    {
        name = "N/A";
    }

    // Constructor:
    public Child(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    // Printing method:
    public void PrintChild()
    {
        Console.WriteLine("{0}, {1} years old.", name, age);
    }
}

class StringTest
{
    static void Main()
    {
        // Create objects by using the new operator:
        Child child1 = new Child("Craig", 11);
        Child child2 = new Child("Sally", 10);

        // Create an object using the default constructor:
        Child child3 = new Child();

        // Display results:
        Console.Write("Child #1: ");
        child1.PrintChild();
        Console.Write("Child #2: ");
        child2.PrintChild();
        Console.Write("Child #3: ");
        child3.PrintChild();
    }
}
/* Output:
   Child #1: Craig, 11 years old.
   Child #2: Sally, 10 years old.
   Child #3: N/A, 0 years old.
*/

```

## 의견

이전 예제에서 `private` 필드(`name` 및 `age`)는 `Child` 클래스의 `public` 메서드를 통해서만 액세스할 수 있습니다. 예를 들어 다음과 같은 문을 사용하여 `Main` 메서드에서 자신의 이름을 출력할 수 없습니다.

```
Console.WriteLine(child1.name); // Error
```

`Main` 이 클래스의 멤버인 경우에만 `Main`에서 `Child`의 `private` 멤버에 액세스할 수 있습니다.

액세스 한정자 없이 클래스 내부에 선언된 형식은 기본적으로 `private`로 설정되므로 키워드가 제거된 경우 이 예제의 데이터 멤버는 `private`입니다.

마지막으로 매개 변수 없는 생성자(`child3`)를 사용하여 만들어진 객체의 경우 `age` 필드는 기본적으로 0으로 초기화되었습니다.

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [참조 형식](#)

# interface(C# 참조)

2021-02-18 • 8 minutes to read • [Edit Online](#)

인터페이스는 계약을 정의합니다. 해당 계약을 구현 하는 `class` 또는 `struct`는 인터페이스에 정의된 구성원의 구현을 제공해야 합니다. C# 8.0부터 인터페이스는 구성원에 대한 기본 구현을 정의할 수 있습니다. 공통적인 기능에 대해 단일 구현을 제공하기 위해 `static` 구성원을 정의할 수도 있습니다.

다음 예제에서 `ImplementationClass` 클래스는 매개 변수가 없고 `void`를 반환하는 `SampleMethod`라는 메서드를 구현해야 합니다.

자세한 내용과 예제는 [인터페이스](#)를 참조하세요.

## 예제

```
interface ISampleInterface
{
    void SampleMethod();
}

class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation:
    void ISampleInterface.SampleMethod()
    {
        // Method implementation.
    }

    static void Main()
    {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();

        // Call the member.
        obj.SampleMethod();
    }
}
```

인터페이스는 네임스페이스 또는 클래스의 구성원일 수 있습니다. 인터페이스 선언에는 다음 구성원의 선언(구현이 없는 서명)을 포함할 수 있습니다.

- [메서드](#)
- [속성](#)
- [인덱서](#)
- [이벤트](#)

이러한 앞의 구성원 선언에는 일반적으로 본문이 포함되지 않습니다. C# 8.0부터는 인터페이스 구성원이 본문을 선언할 수 있습니다. 이를 [기본 구현](#)이라고 합니다. 본문이 있는 구성원은 인터페이스가 재정의 구현을 제공하지 않는 클래스 및 구조체에 대해 "기본" 구현을 제공하도록 허용할 수 있습니다. 또한 C# 8.0부터 인터페이스에는 다음이 포함될 수 있습니다.

- [상수](#)
- [연산자](#)
- [정적 생성자](#).
- [중첩 형식](#)

- 정적 필드, 메서드, 속성, 인덱서 및 이벤트
- 명시적 인터페이스 구현 구문을 사용한 구성원 선언
- 명시적 액세스 한정자(기본 액세스는 `public`)

인터페이스에는 인스턴스 상태를 포함할 수 없습니다. 이제 정적 필드가 허용되지만 인터페이스에서는 인스턴스 필드가 허용되지 않습니다. **인스턴스 자동 속성**은 숨겨진 필드를 암시적으로 선언하므로 인터페이스에서 지원되지 않습니다. 이 규칙은 속성 선언에 미묘한 영향을 미칩니다. 인터페이스 선언에서 다음 코드는 `class` 또는 `struct`에서와 같이 자동으로 구현된 속성을 선언하지 않습니다. 대신, 기본 구현은 없지만 인터페이스를 구현하는 형식에서 구현되어야 하는 속성을 선언합니다.

```
public interface INamed
{
    public string Name {get; set;}
}
```

인터페이스는 하나 이상의 기본 인터페이스에서 상속할 수 있습니다. 인터페이스가 기본 인터페이스에 구현된 **메서드를 재정의**하는 경우 **명시적 인터페이스 구현** 구문을 사용해야 합니다.

기본 형식 목록에 기본 클래스 및 인터페이스가 포함된 경우 기본 클래스가 목록의 첫 번째 항목이어야 합니다.

인터페이스를 구현하는 클래스는 해당 인터페이스의 멤버를 명시적으로 구현할 수 있습니다. 명시적으로 구현된 멤버는 클래스 인스턴스를 통해 액세스할 수 있고 인터페이스 인스턴스를 통해서만 액세스할 수 있습니다. 또한 기본 인터페이스 구성원은 인터페이스의 인스턴스를 통해서만 액세스할 수 있습니다.

명시적 인터페이스 구현에 대한 자세한 내용은 [명시적 인터페이스 구현](#)을 참조하세요.

## 예제

다음 예제에서는 인터페이스의 구현 방법을 보여 줍니다. 이 예제에서 인터페이스에는 속성 선언이 포함되어 있고, 클래스에는 구현이 포함되어 있습니다. `IPoint`를 구현하는 클래스의 모든 인스턴스에는 정수 속성 `x` 및 `y`가 있습니다.

```

interface IPPoint
{
    // Property signatures:
    int X
    {
        get;
        set;
    }

    int Y
    {
        get;
        set;
    }

    double Distance
    {
        get;
    }
}

class Point : IPPoint
{
    // Constructor:
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    // Property implementation:
    public int X { get; set; }

    public int Y { get; set; }

    // Property implementation
    public double Distance =>
        Math.Sqrt(X * X + Y * Y);
}

class MainClass
{
    static void PrintPoint(IPPoint p)
    {
        Console.WriteLine("x={0}, y={1}", p.X, p.Y);
    }

    static void Main()
    {
        IPPoint p = new Point(2, 3);
        Console.Write("My Point: ");
        PrintPoint(p);
    }
}
// Output: My Point: x=2, y=3

```

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 인터페이스 섹션](#) 및 [기본 인터페이스 구성원 - C# 8.0](#)의 기능 사양을 참조하세요.

## 참고 항목

- [C# 참조](#)

- C# 프로그래밍 가이드
- C# 키워드
- 참조 형식
- 인터페이스
- 속성 사용
- 인덱서 사용

# nullable 참조 형식(C# 참조)

2021-02-18 • 14 minutes to read • [Edit Online](#)

## NOTE

이 문서에서는 nullable 참조 형식을 설명합니다. [nullable 값 형식](#)을 선언할 수도 있습니다.

nullable 참조 형식은 'nullable 인식 컨텍스트'에 옵트인된 코드에서 C# 8.0부터 사용할 수 있습니다. nullable 참조 형식, null 정적 분석 경고 및 [null 허용 연산자](#)는 선택적 언어 기능입니다. 모두 기본적으로 꺼져 있습니다. 'nullable 컨텍스트'는 빌드 설정을 사용하여 프로젝트 수준에서 제어되거나 pragma를 사용하여 빌드 설정에서 제어됩니다.

nullable 인식 컨텍스트에서:

- 참조 형식 `T`의 변수는 `null`이 아닌 값으로 초기화되어야 하며, `null`일 수 있는 값이 할당되지 않을 수 있습니다.
- 참조 형식 `T?`의 변수는 `null`로 초기화되거나 `null`이 할당될 수 있지만, 역참조하기 전에 `null`에 대해 확인되어야 합니다.
- `m!`의 경우와 같이 null 허용 연산자를 적용하면 `T?` 형식의 `m` 변수는 `null`이 아닌 것으로 간주합니다.

`null`을 허용하지 않는 참조 형식과 `T` nullable 참조 형식 `T?` 간 차이점은 컴파일러의 이전 규칙 해석에 따라 적용됩니다. `T` 형식의 변수 및 `T?` 형식의 변수는 동일한 .NET 형식으로 나타냅니다. 다음 예제에서는 `null`을 허용하지 않는 문자열 및 nullable 문자열을 선언하고 null 허용 연산자를 사용하여 `null`을 허용하지 않는 문자열에 값을 할당합니다.

```
string notNull = "Hello";
string? nullable = default;
notNull = nullable!; // null forgiveness
```

`notNull` 및 `nullable` 변수는 둘 다 `String` 형식으로 나타냅니다. `null`을 허용하지 않는 형식 및 nullable 형식은 둘 다 같은 형식으로 저장되므로 nullable 참조 형식을 사용할 수 있는 여러 위치가 있습니다. 일반적으로 nullable 참조 형식은 기본 클래스 또는 구현된 인터페이스로 사용할 수 없습니다. nullable 참조 형식은 개체 생성 또는 형식 테스트 식에 사용할 수 없습니다. nullable 참조 형식은 멤버 액세스 식의 형식일 수 없습니다. 다음 예제에서는 다음 구문을 보여 줍니다.

```
public MyClass : System.Object? // not allowed
{
}

var nullEmpty = System.String?.Empty; // Not allowed
var maybeObject = new object?(); // Not allowed
try
{
    if (thing is string? nullableString) // not allowed
        Console.WriteLine(nullableString);
} catch (Exception? e) // Not Allowed
{
    Console.WriteLine("error");
}
```

## nullable 참조 및 정적 분석

이전 섹션의 예제에서는 nullable 참조 형식의 특성을 보여 줍니다. nullable 참조 형식은 새 클래스 형식이 아니라 기존 참조 형식의 주석입니다. 컴파일러는 해당 주석을 사용하여 코드에서 잠재적 null 참조 오류를 찾을 수 있습니다. null을 허용하지 않는 참조 형식과 nullable 참조 형식 간에는 런타임 차이가 없습니다. 컴파일러는 null을 허용하지 않는 참조 형식에 대한 런타임 검사를 추가하지 않습니다. 컴파일 시간 분석에는 이점이 있습니다. 컴파일러는 코드에서 잠재적 null 오류를 찾고 해결하는 데 도움이 되는 경고를 생성합니다. 의도를 선언하고 코드가 해당 의도를 위반하면 컴파일러가 경고를 표시합니다.

nullable 사용 컨텍스트에서 컴파일러는 nullable 참조 형식 및 null을 허용하지 않는 참조 형식의 변수에서 정적 분석을 수행합니다. 컴파일러는 각 참조 변수의 null 상태를 'null이 아님' 또는 'null일 수 있음'으로 추적합니다. null을 허용하지 않는 참조의 기본 상태는 'null이 아님'입니다. nullable 참조의 기본 상태는 'null일 수 있음'입니다.

null을 허용하지 않는 참조 형식은 해당 null 상태가 'null이 아님'이므로 항상 안전하게 역참조할 수 있습니다. 해당 규칙을 적용하기 위해 null을 허용하지 않는 참조 형식이 null이 아닌 값으로 초기화되지 않는 경우 컴파일러는 경고를 실행합니다. 지역 변수는 선언된 위치에 할당되어야 합니다. 모든 생성자는 본문, 호출된 생성자 또는 필드 이니셜라이저를 사용하여 모든 필드를 할당해야 합니다. 상태가 'null일 수 있음'인 참조에 null을 허용하지 않는 참조가 할당되는 경우 컴파일러는 경고를 실행합니다. 그러나 null을 허용하지 않는 참조는 'null이 아님'이므로 해당 변수가 역참조될 때 경고가 실행되지 않습니다.

nullable 참조 형식은 `null`에 초기화되거나 할당될 수 있습니다. 따라서 정적 분석에서는 역참조되기 전에 변수가 'null이 아님'인지 확인해야 합니다. nullable 참조가 'null일 수 있음'으로 확인되면 해당 참조는 null을 허용하지 않는 참조 변수에 할당될 수 없습니다. 다음 클래스는 해당 경고의 예를 보여 줍니다.

```

public class ProductDescription
{
    private string shortDescription;
    private string? detailedDescription;

    public ProductDescription() // Warning! short description not initialized.
    {
    }

    public ProductDescription(string productDescription) =>
        this.shortDescription = productDescription;

    public void SetDescriptions(string productDescription, string? details=null)
    {
        shortDescription = productDescription;
        detailedDescription = details;
    }

    public string GetDescription()
    {
        if (detailedDescription.Length == 0) // Warning! dereference possible null
        {
            return shortDescription;
        }
        else
        {
            return $"{shortDescription}\n{detailedDescription}";
        }
    }

    public string FullDescription()
    {
        if (detailedDescription == null)
        {
            return shortDescription;
        }
        else if (detailedDescription.Length > 0) // OK, detailedDescription can't be null.
        {
            return $"{shortDescription}\n{detailedDescription}";
        }
        return shortDescription;
    }
}

```

다음 코드 조각은 이 클래스를 사용하는 경우 컴파일러가 경고를 내보내는 위치를 보여 줍니다.

```

string shortDescription = default; // Warning! non-nullable set to null;
var product = new ProductDescription(shortDescription); // Warning! static analysis knows shortDescription
maybe null.

string description = "widget";
var item = new ProductDescription(description);

item.SetDescriptions(description, "These widgets will do everything.");

```

앞의 예제에서는 참조 변수의 null 상태를 확인하는 컴파일러의 정적 분석을 보여 줍니다. 컴파일러는 null 검사 및 할당에 대한 언어 규칙을 적용하여 분석에 대해 알립니다. 컴파일러는 메서드 또는 속성의 의미 체계를 가정할 수 없습니다. Null 검사를 수행하는 메서드를 호출하는 경우 컴파일러는 해당 메서드가 변수의 null 상태에 영향을 준다는 것을 알 수 없습니다. 컴파일러에 인수 및 반환 값의 의미 체계를 알리는 여러 가지 특성을 API에 추가할 수 있습니다. 해당 특성은 .NET Core 라이브러리의 여러 일반적인 API에 적용되었습니다. 예를 들어 [IsNullOrEmpty](#)가 업데이트되었으며 컴파일러는 해당 메서드를 null 검사로 올바르게 해석합니다. Null 상태 정적 분석에 적용되는 특성에 대한 자세한 내용은 [nullable 특성](#) 문서를 참조하세요.

## nullable 컨텍스트 설정

두 가지 방법으로 nullable 컨텍스트를 제어할 수 있습니다. 프로젝트 수준에서 `<Nullable>enable</Nullable>` 프로젝트 설정을 추가할 수 있습니다. 단일 C# 소스 파일에서 `#nullable enable` pragma를 추가하여 nullable 컨텍스트를 사용하도록 설정할 수 있습니다. [nullable 전략 설정](#) 문서를 참조하세요.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 다음 제안을 참조하세요.

- [nullable 참조 형식](#)
- [Nullable 참조 형식 사양 초안](#)

## 참조

- [C# 참조](#)
- [Nullable 값 형식](#)

# void(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

`void`를 [메서드](#)(또는 [로컬 함수](#))의 반환 형식으로 사용하여 메서드가 값을 반환하지 않도록 지정합니다.

```
public static void Display(IEnumerable<int> numbers)
{
    if (numbers is null)
    {
        return;
    }

    Console.WriteLine(string.Join(" ", numbers));
}
```

`void`을 참조 형식으로 사용하여 알 수 없는 형식에 대한 포인터를 선언할 수도 있습니다. 자세한 내용은 [포인터 형식](#)을 참조하세요.

`void`은 변수 형식으로 사용할 수 없습니다.

## 참고 항목

- [C# 참조](#)
- [System.Void](#)

# var (C# 참조)

2020-11-02 • 4 minutes to read • [Edit Online](#)

C# 3부터 메서드 범위에서 선언된 변수에 암시적 “형식” `var` 을 사용할 수 있습니다. 암시적 형식 지역 변수는 형식을 직접 선언한 것처럼 강력한 형식이지만 컴파일러가 형식을 결정합니다. `i` 의 다음 두 선언은 기능이 동일합니다.

```
var i = 10; // Implicitly typed.  
int i = 10; // Explicitly typed.
```

## IMPORTANT

`null` 허용 참조 형식을 사용하도록 설정하고 `var` 를 사용하면 식 형식이 `null` 허용이 아니더라도 항상 `null` 허용 참조 형식을 의미합니다.

`var` 키워드는 일반적으로 생성자 호출 식과 함께 사용됩니다. `var` 를 사용하면 다음 예제와 같이 변수 선언 및 개체 인스턴스화에서 형식 이름을 반복하지 않을 수 있습니다.

```
var xs = new List<int>();
```

C# 9.0부터 대상으로 형식화된 `new` 식을 대신 사용할 수 있습니다.

```
List<int> xs = new();  
List<int>? ys = new();
```

## 예제

다음 예제에서는 두 가지 쿼리 식을 보여 줍니다. 첫 번째 식에서는 `var` 을 사용할 수 있지만, 쿼리 결과의 형식을 `IEnumerable<string>` 으로 명시적으로 정의할 수 있기 때문에 필요하지 않습니다. 그러나 두 번째 식에서 `var` 은 결과가 익명 형식의 컬렉션이 되도록 허용하고 해당 형식의 이름은 컴파일러 자체에만 액세스할 수 있습니다. `var` 을 사용하면 결과에 대한 새 클래스를 만들 필요가 없습니다. 예제 #2에서는 `foreach` 반복 변수 `item` 도 암시적 형식이어야 합니다.

```

// Example #1: var is optional when
// the select clause specifies a string
string[] words = { "apple", "strawberry", "grape", "peach", "banana" };
var wordQuery = from word in words
    where word[0] == 'g'
    select word;

// Because each element in the sequence is a string,
// not an anonymous type, var is optional here also.
foreach (string s in wordQuery)
{
    Console.WriteLine(s);
}

// Example #2: var is required because
// the select clause specifies an anonymous type
var custQuery = from cust in customers
    where cust.City == "Phoenix"
    select new { cust.Name, cust.Phone };

// var must be used because each item
// in the sequence is an anonymous type
foreach (var item in custQuery)
{
    Console.WriteLine("Name={0}, Phone={1}", item.Name, item.Phone);
}

```

## 참고 항목

- [C# 참조](#)
- [암시적 형식 지역 변수](#)
- [LINQ 쿼리 작업의 형식 관계](#)

# 기본 제공 형식(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

다음 표에서는 C# 기본 제공 값 형식을 나열합니다.

C# 형식 키워드	.NET 형식
<code>bool</code>	<code>System.Boolean</code>
<code>byte</code>	<code>System.Byte</code>
<code>sbyte</code>	<code>System.SByte</code>
<code>char</code>	<code>System.Char</code>
<code>decimal</code>	<code>System.Decimal</code>
<code>double</code>	<code>System.Double</code>
<code>float</code>	<code>System.Single</code>
<code>int</code>	<code>System.Int32</code>
<code>uint</code>	<code>System.UInt32</code>
<code>long</code>	<code>System.Int64</code>
<code>ulong</code>	<code>System.UInt64</code>
<code>short</code>	<code>System.Int16</code>
<code>ushort</code>	<code>System.UInt16</code>

다음 표에서는 C# 기본 제공 참조 형식을 나열합니다.

C# 형식 키워드	.NET 형식
<code>object</code>	<code>System.Object</code>
<code>string</code>	<code>System.String</code>
<code>dynamic</code>	<code>System.Object</code>

이전 표에서 왼쪽 열의 각 C# 형식 키워드는 해당하는 .NET 형식의 별칭입니다. 서로 교환하여 사용할 수 있습니다. 예를 들어 다음 선언은 동일한 형식의 변수를 선언합니다.

```
int a = 123;  
System.Int32 b = 123;
```

**void** 키워드는 형식이 없음을 나타냅니다. 이 키워드는 값을 반환하지 않는 메서드의 반환 형식으로 사용합니다.

## 참조

- [C# 참조](#)
- [C# 형식의 기본값](#)

# 비관리형 형식(C# 참조)

2020-11-02 • 3 minutes to read • [Edit Online](#)

형식이 다음 형식 중 하나인 경우에는 관리되지 않는 형식입니다.

- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal` 또는 `bool`
- 임의의 [열거형](#) 형식
- 임의의 [포인터](#) 형식
- 관리되지 않는 형식의 필드만 포함하고 C# 7.3 및 이전 버전에서 사용자 정의된 [구조체](#) 형식은 생성 형식(하나 이상의 형식 인수를 포함하는 형식)이 아닙니다.

C# 7.3부터 `unmanaged` 제약 조건을 사용하여 형식 매개 변수가 nullable이 아니며 비포인터 및 비관리형 형식임을 지정할 수 있습니다.

C# 8.0부터는 다음 예와 같이 관리되지 않는 형식의 필드만 포함하는 생성된 구조체 형식도 관리되지 않습니다.

```
using System;

public struct Coords<T>
{
    public T X;
    public T Y;
}

public class UnmanagedTypes
{
    public static void Main()
    {
        DisplaySize<Coords<int>>();
        DisplaySize<Coords<double>>();
    }

    private unsafe static void DisplaySize<T>() where T : unmanaged
    {
        Console.WriteLine($"{typeof(T)} is unmanaged and its size is {sizeof(T)} bytes");
    }
}

// Output:
// Coords`1[System.Int32] is unmanaged and its size is 8 bytes
// Coords`1[System.Double] is unmanaged and its size is 16 bytes
```

제네릭 구조체는 관리되는 구조체 및 관리되지 않는 구조체 형식 모두의 원본일 수 있습니다. 위의 예에서는 제네릭 구조체 `Coords<T>`를 정의하고 관리되지 않는 생성 형식의 예를 보여 줍니다. 관리되지 않는 형식이 아닌 예는 `Coords<object>`입니다. 관리되지 않는 `object` 형식의 필드가 있기 때문에 관리되지 않습니다. 모든 생성 형식을 관리되지 않는 형식으로 하려면 제네릭 구조체의 정의에서 `unmanaged` 제약 조건을 사용합니다.

```
public struct Coords<T> where T : unmanaged
{
    public T X;
    public T Y;
}
```

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

## 참고 항목

- [C# 참조](#)
- [포인터 형식](#)
- [메모리 및 범위 관련 형식](#)
- [sizeof 연산자](#)
- [stackalloc](#)

# C# 형식의 기본값(C# 참조)

2020-04-02 • 3 minutes to read • [Edit Online](#)

다음 표는 C# 형식의 기본값을 보여줍니다.

TYPE	기본값
임의 참조 형식	<code>null</code>
임의 기본 제공 정수 숫자 유형	0(영)
임의 기본 제공 부동 소수점 숫자 유형	0(영)
<code>bool</code>	<code>false</code>
<code>char</code>	<code>'\0'</code> (U+0000)
<code>enum</code>	식 <code>(E)0</code> 로 생성한 값이며 여기서 <code>E</code> 는 열거형 식별자입니다.
<code>struct</code>	모든 값 형식 필드를 기본값으로 설정하고 모든 참조 형식 필드를 <code>null</code> 로 설정하여 생성한 값입니다.
Any Null 허용 값 형식	<code>HasValue</code> 속성은 <code>false</code> 이고 <code>Value</code> 속성은 정의되지 않은 인스턴스입니다. 이 기본값은 null 허용 값 형식의 <code>null</code> 값으로 알려져 있습니다.

`default` 연산자를 사용하여 다음 예제와 같이 형식의 기본값을 생성합니다.

```
int a = default(int);
```

C# 7.1부터 `default` 리터럴을 사용하여 해당 형식의 기본값으로 변수를 초기화할 수 있습니다.

```
int a = default;
```

값 형식의 경우 암시적 매개 변수 없는 생성자를 사용하여 다음 예제와 같은 형식의 기본값도 생성할 수 있습니다.

```
var n = new System.Numerics.Complex();
Console.WriteLine(n); // output: (0, 0)
```

런타임에 `System.Type` 인스턴스가 값 형식을 나타내는 경우, `Activator.CreateInstance(Type)` 메서드를 사용하면 매개 변수가 없는 생성자를 호출하여 형식의 기본값을 가져올 수 있습니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 다음 섹션을 참조하세요.

- [기본값](#)

- 기본 생성자

## 참고 항목

- C# 참조
- 생성자

# C# 키워드

2021-02-18 • 3 minutes to read • [Edit Online](#)

키워드는 컴파일러에 대해 특별한 의미를 갖는, 미리 정의되어 있는 예약된 식별자입니다. 키워드는 프로그램에서 식별자로 사용되려면 접두어로 `@`을 포함해야 합니다. 예를 들어 `@if`는 올바른 식별자이지만 `if`는 `if`가 키워드이므로 식별자로 적절하지 않습니다.

이 항목의 첫 번째 표에는 C# 프로그램의 모든 부분에서 예약된 식별자로 사용되는 키워드가 나와 있습니다. 이 항목의 두 번째 표에는 C#의 상황별 키워드가 나와 있습니다. 상황별 키워드는 제한된 프로그램 컨텍스트에서만 특별한 의미를 가지며 해당 컨텍스트 외부에서는 식별자로 사용될 수 있습니다. 일반적으로 새 키워드는 C# 언어에 추가될 때 이전 버전에서 작성된 프로그램을 중단하지 않도록 하기 위해 상황별 키워드로 추가됩니다.

abstract	as	base	bool
break	byte	case	catch
char	checked	class	const
continue	decimal	default	delegate
do	double	else	enum
event	explicit	extern	false
finally	fixed	float	for
foreach	goto	if	implicit
in	int	interface	internal
is	lock	long	namespace
new	null	object	operator
out	override	params	private
protected	public	readonly	ref
return	sbyte	sealed	short
sizeof	stackalloc	static	string
struct	switch	this	throw
true	try	typeof	uint
ulong	unchecked	unsafe	ushort

using	virtual	void	volatile	
while				

## 상황별 키워드

상황별 키워드는 코드에서 특정 의미를 제공하는 데 사용되지만 C#의 예약어입니다. `partial` 및 `where`과 같은 일부 상황별 키워드는 두 개 이상의 컨텍스트에서 특별한 의미를 갖습니다.

add	alias	ascending
async	await	by
descending	dynamic	equals
from	get	global
group	into	join
let	nameof	notnull
on	orderby	partial(형식)
partial(메서드)	remove	select
set	비관리형 제네릭 형식 제약 조건	value
var	when(필터 조건)	where(제네릭 형식 제약 조건)
where(쿼리 절)	with	yield

## 참조

- [C# 참조](#)

# 액세스 한정자(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

액세스 한정자는 멤버 또는 형식의 선언된 접근성을 지정하는 데 사용되는 키워드입니다. 이 섹션에서는 다음 네 가지 액세스 한정자를 소개합니다.

- `public`
- `protected`
- `internal`
- `private`

액세스 한정자를 사용하여 다음 여섯 가지 접근성 수준을 지정할 수 있습니다.

- `public`: 액세스가 제한되지 않습니다.
- `protected`: 액세스가 포함하는 클래스 또는 포함하는 클래스에서 파생된 형식으로 제한됩니다.
- `internal`: 액세스가 현재 어셈블리로 제한됩니다.
- `protected internal`: 액세스가 현재 어셈블리 또는 포함하는 클래스에서 파생된 형식으로 제한됩니다.
- `private`: 액세스가 포함하는 형식으로 제한됩니다.
- `private protected`: 액세스가 포함하는 클래스 또는 현재 어셈블리 내의 포함하는 클래스에서 파생된 형식으로 제한됩니다.

이 섹션에서는 다음 내용도 소개합니다.

- [접근성 수준](#): 네 가지 액세스 한정자를 사용하여 여섯 가지 접근성 수준을 선언합니다.
- [접근성 도메인](#): 프로그램 섹션에서 멤버를 참조할 수 있는 위치를 지정합니다.
- [접근성 수준 사용에 대한 제한](#): 선언된 접근성 수준 사용에 대한 제한 사항 요약입니다.

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [액세스 한정자](#)
- [액세스 키워드](#)
- [한정자](#)

# 액세스 가능성 수준(C# 참조)

2020-11-02 • 5 minutes to read • [Edit Online](#)

액세스 한정자 `public`, `protected`, `internal` 또는 `private`을 사용하여 멤버에 대해 다음과 같이 선언된 접근성 수준 중 하나를 지정합니다.

선언된 액세스 가능성	의미
<code>public</code>	액세스가 제한되지 않습니다.
<code>protected</code>	액세스가 포함하는 클래스 또는 포함하는 클래스에서 파생된 형식으로 제한됩니다.
<code>internal</code>	액세스가 현재 어셈블리로 제한됩니다.
<code>protected internal</code>	액세스가 현재 어셈블리 또는 포함하는 클래스에서 파생된 형식으로 제한됩니다.
<code>private</code>	액세스가 포함하는 형식으로 제한됩니다.
<code>private protected</code>	액세스가 포함하는 클래스 또는 현재 어셈블리 내의 포함하는 클래스에서 파생된 형식으로 제한됩니다. C# 7.2부터 사용할 수 있습니다.

`protected internal` 또는 `private protected` 조합을 사용할 경우를 제외하고 멤버 또는 형식에는 액세스 한정자가 하나만 허용됩니다.

네임스페이스에는 액세스 한정자가 허용되지 않습니다. 네임스페이스에는 액세스 제한이 없습니다.

멤버 선언이 발생한 컨텍스트에 따라 특정 선언된 액세스 가능성만 허용됩니다. 액세스 한정자가 멤버 선언에서 지정되지 않으면 기본 액세스 가능성이 사용됩니다.

다른 형식에 종첩되지 않은 최상위 형식에는 `internal` 또는 `public` 액세스 가능성만 포함될 수 있습니다. 이러한 형식에 대한 기본 액세스 가능성이 `internal`입니다.

다음 표에 나와 있는 대로 다른 형식의 멤버인 종첩 형식에는 선언된 액세스 가능성이 포함될 수 있습니다.

소속 그룹	기본 멤버 액세스 가능성	멤버의 허용된 선언된 액세스 가능성
<code>enum</code>	<code>public</code>	None
<code>class</code>	<code>private</code>	<code>public</code> <code>protected</code> <code>internal</code> <code>private</code> <code>protected internal</code> <code>private protected</code>

소속 그룹	기본 멤버 액세스 가능성	멤버의 허용된 선언된 액세스 가능성
<code>interface</code>	<code>public</code>	없음
<code>struct</code>	<code>private</code>	<code>public</code> <code>internal</code> <code>private</code>

중첩된 형식의 액세스 가능성은 액세스 가능 도메인에 따라 다릅니다. [액세스 가능성 도메인](#)은 멤버에 대해 선언된 액세스 가능성 및 한 수준 위 형식의 액세스 가능성 도메인에 의해 결정됩니다. 그러나 중첩 형식의 액세스 가능 도메인은 포함하는 형식의 액세스 가능 도메인을 벗어날 수는 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

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [액세스 한정자](#)
- [내게 필요한 옵션 도메인](#)
- [내게 필요한 옵션 수준 사용에 대한 제한](#)
- [액세스 한정자](#)
- [public](#)
- [private](#)
- [protected](#)
- [internal](#)

# 액세스 가능 도메인(C# 참조)

2020-11-02 • 5 minutes to read • [Edit Online](#)

멤버의 액세스 가능 도메인은 멤버가 참조될 수 있는 프로그램 섹션을 지정합니다. 멤버가 다른 형식 내에 중첩 되면 해당 액세스 가능 도메인은 멤버의 [액세스 가능성 수준](#) 및 한 수준 위 형식의 액세스 가능 도메인에 의해 결정됩니다.

최상위 형식의 액세스 가능 도메인은 최소한 최상위 형식이 선언된 프로젝트의 프로그램 텍스트입니다. 즉, 도메인에는 이 프로젝트의 모든 소스 파일이 포함됩니다. 중첩 형식의 액세스 가능 도메인은 최소한 중첩 형식이 선언된 프로젝트의 프로그램 텍스트입니다. 즉, 도메인은 모든 중첩 형식이 포함된 형식 본문입니다. 중첩 형식의 액세스 가능 도메인은 포함하는 형식의 액세스 가능 도메인을 초과하지 않습니다. 다음 예제에서는 이러한 개념을 보여 줍니다.

## 예제

이 예제에는 최상위 형식 `T1` 과 두 개의 중첩 클래스 `M1` 및 `M2` 가 포함됩니다. 클래스에는 여러 가지 선언된 액세스 가능성을 가진 필드가 포함됩니다. `Main` 메서드에서 각 문 뒤에는 각 멤버의 액세스 가능성 도메인을 나타내는 주석이 있습니다. 액세스할 수 없는 멤버를 참조하려고 하는 문은 주석으로 처리됩니다. 액세스할 수 없는 멤버를 참조함으로써 발생한 컴파일러 오류를 확인하려면 한 번에 하나씩 주석을 제거합니다.

```
public class T1
{
    public static int publicInt;
    internal static int internalInt;
    private static int privateInt = 0;

    static T1()
    {
        // T1 can access public or internal members
        // in a public or private (or internal) nested class.
        M1.publicInt = 1;
        M1.internalInt = 2;
        M2.publicInt = 3;
        M2.internalInt = 4;

        // Cannot access the private member privateInt
        // in either class:
        // M1.privateInt = 2; //CS0122
    }

    public class M1
    {
        public static int publicInt;
        internal static int internalInt;
        private static int privateInt = 0;
    }

    private class M2
    {
        public static int publicInt = 0;
        internal static int internalInt = 0;
        private static int privateInt = 0;
    }
}

class MainClass
{
    static void Main()
```

```

{
    // Access is unlimited.
    T1.publicInt = 1;

    // Accessible only in current assembly.
    T1.internalInt = 2;

    // Error CS0122: inaccessible outside T1.
    // T1.privateInt = 3;

    // Access is unlimited.
    T1.M1.publicInt = 1;

    // Accessible only in current assembly.
    T1.M1.internalInt = 2;

    // Error CS0122: inaccessible outside M1.
    //     T1.M1.privateInt = 3;

    // Error CS0122: inaccessible outside T1.
    //     T1.M2.publicInt = 1;

    // Error CS0122: inaccessible outside T1.
    //     T1.M2.internalInt = 2;

    // Error CS0122: inaccessible outside M2.
    //     T1.M2.privateInt = 3;

    // Keep the console open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
}

```

## 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# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [액세스 한정자](#)
- [액세스 수준](#)
- [내계 필요한 옵션 수준 사용에 대한 제한](#)
- [액세스 한정자](#)
- [public](#)
- [private](#)
- [protected](#)
- [internal](#)

# 액세스 가능성 수준 사용에 대한 제한(C# 참조)

2020-11-02 • 6 minutes to read • [Edit Online](#)

선언에서 형식을 지정하는 경우, 형식의 액세스 가능성 수준이 멤버 또는 다른 형식의 액세스 가능성 수준에 따라 달라지는지를 확인합니다. 예를 들어 직접 기본 클래스는 적어도 파생 클래스 수준만큼 액세스 가능해야 합니다. 다음 선언에서는 기본 클래스 `BaseClass` 가 `MyClass` 보다 액세스 가능성이 낮기 때문에 컴파일러 오류가 발생합니다.

```
class BaseClass {...}  
public class MyClass: BaseClass {...} // Error
```

다음 표에는 선언된 액세스 가능성 수준에 대한 제한이 요약되어 있습니다.

CONTEXT	설명
클래스	클래스 형식의 직접 기본 클래스는 적어도 클래스 형식 자체 수준만큼 액세스 가능해야 합니다.
인터페이스	인터페이스 형식의 명시적 기본 인터페이스는 적어도 인터페이스 형식 자체 수준만큼 액세스 가능해야 합니다.
대리자	대리자 형식의 반환 형식 및 매개 변수 형식은 적어도 대리자 형식 자체 수준만큼 액세스 가능해야 합니다.
상수	상수의 형식은 적어도 상수 자체 수준만큼 액세스 가능해야 합니다.
필드	필드의 형식은 적어도 필드 자체 수준만큼 액세스 가능해야 합니다.
메서드	메서드의 반환 형식 및 매개 변수 형식은 적어도 메서드 자체 수준만큼 액세스 가능해야 합니다.
속성	속성의 형식은 적어도 속성 자체 수준만큼 액세스 가능해야 합니다.
이벤트	이벤트의 형식은 적어도 이벤트 자체 수준만큼 액세스 가능해야 합니다.
인덱서	인덱서의 형식 및 매개 변수 형식은 적어도 인덱서 자체 수준만큼 액세스 가능해야 합니다.
연산자	연산자의 반환 형식 및 매개 변수 형식은 적어도 연산자 자체 수준만큼 액세스 가능해야 합니다.
생성자	생성자의 매개 변수 형식은 적어도 생성자 자체 수준만큼 액세스 가능해야 합니다.

## 예제

다음 예제에는 다양한 종류의 잘못된 선언이 포함되어 있습니다. 각 선언 다음의 주석은 예상된 컴파일러 오류

를 나타냅니다.

```
// Restrictions on Using Accessibility Levels
// CS0052 expected as well as CS0053, CS0056, and CS0057
// To make the program work, change access level of both class B
// and MyPrivateMethod() to public.

using System;

// A delegate:
delegate int MyDelegate();

class B
{
    // A private method:
    static int MyPrivateMethod()
    {
        return 0;
    }
}

public class A
{
    // Error: The type B is less accessible than the field A.myField.
    public B myField = new B();

    // Error: The type B is less accessible
    // than the constant A.myConst.
    public readonly B myConst = new B();

    public B MyMethod()
    {
        // Error: The type B is less accessible
        // than the method A.MyMethod.
        return new B();
    }

    // Error: The type B is less accessible than the property A.MyProp
    public B MyProp
    {
        set
        {
        }
    }

    MyDelegate d = new MyDelegate(B.MyPrivateMethod);
    // Even when B is declared public, you still get the error:
    // "The parameter B.MyPrivateMethod is not accessible due to
    // protection level."

    public static B operator +(A m1, B m2)
    {
        // Error: The type B is less accessible
        // than the operator A.operator +(A,B)
        return new B();
    }

    static void Main()
    {
        Console.WriteLine("Compiled successfully");
    }
}
```

C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [액세스 한정자](#)
- [내게 필요한 옵션 도메인](#)
- [액세스 수준](#)
- [액세스 한정자](#)
- [public](#)
- [private](#)
- [protected](#)
- [internal](#)

# internal(C# 참조)

2020-11-02 • 5 minutes to read • [Edit Online](#)

`internal` 키워드는 형식 및 형식 멤버에 대한 [액세스 한정자](#)입니다.

이 페이지에서는 `internal` 액세스를 설명합니다. `internal` 키워드는 `protected internal` 액세스 한정자의 일부이기도 합니다.

내부 형식 또는 멤버는 다음 예제와 같이 동일한 어셈블리의 파일 내에서만 액세스할 수 있습니다.

```
public class BaseClass
{
    // Only accessible within the same assembly.
    internal static int x = 0;
}
```

`internal` 및 다른 액세스 한정자와 비교는 [액세스 가능성 수준](#) 및 [액세스 한정자](#)를 참조하세요.

어셈블리에 대한 자세한 내용은 [.NET 어셈블리](#)를 참조하세요.

내부 액세스는 구성 요소 그룹이 나머지 애플리케이션 코드에 노출되지 않고 비공개 방식으로 상호 작용할 수 있도록 하기 때문에 일반적으로 구성 요소 기반 개발에 사용됩니다. 예를 들어 그래픽 사용자 인터페이스를 빌드하기 위한 프레임워크는 내부 액세스로 멤버를 사용하여 상호 작용하는 `Control` 및 `Form` 클래스를 제공할 수 있습니다. 이러한 멤버는 `internal`이므로 프레임워크를 사용하는 코드에 노출되지 않습니다.

정의 시 사용된 어셈블리 외부에서 내부 액세스로 형식 또는 멤버를 참조하면 오류가 발생합니다.

## 예제

이 예제에는 `Assembly1.cs` 및 `Assembly1_a.cs`의 두 파일이 포함되어 있습니다. 첫 번째 파일에는 내부 기본 클래스인 `BaseClass`가 포함되어 있습니다. 두 번째 파일에서 `BaseClass`를 인스턴스화하려고 하면 오류가 발생합니다.

```
// Assembly1.cs
// Compile with: /target:library
internal class BaseClass
{
    public static int intM = 0;
}
```

```
// Assembly1_a.cs
// Compile with: /reference:Assembly1.dll
class TestAccess
{
    static void Main()
    {
        var myBase = new BaseClass(); // CS0122
    }
}
```

## 예제

이 예제에서는 예제 1에서 사용한 것과 동일한 파일을 사용하고 `BaseClass`의 액세스 가능성 수준을 `public`으로 변경합니다. 또한 `intM` 멤버의 액세스 가능성 수준을 `internal`로 변경합니다. 이 경우 클래스를 인스턴스화할 수 있지만 내부 멤버에는 액세스할 수 없습니다.

```
// Assembly2.cs
// Compile with: /target:library
public class BaseClass
{
    internal static int intM = 0;
}
```

```
// Assembly2_a.cs
// Compile with: /reference:Assembly2.dll
public class TestAccess
{
    static void Main()
    {
        var myBase = new BaseClass(); // Ok.
        BaseClass.intM = 444; // CS0117
    }
}
```

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 선언된 내게 필요한 옵션](#)을 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [액세스 한정자](#)
- [액세스 수준](#)
- [한정자](#)
- [public](#)
- [private](#)
- [protected](#)

# private(C# 참조)

2020-11-02 • 3 minutes to read • [Edit Online](#)

`private` 키워드는 멤버 액세스 한정자입니다.

이 페이지에서는 `private` 액세스를 설명합니다. `private` 키워드는 `private protected` 액세스 한정자의 일부이기도 합니다.

`private` 액세스는 가장 낮은 액세스 수준입니다. `private` 멤버는 이 예제와 같이 선언된 클래스 또는 구조체의 본문 내에서만 액세스할 수 있습니다.

```
class Employee
{
    private int i;
    double d; // private access by default
}
```

동일한 본문에 중첩된 형식도 이러한 `private` 멤버에 액세스할 수 있습니다.

선언된 클래스 또는 구조체 외부에서 `private` 멤버를 참조하면 컴파일 시간 오류가 발생합니다.

`private` 및 다른 액세스 한정자와 비교는 [액세스 가능성 수준](#) 및 [액세스 한정자](#)를 참조하세요.

## 예제

이 예제에서 `Employee` 클래스는 두 전용 데이터 멤버인 `name` 및 `salary`를 포함합니다. `private` 멤버는 멤버 메서드에 의한 경우를 제외하고 액세스할 수 없습니다. `GetName` 및 `Salary`라는 `public` 메서드는 `private` 멤버에 제어된 액세스 권한을 허용하기 위해 추가됩니다. `name` 멤버는 `public` 메서드를 통해 액세스하고, `salary` 멤버는 `public` 읽기 전용 속성을 통해 액세스합니다. 자세한 내용은 [속성](#)을 참조하세요.

```

class Employee2
{
    private string name = "FirstName, LastName";
    private double salary = 100.0;

    public string GetName()
    {
        return name;
    }

    public double Salary
    {
        get { return salary; }
    }
}

class PrivateTest
{
    static void Main()
    {
        var e = new Employee2();

        // The data members are inaccessible (private), so
        // they can't be accessed like this:
        //     string n = e.name;
        //     double s = e.salary;

        // 'name' is indirectly accessed via method:
        string n = e.GetName();

        // 'salary' is indirectly accessed via property
        double s = e.Salary;
    }
}

```

## C# 언어 사양

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

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [액세스 한정자](#)
- [액세스 수준](#)
- [한정자](#)
- [public](#)
- [protected](#)
- [internal](#)

# protected(C# 참조)

2021-02-18 • 3 minutes to read • [Edit Online](#)

`protected` 키워드는 멤버 액세스 한정자입니다.

## NOTE

이 페이지에서는 `protected` 액세스를 설명합니다. `protected` 키워드는 `protected internal` 및 `private protected` 액세스 한정자의 일부이기도 합니다.

`protected` 멤버는 해당 클래스 내에서 파생 클래스 인스턴스가 액세스할 수 있습니다.

`protected` 및 다른 액세스 한정자와 비교는 [액세스 가능성 수준](#)을 참조하세요.

## 예제

기본 클래스의 `protected` 멤버는 파생 클래스 형식을 통해 액세스가 발생하는 경우에만 파생 클래스에서 액세스 할 수 있습니다. 예를 들어 다음 코드 세그먼트를 고려하세요.

```
class A
{
    protected int x = 123;
}

class B : A
{
    static void Main()
    {
        var a = new A();
        var b = new B();

        // Error CS1540, because x can only be accessed by
        // classes derived from A.
        // a.x = 10;

        // OK, because this class derives from A.
        b.x = 10;
    }
}
```

`a.x = 10` 문은 정적 메서드 `Main` 내에서 작성되었으며 클래스 `B`의 인스턴스가 아니므로 오류를 생성합니다.

구조체를 상속할 수 없기 때문에 구조체 멤버는 보호할 수 없습니다.

## 예제

이 예제에서 `DerivedPoint` 클래스는 `Point`에서 파생됩니다. 따라서 파생 클래스에서 직접 기본 클래스의 `protected` 멤버를 액세스할 수 있습니다.

```
class Point
{
    protected int x;
    protected int y;
}

class DerivedPoint: Point
{
    static void Main()
    {
        var dpoint = new DerivedPoint();

        // Direct access to protected members.
        dpoint.x = 10;
        dpoint.y = 15;
        Console.WriteLine($"x = {dpoint.x}, y = {dpoint.y}");
    }
}
// Output: x = 10, y = 15
```

`x` 및 `y`의 액세스 수준을 [private](#)로 변경하는 경우 컴파일러가 오류 메시지를 실행합니다.

```
'Point.y' is inaccessible due to its protection level.
```

```
'Point.x' is inaccessible due to its protection level.
```

## C# 언어 사양

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

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [액세스 한정자](#)
- [액세스 수준](#)
- [한정자](#)
- [public](#)
- [private](#)
- [internal](#)
- [internal virtual 키워드에 대한 보안 문제](#)

# public(C# 참조)

2021-02-18 • 2 minutes to read • [Edit Online](#)

`public` 키워드는 형식 및 형식 멤버에 대한 액세스 한정자입니다. 공용 액세스는 허용 범위가 가장 큰 액세스 수준입니다. 다음 예제와 같이, 공용 멤버 액세스에 대한 제한은 없습니다.

```
class SampleClass
{
    public int x; // No access restrictions.
}
```

자세한 내용은 [액세스 한정자](#) 및 [액세스 가능성 수준](#)을 참조하세요.

## 예제

다음 예제에서는 두 개의 클래스, `PointTest` 및 `Program`를 선언합니다. `PointTest`의 공용 멤버 `x` 및 `y`는 `Program`에서 직접 액세스합니다.

```
class PointTest
{
    public int x;
    public int y;
}

class Program
{
    static void Main()
    {
        var p = new PointTest();
        // Direct access to public members.
        p.x = 10;
        p.y = 15;
        Console.WriteLine($"x = {p.x}, y = {p.y}");
    }
}
// Output: x = 10, y = 15
```

`public` 액세스 수준을 `private` 또는 `protected`로 변경하면 오류 메시지가 표시됩니다.

보호 수준 때문에 'PointTest.y'에 액세스할 수 없습니다.

## C# 언어 사양

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

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [액세스 한정자](#)
- [C# 키워드](#)
- [액세스 가능성 수준](#)

- 액세스 수준
- 한정자
- private
- protected
- internal

# protected internal(C# 참조)

2020-11-02 • 3 minutes to read • [Edit Online](#)

`protected internal` 키워드 조합은 멤버 액세스 한정자입니다. `protected internal` 멤버는 포함하는 클래스에서 파생된 형식이나 현재 어셈블리에서 액세스할 수 있습니다. `protected internal` 및 다른 액세스 한정자와 비교는 [액세스 가능성 수준](#)을 참조하세요.

## 예제

기본 클래스의 `protected internal` 멤버는 포함하는 어셈블리 내의 모든 형식에서 액세스할 수 있습니다. 또한 액세스가 파생된 클래스 형식의 변수를 통해 발생하는 경우에만 다른 어셈블리에 있는 파생 클래스에서 액세스할 수 있습니다. 예를 들어 다음 코드 세그먼트를 고려하세요.

```
// Assembly1.cs
// Compile with: /target:library
public class BaseClass
{
    protected internal int myValue = 0;
}

class TestAccess
{
    void Access()
    {
        var baseObject = new BaseClass();
        baseObject.myValue = 5;
    }
}
```

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass : BaseClass
{
    static void Main()
    {
        var baseObject = new BaseClass();
        var derivedObject = new DerivedClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 10;

        // OK, because this class derives from BaseClass.
        derivedObject.myValue = 10;
    }
}
```

이 예제에는 `Assembly1.cs` 및 `Assembly2.cs`의 두 파일이 포함되어 있습니다. 첫 번째 파일은 공용 기본 클래스인 `BaseClass` 와 다른 클래스인 `TestAccess`를 포함합니다. `BaseClass` 는 `TestAccess` 형식으로 액세스되는 `protected internal` 멤버인 `myValue` 를 소유합니다. 두 번째 파일에서 `BaseClass` 의 인스턴스를 통한 `myValue` 액세스의 시도는 오류를 생성하지만 파생된 클래스 `DerivedClass` 의 인스턴스를 통한 이 멤버로의 액세스는 성공합니다.

구조체를 상속할 수 없기 때문에 구조체 멤버는 `protected internal` 일 수 없습니다.

# C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [액세스 한정자](#)
- [액세스 수준](#)
- [한정자](#)
- [public](#)
- [private](#)
- [internal](#)
- [internal virtual 키워드에 대한 보안 문제](#)

# private protected (C# Reference)

2020-11-02 • 4 minutes to read • [Edit Online](#)

`private protected` 키워드 조합은 멤버 액세스 한정자입니다. `private protected` 멤버는 포함하는 클래스에서 파생된 형식으로 액세스할 수 있지만 포함하는 어셈블리 내에서만 가능합니다. `private protected` 및 다른 액세스 한정자와 비교는 [액세스 가능성 수준](#)을 참조하세요.

## NOTE

`private protected` 액세스 한정자는 C# 7.2 버전에서 유효합니다.

## 예제

기본 클래스의 `private protected` 멤버는 변수의 정적 형식이 파생 클래스 형식인 경우에만 포함 어셈블리의 파생된 형식에서 액세스할 수 있습니다. 예를 들어 다음 코드 세그먼트를 고려하세요.

```
public class BaseClass
{
    private protected int myValue = 0;
}

public class DerivedClass1 : BaseClass
{
    void Access()
    {
        var baseObject = new BaseClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 5;

        // OK, accessed through the current derived class instance
        myValue = 5;
    }
}
```

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass2 : BaseClass
{
    void Access()
    {
        // Error CS0122, because myValue can only be
        // accessed by types in Assembly1
        // myValue = 10;
    }
}
```

이 예제에는 `Assembly1.cs` 및 `Assembly2.cs`의 두 파일이 포함되어 있습니다. 첫 번째 파일은 공용 기본 클래스인 `BaseClass`를 포함하고 여기에서 파생된 형식인 `DerivedClass1`을 포함합니다. `BaseClass`는 `DerivedClass1`이 두 가지 방법으로 액세스를 시도하는 `private protected` 멤버인 `myValue`를 소유합니다. `BaseClass`의 인스턴스를 통한 `myValue` 액세스의 첫 번째 시도는 오류를 생성합니다. 그러나 `DerivedClass1`에서 상속된 멤버로 사용하려는 시도는 성공합니다.

두 번째 파일에서 `DerivedClass2`의 상속된 멤버로 `myValue`에 액세스하는 시도는 Assembly1에서 파생된 형식으로만 액세스할 수 있으므로 오류를 생성합니다.

`Assembly2` 이름을 지정하는 `InternalsVisibleToAttribute`가 `Assembly1.cs`에 포함된 경우 파생 클래스 `DerivedClass1`는 `private protected`에 선언된 `BaseClass` 멤버에 액세스할 수 있습니다. `InternalsVisibleTo`는 `private protected` 멤버가 다른 어셈블리의 파생 클래스에 표시되도록 합니다.

구조체를 상속할 수 없기 때문에 구조체 멤버는 `private protected`일 수 없습니다.

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [액세스 한정자](#)
- [액세스 수준](#)
- [한정자](#)
- [public](#)
- [private](#)
- [internal](#)
- [internal virtual 키워드에 대한 보안 문제](#)

# abstract(C# 참조)

2020-11-02 • 8 minutes to read • [Edit Online](#)

`abstract` 한정자는 수정되는 항목에 누락되거나 불완전한 구현이 있음을 나타냅니다. `abstract` 한정자는 클래스, 메서드, 속성, 인덱서 및 이벤트와 함께 사용될 수 있습니다. 클래스 선언에서 `abstract` 한정자를 사용하여 클래스가 자체에서 인스턴스화되지 않고 다른 클래스의 기본 클래스로만 사용됨을 나타냅니다. 추상으로 표시된 멤버는 추상 클래스에서 파생된 비 추상 클래스에 의해 구현되어야 합니다.

## 예제

이 예제에서 `Square` 클래스는 `Shape`에서 파생되므로 `GetArea` 구현을 제공해야 합니다.

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    int side;

    public Square(int n) => side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => side * side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

다음 기능이 있는 추상 클래스:

- 추상 클래스는 인스턴스화할 수 없습니다.
- 추상 클래스에 추상 메서드 및 접근자가 포함될 수 있습니다.
- `sealed` 한정자를 사용하여 추상 클래스를 수정할 수는 없습니다. 두 한정자가 상반된 의미가 있기 때문입니다. `sealed` 한정자를 사용하면 클래스가 상속되지 않고 `abstract` 한정자를 사용하려면 클래스가 상속되어야 합니다.
- 추상 클래스에서 추상이 아닌 파생 클래스에는 모든 상속된 메서드 및 접근자의 실제 구현이 포함되어야 합니다.

메서드 또는 속성 선언에서 `abstract` 한정자를 사용하여 메서드 또는 속성에 구현이 포함되지 않음을 나타냅니다.

추상 메서드에는 다음과 같은 기능이 있습니다.

- 추상 메서드는 암시적으로 가상 메서드입니다.
- 추상 메서드 선언은 추상 클래스에서만 허용됩니다.
- 추상 메서드 선언은 실제 구현을 제공하지 않으므로 메서드 본문이 없습니다. 메서드 선언은 세미콜론으로 끝나며, 구현은 다른 파일이나 같은 파일의 다른 위치에서 이루어집니다.

로 끝나고 시그니처 뒤에 중괄호( { })가 없습니다. 예를 들면 다음과 같습니다.

```
public abstract void MyMethod();
```

구현은 비추상 클래스의 멤버인 메서드 `override`에 의해 제공됩니다.

- 추상 메서드 선언에서 `static` 또는 `virtual` 한정자를 사용하는 것은 오류입니다.

선언 및 호출 구문의 차이점을 제외하고 추상 속성은 추상 메서드처럼 동작합니다.

- 정적 속성에서 `abstract` 한정자를 사용하는 것은 오류입니다.
- `override` 한정자를 사용하는 속성 선언을 포함하여 상속된 추상 속성을 파생 클래스에서 재정의할 수 있습니다.

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

추상 클래스는 모든 인터페이스 멤버에 대한 구현을 제공해야 합니다.

인터페이스를 구현하는 추상 클래스는 인터페이스 메서드를 추상 메서드에 맵핑 할 수 있습니다. 예를 들어:

```
interface I
{
    void M();
}

abstract class C : I
{
    public abstract void M();
}
```

## 예제

이 예제에서 `DerivedClass` 클래스는 추상 클래스 `BaseClass`에서 파생됩니다. 추상 클래스에는 추상 메서드 `AbstractMethod` 및 두 개의 추상 속성 `x` 및 `y`가 포함됩니다.

```

abstract class BaseClass // Abstract class
{
    protected int _x = 100;
    protected int _y = 150;
    public abstract void AbstractMethod(); // Abstract method
    public abstract int X { get; }
    public abstract int Y { get; }
}

class DerivedClass : BaseClass
{
    public override void AbstractMethod()
    {
        _x++;
        _y++;
    }

    public override int X // overriding property
    {
        get
        {
            return _x + 10;
        }
    }

    public override int Y // overriding property
    {
        get
        {
            return _y + 10;
        }
    }
}

static void Main()
{
    var o = new DerivedClass();
    o.AbstractMethod();
    Console.WriteLine($"x = {o.X}, y = {o.Y}");
}
}

// Output: x = 111, y = 161

```

앞의 예제에서 다음과 같이 문을 사용하여 추상 클래스를 인스턴스화하려고 하면,

```
BaseClass bc = new BaseClass(); // Error
```

컴파일러가 추상 클래스 'BaseClass'의 인스턴스를 만들 수 없다는 오류가 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# 참조](#)
- [C# 프로그래밍 가이드](#)
- [한정자](#)
- [virtual](#)
- [override](#)

- C# 키워드

# async(C# 참조)

2020-11-02 • 11 minutes to read • [Edit Online](#)

`async` 한정자를 사용하여 메서드, [람다 식](#) 또는 [무명 메서드](#)를 비동기로 지정합니다. 메서드 또는 식에 이 한정자를 사용하면 [비동기 메서드](#)라고 합니다. 다음 예제에서는 `ExampleMethodAsync`라는 비동기 메서드를 정의합니다.

```
public async Task<int> ExampleMethodAsync()
{
    //...
}
```

비동기 프로그래밍이 처음이거나 비동기 메서드가 `await` 연산자를 사용하여 호출자의 스레드를 차단하지 않고 장기 실행 작업을 수행할 수 있는 방법을 잘 모르겠으면 [async 및 await를 사용한 비동기 프로그래밍의 소개](#) 내용을 참조하세요. 다음 코드는 비동기 메서드 안에 있으며 [HttpClient.GetStringAsync](#) 메서드를 호출합니다.

```
string contents = await httpClient.GetStringAsync(requestUrl);
```

비동기 메서드는 대기 중인 작업이 완료될 때까지 메서드가 일시 중단되는 지점인 첫 번째 `await` 식에 도달하기 전에는 동기적으로 실행됩니다. 다음 단원의 예제에서처럼 그 동안에는 제어가 메서드 호출자에게 반환됩니다.

`async` 키워드에서 수정하는 메서드에 `await` 식 또는 문이 없는 경우 해당 메서드가 동기적으로 실행됩니다. `await` 문이 포함되지 않은 모든 비동기 메서드에서는 오류가 발생할 수 있으므로 컴파일러 경고가 나타납니다. [컴파일러 경고\(수준 1\) CS4014](#)를 참조하세요.

`async` 키워드는 메서드, 람다 식 또는 무명 메서드를 수정할 때만 키워드로 사용됩니다. 다른 모든 컨텍스트에서는 식별자로 해석됩니다.

## 예제

다음 예제에서는 비동기 이벤트 처리기, `startButton_Click`, 비동기 메서드 및 `ExampleMethodAsync` 간의 제어 흐름과 구조를 보여 줍니다. 비동기 메서드의 결과는 웹 페이지의 문자 수입니다. 이 코드는 Visual Studio에서 만든 WPF(Windows Presentation Foundation) 앱 또는 Windows 스토어 앱에 적합합니다. 앱을 설정하는 방법은 코드 주석을 참조하세요.

Visual Studio에서 이 코드를 WPF(Windows Presentation Foundation) 앱 또는 Windows 스토어 앱으로 실행할 수 있습니다. `StartButton`이라는 Button 컨트롤과 `ResultsTextBox`라는 TextBox 컨트롤이 필요합니다. 다음과 같이 작성되도록 이름과 처리기를 설정해야 합니다.

```
<Button Content="Button" HorizontalAlignment="Left" Margin="88,77,0,0" VerticalAlignment="Top" Width="75"
       Click="StartButton_Click" Name="StartButton"/>
<TextBox HorizontalAlignment="Left" Height="137" Margin="88,140,0,0" TextWrapping="Wrap"
         Text="&lt;Enter a URL&gt;" VerticalAlignment="Top" Width="310" Name="ResultsTextBox"/>
```

코드를 WPF 앱으로 실행하려면

- 이 코드를 `MainWindow.xaml.cs`의 `MainWindow` 클래스에 붙여 넣습니다.
- `System.Net.Http`에 대한 참조를 추가합니다.
- `System.Net.Http`에 대한 `using` 지시문을 추가합니다.

코드를 Windows 스토어 앱으로 실행하려면

- 이 코드를 MainPage.xaml.cs의 `MainPage` 클래스에 붙여넣습니다.
- System.Net.Http 및 System.Threading.Tasks에 대한 using 지시문을 추가합니다.

```
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    // ExampleMethodAsync returns a Task<int>, which means that the method
    // eventually produces an int result. However, ExampleMethodAsync returns
    // the Task<int> value as soon as it reaches an await.
    ResultsTextBox.Text += "\n";

    try
    {
        int length = await ExampleMethodAsync();
        // Note that you could put "await ExampleMethodAsync()" in the next line where
        // "length" is, but due to when '+=' fetches the value of ResultsTextBox, you
        // would not see the global side effect of ExampleMethodAsync setting the text.
        ResultsTextBox.Text += String.Format("Length: {0:N0}\n", length);
    }
    catch (Exception)
    {
        // Process the exception if one occurs.
    }
}

public async Task<int> ExampleMethodAsync()
{
    var httpClient = new HttpClient();
    int exampleInt = (await httpClient.GetStringAsync("http://msdn.microsoft.com")).Length;
    ResultsTextBox.Text += "Preparing to finish ExampleMethodAsync.\n";
    // After the following return statement, any method that's awaiting
    // ExampleMethodAsync (in this case, StartButton_Click) can get the
    // integer result.
    return exampleInt;
}
// The example displays the following output:
// Preparing to finish ExampleMethodAsync.
// Length: 53292
```

#### IMPORTANT

작업 및 작업 완료를 기다리는 동안 실행되는 코드에 관한 자세한 내용은 [async 및 await를 사용한 비동기 프로그래밍](#)을 참조하세요. 비슷한 요소를 사용하는 전체 콘솔 예제는 [완료되면 비동기 작업 처리\(C#\)](#)를 참조하세요.

## 반환 형식

비동기 메서드의 반환 형식은 다음과 같을 수 있습니다.

- `Task`
- `Task<TResult>`
- `void`. `async void` 메서드는 호출자가 `await` 해당 메서드를 사용할 수 없으며 성공적으로 완료 또는 오류 조건을 보고하는 다른 메커니즘을 구현해야 하기 때문에 이벤트 처리기 이외의 코드에 대해 일반적으로 사용되지 않습니다.
- C# 7.0부터 액세스 가능한 `GetAwaiter` 메서드가 있는 모든 형식. `System.Threading.Tasks.ValueTask<TResult>` 형식은 이러한 구현 중 하나입니다. NuGet 패키지 `System.Threading.Tasks.Extensions`를 추가하면 사용할 수 있습니다.

비동기 메서드는 모든 `in`, `ref` 또는 `out` 매개 변수를 선언할 수 없고 [참조 반환 값](#)을 가질 수도 없지만, 이러한 매

개 변수가 있는 메서드를 호출할 수는 있습니다.

메서드의 `return` 문에서 `TResult` 형식의 피연산자를 지정할 경우 비동기 메서드의 반환 형식으로 `Task<TResult>`를 지정합니다. 메서드가 완료되었을 때 의미 있는 값이 반환되지 않을 경우 `Task`를 사용합니다. 즉, 이 메서드를 호출하면 `Task`가 반환되지만 `Task`가 완료되면 `await`를 기다리는 모든 `Task` 식이 `void` 됩니다.

`void` 반환 형식은 주로 해당 반환 형식이 필요한 이벤트 처리기를 정의할 때 사용합니다. `void` 반환 비동기 메서드의 호출자는 기다릴 수 없으므로 메서드가 `throw`하는 예외를 `catch`할 수 없습니다.

C# 7.0부터 `GetAwaiter` 메서드가 있는 다른 형식(일반적으로 값 형식)을 반환하여 성능이 중요한 코드 셙션에서 메모리 할당을 최소화합니다.

자세한 내용과 예제는 [비동기 반환 형식](#)을 참조하세요.

## 참고 항목

- [AsyncStateMachineAttribute](#)
- [await](#)
- [async 및 await를 사용한 비동기 프로그래밍](#)
- [완료되면 비동기 작업 처리](#)

# const(C# 참조)

2020-11-02 • 5 minutes to read • [Edit Online](#)

상수 필드 또는 지역 상수를 선언할 때는 `const` 키워드를 사용합니다. 상수 필드 및 지역 상수는 변수가 아니며 수정할 수 없습니다. 상수는 숫자, 부울 값, 문자열 또는 `null` 참조일 수 있습니다. 언제든지 변경될 수 있는 정보를 나타낼 때는 상수를 만들지 마세요. 예를 들어, 상수 필드를 사용하여 서비스의 가격, 제품 버전 번호 또는 회사의 브랜드 이름을 저장하지 마세요. 이러한 값은 시간이 지남에 따라 변경될 수 있으며, 컴파일러는 상수를 전파하므로 변경 내용을 보기 위해서는 라이브러리를 사용하여 컴파일된 다른 코드를 다시 컴파일해야 합니다.

`readonly` 키워드를 참조하세요. 다음은 그 예입니다.

```
const int X = 0;
public const double GravitationalConstant = 6.673e-11;
private const string ProductName = "Visual C#";
```

## 설명

상수 선언 형식은 선언에서 제공하는 멤버의 형식을 지정합니다. 지역 상수 또는 상수 필드의 이니셜라이저는 대상 형식으로 암시적으로 변환될 수 있는 상수 식이어야 합니다.

상수 식은 컴파일 시간에 완전히 계산될 수 있는 식입니다. 따라서 참조 형식의 상수에 대해 가능한 값은 `string` 및 `null` 참조뿐입니다.

상수 선언에서는 다음과 같이 여러 상수를 선언할 수 있습니다.

```
public const double X = 1.0, Y = 2.0, Z = 3.0;
```

`static` 한정자는 상수 선언에는 허용되지 않습니다.

상수는 다음과 같이 상수 식에 참여할 수 있습니다.

```
public const int C1 = 5;
public const int C2 = C1 + 100;
```

### NOTE

`readonly` 키워드는 `const` 키워드와 다릅니다. `const` 필드는 필드 선언에서만 초기화될 수 있습니다. `readonly` 필드는 선언이나 생성자에서 초기화될 수 있습니다. 따라서 `readonly` 필드는 사용된 생성자에 따라 다른 값을 가질 수 있습니다. 또한 `const` 필드가 컴파일 시간 상수라고 하더라도 `readonly` 필드는 다음 줄에서와 같이 런타임 상수에 사용될 수 있습니다. `public static readonly uint li = (uint)DateTime.Now.Ticks;`

## 예제

```

public class ConstTest
{
    class SampleClass
    {
        public int x;
        public int y;
        public const int C1 = 5;
        public const int C2 = C1 + 5;

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

    static void Main()
    {
        var mC = new SampleClass(11, 22);
        Console.WriteLine($"x = {mC.x}, y = {mC.y}");
        Console.WriteLine($"C1 = {SampleClass.C1}, C2 = {SampleClass.C2}");
    }
}

/* Output
   x = 11, y = 22
   C1 = 5, C2 = 10
*/

```

## 예제

이 예제에서는 상수를 지역 변수로 사용하는 방법을 보여 줍니다.

```

public class SealedTest
{
    static void Main()
    {
        const int C = 707;
        Console.WriteLine($"My local constant = {C}");
    }
}

// Output: My local constant = 707

```

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [한정자](#)
- [readonly](#)

# event(C# 참조)

2020-11-02 • 7 minutes to read • [Edit Online](#)

`event` 키워드는 게시자 클래스에서 이벤트를 선언하는데 사용됩니다.

## 예제

다음 예제에서는 [EventHandler](#)를 기본 대리자 형식으로 사용하는 이벤트를 선언하고 발생시키는 방법을 보여 줍니다. 제네릭 [EventHandler<TEventArgs>](#) 대리자 형식을 사용하는 방법 및 이벤트를 구독하고 이벤트 처리기 메서드를 만드는 방법을 보여 주는 전체 코드 예제는 [.NET 지침을 따르는 이벤트를 게시하는 방법](#)을 참조하세요.

```
public class SampleEventArgs
{
    public SampleEventArgs(string text) { Text = text; }
    public string Text { get; } // readonly
}

public class Publisher
{
    // Declare the delegate (if using non-generic pattern).
    public delegate void SampleEventHandler(object sender, SampleEventArgs e);

    // Declare the event.
    public event SampleEventHandler SampleEvent;

    // Wrap the event in a protected virtual method
    // to enable derived classes to raise the event.
    protected virtual void RaiseSampleEvent()
    {
        // Raise the event in a thread-safe manner using the ?. operator.
        SampleEvent?.Invoke(this, new SampleEventArgs("Hello"));
    }
}
```

이벤트는 해당 이벤트가 선언되는 클래스(게시자 클래스) 또는 구조체 내에서만 호출할 수 있는 특수한 멀티캐스트 대리자입니다. 다른 클래스 또는 구조체에서 이벤트를 구독하는 경우 해당 이벤트 처리기 메서드는 게시자 클래스에서 이벤트를 발생시킬 때 호출됩니다. 자세한 내용 및 코드 예제는 [이벤트 및 대리자](#)를 참조하세요.

이벤트는 [public](#), [private](#), [protected](#), [internal](#), [protected internal](#) 또는 [private protected](#)로 표시될 수 있습니다. 이러한 액세스 한정자는 클래스 사용자가 이벤트에 액세스하는 방법을 정의합니다. 자세한 내용은 [액세스 한정자](#)를 참조하세요.

## 키워드 및 이벤트

이벤트에 적용되는 키워드는 다음과 같습니다.

키워드	DESCRIPTION	참조 항목
<a href="#">static</a>	클래스의 인스턴스가 없는 경우에도 언제든지 호출자가 이벤트를 사용할 수 있도록 설정합니다.	<a href="#">정적 클래스 및 정적 클래스 멤버</a>

키워드	DESCRIPTION	참조 항목
<a href="#">virtual</a>	파생 클래스에서 <a href="#">override</a> 키워드를 사용하여 이벤트 동작을 재정의할 수 있도록 합니다.	<a href="#">상속</a>
<a href="#">sealed</a>	파생 클래스에 대해 더 이상 가상이 아니도록 지정합니다.	
<a href="#">abstract</a>	컴파일러에서 <code>add</code> 및 <code>remove</code> 이벤트 접근자 블록을 생성하지 않으므로 파생 클래스는 자체 구현을 제공해야 합니다.	

이벤트는 [static](#) 키워드를 사용하여 정적 이벤트로 선언할 수 있습니다. 그러면 클래스의 인스턴스가 없는 경우에도 언제든지 호출자가 이벤트를 사용할 수 있습니다. 자세한 내용은 [static 클래스 및 static 클래스 멤버](#)를 참조하세요.

이벤트는 [virtual](#) 키워드를 사용하여 가상 이벤트로 표시할 수 있습니다. 그러면 파생 클래스에서 [override](#) 키워드를 사용하여 이벤트 동작을 재정의할 수 있습니다. 자세한 내용은 [상속](#)을 참조하세요. 또한 가상 이벤트를 재정의하는 이벤트는 [sealed](#)일 수 있으며, 파생 클래스에 대해 더 이상 가상이 아니도록 지정합니다. 마지막으로 이벤트를 [abstract](#)로 선언할 수 있으며, 컴파일러에서 `add` 및 `remove` 이벤트 접근자 블록을 생성하지 않는다는 의미입니다. 따라서 파생 클래스는 자체 구현을 제공해야 합니다.

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [add](#)
- [remove](#)
- [한정자](#)
- [대리자를 결합하는 방법\(멀티캐스트 대리자\)](#)

# extern(C# 참조)

2021-02-18 • 5 minutes to read • [Edit Online](#)

`extern` 한정자는 외부에서 구현되는 메서드를 선언하는 데 사용됩니다. `extern` 한정자는 일반적으로 Interop 서비스를 사용하여 비관리 코드를 호출할 때 `[DllImport]` 특성과 함께 사용됩니다. 이 경우 다음 예제에서와 같이 메서드를 `static`으로 선언해야 합니다.

```
[DllImport("avifil32.dll")]
private static extern void AVIFileInit();
```

`extern` 키워드는 외부 어셈블리 별칭도 정의하여 단일 어셈블리 내에서 동일한 구성 요소의 다른 버전을 참조 할 수 있도록 합니다. 자세한 내용은 [extern alias](#)를 참조하세요.

`abstract` 및 `extern` 한정자를 함께 사용하여 같은 멤버를 수정할 수는 없습니다. `extern` 한정자는 메서드가 C# 코드 외부에서 구현됨을 나타내고 `abstract` 한정자는 해당 클래스에서 메서드가 구현되지 않음을 나타냅니다.

`extern` 키워드는 C++보다 C#에서 사용이 제한적입니다. C# 키워드를 C++ 키워드와 비교하려면 `extern`을 사용 하여 C++ 언어 참조에 링크 지정을 참조하십시오.

## 예 1

이 예제에서는 프로그램이 사용자로부터 문자열을 수신하여 메시지 상자에 표시합니다. 이 프로그램은 User32.dll 라이브러리에서 가져온 `MessageBox` 메서드를 사용합니다.

```
//using System.Runtime.InteropServices;
class ExternTest
{
    [DllImport("User32.dll", CharSet=CharSet.Unicode)]
    public static extern int MessageBox(IntPtr h, string m, string c, int type);

    static int Main()
    {
        string myString;
        Console.Write("Enter your message: ");
        myString = Console.ReadLine();
        return MessageBox((IntPtr)0, myString, "My Message Box", 0);
    }
}
```

## 예제 2

이 예제는 C 라이브러리(네이티브 DLL)로 호출되는 C# 프로그램을 보여 줍니다.

1. 다음 C 파일을 만들고 이름을 `cmdll.c`로 지정합니다.

```
// cmdll.c
// Compile with: -LD
int __declspec(dllexport) SampleMethod(int i)
{
    return i*10;
}
```

2. Visual Studio 설치 디렉터리에서 Visual Studio x64(또는 x32) 네이티브 도구 명령 프롬프트 창을 열고 명

령 프롬프트에서 `cl -LD cmdll.c` 를 입력하여 `cmdll.c` 파일을 컴파일합니다.

3. 같은 디렉터리에서 다음 C# 파일을 만들고 이름을 `cm.cs`로 지정합니다.

```
// cm.cs
using System;
using System.Runtime.InteropServices;
public class MainClass
{
    [DllImport("Cmdll.dll")]
    public static extern int SampleMethod(int x);

    static void Main()
    {
        Console.WriteLine("SampleMethod() returns {0}.", SampleMethod(5));
    }
}
```

4. Visual Studio 설치 디렉터리에서 Visual Studio x64(또는 x32) 네이티브 도구 명령 프롬프트 창을 열고 명령 프롬프트에서 다음을 입력하여 `cm.cs` 파일을 컴파일합니다.

```
csc cm.cs(x64 명령 프롬프트의 경우) —또는— csc -platform:x86 cm.cs(x32 명령 프롬프트의 경우)
```

이렇게 하면 실행 파일 `cm.exe` 가 만들어집니다.

5. `cm.exe` 을 실행합니다. `SampleMethod` 메서드가 값 5를 DLL 파일에 전달하면 10을 곱한 값이 반환됩니다. 프로그램에서는 다음이 출력됩니다.

```
SampleMethod() returns 50.
```

## C# 언어 사양

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

## 참조

- [System.Runtime.InteropServices.DllImportAttribute](#)
- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [한정자](#)

# in(제네릭 한정자)(C# 참조)

2020-11-02 • 5 minutes to read • [Edit Online](#)

제네릭 형식 매개 변수에서 `in` 키워드는 형식 매개 변수를 반공변(contravariant)으로 지정합니다. 제네릭 인터페이스 및 대리자에서 `in` 키워드를 사용할 수 있습니다.

반공변성(Contravariance)을 통해 제네릭 매개 변수에 지정된 것보다 적은 파생 형식을 사용할 수 있습니다. 따라서 공변(contravariant) 인터페이스를 구현하는 클래스의 암시적 변환과 대리자 형식의 암시적 변환이 허용됩니다. 제네릭 형식 매개 변수의 공변성(Covariance) 및 반공변성(Contravariance)은 참조 형식에 대해 지원되고 값 형식에 대해서는 지원되지 않습니다.

메서드 매개 변수의 형식을 정의하고 메서드의 반환 형식을 정의하지 않는 경우에만 형식을 제네릭 인터페이스 또는 대리자에서 반공변(contravariant)으로 선언할 수 있습니다. `In`, `ref` 및 `out` 매개 변수는 고정이어야 합니다. 즉 공변(covariant) 또는 반공변(contravariant)입니다.

반공변(contravariant) 형식 매개 변수가 있는 인터페이스는 해당 메서드가 인터페이스 형식 매개 변수에 지정된 형식보다 덜 파생된 형식의 인수를 사용할 수 있도록 합니다. 예를 들어 `IComparer<T>` 인터페이스에서 `T` 형식은 반공변(contravariant)이므로 `Employee` 가 `Person` 을 상속하는 경우 특수 변환 메서드를 사용하지 않고 `IComparer<Person>` 형식의 개체를 `IComparer<Employee>` 형식의 개체에 할당할 수 있습니다.

반공변(contravariant) 대리자에 동일한 형식의 다른 대리자를 할당할 수 있지만 덜 파생된 제네릭 형식 매개 변수가 필요합니다.

자세한 내용은 [공변성\(Covariance\)](#) 및 [반공변성\(Contravariance\)](#)을 참조하세요.

## 반공변(contravariant) 제네릭 인터페이스

다음 예제에서는 반공변(contravariant) 제네릭 인터페이스를 선언, 확장 및 구현하는 방법을 보여 줍니다. 또한 이 인터페이스를 구현하는 클래스에 대해 암시적 변환을 사용하는 방법을 보여 줍니다.

```
// Contravariant interface.  
interface IContravariant<in A> { }  
  
// Extending contravariant interface.  
interface IExtContravariant<in A> : IContravariant<A> { }  
  
// Implementing contravariant interface.  
class Sample<A> : IContravariant<A> { }  
  
class Program  
{  
    static void Test()  
    {  
        IContravariant<Object> iobj = new Sample<Object>();  
        IContravariant<String> istr = new Sample<String>();  
  
        // You can assign iobj to istr because  
        // the IContravariant interface is contravariant.  
        istr = iobj;  
    }  
}
```

## 반공변(contravariant) 제네릭 대리자

다음 예제에서는 반공변(contravariant) 제네릭 대리자를 선언, 인스턴스화 및 호출하는 방법을 보여 줍니다. 또

한 대리자 형식을 암시적으로 변환하는 방법을 보여 줍니다.

```
// Contravariant delegate.  
public delegate void DContravariant<in A>(A argument);  
  
// Methods that match the delegate signature.  
public static void SampleControl(Control control)  
{ }  
public static void SampleButton(Button button)  
{ }  
  
public void Test()  
{  
  
    // Instantiating the delegates with the methods.  
    DContravariant<Control> dControl = SampleControl;  
    DContravariant<Button> dButton = SampleButton;  
  
    // You can assign dControl to dButton  
    // because the DContravariant delegate is contravariant.  
    dButton = dControl;  
  
    // Invoke the delegate.  
    dButton(new Button());  
}
```

## C# 언어 사양

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

## 참조

- [out](#)
- [공 분산 및 반공 분산](#)
- [한정자](#)

# new 한정자(C# 참조)

2020-11-02 • 8 minutes to read • [Edit Online](#)

선언 한정자로 사용되는 `new` 키워드는 기본 클래스에서 상속된 멤버를 명시적으로 숨깁니다. 상속된 멤버를 숨기면 파생 버전의 멤버로 기본 클래스 버전의 멤버를 대신하게 됩니다. `new` 한정자를 사용하지 않고 멤버를 숨길 수도 있지만 컴파일러 경고가 발생합니다. `new`를 사용하여 멤버를 명시적으로 숨기면 이 경고가 발생하지 않습니다.

`new` 키워드를 사용하여 형식의 인스턴스를 만들거나 제네릭 형식 제약 조건으로 만들 수도 있습니다.

상속된 멤버를 숨기려면 동일한 멤버 이름을 사용하여 파생된 클래스에 해당 멤버를 선언한 다음 `new` 키워드를 사용하여 이를 한정합니다. 예를 들어:

```
public class BaseC
{
    public int x;
    public void Invoke() { }
}
public class DerivedC : BaseC
{
    new public void Invoke() { }
}
```

이 예제에서 `BaseC.Invoke` 는 `DerivedC.Invoke`에 의해 숨겨집니다. `x` 필드는 비슷한 이름으로 숨겨져 있지 않기 때문에 영향을 받지 않습니다.

상속을 사용한 이름 숨기기는 다음 중 한 가지 형식을 취합니다.

- 일반적으로 클래스 또는 구조체에 파생된 상수, 필드, 속성 또는 형식은 동일한 이름의 모든 기본 클래스 멤버를 숨깁니다. 이 사항이 적용되지 않는 경우가 있습니다. 예를 들어 호출할 수 없는 형식이 포함된 `N`이라는 이름의 새 필드를 선언하고 기본 형식에서 `N`을 메서드로 선언하면 새 필드가 호출 구문에서 기본 선언을 숨기지 않습니다. 자세한 내용은 [C# 언어 사양의 멤버 조회](#) 섹션을 참조하세요.
- 클래스 또는 구조체에 파생된 메서드는 기본 클래스에서 동일한 이름의 속성, 필드 및 형식을 숨깁니다. 또한 시그니처가 동일한 기본 클래스 메서드도 모두 숨깁니다.
- 클래스 또는 구조체에 파생된 인덱서는 시그니처가 동일한 기본 클래스 인덱서를 모두 숨깁니다.

동일한 멤버에 대해 `new` 와 `override`를 모두 사용하면 오류가 발생합니다. 두 한정자는 함께 사용할 수 없는 의미를 지니고 있기 때문입니다. `new` 한정자는 동일한 이름의 새 멤버를 만들고 원래 멤버를 숨기도록 하는 반면, `override` 한정자는 상속된 멤버에 대한 구현을 확장합니다.

상속된 멤버를 숨기지 않는 선언에 `new` 한정자를 사용하면 경고가 발생합니다.

## 예제

이 예제에서 기본 클래스 `BaseC` 및 파생 클래스 `DerivedC`는 동일한 필드 이름 `x`를 사용하므로 상속된 필드의 값이 숨겨집니다. 이 예제에서는 `new` 한정자의 사용법을 보여 줍니다. 또한 정규화된 이름을 사용하여 기본 클래스의 숨겨진 멤버에 액세스하는 방법을 보여 줍니다.

```
public class BaseC
{
    public static int x = 55;
    public static int y = 22;
}

public class DerivedC : BaseC
{
    // Hide field 'x'.
    new public static int x = 100;

    static void Main()
    {
        // Display the new value of x:
        Console.WriteLine(x);

        // Display the hidden value of x:
        Console.WriteLine(BaseC.x);

        // Display the unhidden member y:
        Console.WriteLine(y);
    }
}
/*
Output:
100
55
22
*/
```

## 예제

이 예제에서 중첩 클래스는 기본 클래스에서 이름이 동일한 클래스를 숨깁니다. 이 예제에서는 `new` 키워드를 사용하여 경고 메시지를 제거하는 방법과 정규화된 이름을 사용하여 숨겨진 클래스 멤버에 액세스하는 방법을 보여 줍니다.

```

public class BaseC
{
    public class NestedC
    {
        public int x = 200;
        public int y;
    }
}

public class DerivedC : BaseC
{
    // Nested type hiding the base type members.
    new public class NestedC
    {
        public int x = 100;
        public int y;
        public int z;
    }

    static void Main()
    {
        // Creating an object from the overlapping class:
        NestedC c1 = new NestedC();

        // Creating an object from the hidden class:
        BaseC.NestedC c2 = new BaseC.NestedC();

        Console.WriteLine(c1.x);
        Console.WriteLine(c2.x);
    }
}
/*
Output:
100
200
*/

```

**new** 한정자를 제거해도 프로그램은 컴파일되고 실행되지만 다음과 같은 경고가 발생합니다.

The keyword new is required on 'MyDerivedC.x' because it hides inherited member ' MyBaseC.x'.

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [한정자](#)
- [Override 및 New 키워드를 사용하여 버전 관리](#)
- [Override 및 New 키워드를 사용해야 하는 경우](#)

# out(제네릭 한정자)(C# 참조)

2020-11-02 • 7 minutes to read • [Edit Online](#)

제네릭 형식 매개 변수에서 `out` 키워드는 형식 매개 변수를 공변(covariant)으로 지정합니다. 제네릭 인터페이스 및 대리자에서 `out` 키워드를 사용할 수 있습니다.

공변성(covariance)을 통해 제네릭 매개 변수에 지정된 것보다 많은 파생 형식을 사용할 수 있습니다. 따라서 공변(covariant) 인터페이스를 구현하는 클래스의 암시적 변환과 대리자 형식의 암시적 변환이 허용됩니다. 공변성(Covariance) 및 반공변성(Contravariance)은 참조 형식에 대해 지원되고 값 형식에 대해서는 지원되지 않습니다.

공변(covariant) 형식 매개 변수가 있는 인터페이스는 해당 메서드가 형식 인터페이스에 지정된 것보다 많은 파생 형식을 반환할 수 있도록 합니다. 예를 들어 .NET Framework 4의 `IEnumerable<T>`에서 `T` 형식은 공변(covariant)이므로 특수 변환 메서드를 사용하지 않고 `IEnumerable<Object>` 형식의 개체를 `IEnumerable<String>` 형식의 개체에 할당할 수 있습니다.

공변(covariant) 대리자에 동일한 형식의 다른 대리자를 할당할 수 있지만 더 파생된 제네릭 형식 매개 변수가 필요합니다.

자세한 내용은 [공변성\(Covariance\)](#) 및 [반공변성\(Contravariance\)](#)을 참조하세요.

## 예제 - 공변(covariant) 제네릭 인터페이스

다음 예제에서는 공변(covariant) 제네릭 인터페이스를 선언, 확장 및 구현하는 방법을 보여 줍니다. 또한 공변(covariant) 인터페이스를 구현하는 클래스에 대해 암시적 변환을 사용하는 방법을 보여 줍니다.

```
// Covariant interface.  
interface ICovariant<out R> { }  
  
// Extending covariant interface.  
interface IExtCovariant<out R> : ICovariant<R> { }  
  
// Implementing covariant interface.  
class Sample<R> : ICovariant<R> { }  
  
class Program  
{  
    static void Test()  
    {  
        ICovariant<Object> iobj = new Sample<Object>();  
        ICovariant<String> istr = new Sample<String>();  
  
        // You can assign istr to iobj because  
        // the ICovariant interface is covariant.  
        iobj = istr;  
    }  
}
```

제네릭 인터페이스에서 형식 매개 변수가 다음 조건을 충족하는 경우 공변(covariant)으로 선언할 수 있습니다.

- 형식 매개 변수는 인터페이스 메서드의 반환 형식으로만 사용되고 메서드 인수의 형식으로 사용되지 않습니다.

#### NOTE

그러나 이 규칙에는 한 가지 예외가 있습니다. 공변(covariant) 인터페이스에 메서드 매개 변수로 반공변(contravariant) 제네릭 대리자가 있는 경우 공변(covariant) 형식을 이 대리자에 대한 제네릭 형식 매개 변수로 사용할 수 있습니다. 공변(covariant) 및 반공변(contravariant) 제네릭 대리자에 대한 자세한 내용은 [대리자의 가변성](#) 및 [Func 및 Action 제네릭 대리자에 가변성 사용](#)을 참조하세요.

- 형식 매개 변수는 인터페이스 메서드에 대한 제네릭 제약 조건으로 사용되지 않습니다.

## 예제 - 공변(covariant) 제네릭 대리자

다음 예제에서는 공변(covariant) 제네릭 대리자를 선언, 인스턴스화 및 호출하는 방법을 보여 줍니다. 또한 대리자 형식을 암시적으로 변환하는 방법을 보여 줍니다.

```
// Covariant delegate.  
public delegate R DCovariant<out R>();  
  
// Methods that match the delegate signature.  
public static Control SampleControl()  
{ return new Control(); }  
  
public static Button SampleButton()  
{ return new Button(); }  
  
public void Test()  
{  
    // Instantiate the delegates with the methods.  
    DCovariant<Control> dControl = SampleControl;  
    DCovariant<Button> dButton = SampleButton;  
  
    // You can assign dButton to dControl  
    // because the DCovariant delegate is covariant.  
    dControl = dButton;  
  
    // Invoke the delegate.  
    dControl();  
}
```

제네릭 대리자에서 형식이 메서드 반환 형식으로만 사용되고 메서드 인수에 사용되지 않는 경우 공변(covariant)으로 선언할 수 있습니다.

## C# 언어 사양

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

## 참조

- [제네릭 인터페이스의 가변성](#)
- [in](#)
- [한정자](#)

# override(C# 참조)

2020-11-02 • 6 minutes to read • [Edit Online](#)

`override` 한정자는 상속된 메서드, 속성, 인덱서 또는 이벤트의 추상 또는 가상 구현을 확장하거나 수정하는 데 필요합니다.

다음 예제에서 `Square` 클래스는 `GetArea` 가 추상 `Shape` 클래스에서 상속되기 때문에 `GetArea` 의 재정의된 구현을 제공해야 합니다.

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    int side;

    public Square(int n) => side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => side * side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

`override` 메서드는 기본 클래스에서 상속된 멤버의 새 구현을 제공합니다. `override` 선언에서 재정의된 메서드를 재정의된 기본 메서드라고 합니다. `override` 메서드에는 재정의된 기본 메서드와 동일한 시그니처가 있어야 합니다. C# 9.0부터 `override` 메서드는 공변 반환 형식을 지원합니다. 특히 `override` 메서드의 반환 형식은 해당하는 기본 메서드의 반환 형식에서 파생될 수 있습니다. C# 8.0 이전 버전에서는 `override` 메서드의 반환 형식과 재정의된 기본 메서드가 동일해야 합니다.

비가상 또는 정적 메서드는 재정의할 수 없습니다. 재정의된 기본 메서드는 `virtual`, `abstract` 또는 `override`여야 합니다.

`override` 선언에서는 `virtual` 메서드의 액세스 가능성을 변경할 수 없습니다. `override` 메서드 및 `virtual` 메서드 둘 다에 동일한 [액세스 수준 한정자](#)가 있어야 합니다.

`new`, `static` 또는 `virtual` 한정자를 사용하여 `override` 메서드를 수정할 수 없습니다.

재정의 속성 선언은 상속된 속성과 동일한 액세스 한정자, 형식 및 이름을 지정해야 합니다. C# 9.0부터 읽기 전용 재정의 속성은 공변 반환 형식을 지원합니다. 재정의된 속성은 `virtual`, `abstract` 또는 `override`여야 합니다.

`override` 키워드 사용 방법에 대한 자세한 내용은 [Override 및 New 키워드를 사용하여 버전 관리](#) 및 [Override 및 New 키워드를 사용해야 하는 경우](#)를 참조하세요. 상속에 대한 자세한 내용은 [상속](#)을 참조하세요.

## 예제

이 예제에서는 `Employee`라는 기본 클래스와 `SalesEmployee`라는 파생 클래스를 정의합니다. `SalesEmployee` 클

래스에는 추가 필드 `salesbonus` 가 포함되어 있고, 이를 고려하기 위해 `CalculatePay` 메서드를 재정의합니다.

```
class TestOverride
{
    public class Employee
    {
        public string name;

        // Basepay is defined as protected, so that it may be
        // accessed only by this class and derived classes.
        protected decimal basepay;

        // Constructor to set the name and basepay values.
        public Employee(string name, decimal basepay)
        {
            this.name = name;
            this.basepay = basepay;
        }

        // Declared virtual so it can be overridden.
        public virtual decimal CalculatePay()
        {
            return basepay;
        }
    }

    // Derive a new class from Employee.
    public class SalesEmployee : Employee
    {
        // New field that will affect the base pay.
        private decimal salesbonus;

        // The constructor calls the base-class version, and
        // initializes the salesbonus field.
        public SalesEmployee(string name, decimal basepay,
                             decimal salesbonus) : base(name, basepay)
        {
            this.salesbonus = salesbonus;
        }

        // Override the CalculatePay method
        // to take bonus into account.
        public override decimal CalculatePay()
        {
            return basepay + salesbonus;
        }
    }

    static void Main()
    {
        // Create some new employees.
        var employee1 = new SalesEmployee("Alice",
                                         1000, 500);
        var employee2 = new Employee("Bob", 1200);

        Console.WriteLine($"Employee1 {employee1.name} earned: {employee1.CalculatePay()}");
        Console.WriteLine($"Employee2 {employee2.name} earned: {employee2.CalculatePay()}");
    }
}
/*
Output:
Employee1 Alice earned: 1500
Employee2 Bob earned: 1200
*/
```

# C# 언어 사양

자세한 내용은 [C# 언어 사양의 재정의 메서드 섹션](#)을 참조하세요.

공변 반환 형식에 대한 자세한 내용은 [기능 제안 노트](#)를 참조하세요.

## 참고 항목

- [C# 참조](#)
- [상속](#)
- [C# 키워드](#)
- [한정자](#)
- [abstract](#)
- [virtual](#)
- [new\(한정자\)](#)
- [다형성](#)

# readonly(C# 참조)

2020-11-02 • 10 minutes to read • [Edit Online](#)

`readonly` 키워드는 다음 네 가지 컨텍스트에서 사용할 수 있는 한정자입니다.

- 필드 선언에서 필드에 대한 할당을 나타내는 `readonly`은 선언의 일부로 또는 동일한 클래스의 생성자에서만 발생할 수 있습니다. 필드 선언과 생성자 내에서 읽기 전용 필드를 여러 번 할당 및 재할당할 수 있습니다.

생성자가 종료된 후에는 `readonly` 필드를 할당할 수 없습니다. 이 규칙의 의미는 값 형식과 참조 형식에서 서로 다릅니다.

- 값 형식에는 해당 데이터가 직접 포함되므로, `readonly` 값 형식인 필드는 변경할 수 없습니다.
- 참조 형식에는 해당 데이터에 대한 참조가 포함되므로, `readonly` 참조 형식인 필드는 항상 같은 개체를 참조해야 합니다. 해당 개체는 변경할 수 있습니다. `readonly` 한정자는 필드가 참조 형식의 다른 인스턴스로 바뀌지 않도록 합니다. 그러나 이 한정자는 필드의 인스턴스 데이터가 읽기 전용 필드를 통해 수정되는 것을 방지하지는 않습니다.

## WARNING

변경 가능한 참조 형식인, 외부에서 볼 수 있는 읽기 전용 필드가 포함된 외부에서 볼 수 있는 형식은 보안상 취약할 수 있으며 경고 CA2104 : “변경 가능한 읽기 전용 참조 형식을 선언하지 마세요.”를 실행할 수 있습니다.

- `readonly struct` 형식 정의에서 `readonly`은 구조체 형식을 변경할 수 없음을 나타냅니다. 자세한 내용은 [구조체 형식 문서](#)의 `readonly` 구조체 섹션을 참조하세요.
- 구조체 형식 내 인스턴스 멤버 선언에서 `readonly`은 인스턴스 멤버가 구조체의 상태를 수정하지 않음을 나타냅니다. 자세한 내용은 [구조체 형식 문서](#)의 `readonly` 인스턴스 멤버 섹션을 참조하세요.
- `ref readonly` 메서드 반환에서 `readonly` 한정자는 메서드가 참조를 반환하고 해당 참조에 쓰기가 허용되지 않음을 나타냅니다.

`readonly struct` 및 `ref readonly` 컨텍스트는 C# 7.2에서 추가되었습니다. `readonly` 구조체 멤버는 C# 8.0에서 추가되었습니다.

## 읽기 전용 필드 예제

이 예제에서 `year` 필드의 값은 클래스 생성자에서 할당되었지만 `ChangeYear` 메서드에서 변경할 수 없습니다.

```
class Age
{
    readonly int year;
    Age(int year)
    {
        this.year = year;
    }
    void ChangeYear()
    {
        //year = 1967; // Compile error if uncommented.
    }
}
```

다음 컨텍스트에서만 `readonly` 필드에 값을 할당할 수 있습니다.

- 변수가 선언에서 초기화될 때. 예를 들면 다음과 같습니다.

```
public readonly int y = 5;
```

- 인스턴스 필드 선언을 포함하는 클래스의 인스턴스 생성자.
- 정적 필드 선언을 포함하는 클래스의 정적 생성자.

또한 이러한 생성자 컨텍스트에서는 `readonly` 필드를 `out` 또는 `ref` 매개 변수로 전달하는 것이 유효합니다.

#### NOTE

`readonly` 키워드와 `const` 키워드와 다릅니다. `const` 필드는 필드 선언에서만 초기화될 수 있습니다. 필드 선언과 임의 생성자에서 `readonly` 필드를 여러 번 할당할 수 있습니다. 따라서 `readonly` 필드는 사용된 생성자에 따라 다른 값을 가질 수 있습니다. 또한 `const` 필드는 컴파일 시간 상수인 반면, `readonly` 필드는 다음 예제와 같이 런타임 상수에 사용될 수 있습니다.

```
public static readonly uint timeStamp = (uint)DateTime.Now.Ticks;
```

```
public class SamplePoint
{
    public int x;
    // Initialize a readonly field
    public readonly int y = 25;
    public readonly int z;

    public SamplePoint()
    {
        // Initialize a readonly instance field
        z = 24;
    }

    public SamplePoint(int p1, int p2, int p3)
    {
        x = p1;
        y = p2;
        z = p3;
    }

    public static void Main()
    {
        SamplePoint p1 = new SamplePoint(11, 21, 32);    // OK
        Console.WriteLine($"p1: x={p1.x}, y={p1.y}, z={p1.z}");
        SamplePoint p2 = new SamplePoint();
        p2.x = 55;    // OK
        Console.WriteLine($"p2: x={p2.x}, y={p2.y}, z={p2.z}");
    }
    /*
    Output:
    p1: x=11, y=21, z=32
    p2: x=55, y=25, z=24
    */
}
```

위 예제에서 다음 예제와 같은 명령문을 사용하는 경우

```
p2.y = 66;           // Error
```

컴파일러 오류 메시지가 표시됩니다.

읽기 전용 필드에는 할당할 수 없습니다. 단 생성자 또는 변수 이니셜라이저에서는 예외입니다.

## 참조 읽기 전용 반환 예

`ref return` 의 `readonly` 한정자는 반환된 참조를 수정할 수 없음을 나타냅니다. 다음 예제는 원점에 대한 참조를 반환합니다. 예제에서는 `readonly` 한정자를 사용하여 호출자가 원본을 수정할 수 없음을 나타냅니다.

```
private static readonly SamplePoint origin = new SamplePoint(0, 0, 0);
public static ref readonly SamplePoint Origin => ref origin;
```

반환된 유형은 `readonly struct` 일 필요는 없습니다. `ref readonly` 를 통해 `ref`에서 반환될 수 있는 모든 형식을 반환할 수 있습니다.

## C# 언어 사양

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

언어 사양 제안도 확인할 수 있습니다.

- [readonly ref 및 readonly 구조체](#)
- [readonly 구조체 멤버](#)

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [한정자](#)
- [const](#)
- [필드](#)

# sealed(C# 참조)

2020-11-02 • 5 minutes to read • [Edit Online](#)

클래스에 적용된 경우 `sealed` 한정자는 다른 클래스가 해당 클래스에서 상속하지 못하도록 합니다. 다음 예제에서 `B` 클래스는 `A` 클래스에서 상속하지만 `B` 클래스에서 상속할 수 있는 클래스는 없습니다.

```
class A {}
sealed class B : A {}
```

기본 클래스의 가상 메서드 또는 속성을 재정의하는 메서드 또는 속성에 `sealed` 한정자를 사용할 수도 있습니다. 이렇게 하면 사용자 클래스에서 클래스가 파생되고 특정 가상 메서드 또는 속성을 재정의하지 못하도록 할 수 있습니다.

## 예제

다음 예제에서 `z`는 `y`에서 상속하지만 `z`는 `x`에서 선언되고 `y`에서 봉인된 가상 함수 `F`를 재정의할 수 없습니다.

```
class X
{
    protected virtual void F() { Console.WriteLine("X.F"); }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}

class Y : X
{
    sealed protected override void F() { Console.WriteLine("Y.F"); }
    protected override void F2() { Console.WriteLine("Y.F2"); }
}

class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("Z.F"); }

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

클래스에서 새 메서드 또는 속성을 정의할 때 `virtual`로 선언하지 않으면 파생 클래스가 재정의하지 못하도록 할 수 있습니다.

추상 클래스는 추상 메서드 또는 속성 구현을 제공하는 클래스에 상속되어야 하므로 봉인 클래스에 `abstract` 한정자를 사용하면 오류가 발생합니다.

메서드 또는 속성에 적용된 경우 `sealed` 한정자는 항상 `override`와 함께 사용해야 합니다.

구조체는 암시적으로 봉인되므로 상속할 수 없습니다.

자세한 내용은 [상속](#)을 참조하세요.

더 많은 예제를 보려면 [추상 및 봉인 클래스와 클래스 멤버](#)를 참조하세요.

## 예제

```

sealed class SealedClass
{
    public int x;
    public int y;
}

class SealedTest2
{
    static void Main()
    {
        var sc = new SealedClass();
        sc.x = 110;
        sc.y = 150;
        Console.WriteLine($"x = {sc.x}, y = {sc.y}");
    }
}
// Output: x = 110, y = 150

```

앞의 예제에서 다음 문을 사용하여 봉인된 클래스에서 상속을 시도할 수 있습니다.

```
class MyDerivedC: SealedClass {} // Error
```

그 결과 다음 오류 메시지가 표시됩니다.

```
'MyDerivedC': cannot derive from sealed type 'SealedClass'
```

## 설명

클래스, 메서드 또는 속성을 봉인할지 여부를 결정하려면 일반적으로 다음 두 가지 사항을 고려해야 합니다.

- 파생 클래스가 사용자 클래스의 사용자 지정을 통해 얻을 수 있는 잠재적인 이점
- 파생 클래스가 사용자 클래스를 수정하여 더 이상 올바르게 또는 예상대로 작동하지 않을 가능성

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [정적 클래스 및 정적 클래스 멤버](#)
- [추상/봉인된 클래스 및 클래스 멤버](#)
- [액세스 한정자](#)
- [한정자](#)
- [override](#)
- [virtual](#)

# static(C# 참조)

2020-11-02 • 8 minutes to read • [Edit Online](#)

이 페이지에서는 `static` 한정자 키워드를 다룹니다. `static` 키워드는 [using static](#) 지시문의 일부이기도 합니다.

`static` 한정자를 사용하여 특정 개체가 아니라 형식 자체에 속하는 정적 멤버를 선언할 수 있습니다. `static` 한정자를 사용하여 `static` 클래스를 선언할 수 있습니다. 클래스, 인터페이스 및 구조체에서 필드, 메서드, 속성, 연산자, 이벤트 및 생성자에 `static` 한정자를 추가할 수 있습니다. `static` 한정자는 인덱서 또는 종료자와 함께 사용할 수 없습니다. 자세한 내용은 [static 클래스 및 static 클래스 멤버](#)를 참조하세요.

C# 8.0부터 `static` 한정자를 [로컬 함수](#)에 추가할 수 있습니다. 정적 로컬 함수는 지역 변수 또는 인스턴스 상태를 캡처할 수 없습니다.

C# 9.0부터 [람다 식](#) 또는 [무명 메서드](#)에 `static` 한정자를 추가할 수 있습니다. 정적 람다 또는 무명 메서드는 지역 변수 또는 인스턴스 상태를 캡처할 수 없습니다.

## 예제 - 정적 클래스

다음 클래스는 `static`으로 선언되며 `static` 메서드만 포함합니다.

```
static class CompanyEmployee
{
    public static void DoSomething() { /*...*/ }
    public static void DoSomethingElse() { /*...*/ }
}
```

상수 또는 형식 선언은 암시적으로 `static` 구성원입니다. `static` 구성원은 인스턴스를 통해 참조할 수 없습니다. 대신, 형식 이름을 통해 참조됩니다. 예를 들어 다음 클래스를 예로 들어 볼 수 있습니다.

```
public class MyBaseC
{
    public struct MyStruct
    {
        public static int x = 100;
    }
}
```

`static` 구성원 `x`를 참조하려면 동일한 범위에서 구성원에 액세스할 수 없는 경우 정규화된 이름인 `MyBaseC.MyStruct.x`를 사용합니다.

```
Console.WriteLine(MyBaseC.MyStruct.x);
```

클래스 인스턴스에는 클래스의 모든 인스턴스 필드에 대한 별도 복사본이 포함되지만 각 `static` 필드의 복사본은 한 개만 있습니다.

`this`를 사용하여 `static` 메서드 또는 속성 접근자를 참조할 수는 없습니다.

`static` 키워드가 클래스에 적용된 경우 클래스의 모든 구성원은 `static`이어야 합니다.

클래스, 인터페이스 및 `static` 클래스에 `static` 생성자가 있을 수 있습니다. 프로그램이 시작되어 클래스가 인스턴스화되기 전에 `static` 생성자가 호출됩니다.

**NOTE**

`static` 키워드는 C++보다 사용이 제한적입니다. C++ 키워드와 비교하려면 [스토리지 클래스\(C++\)](#)를 참조하세요.

`static` 구성원을 보여 주려면 회사 직원을 나타내는 클래스를 고려해 보세요. 클래스에 직원 수를 구하는 메서드와 직원 수를 저장하는 필드가 포함되어 있다고 가정합니다. 메서드와 필드는 둘 다 직원 인스턴스에 속하지 않습니다. 대신 직원의 클래스에 전체적으로 속합니다. 클래스의 `static` 구성원으로 선언해야 합니다.

## 예제 - 정적 필드 및 메서드

이 예제에서는 새 직원의 이름 및 ID를 읽고, 직원 카운터를 1만큼 늘린 다음 새 직원에 대한 정보와 새 직원 수를 표시합니다. 이 프로그램은 키보드에서 현재 직원 수를 읽습니다.

```

public class Employee4
{
    public string id;
    public string name;

    public Employee4()
    {
    }

    public Employee4(string name, string id)
    {
        this.name = name;
        this.id = id;
    }

    public static int employeeCounter;

    public static int AddEmployee()
    {
        return ++employeeCounter;
    }
}

class MainClass : Employee4
{
    static void Main()
    {
        Console.Write("Enter the employee's name: ");
        string name = Console.ReadLine();
        Console.Write("Enter the employee's ID: ");
        string id = Console.ReadLine();

        // Create and configure the employee object.
        Employee4 e = new Employee4(name, id);
        Console.Write("Enter the current number of employees: ");
        string n = Console.ReadLine();
        Employee4.employeeCounter = Int32.Parse(n);
        Employee4.AddEmployee();

        // Display the new information.
        Console.WriteLine($"Name: {e.name}");
        Console.WriteLine($"ID: {e.id}");
        Console.WriteLine($"New Number of Employees: {Employee4.employeeCounter}");
    }
}
/*
Input:
Matthias Berndt
AF643G
15
*
Sample Output:
Enter the employee's name: Matthias Berndt
Enter the employee's ID: AF643G
Enter the current number of employees: 15
Name: Matthias Berndt
ID: AF643G
New Number of Employees: 16
*/

```

## 예제 - 정적 초기화

이 예제에서는 아직 선언되지 않은 다른 `static` 필드를 사용하여 `static` 필드를 초기화할 수 있음을 보여 줍니다. `static` 필드에 값을 명시적으로 할당할 때까지 결과는 정의되지 않습니다.

```
class Test
{
    static int x = y;
    static int y = 5;

    static void Main()
    {
        Console.WriteLine(Test.x);
        Console.WriteLine(Test.y);

        Test.x = 99;
        Console.WriteLine(Test.x);
    }
}
/*
Output:
0
5
99
*/
```

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [한정자](#)
- [using 정적 지시문](#)
- [정적 클래스 및 정적 클래스 멤버](#)

# unsafe(C# 참조)

2020-11-02 • 3 minutes to read • [Edit Online](#)

`unsafe` 키워드는 포인터와 관련된 모든 작업에 필요한 안전하지 않은 컨텍스트를 나타냅니다. 자세한 내용은 [안전하지 않은 코드 및 포인터](#)를 참조하세요.

형식 또는 멤버 선언에서 `unsafe` 한정자를 사용할 수 있습니다. 따라서 형식 또는 멤버의 전체 텍스트 범위가 안전하지 않은 컨텍스트로 간주됩니다. 예를 들어 다음은 `unsafe` 한정자를 사용하여 선언된 메서드입니다.

```
unsafe static void FastCopy(byte[] src, byte[] dst, int count)
{
    // Unsafe context: can use pointers here.
}
```

안전하지 않은 컨텍스트의 범위는 매개 변수 목록에서 메서드의 끝까지 확장되므로 매개 변수 목록에 포인터를 사용할 수도 있습니다.

```
unsafe static void FastCopy ( byte* ps, byte* pd, int count ) {...}
```

안전하지 않은 블록을 통해 이 블록 내에서 안전하지 않은 코드를 사용할 수도 있습니다. 예:

```
unsafe
{
    // Unsafe context: can use pointers here.
}
```

안전하지 않은 코드를 컴파일하려면 `-unsafe` 컴파일러 옵션을 지정해야 합니다. 안전하지 않은 코드는 공용 언어 런타임에서 확인할 수 없습니다.

## 예제

```
// compile with: -unsafe
class UnsafeTest
{
    // Unsafe method: takes pointer to int.
    unsafe static void SquarePtrParam(int* p)
    {
        *p *= *p;
    }

    unsafe static void Main()
    {
        int i = 5;
        // Unsafe method: uses address-of operator (&).
        SquarePtrParam(&i);
        Console.WriteLine(i);
    }
}
// Output: 25
```

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 안전하지 않은 코드](#)를 참조하세요. 언어 사양은 C# 구문 및 사용법에 대한 신뢰할 수 있는 소스 됩니다.

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [fixed 문](#)
- [안전하지 않은 코드 및 포인터](#)
- [고정 크기 버퍼](#)

# virtual(C# 참조)

2020-11-02 • 6 minutes to read • [Edit Online](#)

`virtual` 키워드는 메서드, 속성, 인덱서 또는 이벤트 선언을 수정하고 파생 클래스에서 재정의하도록 허용하는데 사용됩니다. 예를 들어 이 메서드는 이를 상속하는 모든 클래스에서 재정의할 수 있습니다.

```
public virtual double Area()
{
    return x * y;
}
```

가상 멤버의 구현은 파생 클래스의 **재정의 멤버**로 변경할 수 있습니다. `virtual` 키워드 사용 방법에 대한 자세한 내용은 [Override 및 New 키워드를 사용하여 버전 관리](#) 및 [Override 및 New 키워드를 사용해야 하는 경우를 참조하세요](#).

## 설명

가상 메서드가 호출되면 재정의 멤버에 대해 개체의 런타임 형식이 확인됩니다. 파생 클래스가 멤버를 재정의하지 않은 경우 가장 많이 파생된 클래스의 재정의 멤버(원래 멤버일 수 있음)가 호출됩니다.

기본적으로 메서드는 가상이 아닙니다. 가상이 아닌 메서드는 재정의할 수 없습니다.

`virtual` 한정자는 `static`, `abstract`, `private` 또는 `override` 한정자와 사용할 수 없습니다. 다음 예제에서는 가상 속성을 보여 줍니다.

```

class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int num;
    public virtual int Number
    {
        get { return num; }
        set { num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string name;

    // Override auto-implemented property with ordinary property
    // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                name = value;
            }
            else
            {
                name = "Unknown";
            }
        }
    }
}

```

선언과 호출 구문에서의 차이점을 제외하면 가상 속성은 가상 메서드처럼 작동합니다.

- 정적 속성에서 `virtual` 한정자를 사용하는 것은 오류입니다.
- `override` 한정자를 사용하는 속성 선언을 포함함으로써 상속된 가상 속성을 파생 클래스에서 재정의할 수 있습니다.

## 예제

이 예제에서 `Shape` 클래스는 두 개의 좌표 `x`, `y` 와 `Area()` 가상 메서드를 포함합니다. `Circle`, `Cylinder` 및 `Sphere` 와 같은 서로 다른 도형의 클래스는 `Shape` 클래스를 상속하며, 각 모양에 대해 노출 영역이 계산됩니다. 각 파생 클래스에는 `Area()` 의 자체 재정의 구현이 있습니다.

다음 선언에 나와 있는 것처럼 상속된 클래스 `Circle`, `Sphere` 및 `Cylinder`는 모두 기본 클래스를 초기화하는 생성자를 사용합니다.

```
public Cylinder(double r, double h): base(r, h) {}
```

다음 프로그램은 메서드와 연결된 개체에 따라 `Area()` 메서드의 적절한 구현을 호출하여 각 그림의 해당 영역을 계산하고 표시합니다.

```

class TestClass
{
    public class Shape
    {
        public const double PI = Math.PI;
        protected double x, y;

        public Shape()
        {
        }

        public Shape(double x, double y)
        {
            this.x = x;
            this.y = y;
        }

        public virtual double Area()
        {
            return x * y;
        }
    }

    public class Circle : Shape
    {
        public Circle(double r) : base(r, 0)
        {
        }

        public override double Area()
        {
            return PI * x * x;
        }
    }

    class Sphere : Shape
    {
        public Sphere(double r) : base(r, 0)
        {
        }

        public override double Area()
        {
            return 4 * PI * x * x;
        }
    }

    class Cylinder : Shape
    {
        public Cylinder(double r, double h) : base(r, h)
        {
        }

        public override double Area()
        {
            return 2 * PI * x * x + 2 * PI * x * y;
        }
    }

    static void Main()
    {
        double r = 3.0, h = 5.0;
        Shape c = new Circle(r);
        Shape s = new Sphere(r);
        Shape l = new Cylinder(r, h);
        // Display results.
        Console.WriteLine("Area of Circle    = {0:F2}", c.Area());
        Console.WriteLine("Area of Sphere   = {0:F2}", s.Area());
    }
}

```

```
        Console.WriteLine("Area of Cylinder = {0:F2}", l.Area());
    }
}
/*
Output:
Area of Circle    = 28.27
Area of Sphere    = 113.10
Area of Cylinder = 150.80
*/
```

## C# 언어 사양

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

## 참조

- [다형성](#)
- [abstract](#)
- [override](#)
- [new\(한정자\)](#)

# volatile(C# 참조)

2020-11-02 • 7 minutes to read • [Edit Online](#)

`volatile` 키워드는 동시에 실행되는 여러 스레드에 의해 필드가 수정될 수 있음을 나타냅니다. 컴파일러, 런타임 시스템 및 하드웨어는 성능상의 이유로 메모리 위치에 대한 읽기 및 쓰기를 다시 정렬할 수 있습니다.

`volatile`로 선언된 필드에는 이러한 최적화가 적용되지 않습니다. `volatile` 한정자를 추가하면 모든 스레드가 수행된 순서대로 다른 스레드가 휘발성 쓰기를 수행합니다. 모든 실행 스레드에서처럼 휘발성 쓰기의 단일 순서가 모두 보장되는 것은 아닙니다.

`volatile` 키워드는 다음 형식의 필드에 적용될 수 있습니다.

- 참조 형식.
- 포인터 형식(안전하지 않은 컨텍스트에서). 포인터 자체는 volatile이 될 수 있지만, 포인터가 가리키는 개체는 volatile이 될 수 없습니다. 즉, "pointer to volatile"을 선언할 수 없습니다.
- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `char`, `float` 및 `bool`과 같은 단순 형식.
- 기본 형식 `byte`, `sbyte`, `short`, `ushort`, `int` 또는 `uint` 중 하나가 있는 `enum` 형식.
- 참조 형식으로 알려진 제네릭 형식 매개 변수.
- `IntPtr`와 `UIntPtr`를 참조하세요.

`double` 및 `long`을 포함한 기타 형식은 해당 형식의 필드에 대한 읽기 및 쓰기가 원자성임을 보장할 수 없기 때문에 `volatile`로 표시될 수 없습니다. 이러한 형식의 필드에 대한 다중 스레드 액세스를 보호하려면 `Interlocked` 클래스 멤버를 사용하거나 `lock` 문을 통해 액세스를 보호합니다.

`volatile` 키워드는 `class` 또는 `struct`의 필드에만 적용할 수 있습니다. 지역 변수는 `volatile`로 선언할 수 없습니다.

## 예제

다음 예제에서는 공용 필드 변수를 `volatile`로 선언하는 방법을 보여 줍니다.

```
class VolatileTest
{
    public volatile int sharedStorage;

    public void Test(int _i)
    {
        sharedStorage = _i;
    }
}
```

다음 예제에서는 보조 또는 작업자 스레드를 만들어 기본 스레드와 병렬로 처리하는 데 사용하는 방법을 보여 줍니다. 다중 스레딩에 대한 자세한 내용은 [관리되는 스레딩](#)을 참조하세요.

```

public class Worker
{
    // This method is called when the thread is started.
    public void DoWork()
    {
        bool work = false;
        while (!_shouldStop)
        {
            work = !work; // simulate some work
        }
        Console.WriteLine("Worker thread: terminating gracefully.");
    }
    public void RequestStop()
    {
        _shouldStop = true;
    }
    // Keyword volatile is used as a hint to the compiler that this data
    // member is accessed by multiple threads.
    private volatile bool _shouldStop;
}

public class WorkerThreadExample
{
    public static void Main()
    {
        // Create the worker thread object. This does not start the thread.
        Worker workerObject = new Worker();
        Thread workerThread = new Thread(workerObject.DoWork);

        // Start the worker thread.
        workerThread.Start();
        Console.WriteLine("Main thread: starting worker thread...");

        // Loop until the worker thread activates.
        while (!workerThread.IsAlive)
        {

        }

        // Put the main thread to sleep for 500 milliseconds to
        // allow the worker thread to do some work.
        Thread.Sleep(500);

        // Request that the worker thread stop itself.
        workerObject.RequestStop();

        // Use the Thread.Join method to block the current thread
        // until the object's thread terminates.
        workerThread.Join();
        Console.WriteLine("Main thread: worker thread has terminated.");
    }
    // Sample output:
    // Main thread: starting worker thread...
    // Worker thread: terminating gracefully.
    // Main thread: worker thread has terminated.
}

```

`_shouldStop`의 선언에 `volatile` 한정자를 추가하면 항상 동일한 결과가 표시됩니다(앞의 코드에 표시된 것과 유사함). 그러나 `_shouldStop` 멤버의 해당 한정자가 없으면 동작을 예측할 수 없습니다. `DoWork` 메서드가 멤버 액세스를 최적화할 수 있으므로 부실 데이터를 읽게 됩니다. 다중 스레드 프로그래밍의 특성으로 인해 부실 읽기 수는 예측할 수 없습니다. 프로그램의 실행에 따라 약간 다른 결과가 생성됩니다.

## C# 언어 사양

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

## 참조

- C# 언어 사양: volatile 키워드
- C# 참조
- C# 프로그래밍 가이드
- C# 키워드
- 한정자
- lock 문
- Interlocked

# 문 키워드(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

문은 프로그램 명령입니다. 다음 표에서 참조된 항목에 설명된 경우를 제외하고 문은 순서대로 실행됩니다. 다음 표에는 C# 문 키워드가 나와 있습니다. 키워드로 표현되지 않은 문에 대한 자세한 내용은 [문](#)을 참조하세요.

범주	C# 키워드
선택 문	<code>if, else, switch, case</code>
반복 문	<code>do, for, foreach, in, while</code>
점프 문	<code>break, continue, default, goto, return, yield</code>
예외 처리 문	<code>throw, try-catch, try-finally, try-catch-finally</code>
Checked 및 Unchecked	<code>checked, unchecked</code>
fixed 문	<code>fixed</code>
lock 문	<code>lock</code>

## 참고 항목

- [C# 참조](#)
- [문](#)
- [C# 키워드](#)

# if-else(C# 참조)

2020-11-02 • 9 minutes to read • [Edit Online](#)

`if` 문은 부울 식의 값에 따라 실행할 문을 식별합니다. 다음 예제에서는 `bool` 변수 `condition` 가 `true` 로 설정된 다음 `if` 문에서 확인됩니다. 출력은 `The variable is set to true.` 입니다.

```
bool condition = true;

if (condition)
{
    Console.WriteLine("The variable is set to true.");
}
else
{
    Console.WriteLine("The variable is set to false.");
}
```

콘솔 앱의 `Main` 메서드에 배치하여 이 항목의 예제를 실행할 수 있습니다.

C#의 `if` 문은 다음 예제와 같이 두 가지 형식을 사용할 수 있습니다.

```
// if-else statement
if (condition)
{
    then-statement;
}
else
{
    else-statement;
}
// Next statement in the program.

// if statement without an else
if (condition)
{
    then-statement;
}
// Next statement in the program.
```

`if-else` 문에서 `condition` 이 `true`로 평가되는 경우 `then-statement` 가 실행됩니다. `condition` 이 `false`이면 `else-statement` 가 실행됩니다. `condition` 이 `true`인 동시에 `false`일 수는 없으므로 `then-statement` 문의 `else-statement` 및 `if-else` 를 둘 다 실행할 수는 없습니다. `then-statement` 또는 `else-statement` 가 실행된 후 제어가 `if` 문 뒤의 다음 문으로 전송됩니다.

`if` 문을 포함하지 않는 `else` 문에서 `condition` 이 `true`이면 `then-statement` 가 실행됩니다. `condition` 이 `false`이면 제어가 `if` 문 뒤의 다음 문으로 전송됩니다.

`then-statement` 및 `else-statement` 는 둘 다 단일 문이나 중괄호(`{}`)로 묶인 여러 문으로 구성될 수 있습니다. 단일 문의 경우 중괄호는 선택 사항이지만 권장됩니다.

`then-statement` 및 `else-statement` 의 문은 원래 `if` 문 내부에 중첩된 다른 `if` 문을 포함하여 모든 종류일 수 있습니다. 중첩된 `if` 문에서 각 `else` 절은 해당 `if` 가 없는 마지막 `else`에 속합니다. 다음 예제에서 `Result1` 및 `m > 10` 이 둘 다 `true`이면 `n > 20` 이 나타납니다. `m > 10` 이 `true`이지만 `n > 20` 이 `false`이면 `Result2` 가 나타납니다.

```
// Try with m = 12 and then with m = 8.  
int m = 12;  
int n = 18;  
  
if (m > 10)  
    if (n > 20)  
    {  
        Console.WriteLine("Result1");  
    }  
else  
{  
    Console.WriteLine("Result2");  
}
```

Result2 0| false일 때 `(m > 10)` 를 대신 표시하려는 경우 다음 예제와 같이 중괄호로 중첩된 `if` 문의 시작 및 끝을 설정하여 해당 연결을 지정할 수 있습니다.

```
// Try with m = 12 and then with m = 8.  
if (m > 10)  
{  
    if (n > 20)  
        Console.WriteLine("Result1");  
}  
else  
{  
    Console.WriteLine("Result2");  
}
```

조건 `(m > 10)` 0| false이면 Result2 가 나타납니다.

## 예제

다음 예제에서는 키보드에서 문자 하나를 입력하며, 프로그램이 중첩된 `if` 문을 사용하여 입력 문자가 영문자인지 여부를 확인합니다. 입력 문자가 영문자이면 프로그램에서 입력 문자가 소문자인지 대문자인지를 확인합니다. 각 경우에 대한 메시지가 나타납니다.

```

Console.Write("Enter a character: ");
char c = (char)Console.Read();
if (Char.IsLetter(c))
{
    if (Char.ToLower(c))
    {
        Console.WriteLine("The character is lowercase.");
    }
    else
    {
        Console.WriteLine("The character is uppercase.");
    }
}
else
{
    Console.WriteLine("The character isn't an alphabetic character.");
}

//Sample Output:

//Enter a character: 2
//The character isn't an alphabetic character.

//Enter a character: A
//The character is uppercase.

//Enter a character: h
//The character is lowercase.

```

## 예제

다음 부분 코드와 같이 `if` 문을 `else` 블록 내에 중첩할 수도 있습니다. 예제에서는 `if` 문을 `else` 블록 2개와 `then` 블록 1개 안에 중첩합니다. 주석은 각 블록에서 `true` 또는 `false`인 조건을 지정합니다.

```

// Change the values of these variables to test the results.
bool Condition1 = true;
bool Condition2 = true;
bool Condition3 = true;
bool Condition4 = true;

if (Condition1)
{
    // Condition1 is true.
}
else if (Condition2)
{
    // Condition1 is false and Condition2 is true.
}
else if (Condition3)
{
    if (Condition4)
    {
        // Condition1 and Condition2 are false. Condition3 and Condition4 are true.
    }
    else
    {
        // Condition1, Condition2, and Condition4 are false. Condition3 is true.
    }
}
else
{
    // Condition1, Condition2, and Condition3 are false.
}

```

## 예제

다음 예제에서는 입력 문자가 소문자, 대문자 또는 숫자인지를 확인합니다. 세 가지 조건이 모두 false이면 문자는 영숫자 문자가 아닙니다. 예제에서는 각 경우에 대한 메시지가 표시됩니다.

```
Console.WriteLine("Enter a character: ");
char ch = (char)Console.Read();

if (Char.IsUpper(ch))
{
    Console.WriteLine("The character is an uppercase letter.");
}
else if (Char.IsLower(ch))
{
    Console.WriteLine("The character is a lowercase letter.");
}
else if (Char.IsDigit(ch))
{
    Console.WriteLine("The character is a number.");
}
else
{
    Console.WriteLine("The character is not alphanumeric.");
}

//Sample Input and Output:
//Enter a character: E
//The character is an uppercase letter.

//Enter a character: e
//The character is a lowercase letter.

//Enter a character: 4
//The character is a number.

//Enter a character: =
//The character is not alphanumeric.
```

`else` 블록 또는 `then` 블록의 문이 유효한 모든 문일 수 있는 것과 마찬가지로 유효한 모든 부울 식을 조건에 대해 사용할 수 있습니다. `!`, `&&`, `||`, `&`, `|` 및 `^`와 같은 논리 연산자를 사용하여 복합 조건을 만들 수 있습니다. 다음 코드는 예제를 보여 줍니다.

```

// NOT
bool result = true;
if (!result)
{
    Console.WriteLine("The condition is true (result is false).");
}
else
{
    Console.WriteLine("The condition is false (result is true).");
}

// Short-circuit AND
int m = 9;
int n = 7;
int p = 5;
if (m >= n && m >= p)
{
    Console.WriteLine("Nothing is larger than m.");
}

// AND and NOT
if (m >= n && !(p > m))
{
    Console.WriteLine("Nothing is larger than m.");
}

// Short-circuit OR
if (m > n || m > p)
{
    Console.WriteLine("m isn't the smallest.");
}

// NOT and OR
m = 4;
if (!(m >= n || m >= p))
{
    Console.WriteLine("Now m is the smallest.");
}
// Output:
// The condition is false (result is true).
// Nothing is larger than m.
// Nothing is larger than m.
// m isn't the smallest.
// Now m is the smallest.

```

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [?: 연산자](#)
- [if-else 문\(C++\)](#)
- [switch](#)

# switch(C# 참조)

2020-11-02 • 31 minutes to read • [Edit Online](#)

이 문서에서는 `switch` 문에 대해 설명합니다. C# 8.0에 도입된 `switch` 식에 대한 자세한 내용은 [식 및 연산자](#) 섹션에서 `switch` 식 문서를 참조하세요.

`switch`는 [일치](#) 식을 사용한 패턴 일치를 기반으로 하여 후보 목록에서 실행 할 `switch` 섹션 하나를 선택하는 선택 문입니다.

```
using System;

public class Example
{
    public static void Main()
    {
        int caseSwitch = 1;

        switch (caseSwitch)
        {
            case 1:
                Console.WriteLine("Case 1");
                break;
            case 2:
                Console.WriteLine("Case 2");
                break;
            default:
                Console.WriteLine("Default case");
                break;
        }
    }
}

// The example displays the following output:
//      Case 1
```

단일 식을 3개 이상의 조건에 대해 테스트하는 경우 `switch` 문이 [if-else](#) 구문 대신 사용되는 경우가 많습니다. 예를 들어 다음 `switch` 문은 `color` 형식의 변수에 다음 세 가지 값 중 하나가 있는지 여부를 확인합니다.

```

using System;

public enum Color { Red, Green, Blue }

public class Example
{
    public static void Main()
    {
        Color c = (Color) (new Random()).Next(0, 3);
        switch (c)
        {
            case Color.Red:
                Console.WriteLine("The color is red");
                break;
            case Color.Green:
                Console.WriteLine("The color is green");
                break;
            case Color.Blue:
                Console.WriteLine("The color is blue");
                break;
            default:
                Console.WriteLine("The color is unknown.");
                break;
        }
    }
}

```

이 문은 `if - else` 구문을 사용하는 다음 예제와 같습니다.

```

using System;

public enum Color { Red, Green, Blue }

public class Example
{
    public static void Main()
    {
        Color c = (Color) (new Random()).Next(0, 3);
        if (c == Color.Red)
            Console.WriteLine("The color is red");
        else if (c == Color.Green)
            Console.WriteLine("The color is green");
        else if (c == Color.Blue)
            Console.WriteLine("The color is blue");
        else
            Console.WriteLine("The color is unknown.");
    }
}

// The example displays the following output:
//      The color is red

```

## 일치 식

일치 식은 `case` 레이블의 패턴과 일치시킬 값을 제공합니다. 구문은 다음과 같습니다.

```
switch (expr)
```

C# 6 이하에서 일치 식은 다음 형식의 값을 반환하는 식이어야 합니다.

- `char`
- `string`

- `bool`
- 정수 값(예: `int` 또는 `long`)입니다.
- `enum` 값

C# 7.0부터 일치 식은 null이 아닌 모든 식일 수 있습니다.

## switch 섹션

`switch` 문에는 스위치 섹션이 하나 이상 포함됩니다. 각 `switch` 섹션에는 하나 이상의 `case` 레이블(사례 또는 기본 레이블) 및 하나 이상의 명령문이 있습니다. `switch` 문은 `switch` 섹션에 있는 기본 레이블이 하나만 포함될 수 있습니다. 다음 예제에서는 각각 두 개의 명령문을 포함하는 세 개의 `switch` 섹션이 있는 간단한 `switch` 문을 보여 줍니다. 두 번째 `switch` 섹션에는 `case 2:` 및 `case 3:` 레이블이 포함되어 있습니다.

`switch` 문에 포함할 수 있는 `switch` 섹션 수에는 제한이 없으며 각 섹션에 하나 이상의 `case` 레이블을 포함할 수 있습니다(다음 예제 참조). 하지만 두 `case` 레이블에 동일한 식을 포함할 수는 없습니다.

```
using System;

public class Example
{
    public static void Main()
    {
        Random rnd = new Random();
        int caseSwitch = rnd.Next(1,4);

        switch (caseSwitch)
        {
            case 1:
                Console.WriteLine("Case 1");
                break;
            case 2:
            case 3:
                Console.WriteLine($"Case {caseSwitch}");
                break;
            default:
                Console.WriteLine($"An unexpected value ({caseSwitch})");
                break;
        }
    }
}

// The example displays output like the following:
//      Case 1
```

`switch` 문에서 하나의 `switch` 섹션만 실행됩니다. C#은 한 `switch` 섹션에서 다음 `switch` 섹션으로 계속 실행하도록 허용하지 않습니다. 이로 인해 다음 코드는 다음과 같은 컴파일러 오류를 생성합니다. CS0163: "한 `case` 레이블에서 다른 `case` 레이블(<code label>)로 제어를 이동할 수 없습니다."

```
switch (caseSwitch)
{
    // The following switch section causes an error.
    case 1:
        Console.WriteLine("Case 1...");
        // Add a break or other jump statement here.
    case 2:
        Console.WriteLine("... and/or Case 2");
        break;
}
```

이 요구 사항은 일반적으로 `break`, `goto` 또는 `return` 문을 통해 `switch` 섹션을 명시적으로 종료하여 충족됩니다. 그러나 다음 코드는 프로그램 제어가 `default` `switch` 섹션으로 이동할 수 없도록 하기 때문에 유효합니다.

```

switch (caseSwitch)
{
    case 1:
        Console.WriteLine("Case 1...");
        break;
    case 2:
    case 3:
        Console.WriteLine("... and/or Case 2");
        break;
    case 4:
        while (true)
            Console.WriteLine("Endless looping. . .");
    default:
        Console.WriteLine("Default value...");
        break;
}

```

case 레이블이 일치 식과 일치하는 switch 섹션에서 문 목록의 실행은 첫 번째 문으로 시작하고 일반적으로 `break`, `goto case`, `goto label`, `return` 또는 `throw` 같은 점프 문에 도달할 때까지 문 목록 전체를 진행합니다. 이 경우 `switch` 문 외부 또는 다른 case 레이블로 제어를 보냅니다. 사용되는 경우 `goto` 문은 constant 레이블에 컨트롤을 전달해야 합니다. 이 제한 사항이 필요합니다. constant가 아닌 레이블에 컨트롤을 전달하려고 하면 코드의 의도치 않은 위치에 컨트롤을 전달하거나 무한 루프를 만드는 것과 같은 원치 않는 부작용이 발생할 수 있기 때문입니다.

## case 레이블

각 case 레이블은 일치 식과 비교할 패턴(이전 예제의 `caseSwitch` 변수)을 지정합니다. 일치하는 경우 제어가 일치하는 첫 번째 case 레이블을 포함하는 switch 섹션으로 전송됩니다. 일치 식과 일치하는 case 레이블 패턴이 없는 경우 제어가 `default` case 레이블을 포함하는 섹션으로 전송됩니다(있는 경우). `default` case가 없는 경우 switch 섹션의 문이 실행되지 않으며, 제어가 `switch` 문 외부로 전송됩니다.

`switch` 문과 패턴 일치에 대한 자세한 내용은 [switch 문을 사용한 패턴 일치](#) 섹션을 참조하세요.

C# 6에서 상수 패턴만 지원하고 상수 값의 반복을 허용하지 않는 경우 case 레이블은 상호 배타적인 값을 정의하며 하나의 패턴만 일치 식과 일치할 수 있습니다. 따라서 `case` 문이 표시되는 순서는 중요하지 않습니다.

그러나 C# 7.0에서는 다른 패턴도 지원되므로 case 레이블에서 상호 배타적인 값을 정의할 필요가 없으며 여러 패턴이 일치 식과 일치할 수 있습니다. 일치 패턴을 포함하는 첫 번째 switch 섹션의 문만 실행되기 때문에 이제 `case` 문이 나타나는 순서가 중요합니다. C#에서 case 문이 이전 문과 같거나 이전 문의 하위 집합인 switch 섹션을 검색할 경우 컴파일러 오류, CS8120, "Switch case가 이전 case에서 이미 처리되었습니다."를 생성합니다.

다음 예제에서는 상호 배타적이 아닌 다양한 패턴을 사용하는 `switch` 문을 보여 줍니다. 더 이상 `switch` 문의 첫 번째 섹션이 아니도록 `case 0:` switch 섹션을 이동하는 경우 해당 값이 0인 정수는 `case int val` 문에 정의된 패턴인 모든 정수의 하위 집합이므로 C#에서 컴파일러 오류를 생성합니다.

```

using System;
using System.Collections.Generic;
using System.Linq;

public class Example
{
    public static void Main()
    {
        var values = new List<object>();
        for (int ctr = 0; ctr <= 7; ctr++) {
            if (ctr == 2)
                values.Add(DiceLibrary.Roll2());
            else if (ctr == 4)
                values.Add(DiceLibrary.Pass());
            else

```

```

        values.Add(DiceLibrary.Roll());
    }

    Console.WriteLine($"The sum of { values.Count } die is { DiceLibrary.DiceSum(values) }");
}
}

public static class DiceLibrary
{
    // Random number generator to simulate dice rolls.
    static Random rnd = new Random();

    // Roll a single die.
    public static int Roll()
    {
        return rnd.Next(1, 7);
    }

    // Roll two dice.
    public static List<object> Roll2()
    {
        var rolls = new List<object>();
        rolls.Add(Roll());
        rolls.Add(Roll());
        return rolls;
    }

    // Calculate the sum of n dice rolls.
    public static int DiceSum(IEnumerable<object> values)
    {
        var sum = 0;
        foreach (var item in values)
        {
            switch (item)
            {
                // A single zero value.
                case 0:
                    break;
                // A single value.
                case int val:
                    sum += val;
                    break;
                // A non-empty collection.
                case IEnumerable<object> subList when subList.Any():
                    sum += DiceSum(subList);
                    break;
                // An empty collection.
                case IEnumerable<object> subList:
                    break;
                // A null reference.
                case null:
                    break;
                // A value that is neither an integer nor a collection.
                default:
                    throw new InvalidOperationException("unknown item type");
            }
        }
        return sum;
    }

    public static object Pass()
    {
        if (rnd.Next(0, 2) == 0)
            return null;
        else
            return new List<object>();
    }
}

```

이 문제를 해결하고 다음 두 가지 방법 중 하나로 컴파일러 경고를 제거할 수 있습니다.

- switch 섹션의 순서 변경
- `case` 레이블에서 `when` 절 사용

## default case

`default case`는 일치 식이 다른 `case` 레이블과 일치하지 않는 경우 실행할 switch 섹션을 지정합니다.

`default case`가 없고 일치 식이 다른 `case` 레이블과 일치하지 않는 경우 프로그램 흐름이 `switch` 문으로 이동합니다.

`default case`는 `switch` 문에 임의 순서로 표시될 수 있습니다. 소스 코드에서 해당 순서와 관계없이 항상 모든 `case` 레이블이 평가된 후 마지막에 평가됩니다.

## switch 문과 일치하는 패턴

각 `case` 문은 일치 식과 일치할 경우 포함하는 switch 섹션이 실행되는 패턴을 정의합니다. 상수 패턴은 모든 버전의 C#에서 지원됩니다. 나머지 패턴은 C# 7.0부터 지원됩니다.

### 상수 패턴

상수 패턴은 일치 식이 지정된 상수와 같은지 여부를 테스트합니다. 구문은 다음과 같습니다.

```
case constant:
```

여기서 `constant`는 테스트 할 값입니다. `constant`는 다음 상수 식 중 하나가 될 수 있습니다.

- `bool` 리터럴( `true` 또는 `false` ).
- 정수 상수(예: `int`, `long` 또는 `byte` )입니다.
- 선언된 `const` 변수의 이름
- 열거형 상수
- `char` 리터럴
- `string` 리터럴

상수 식은 다음과 같이 계산됩니다.

- `expr` 및 `constant`가 정수 형식인 경우 C# 같은 연산자는 식에서 `true`를 반환하는지 여부 즉, `expr == constant`인지 여부를 확인합니다.
- 정수 형식이 아니면 static `Object.Equals(expr, constant)` 메서드 호출을 통해 식의 값이 결정됩니다.

다음 예제에서는 상수 패턴을 사용하여 특정 날짜가 주말인지, 작업 주의 첫째 날인지, 작업 주의 마지막 날인지 또는 작업 주의 중간인지를 확인합니다. 현재 날짜의 `DateTime.DayOfWeek` 속성을 `DayOfWeek` 열거형의 멤버와 비교해서 평가합니다.

```
using System;

class Program
{
    static void Main()
    {
        switch (DateTime.Now.DayOfWeek)
        {
            case DayOfWeek.Sunday:
            case DayOfWeek.Saturday:
                Console.WriteLine("The weekend");
                break;
            case DayOfWeek.Monday:
                Console.WriteLine("The first day of the work week.");
                break;
            case DayOfWeek.Friday:
                Console.WriteLine("The last day of the work week.");
                break;
            default:
                Console.WriteLine("The middle of the work week.");
                break;
        }
    }
}

// The example displays output like the following:
//      The middle of the work week.
```

다음 예제에서는 상수 패턴을 사용하여 자동 커피 머신을 시뮬레이트하는 콘솔 애플리케이션의 사용자 입력을 처리합니다.

```

using System;

class Example
{
    static void Main()
    {
        Console.WriteLine("Coffee sizes: 1=small 2=medium 3=large");
        Console.Write("Please enter your selection: ");
        string str = Console.ReadLine();
        int cost = 0;

        // Because of the goto statements in cases 2 and 3, the base cost of 25
        // cents is added to the additional cost for the medium and large sizes.
        switch (str)
        {
            case "1":
            case "small":
                cost += 25;
                break;
            case "2":
            case "medium":
                cost += 25;
                goto case "1";
            case "3":
            case "large":
                cost += 50;
                goto case "1";
            default:
                Console.WriteLine("Invalid selection. Please select 1, 2, or 3.");
                break;
        }
        if (cost != 0)
        {
            Console.WriteLine("Please insert {0} cents.", cost);
        }
        Console.WriteLine("Thank you for your business.");
    }
}

// The example displays output like the following:
//      Coffee sizes: 1=small 2=medium 3=large
//      Please enter your selection: 2
//      Please insert 50 cents.
//      Thank you for your business.

```

## 형식 패턴

형식 패턴은 간결한 형식 평가 및 변환을 사용하도록 설정합니다. `switch` 문과 함께 사용하여 패턴 일치를 수행하는 경우 식을 지정된 형식으로 변환할 수 있는지 여부를 테스트하고, 변환할 수 있으면 해당 형식의 변수로 캐스팅합니다. 구문은 다음과 같습니다.

```
case type varname
```

여기서 `type`은 `expr`의 결과가 변환되는 형식의 이름이고, `varname`은 일치에 성공할 경우 `expr`의 결과가 변환되는 개체입니다. `expr`의 컴파일 시간 형식은 C# 7.1부터 제네릭 형식 매개 변수일 수 있습니다.

다음 중 하나가 `true`일 경우 `case` 식은 `true`입니다.

- `expr0 | type`과 동일한 형식의 인스턴스입니다.
- `expr0 | type`에서 파생된 형식의 인스턴스입니다. 즉, `expr`의 결과를 `type`의 인스턴스로 업캐스트할 수 있습니다.
- `expr`의 컴파일 시간 형식은 `type`의 기본 클래스이고 `expr`의 런타임 형식은 `type0`이나 `type`에서 파생됨

다. 변수의 **컴파일 시간 형식**은 해당 형식 선언에 정의된 변수의 형식입니다. 변수의 **런타임 형식**은 해당 변수에 할당된 인스턴스의 형식입니다.

- *expr* | *type* 인터페이스를 구현하는 형식의 인스턴스입니다.

case 식이 true이면 *varname*이 한정적으로 할당되고 switch 셕션 내의 로컬 범위만 갖습니다.

`null`은 형식과 일치하지 않습니다. `null`과 일치하려면 다음 `case` 레이블을 사용합니다.

```
case null:
```

다음 예제에서는 형식 패턴을 사용하여 다양한 종류의 컬렉션 형식에 대한 정보를 제공합니다.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

class Example
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8, 10 };
        ShowCollectionInformation(values);

        var names = new List<string>();
        names.AddRange(new string[] { "Adam", "Abigail", "Bertrand", "Bridgette" });
        ShowCollectionInformation(names);

        List<int> numbers = null;
        ShowCollectionInformation(numbers);
    }

    private static void ShowCollectionInformation(object coll)
    {
        switch (coll)
        {
            case Array arr:
                Console.WriteLine($"An array with {arr.Length} elements.");
                break;
            case IEnumerable<int> ieInt:
                Console.WriteLine($"Average: {ieInt.Average(s => s)}");
                break;
            case IList list:
                Console.WriteLine($"{list.Count} items");
                break;
            case IEnumerable ie:
                string result = "";
                foreach (var e in ie)
                    result += $"{e} ";
                Console.WriteLine(result);
                break;
            case null:
                // Do nothing for a null.
                break;
            default:
                Console.WriteLine($"A instance of type {coll.GetType().Name}");
                break;
        }
    }
}

// The example displays the following output:
//     An array with 5 elements.
//     4 items
```

`object` 대신, 다음 코드와 같이 컬렉션 형식을 형식 매개 변수로 사용하여 제네릭 메서드를 만들 수 있습니다.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

class Example
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8, 10 };
        ShowCollectionInformation(values);

        var names = new List<string>();
        names.AddRange(new string[] { "Adam", "Abigail", "Bertrand", "Bridgette" });
        ShowCollectionInformation(names);

        List<int> numbers = null;
        ShowCollectionInformation(numbers);
    }

    private static void ShowCollectionInformation<T>(T coll)
    {
        switch (coll)
        {
            case Array arr:
                Console.WriteLine($"An array with {arr.Length} elements.");
                break;
            case IEnumerable<int> ieInt:
                Console.WriteLine($"Average: {ieInt.Average(s => s)}");
                break;
            case IList list:
                Console.WriteLine($"{list.Count} items");
                break;
            case IEnumerable ie:
                string result = "";
                foreach (var e in ie)
                    result += $"{e} ";
                Console.WriteLine(result);
                break;
            case object o:
                Console.WriteLine($"A instance of type {o.GetType().Name}");
                break;
            default:
                Console.WriteLine("Null passed to this method.");
                break;
        }
    }
}

// The example displays the following output:
//      An array with 5 elements.
//      4 items
//      Null passed to this method.
```

제네릭 버전은 두 가지 측면에서 첫 번째 샘플과 다릅니다. 먼저 `null` `case`를 사용할 수 없습니다. 컴파일러가 임의의 형식 `T`를 `object` 이외의 형식으로 변환할 수 없으므로 `constant case`를 사용할 수 없습니다. `default case`였던 항목이 이제 `Null`이 아닌 `object`에 대해 테스트됩니다. 즉, `default case`는 `null`에 대해서만 테스트 됩니다.

패턴 일치를 사용하지 않을 경우 이 코드를 다음과 같이 작성할 수 있습니다. 형식 패턴 일치를 사용하면 변환 결과가 `null`인지 여부를 테스트하거나 반복된 캐스트를 수행할 필요가 없으므로 더욱 단순하고 읽기 쉬운 코드가 생성됩니다.

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

class Example
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8, 10 };
        ShowCollectionInformation(values);

        var names = new List<string>();
        names.AddRange(new string[] { "Adam", "Abigail", "Bertrand", "Bridgette" });
        ShowCollectionInformation(names);

        List<int> numbers = null;
        ShowCollectionInformation(numbers);
    }

    private static void ShowCollectionInformation(object coll)
    {
        if (coll is Array)
        {
            Array arr = (Array) coll;
            Console.WriteLine($"An array with {arr.Length} elements.");
        }
        else if (coll is IEnumerable<int>)
        {
            IEnumerable<int> ieInt = (IEnumerable<int>) coll;
            Console.WriteLine($"Average: {ieInt.Average(s => s)}");
        }
        else if (coll is IList)
        {
            IList list = (IList) coll;
            Console.WriteLine($"{list.Count} items");
        }
        else if (coll is IEnumerable)
        {
            IEnumerable ie = (IEnumerable) coll;
            string result = "";
            foreach (var e in ie)
                result += $"{e} ";
            Console.WriteLine(result);
        }
        else if (coll == null)
        {
            // Do nothing.
        }
        else
        {
            Console.WriteLine($"An instance of type {coll.GetType().Name}");
        }
    }
}

// The example displays the following output:
//     An array with 5 elements.
//     4 items

```

## case 문과 when 절

C# 7.0부터 case 문이 상호 배타적일 필요가 없으므로 when 절을 추가하여 case 문이 true로 평가되기 위해 총 족해야 하는 추가 조건을 지정할 수 있습니다. when 절은 부울 값을 반환하는 모든 식일 수 있습니다.

다음 예제에서는 기본 `Shape` 클래스, `Shape`에서 파생된 `Rectangle` 클래스 및 `Rectangle`에서 파생된 `Square` 클래스를 정의합니다. `when` 절을 사용하여 `ShowShapeInfo`에서 `Square` 개체로 인스턴스화되지 않은 경우에도 동일한 길이 및 너비가 할당된 `Rectangle` 개체를 `Square`로 처리하도록 합니다. 이 메서드는 `null`인 개체나 면적이 0인 도형에 대한 정보를 표시하지 않습니다.

```
using System;

public abstract class Shape
{
    public abstract double Area { get; }
    public abstract double Circumference { get; }
}

public class Rectangle : Shape
{
    public Rectangle(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public double Length { get; set; }
    public double Width { get; set; }

    public override double Area
    {
        get { return Math.Round(Length * Width,2); }
    }

    public override double Circumference
    {
        get { return (Length + Width) * 2; }
    }
}

public class Square : Rectangle
{
    public Square(double side) : base(side, side)
    {
        Side = side;
    }

    public double Side { get; set; }
}

public class Circle : Shape
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Radius { get; set; }

    public override double Circumference
    {
        get { return 2 * Math.PI * Radius; }
    }

    public override double Area
    {
        get { return Math.PI * Math.Pow(Radius, 2); }
    }
}

public class Example
{
```

```

    public static void Main()
    {
        Shape sh = null;
        Shape[] shapes = { new Square(10), new Rectangle(5, 7),
                           sh, new Square(0), new Rectangle(8, 8),
                           new Circle(3) };
        foreach (var shape in shapes)
            ShowShapeInfo(shape);
    }

    private static void ShowShapeInfo(Shape sh)
    {
        switch (sh)
        {
            // Note that this code never evaluates to true.
            case Shape shape when shape == null:
                Console.WriteLine($"An uninitialized shape (shape == null)");
                break;
            case null:
                Console.WriteLine($"An uninitialized shape");
                break;
            case Shape shape when sh.Area == 0:
                Console.WriteLine($"The shape: {sh.GetType().Name} with no dimensions");
                break;
            case Square sq when sh.Area > 0:
                Console.WriteLine("Information about square:");
                Console.WriteLine($"    Length of a side: {sq.Side}");
                Console.WriteLine($"    Area: {sq.Area}");
                break;
            case Rectangle r when r.Length == r.Width && r.Area > 0:
                Console.WriteLine("Information about square rectangle:");
                Console.WriteLine($"    Length of a side: {r.Length}");
                Console.WriteLine($"    Area: {r.Area}");
                break;
            case Rectangle r when sh.Area > 0:
                Console.WriteLine("Information about rectangle:");
                Console.WriteLine($"    Dimensions: {r.Length} x {r.Width}");
                Console.WriteLine($"    Area: {r.Area}");
                break;
            case Shape shape when sh != null:
                Console.WriteLine($"A {sh.GetType().Name} shape");
                break;
            default:
                Console.WriteLine($"The {nameof(sh)} variable does not represent a Shape.");
                break;
        }
    }
}

// The example displays the following output:
//      Information about square:
//          Length of a side: 10
//          Area: 100
//      Information about rectangle:
//          Dimensions: 5 x 7
//          Area: 35
//      An uninitialized shape
//      The shape: Square with no dimensions
//      Information about square rectangle:
//          Length of a side: 8
//          Area: 64
//      A Circle shape

```

`Shape` 개체가 `null`인지 여부를 테스트하는 예제의 `when` 절은 실행되지 않습니다. `null`인지 테스트하는 올바른 형식 패턴은 `case null:`입니다.

# C# 언어 사양

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

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [if-else](#)
- [패턴 일치](#)

# do(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

**do** 문은 지정된 부울 식이 `true`로 계산되는 동안 문 또는 문 블록을 실행합니다. 이 식은 각 루프 실행 후 평가되기 때문에 `do-while` 루프가 한 번 이상 실행됩니다. 이는 0번 이상 실행되는 `while` 루프와 다릅니다.

**do** 문 블록 내의 어느 지점에서나 `break` 문을 사용하여 루프를 중단할 수 있습니다.

`continue` 문을 사용하여 `while` 식의 계산을 직접 실행할 수 있습니다. 식이 `true`로 계산될 경우 루프의 첫 번째 문에서 계속 실행됩니다. 그렇지 않으면 실행은 루프 뒤에 나오는 첫 번째 문에서 계속됩니다.

`goto`, `return` 또는 `throw` 문으로 `do-while` 루프를 종료할 수도 있습니다.

## 예제

다음 예제에서는 **do** 문의 사용법을 보여 줍니다. **Run**을 선택하여 예제 코드를 실행합니다. 그런 다음, 코드를 수정하고 다시 실행할 수 있습니다.

```
int n = 0;
do
{
    Console.WriteLine(n);
    n++;
} while (n < 5);
```

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 `do` 문 섹션을 참조하세요.

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [while 문](#)

# for(C# 참조)

2020-11-02 • 6 minutes to read • [Edit Online](#)

`for` 문은 지정된 부울 식이 `true`로 계산되는 동안 문 또는 문 블록을 실행합니다.

`for` 문 블록 내 원하는 지점에서 `break` 문을 사용하여 루프를 중단하거나 `continue` 문을 사용하여 루프의 다음 반복을 한 단계 실행할 수 있습니다. `goto`, `return` 또는 `throw` 문으로 `for` 루프를 종료할 수도 있습니다.

## for 문의 구조

`for` 문은 *initializer*, *condition* 및 *iterator* 섹션을 정의합니다.

```
for (initializer; condition; iterator)
    body
```

세 개의 섹션은 모두 선택 사항입니다. 루프의 본문은 명령문 또는 명령문 블록입니다.

다음 예제에서는 모든 섹션이 정의된 `for` 문을 보여 줍니다.

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

### initializer 섹션

*initializer* 섹션의 명령문은 루프로 유입되기 전에 한 번만 실행됩니다. *initializer* 섹션은 다음 중 하나입니다.

- 루프 외부에서 액세스할 수 없는 로컬 루프 변수의 선언 및 초기화
- 쉼표로 구분된 다음 목록의 0개 이상의 명령문 식:
  - `assignment` 문
  - 메서드 호출
  - 접두사 또는 후위 `increment` 식(예: `++i` 또는 `i++`)
  - 접두사 또는 후위 `decrement` 식(예: `--i` 또는 `i--`)
  - `new` 연산자를 사용하여 개체 만들기
  - `await` 식

위의 예제에서 *initializer* 섹션은 로컬 루프 변수 `i`를 선언하고 초기화합니다.

```
int i = 0
```

### condition 섹션

*condition* 섹션이 있으면 부울 식이어야 합니다. 이 식은 모든 루프 반복 전에 평가됩니다. *condition* 섹션이 없거나 부울 식이 `true`로 평가되는 경우, 다음 루프 반복이 실행되고 그렇지 않으면 루프가 종료됩니다.

위의 예제에서 *condition* 섹션은 로컬 루프 변수의 값에 따라 루프가 종료되는지 여부를 결정합니다.

```
i < 5
```

### iterator 섹션

*iterator* 섹션은 루프의 본문을 반복할 때마다 수행되는 작업을 정의합니다. *iterator* 섹션에는 쉼표로 구분된 다음 명령문 식이 0개 이상 포함됩니다.

- [assignment](#) 문
- 메서드 호출
- 접두사 또는 후위 [increment](#) 식(예: `++i` 또는 `i++`)
- 접두사 또는 후위 [decrement](#) 식(예: `--i` 또는 `i--`)
- [new](#) 연산자를 사용하여 개체 만들기
- [await](#) 식

위 예제의 *iterator* 섹션은 로컬 루프 변수를 증가시킵니다.

```
i++
```

### 예

다음 예제에서는 *initializer* 섹션에서 외부 루프 변수에 값 할당, *initializer* 및 *iterator* 섹션에서 메서드 호출, *iterator* 섹션에서 두 변수의 값 변경과 같이 `for` 문 섹션의 여러 가지 덜 일반적인 사용법을 보여 줍니다.

[Run](#)을 선택하여 예제 코드를 실행합니다. 그런 다음, 코드를 수정하고 다시 실행할 수 있습니다.

```
int i;
int j = 10;
for (i = 0, Console.WriteLine($"Start: i={i}, j={j}"); i < j; i++, j--, Console.WriteLine($"Step: i={i}, j={j}"))
{
    // Body of the loop.
}
```

다음 예제에서는 무한 `for` 루프를 정의합니다.

```
for ( ; ; )
{
    // Body of the loop.
}
```

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 [for 문](#) 섹션을 참조하세요.

### 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [foreach, in](#)

# foreach, in(C# 참조)

2020-11-02 • 8 minutes to read • [Edit Online](#)

`foreach` 문은 다음 예제와 같이 `System.Collections.IEnumerable` 또는 `System.Collections.Generic.IEnumerable<T>` 인터페이스를 구현하는 형식의 인스턴스에 있는 각 요소에 대해 문 또는 문 블록을 실행합니다.

```
var fibNumbers = new List<int> { 0, 1, 1, 2, 3, 5, 8, 13 };
int count = 0;
foreach (int element in fibNumbers)
{
    Console.WriteLine($"Element #{count}: {element}");
    count++;
}
Console.WriteLine($"Number of elements: {count}");
```

`foreach` 문은 이러한 형식으로 제한되지 않습니다. 다음 조건을 충족하는 모든 형식의 인스턴스와 함께 사용할 수 있습니다.

- 형식에는 반환 형식이 클래스, 구조체 또는 인터페이스 유형인 `public` 매개 변수가 없는 `GetEnumerator` 메서드가 포함됩니다. C# 9.0부터는 `GetEnumerator` 메서드가 형식의 **확장 메서드**일 수 있습니다.
- `GetEnumerator` 메서드의 반환 형식이 `public Current` 속성과 반환 형식이 `Boolean`인 `public` 매개 변수가 없는 `MoveNext` 메서드를 포함합니다.

다음 예제에서는 인터페이스를 구현하지 않는 `System.Span<T>` 형식의 인스턴스와 함께 `foreach` 문을 사용합니다.

```
public class IterateSpanExample
{
    public static void Main()
    {
        Span<int> numbers = new int[] { 3, 14, 15, 92, 6 };
        foreach (int number in numbers)
        {
            Console.Write($"{number} ");
        }
        Console.WriteLine();
    }
}
```

C# 7.3부터는 열거자의 `Current` 속성이 **참조 반환 값**(`ref T` 여기서 `T`는 컬렉션 요소의 형식임)을 반환하는 경우 다음 예제와 같이 `ref` 또는 `ref readonly` 한정자를 사용하여 반복 변수를 선언할 수 있습니다.

```

public class ForeachRefExample
{
    public static void Main()
    {
        Span<int> storage = stackalloc int[10];
        int num = 0;
        foreach (ref int item in storage)
        {
            item = num++;
        }

        foreach (ref readonly var item in storage)
        {
            Console.WriteLine($"{item} ");
        }
        // Output:
        // 0 1 2 3 4 5 6 7 8 9
    }
}

```

C# 8.0부터는 `await foreach` 문을 사용하여 비동기 데이터 스트림 즉, `IAsyncEnumerable<T>` 인터페이스를 구현하는 컬렉션 형식을 사용할 수 있습니다. 루프의 각 반복은 다음 요소가 비동기적으로 검색되는 동안 일시 중단될 수도 있습니다. 다음 예제에서는 `await foreach` 문을 사용하는 방법을 보여줍니다.

```

await foreach (var item in GenerateSequenceAsync())
{
    Console.WriteLine(item);
}

```

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](#)을 참조하세요. 비동기 스트림에 대한 자세한 내용은 [C# 8.0의 새로운 기능](#) 문서의 [비동기 스트림](#) 섹션을 참조하세요.

`foreach` 문 블록 내 원하는 지점에서 `break` 문을 사용하여 루프를 중단하거나 `continue` 문을 사용하여 루프의 다음 반복을 한 단계 실행할 수 있습니다. `goto, return` 또는 `throw` 문으로 `foreach` 루프를 종료할 수도 있습니다.

`foreach` 문이 `null`에 적용되면 `NullReferenceException`이 `throw`됩니다. `foreach` 문의 소스 컬렉션이 비어 있으면 `foreach` 루프의 본문이 실행되지 않고 건너뜁니다.

## 반복 변수의 형식

다음 코드와 같이 `var` 키워드를 사용하여 컴파일러가 `foreach` 문에서 반복 변수의 형식을 유추할 수 있습니다.

```

foreach (var item in collection) { }

```

다음 코드와 같이 반복 변수의 형식을 명시적으로 지정할 수도 있습니다.

```

IEnumerable<T> collection = new T[5];
foreach (V item in collection) { }

```

위의 양식에서 컬렉션 요소의 `T` 형식은 암시적 또는 명시적으로 반복 변수의 `V` 형식으로 변환할 수 있어야 합니다. `T`에서 `V`로의 명시적 변환이 런타임에 실패하는 경우 `foreach` 문은 `InvalidOperationException`를 `throw`합니다. 예를 들어 `T`가 봉인되지 않은 클래스 형식인 경우 `V`는 `T`가 구현하지 않는 형식을 포함하여 모든 인터페

이스 형식일 수 있습니다. 런타임에 컬렉션 요소의 형식은 `T`에서 파생되어 실제로 `V`를 구현하는 형식일 수 있습니다. 그렇지 않은 경우에는 `InvalidCastException`을 throw합니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 `foreach` 문 섹션을 참조하세요.

C# 8.0 이상에 추가된 기능에 대한 자세한 내용은 다음 기능 제안 노트를 참조하세요.

- [비동기 스트림\(C# 8.0\)](#)
- [foreach 루프에 대한 확장 GetEnumerator 지원\(C# 9.0\)](#)

## 참고 항목

- [C# 참조](#)
- [C# 키워드](#)
- [배열에 foreach 사용](#)
- [for 문](#)

# while(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

`while` 문은 지정된 부울 식이 `true`로 계산되는 동안 문 또는 문 블록을 실행합니다. 이 식은 각 루프를 실행하기 전에 평가되기 때문에 `while` 루프는 0번 이상 실행됩니다. 이는 한 번 이상 실행되는 `do` 루프와 다릅니다.

`while` 문 블록 내의 어느 지점에서나 `break` 문을 사용하여 루프를 중단할 수 있습니다.

`continue` 문을 사용하여 `while` 식의 계산을 직접 실행할 수 있습니다. 식이 `true`로 계산될 경우 루프의 첫 번째 문에서 계속 실행됩니다. 그렇지 않으면 실행은 루프 뒤에 나오는 첫 번째 문에서 계속됩니다.

`goto`, `return` 또는 `throw` 문으로 `while` 루프를 종료할 수도 있습니다.

## 예제

다음 예제에서는 `while` 문의 사용법을 보여 줍니다. **Run**을 선택하여 예제 코드를 실행합니다. 그런 다음, 코드를 수정하고 다시 실행할 수 있습니다.

```
int n = 0;
while (n < 5)
{
    Console.WriteLine(n);
    n++;
}
```

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 `while` 문 섹션을 참조하세요.

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [do 문](#)

# break(C# 참조)

2020-11-02 • 4 minutes to read • [Edit Online](#)

`break` 문은 배치된 지점에서 가장 가까운 바깥쪽 루프 또는 `switch` 문을 종료합니다. 제어는 종료된 문 뒤의 문으로 전달됩니다(있는 경우).

## 예제

이 예제의 조건문에는 1부터 100까지 개수를 계산하는 카운터가 포함되지만 `break` 문은 4회 계산 후에 루프를 종료합니다.

```
class BreakTest
{
    static void Main()
    {
        for (int i = 1; i <= 100; i++)
        {
            if (i == 5)
            {
                break;
            }
            Console.WriteLine(i);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Output:
1
2
3
4
*/
```

## 예제

이 예제에서는 `switch` 문에서 `break`의 사용을 보여 줍니다.

```

class Switch
{
    static void Main()
    {
        Console.WriteLine("Enter your selection (1, 2, or 3): ");
        string s = Console.ReadLine();
        int n = Int32.Parse(s);

        switch (n)
        {
            case 1:
                Console.WriteLine("Current value is 1");
                break;
            case 2:
                Console.WriteLine("Current value is 2");
                break;
            case 3:
                Console.WriteLine("Current value is 3");
                break;
            default:
                Console.WriteLine("Sorry, invalid selection.");
                break;
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Sample Input: 1

Sample Output:
Enter your selection (1, 2, or 3): 1
Current value is 1
*/

```

4 를 입력하면 다음과 같이 출력됩니다.

```

Enter your selection (1, 2, or 3): 4
Sorry, invalid selection.

```

## 예제

이 예제에서 `break` 문은 내부 중첩 루프를 종단하고 제어를 외부 루프에 반환하는 데 사용됩니다. 컨트롤은 중첩된 루프에서 오직 반환된 한 레벨 업입니다.

```

class BreakInNestedLoops
{
    static void Main(string[] args)
    {

        int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        char[] letters = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j' };

        // Outer loop.
        for (int i = 0; i < numbers.Length; i++)
        {
            Console.WriteLine($"num = {numbers[i]}");

            // Inner loop.
            for (int j = 0; j < letters.Length; j++)
            {
                if (j == i)
                {
                    // Return control to outer loop.
                    break;
                }
                Console.Write($"{letters[j]} ");
            }
            Console.WriteLine();
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
 * Output:
 num = 0

 num = 1
 a
 num = 2
 a b
 num = 3
 a b c
 num = 4
 a b c d
 num = 5
 a b c d e
 num = 6
 a b c d e f
 num = 7
 a b c d e f g
 num = 8
 a b c d e f g h
 num = 9
 a b c d e f g h i
*/

```

## 예제

이 예제에서 `break` 문은 루프의 각 반복 중에 현재 분기를 중단하는 데만 사용됩니다. 루프 자체는 종점된 `switch` 문에 속하는 `break` 의 인스턴스에 영향을 받지 않습니다.

```

class BreakFromSwitchInsideLoop
{
    static void Main(string[] args)
    {
        // loop 1 to 3
        for (int i = 1; i <= 3; i++)
        {
            switch(i)
            {
                case 1:
                    Console.WriteLine("Current value is 1");
                    break;
                case 2:
                    Console.WriteLine("Current value is 2");
                    break;
                case 3:
                    Console.WriteLine("Current value is 3");
                    break;
                default:
                    Console.WriteLine("This shouldn't happen.");
                    break;
            }
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
 * Output:
 * Current value is 1
 * Current value is 2
 * Current value is 3
 */

```

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [switch](#)

# continue(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

`continue` 문은 자신이 나타나는 바깥쪽 `while`, `do`, `for` 또는 `foreach` 문의 다음 반복으로 제어를 전달합니다.

## 예제

이 예제에서 카운터는 1에서 10까지 계산하도록 초기화됩니다. `continue` 문을 `(i < 9)` 식과 함께 사용하면 `i`가 9보다 작은 반복에서 `continue` 와 `for` 본문 끝 사이에 있는 문을 건너뜁니다. `for` 루프의 마지막 두 반복 (`i == 9` 및 `i == 10`)에서는 `continue` 문이 실행되지 않고 `i` 값이 콘솔에 출력됩니다.

```
class ContinueTest
{
    static void Main()
    {
        for (int i = 1; i <= 10; i++)
        {
            if (i < 9)
            {
                continue;
            }
            Console.WriteLine(i);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Output:
9
10
*/
```

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [break 문](#)

# goto(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

`goto` 문은 레이블 지정된 문으로 직접 프로그램 컨트롤을 전송합니다.

`goto`는 일반적으로 `switch` 문에서 switch-case 레이블 또는 기본 레이블로 컨트롤을 전송하는 데 사용됩니다.

`goto` 문은 많이 중첩된 루프를 회피하는 데도 유용합니다.

## 예제

다음 예제에서는 `switch` 문에서 `goto`의 사용 방법을 보여 줍니다.

```
class SwitchTest
{
    static void Main()
    {
        Console.WriteLine("Coffee sizes: 1=Small 2=Medium 3=Large");
        Console.Write("Please enter your selection: ");
        string s = Console.ReadLine();
        int n = int.Parse(s);
        int cost = 0;
        switch (n)
        {
            case 1:
                cost += 25;
                break;
            case 2:
                cost += 25;
                goto case 1;
            case 3:
                cost += 50;
                goto case 1;
            default:
                Console.WriteLine("Invalid selection.");
                break;
        }
        if (cost != 0)
        {
            Console.WriteLine($"Please insert {cost} cents.");
        }
        Console.WriteLine("Thank you for your business.");

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

Sample Input: 2

Sample Output:

```
Coffee sizes: 1=Small 2=Medium 3=Large
Please enter your selection: 2
Please insert 50 cents.
Thank you for your business.
```

## 예제

다음 예제에서는 `goto` 를 사용하여 중첩된 루프에서 벗어나는 방법을 보여 줍니다.

```
public class GotoTest1
{
    static void Main()
    {
        int x = 200, y = 4;
        int count = 0;
        string[,] array = new string[x, y];

        // Initialize the array.
        for (int i = 0; i < x; i++)
            for (int j = 0; j < y; j++)
                array[i, j] = (++count).ToString();

        // Read input.
        Console.Write("Enter the number to search for: ");

        // Input a string.
        string myNumber = Console.ReadLine();

        // Search.
        for (int i = 0; i < x; i++)
        {
            for (int j = 0; j < y; j++)
            {
                if (array[i, j].Equals(myNumber))
                {
                    goto Found;
                }
            }
        }

        Console.WriteLine($"The number {myNumber} was not found.");
        goto Finish;

    Found:
        Console.WriteLine($"The number {myNumber} is found.");

    Finish:
        Console.WriteLine("End of search.");

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
Sample Input: 44

Sample Output
Enter the number to search for: 44
The number 44 is found.
End of search.
*/
```

## C# 언어 사양

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

## 참조

- C# 참조
- C# 프로그래밍 가이드
- C# 키워드
- goto 문(C++)

# return(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

`return` 문은 해당 문이 나타나는 메서드의 실행을 종료하고 제어를 호출 메서드로 반환합니다. 선택적 값을 반환할 수도 있습니다. 메서드가 `void` 형식인 경우 `return` 문을 생략할 수 있습니다.

`return` 문이 `try` 블록 안에 있으면 `finally` 블록(있는 경우)은 제어가 호출 메서드로 반환되기 전에 실행됩니다.

## 예제

다음 예제에서 `CalculateArea()` 메서드는 `area` 로컬 변수를 `double` 값으로 반환합니다.

```
class ReturnTest
{
    static double CalculateArea(int r)
    {
        double area = r * r * Math.PI;
        return area;
    }

    static void Main()
    {
        int radius = 5;
        double result = CalculateArea(radius);
        Console.WriteLine("The area is {0:0.00}", result);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: The area is 78.54
```

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [return 문](#)

# throw(C# 참조)

2020-11-02 • 6 minutes to read • [Edit Online](#)

프로그램 실행 중 예외 발생 신호를 보냅니다.

## 설명

`throw` 의 구문은 다음과 같습니다.

```
throw [e];
```

여기서 `e`는 `System.Exception`에서 파생된 클래스의 인스턴스입니다. 다음 예제에서는 `throw` 문을 사용하여 `GetNumber`라는 메서드에 전달된 인수가 내부 배열의 유효한 인덱스에 해당하지 않는 경우 `IndexOutOfRangeException`을 `throw`합니다.

```
using System;

public class NumberGenerator
{
    int[] numbers = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

    public int GetNumber(int index)
    {
        if (index < 0 || index >= numbers.Length) {
            throw new IndexOutOfRangeException();
        }
        return numbers[index];
    }
}
```

그러면 메서드 호출자가 `try-catch` 또는 `try-catch-finally` 블록을 사용하여 `throw`된 예외를 처리합니다. 다음 예제에서는 `GetNumber` 메서드에 의해 `throw`된 예외를 처리합니다.

```
using System;

public class Example
{
    public static void Main()
    {
        var gen = new NumberGenerator();
        int index = 10;
        try {
            int value = gen.GetNumber(index);
            Console.WriteLine($"Retrieved {value}");
        }
        catch (IndexOutOfRangeException e)
        {
            Console.WriteLine($"{e.GetType().Name}: {index} is outside the bounds of the array");
        }
    }
}

// The example displays the following output:
//      IndexOutOfRangeException: 10 is outside the bounds of the array
```

## 예외 다시 throw

`catch` 블록에서 `throw`를 사용하여 `catch` 블록에서 처리된 예외를 다시 `throw`할 수도 있습니다. 이 경우 `throw`는 예외 피연산자를 사용하지 않습니다. 이는 메서드가 호출자의 인수를 다른 일부 라이브러리 메서드에 전달하고 라이브러리 메서드가 호출자에게 전달되어야 하는 예외를 `throw`하는 경우에 가장 유용합니다. 예를 들어 다음 예제에서는 초기화되지 않은 문자열의 첫 번째 문자를 검색하려고 할 때 `throw`되는 `NullReferenceException`을 다시 `throw`합니다.

```
using System;

public class Sentence
{
    public Sentence(string s)
    {
        Value = s;
    }

    public string Value { get; set; }

    public char GetFirstCharacter()
    {
        try {
            return Value[0];
        }
        catch (NullReferenceException e) {
            throw;
        }
    }
}

public class Example
{
    public static void Main()
    {
        var s = new Sentence(null);
        Console.WriteLine($"The first character is {s.GetFirstCharacter()}");
    }
}
// The example displays the following output:
//     Unhandled Exception: System.NullReferenceException: Object reference not set to an instance of an
object.
//         at Sentence.GetFirstCharacter()
//         at Example.Main()
```

### IMPORTANT

`catch` 블록의 `throw e` 구문을 사용하여 호출자에게 전달하는 새 예외를 인스턴스화할 수도 있습니다. 이 경우 `StackTrace` 속성에서 제공되는 원래 예외의 스택 추적이 유지되지 않습니다.

## throw 식

C# 7.0부터 `throw`를 명령문뿐만 아니라 식으로도 사용할 수 있습니다. 이렇게 하면 이전에 지원되지 않은 상황에서 예외를 `throw`할 수 있습니다. 여기에는 다음이 포함됩니다.

- **조건 연산자.** 다음 예제에서는 `throw` 식을 사용하여 메서드에 빈 문자열 배열이 전달된 경우 `ArgumentException`을 `throw`합니다. C# 7.0 이전에는 이 논리가 `if / else` 문에 표시되어야 합니다.

```

private static void DisplayFirstNumber(string[] args)
{
    string arg = args.Length >= 1 ? args[0] :
        throw new ArgumentException("You must supply an argument");
    if (Int64.TryParse(arg, out var number))
        Console.WriteLine($"You entered {number:F0}");
    else
        Console.WriteLine($"{arg} is not a number.");
}

```

- **Null 병합 연산자.** 다음 예제에서는 `throw` 식에 Null 병합 연산자를 사용하여 `Name` 속성에 할당된 문자열이 `null`인 경우 예외를 `throw`합니다.

```

public string Name
{
    get => name;
    set => name = value ??
        throw new ArgumentNullException(paramName: nameof(value), message: "Name cannot be null");
}

```

- 식 본문 `람다` 또는 메서드. 다음 예제에서는 `DateTime` 값으로 변환이 지원되지 않기 때문에 `InvalidCastException`을 `throw`하는 식 본문 메서드를 보여 줍니다.

```

DateTime ToDateTime(IFormatProvider provider) =>
    throw new InvalidCastException("Conversion to a DateTime is not supported.");

```

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [try-catch](#)
- [C# 키워드](#)
- [방법: 명시적으로 예외 Throw](#)

# try-catch(C# 참조)

2020-11-02 • 19 minutes to read • [Edit Online](#)

try-catch 문은 `try` 블록에 이어 서로 다른 예외에 대한 처리기를 지정하는 하나 이상의 `catch` 절로 구성됩니다.

예외가 `throw`되면 CLR(공용 언어 런타임)에서는 이 예외를 처리하는 `catch` 문을 검색합니다. 현재 실행 중인 메서드에 `catch` 블록이 포함되지 않으면 CLR에서는 현재 메서드를 호출한 메서드 등에서 호출 스택까지 확인합니다. `catch` 블록을 찾을 수 없으면 CLR에서는 처리되지 않은 예외 메시지를 사용자에게 표시하고 프로그램의 예외를 종지합니다.

`try` 블록에는 예외를 가져올 수 있는 보호된 코드가 포함됩니다. 블록은 예외가 `throw`되거나 성공적으로 완료될 때까지 실행됩니다. 예를 들어 `null` 개체를 캐스팅하는 다음 시도에서 `NullReferenceException` 예외가 발생합니다.

```
object o2 = null;
try
{
    int i2 = (int)o2;    // Error
}
```

`catch` 절은 예외 형식을 `catch`하는 인수 없이 사용할 수 있지만 이 사용은 권장되지 않습니다. 일반적으로 복구하는 방법을 알고 있는 예외만 `catch`해야 합니다. 따라서 항상 `System.Exception`에서 파생된 개체 인수를 지정해야 합니다. 예를 들면 다음과 같습니다.

```
catch (InvalidOperationException e)
{
}
```

같은 try-catch 문에서 특정 `catch` 절을 두 개 이상 사용할 수 있습니다. 이 경우 `catch` 절은 순서대로 검사되므로 `catch` 절의 순서가 중요합니다. 더 구체적인 예외를 덜 구체적인 예외보다 먼저 `catch`합니다. 나중에 나타나는 블록에 도달할 수 없도록 `catch` 블록 순서를 지정하면 컴파일러에서 오류가 발생합니다.

`catch` 인수를 사용하여 처리할 예외를 필터링할 수 있습니다. 예외를 추가로 검사하는 예외 필터를 사용하여 예외를 처리할지 결정할 수도 있습니다. 예외 필터가 `false`를 반환하면 처리기 검색이 계속됩니다.

```
catch (ArgumentException e) when (e.ParamName == "...")
{}
```

필터 덕분에 스택이 손상되지 않으므로 예외 필터는 `catch`하고 다시 `throw`하는 것이 좋습니다(아래 내용 참조). 나중에 나타나는 처리기가 스택을 덤프하면 예외가 다시 `throw`된 마지막 위치가 아니라 예외가 원래 발생한 위치를 확인할 수 있습니다. 예외 필터 식의 일반적으로 사용법은 로깅입니다. 로그에도 출력되는 항상 `false`를 반환하는 필터를 만들 수 있고, 예외를 처리하고 다시 `throw`할 필요 없이 진행되는 대로 예외를 기록할 수 있습니다.

`throw` 문을 `catch` 블록에서 사용하여 `catch` 문에 의해 `catch`된 예외를 다시 `throw`할 수 있습니다. 다음 예제에서는 `IOException` 예외에서 소스 정보를 추출하고 예외를 부모 메서드에 `throw`합니다.

```

catch (FileNotFoundException e)
{
    // FileNotFoundExceptions are handled here.
}
catch (IOException e)
{
    // Extract some information from this exception, and then
    // throw it to the parent method.
    if (e.Source != null)
        Console.WriteLine("IOException source: {0}", e.Source);
    throw;
}

```

하나의 예외를 catch하고 다른 예외를 throw할 수 있습니다. 이 작업을 할 때 다음 예제와 같이 내부 예외로 catch한 예외를 지정합니다.

```

catch (InvalidOperationException e)
{
    // Perform some action here, and then throw a new exception.
    throw new YourCustomException("Put your error message here.", e);
}

```

다음 예제와 같이 지정된 조건이 true이면 예외를 다시 throw할 수도 있습니다.

```

catch (InvalidOperationException e)
{
    if (e.Data == null)
    {
        throw;
    }
    else
    {
        // Take some action.
    }
}

```

#### NOTE

예외 필터를 사용하여 유사한 결과를 더 깔끔하게 얻을 수도 있습니다(물론 이 문서의 앞 부분에서 설명한 것처럼 스택을 수정하지 않음). 다음 예제에서는 이전 예제와 같은 호출자에 대한 유사한 동작을 보여 줍니다. `e.Data` 가 `null` 일 때 함수는 호출자에게 `InvalidOperationException` 을 다시 throw합니다.

```

catch (InvalidOperationException e) when (e.Data != null)
{
    // Take some action.
}

```

`try` 블록 내부에서 선언된 변수만 초기화합니다. 그렇지 않으면 블록의 예외가 완료되기 전에 다른 예외가 발생할 수 있습니다. 예를 들어 다음 코드 예제에서 `n` 변수는 `try` 블록 내부에서 초기화됩니다. `Write(n)` 문의 `try` 블록 외부에서 이 변수를 사용하려고 하면 컴파일러 오류가 발생합니다.

```
static void Main()
{
    int n;
    try
    {
        // Do not initialize this variable here.
        n = 123;
    }
    catch
    {
    }
    // Error: Use of unassigned local variable 'n'.
    Console.WriteLine(n);
}
```

catch에 대한 자세한 내용은 [try-catch-finally](#)를 참조하세요.

## 비동기 메서드의 예외

비동기 메서드는 `async` 한정자를 통해 표시되고 대개 하나 이상의 `await` 식 및 문을 포함합니다. `await` 식은 `await` 연산자를 `Task` 또는 `Task<TResult>`에 적용합니다.

컨트롤이 비동기 메서드의 `await`에 도달하면 대기 중인 작업이 완료될 때까지 메서드의 진행이 일시 중단됩니다. 작업이 완료되면 메서드가 실행이 다시 시작될 수 있습니다. 자세한 내용은 [async 및 await를 사용한 비동기 프로그래밍](#)을 참조하세요.

`await` 가 적용되는 완료된 작업은 작업을 반환하는 메서드의 처리되지 않은 예외로 인해 오류 상태에 있을 수 있습니다. 작업을 기다리면 예외가 `throw`됩니다. 작업을 반환하는 비동기 프로세스가 취소되면 작업이 취소됨 상태로 종료될 수도 있습니다. 취소된 작업을 기다리면 `OperationCanceledException`이 `throw`됩니다.

예외를 `catch`하려면 `try` 블록에서 작업을 기다리고 연결된 `catch` 블록에서 예외를 `catch`합니다. 예제에 대해서는 [비동기 메서드 예제](#) 섹션을 참조하세요.

대기 중인 비동기 메서드에서 여러 예외가 발생했기 때문에 작업이 오류 상태에 있을 수 있습니다. 예를 들어 작업은 `Task.WhenAll` 호출의 결과일 수 있습니다. 작업을 기다릴 때 예외 중 하나만 `catch`되고 `catch`될 예외를 예상할 수 없습니다. 예제에 대해서는 [Task.WhenAll 예제](#) 섹션을 참조하세요.

## 예제

다음 예제에서 `try` 블록에는 예외를 가져올 수 있는 `ProcessString` 메서드에 대한 호출이 포함됩니다. `catch` 절에는 화면에 메시지만 표시하는 예외 처리기가 포함됩니다. `throw` 문이 `ProcessString` 내부에서 호출되면 시스템에서는 `catch` 문을 검색하고 메시지 `Exception caught`를 표시합니다.

```

class TryFinallyTest
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException();
        }
    }

    static void Main()
    {
        string s = null; // For demonstration purposes.

        try
        {
            ProcessString(s);
        }
        catch (Exception e)
        {
            Console.WriteLine("{0} Exception caught.", e);
        }
    }
}

/*
Output:
System.ArgumentNullException: Value cannot be null.
at TryFinallyTest.Main() Exception caught.
*/

```

## 두 개의 catch 블록 예제

다음 예제에서는 두 catch 블록이 사용되고 먼저 나오는 가장 구체적인 예외가 catch됩니다.

가장 덜 구체적인 예외를 catch하려면 `ProcessString`의 `throw` 문을 `throw new Exception()` 문으로 바꿔야 합니다.

가장 덜 구체적인 catch 블록을 예제에 첫 번째로 배치하면 다음 오류 메시지가 표시됩니다.

```
A previous catch clause already catches all exceptions of this or a super type ('System.Exception').
```

```

class ThrowTest3
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException();
        }
    }

    static void Main()
    {
        try
        {
            string s = null;
            ProcessString(s);
        }
        // Most specific:
        catch (ArgumentNullException e)
        {
            Console.WriteLine("{0} First exception caught.", e);
        }
        // Least specific:
        catch (Exception e)
        {
            Console.WriteLine("{0} Second exception caught.", e);
        }
    }
}
/*
Output:
System.ArgumentNullException: Value cannot be null.
at Test.ThrowTest3.ProcessString(String s) ... First exception caught.
*/

```

## 비동기 메서드 예제

다음 예제에서는 비동기 메서드에 대한 예외 처리를 보여 줍니다. 비동기 작업에서 `throw`하는 예외를 `catch`하려면 `try` 블록에 `await` 식을 배치하고 `catch` 블록에서 예외를 `catch`합니다.

예제에서 `throw new Exception` 줄의 주석 처리를 제거하여 예외 처리를 보여 줍니다. 작업의 `IsFaulted` 속성이 `True`로 설정되고, 작업의 `Exception.InnerException` 속성이 예외로 설정되고, 예외가 `catch` 블록에서 `catch`됩니다.

`throw new OperationCanceledException` 줄의 주석 처리를 제거하여 비동기 프로세스를 취소할 수 있을 때 발생하는 동작을 보여 줍니다. 작업의 `IsCanceled` 속성이 `true`로 설정되고, 예외가 `catch` 블록에서 `catch`됩니다. 이 예제에 적용되지 않는 몇몇 조건에서는 작업의 `IsFaulted` 속성이 `true`로 설정되고 `IsCanceled`가 `false`로 설정됩니다.

```

public async Task DoSomethingAsync()
{
    Task<string> theTask = DelayAsync();

    try
    {
        string result = await theTask;
        Debug.WriteLine("Result: " + result);
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception Message: " + ex.Message);
    }
    Debug.WriteLine("Task IsCanceled: " + theTask.IsCanceled);
    Debug.WriteLine("Task IsFaulted: " + theTask.IsFaulted);
    if (theTask.Exception != null)
    {
        Debug.WriteLine("Task Exception Message: "
            + theTask.Exception.Message);
        Debug.WriteLine("Task Inner Exception Message: "
            + theTask.Exception.InnerException.Message);
    }
}

private async Task<string> DelayAsync()
{
    await Task.Delay(100);

    // Uncomment each of the following lines to
    // demonstrate exception handling.

    //throw new OperationCanceledException("canceled");
    //throw new Exception("Something happened.");
    return "Done";
}

// Output when no exception is thrown in the awaited method:
//   Result: Done
//   Task IsCanceled: False
//   Task IsFaulted: False

// Output when an Exception is thrown in the awaited method:
//   Exception Message: Something happened.
//   Task IsCanceled: False
//   Task IsFaulted: True
//   Task Exception Message: One or more errors occurred.
//   Task Inner Exception Message: Something happened.

// Output when a OperationCanceledException or TaskCanceledException
// is thrown in the awaited method:
//   Exception Message: canceled
//   Task IsCanceled: True
//   Task IsFaulted: False

```

## Task.WhenAll 예제

다음 예제에서는 여러 작업에서 여러 예외가 발생할 수 있는 경우 예외 처리를 보여 줍니다. `try` 블록은 `Task.WhenAll`에 대한 호출에서 반환된 작업을 기다립니다. `WhenAll`이 적용된 작업 세 개가 완료되면 작업이 완료됩니다.

세 작업에서 각각 예외가 발생합니다. `catch` 블록은 `Task.WhenAll`에서 반환된 작업의 `Exception.InnerExceptions` 속성에 있는 예외를 반복합니다.

```

public async Task DoMultipleAsync()
{
    Task theTask1 = ExcAsync(info: "First Task");
    Task theTask2 = ExcAsync(info: "Second Task");
    Task theTask3 = ExcAsync(info: "Third Task");

    Task allTasks = Task.WhenAll(theTask1, theTask2, theTask3);

    try
    {
        await allTasks;
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception: " + ex.Message);
        Debug.WriteLine("Task IsFaulted: " + allTasks.IsFaulted);
        foreach (var inEx in allTasks.Exception.InnerException)
        {
            Debug.WriteLine("Task Inner Exception: " + inEx.Message);
        }
    }
}

private async Task ExcAsync(string info)
{
    await Task.Delay(100);

    throw new Exception("Error-" + info);
}

// Output:
//   Exception: Error-First Task
//   Task IsFaulted: True
//   Task Inner Exception: Error-First Task
//   Task Inner Exception: Error-Second Task
//   Task Inner Exception: Error-Third Task

```

## C# 언어 사양

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

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [try, throw 및 catch 문\(C++\)](#)
- [throw](#)
- [try-finally](#)
- [방법: 명시적으로 예외 Throw](#)

# try-finally(C# 참조)

2020-11-02 • 6 minutes to read • [Edit Online](#)

`finally` 블록을 사용하면 `try` 블록에서 할당된 리소스를 정리할 수 있으며, `try` 블록에서 예외가 발생하는 경우에도 코드를 실행할 수 있습니다. 일반적으로 `finally` 블록의 문은 제어가 `try` 문을 벗어날 때 실행됩니다. 제어 전송은 정상적인 실행 결과, `break`, `continue`, `goto` 또는 `return` 문의 실행 결과 또는 `try` 문에서 예외 전파의 결과로 발생할 수 있습니다.

처리된 예외 내에서는 연결된 `finally` 블록이 항상 실행됩니다. 그러나 예외가 처리되지 않은 경우 `finally` 블록의 실행 여부는 예외 해제 작업의 트리거 방법에 따라 달라집니다. 트리거 방법은 다시 사용자 컴퓨터의 설정 방법에 따라 달라집니다.

일반적으로 처리되지 않은 예외로 애플리케이션이 종료되는 경우 `finally` 블록의 실행 여부는 중요하지 않습니다. 그러나 해당 상황에서도 실행해야 하는 문이 `finally` 블록에 있는 경우 한 가지 솔루션은 `catch` 블록을 `try - finally` 문에 추가하는 것입니다. 또는 호출 스택에서 상위 `try - finally` 문의 `try` 블록에 `throw`될 수 있는 예외를 `catch`할 수 있습니다. 즉, `try - finally` 문을 포함하는 메서드를 호출하는 메서드, 해당 메서드를 호출하는 메서드 또는 호출 스택의 임의 메서드에 예외를 `catch`할 수 있습니다. 예외가 `catch`되지 않는 경우 `finally`의 실행은 운영 체제에서 예외 해제 작업 트리거를 선택하는지 여부에 따라 달라집니다.

## 예제

다음 예제에서는 잘못된 변환 문으로 인해 `System.InvalidCastException` 예외가 발생합니다. 예외가 처리되지 않습니다.

```
public class ThrowTestA
{
    static void Main()
    {
        int i = 123;
        string s = "Some string";
        object obj = s;

        try
        {
            // Invalid conversion; obj contains a string, not a numeric type.
            i = (int)obj;

            // The following statement is not run.
            Console.WriteLine("WriteLine at the end of the try block.");
        }
        finally
        {
            // To run the program in Visual Studio, type CTRL+F5. Then
            // click Cancel in the error dialog.
            Console.WriteLine("\nExecution of the finally block after an unhandled\n" +
                "error depends on how the exception unwind operation is triggered.");
            Console.WriteLine("i = {0}", i);
        }
    }
    // Output:
    // Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
    //
    // Execution of the finally block after an unhandled
    // error depends on how the exception unwind operation is triggered.
    // i = 123
}
```

다음 예제에서는 TryCast 메서드의 예외가 호출 스택의 위에 있는 메서드에서 catch됩니다.

```
public class ThrowTestB
{
    static void Main()
    {
        try
        {
            // TryCast produces an unhandled exception.
            TryCast();
        }
        catch (Exception ex)
        {
            // Catch the exception that is unhandled in TryCast.
            Console.WriteLine
                ("Catching the {0} exception triggers the finally block.",
                ex.GetType());

            // Restore the original unhandled exception. You might not
            // know what exception to expect, or how to handle it, so pass
            // it on.
            throw;
        }
    }

    public static void TryCast()
    {
        int i = 123;
        string s = "Some string";
        object obj = s;

        try
        {
            // Invalid conversion; obj contains a string, not a numeric type.
            i = (int)obj;

            // The following statement is not run.
            Console.WriteLine("WriteLine at the end of the try block.");
        }
        finally
        {
            // Report that the finally block is run, and show that the value of
            // i has not been changed.
            Console.WriteLine("\nIn the finally block in TryCast, i = {0}.\n", i);
        }
    }
    // Output:
    // In the finally block in TryCast, i = 123.

    // Catching the System.InvalidCastException exception triggers the finally block.

    // Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
}
```

finally에 대한 자세한 내용은 [try-catch-finally](#)를 참조하세요.

C#에는 IDisposable 개체에 대한 유사한 기능을 편리한 구문으로 제공하는 [using 문](#)도 포함되어 있습니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 [try 문](#) 섹션을 참조하세요.

## 참고 항목

- C# 참조
- C# 프로그래밍 가이드
- C# 키워드
- try, throw 및 catch 문(C++)
- throw
- try-catch
- 방법: 명시적으로 예외 Throw

# try-catch-finally(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

`catch` 및 `finally` 를 함께 사용하는 일반적인 경우는 `try` 블록에서 리소스를 얻어 사용하고, `catch` 블록에서 예외 상황을 처리하고, `finally` 블록에서 리소스를 해제하는 것입니다.

예외를 다시 `throw`하는 방법에 대한 자세한 내용과 예제는 [try-catch](#) 및 [예외 throw](#)를 참조하세요. `finally` 블록에 대한 자세한 내용은 [try-finally](#)를 참조하세요.

## 예제

```
public class EHClass
{
    void ReadFile(int index)
    {
        // To run this code, substitute a valid path from your local machine
        string path = @"c:\users\public\test.txt";
        System.IO.StreamReader file = new System.IO.StreamReader(path);
        char[] buffer = new char[10];
        try
        {
            file.ReadBlock(buffer, index, buffer.Length);
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine("Error reading from {0}. Message = {1}", path, e.Message);
        }

        finally
        {
            if (file != null)
            {
                file.Close();
            }
        }
        // Do something with buffer...
    }
}
```

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 `try` 문 섹션을 참조하세요.

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [try, throw 및 catch 문\(C++\)](#)
- [throw](#)
- [방법: 명시적으로 예외 Throw](#)
- [using 문](#)

# Checked 및 Unchecked(C# 참조)

2020-11-02 • 3 minutes to read • [Edit Online](#)

checked 컨텍스트 또는 unchecked 컨텍스트에서 C# 문을 실행할 수 있습니다. checked 컨텍스트에서는 산술 오버플로가 있으면 예외가 발생합니다. unchecked 컨텍스트에서는 산술 오버플로가 무시되고 대상 형식에 맞지 않는 상위 비트가 삭제되어 해당 결과가 잘립니다.

- `checked` checked 컨텍스트를 지정합니다.
- `unchecked` unchecked 컨텍스트를 지정합니다.

오버플로 검사의 영향을 받는 작업은 다음과 같습니다.

- 정수 계열 형식에 다음의 미리 정의된 연산자를 사용하는 식

`++`, `--`, `-`(단항), `+`, `-`, `*`, `/`

- 정수 형식 간이나 `float` 또는 `double`에서 정수 형식으로의 명시적 숫자 변환

`checked`도 `unchecked`도 지정하지 않으면 상수가 아닌 식(런타임에 계산되는 식)의 기본 컨텍스트는 `-checked` 컴파일러 옵션의 값으로 정의됩니다. 기본적으로 이 옵션의 값은 설정되지 않으며 `unchecked` 컨텍스트에서 산술 연산이 실행됩니다.

상수 식(컴파일 시간에 완전히 계산될 수 있는 식)의 경우 기본 컨텍스트는 항상 `checked`입니다. 상수 식이 `unchecked` 컨텍스트에 명시적으로 배치되지 않는 경우 식에 대한 컴파일 시간 계산 중 발생하는 오버플로로 인해 컴파일 시간 오류가 발생합니다.

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [문 키워드](#)

# checked(C# 참조)

2020-11-02 • 5 minutes to read • [Edit Online](#)

`checked` 키워드는 정수 형식 산술 연산 및 변환에 대한 오버플로 검사를 명시적으로 사용하도록 설정하는 데 사용됩니다.

상수 값만 포함된 식이 대상 형식의 범위를 벗어난 값을 생성할 경우 기본적으로 이 식에서는 컴파일러 오류가 발생합니다. 식에 하나 이상의 상수가 아닌 값이 포함된 경우 컴파일러에서는 오버플로를 감지하지 않습니다. 다음 예제에서 `i2`에 할당된 식을 계산하면 컴파일러 오류가 발생하지 않습니다.

```
// The following example causes compiler error CS0220 because 2147483647
// is the maximum value for integers.
//int i1 = 2147483647 + 10;

// The following example, which includes variable ten, does not cause
// a compiler error.
int ten = 10;
int i2 = 2147483647 + ten;

// By default, the overflow in the previous statement also does
// not cause a run-time exception. The following line displays
// -2,147,483,639 as the sum of 2,147,483,647 and 10.
Console.WriteLine(i2);
```

기본적으로 이러한 상수가 아닌 식은 런타임에 오버플로가 있는지 검사되지 않고 오버플로 예외를 일으키지 않습니다. 이전 예제는 양의 정수 2개의 합계로 -2,147,483,639를 표시합니다.

오버플로 검사는 컴파일러 옵션, 환경 구성 또는 `checked` 키워드 사용을 통해 사용하도록 설정될 수 있습니다. 다음 예제에서는 `checked` 식 또는 `checked` 블록을 사용하여 런타임에 이전 합계에 의해 생성되는 오버플로를 감지하는 방법을 보여 줍니다. 두 예제는 모두 오버플로 예외를 일으킵니다.

```
// If the previous sum is attempted in a checked environment, an
// OverflowException error is raised.

// Checked expression.
Console.WriteLine(checked(2147483647 + ten));

// Checked block.
checked
{
    int i3 = 2147483647 + ten;
    Console.WriteLine(i3);
}
```

`unchecked` 키워드는 오버플로 검사를 피하는 데 사용될 수 있습니다.

## 예제

이 샘플에서는 `checked` 를 사용하여 런타임에 오버플로 검사를 사용하도록 설정하는 방법을 보여 줍니다.

```

class OverFlowTest
{
    // Set maxValue to the maximum value for integers.
    static int maxValue = 2147483647;

    // Using a checked expression.
    static int CheckedMethod()
    {
        int z = 0;
        try
        {
            // The following line raises an exception because it is checked.
            z = checked(maxValue + 10);
        }
        catch (System.OverflowException e)
        {
            // The following line displays information about the error.
            Console.WriteLine("CHECKED and CAUGHT: " + e.ToString());
        }
        // The value of z is still 0.
        return z;
    }

    // Using an unchecked expression.
    static int UncheckedMethod()
    {
        int z = 0;
        try
        {
            // The following calculation is unchecked and will not
            // raise an exception.
            z = maxValue + 10;
        }
        catch (System.OverflowException e)
        {
            // The following line will not be executed.
            Console.WriteLine("UNCHECKED and CAUGHT: " + e.ToString());
        }
        // Because of the undetected overflow, the sum of 2147483647 + 10 is
        // returned as -2147483639.
        return z;
    }

    static void Main()
    {
        Console.WriteLine("\nCHECKED output value is: {0}",
                        CheckedMethod());
        Console.WriteLine("UNCHECKED output value is: {0}",
                        UncheckedMethod());
    }
}

/*
Output:
CHECKED and CAUGHT: System.OverflowException: Arithmetic operation resulted
in an overflow.
    at ConsoleApplication1.OverFlowTest.CheckedMethod()

CHECKED output value is: 0
UNCHECKED output value is: -2147483639
*/
}

```

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [Checked 및 Unchecked](#)
- [unchecked](#)

# unchecked(C# 참조)

2020-11-02 • 4 minutes to read • [Edit Online](#)

`unchecked` 키워드는 정수 형식 산술 연산 및 변환에 대한 오버플로 검사를 비활성화하는 데 사용됩니다.

`unchecked` 컨텍스트에서 식이 대상 형식의 범위를 벗어난 값을 생성하는 경우 오버플로에 플래그가 지정되지 않습니다. 예를 들어 다음 예제의 계산은 `unchecked` 블록 또는 식에서 수행되므로 결과가 정수에 비해 너무 크다는 사실이 무시되며 `int1`에 -2,147,483,639 값이 할당됩니다.

```
unchecked
{
    int1 = 2147483647 + 10;
}
int1 = unchecked(ConstantMax + 10);
```

`unchecked` 환경을 제거하면 컴파일 오류가 발생합니다. 식의 모든 항이 상수이기 때문에 컴파일 시간에 오버플로가 검색될 수 있습니다.

상수가 아닌 항을 포함하는 식은 컴파일 시간 및 런타임에 기본적으로 확인되지 않습니다. `checked` 환경을 사용하도록 설정하는 방법에 대한 자세한 내용은 [checked](#)를 참조하세요.

오버플로를 확인하는 데 시간이 걸리기 때문에 오버플로 위험이 없는 상황에서는 `unchecked` 코드를 사용하여 성능을 향상할 수 있습니다. 그러나 오버플로가 발생할 가능성이 있는 경우 `checked` 환경을 사용해야 합니다.

## 예제

이 샘플에서는 `unchecked` 키워드를 사용하는 방법을 보여 줍니다.

```

class UncheckedDemo
{
    static void Main(string[] args)
    {
        // int.MaxValue is 2,147,483,647.
        const int ConstantMax = int.MaxValue;
        int int1;
        int int2;
        int variableMax = 2147483647;

        // The following statements are checked by default at compile time. They do not
        // compile.
        //int1 = 2147483647 + 10;
        //int1 = ConstantMax + 10;

        // To enable the assignments to int1 to compile and run, place them inside
        // an unchecked block or expression. The following statements compile and
        // run.
        unchecked
        {
            int1 = 2147483647 + 10;
        }
        int1 = unchecked(ConstantMax + 10);

        // The sum of 2,147,483,647 and 10 is displayed as -2,147,483,639.
        Console.WriteLine(int1);

        // The following statement is unchecked by default at compile time and run
        // time because the expression contains the variable variableMax. It causes
        // overflow but the overflow is not detected. The statement compiles and runs.
        int2 = variableMax + 10;

        // Again, the sum of 2,147,483,647 and 10 is displayed as -2,147,483,639.
        Console.WriteLine(int2);

        // To catch the overflow in the assignment to int2 at run time, put the
        // declaration in a checked block or expression. The following
        // statements compile but raise an overflow exception at run time.
        checked
        {
            //int2 = variableMax + 10;
        }
        //int2 = checked(variableMax + 10);

        // Unchecked sections frequently are used to break out of a checked
        // environment in order to improve performance in a portion of code
        // that is not expected to raise overflow exceptions.
        checked
        {
            // Code that might cause overflow should be executed in a checked
            // environment.
            unchecked
            {
                // This section is appropriate for code that you are confident
                // will not result in overflow, and for which performance is
                // a priority.
            }
            // Additional checked code here.
        }
    }
}

```

## C# 언어 사양

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

다.

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [Checked 및 Unchecked](#)
- [checked](#)

# fixed 문(C# 참조)

2020-11-02 • 7 minutes to read • [Edit Online](#)

`fixed` 문은 가비지 수집기에서 이동 가능한 변수를 재배치할 수 없도록 합니다. `fixed` 문은 `unsafe` 컨텍스트에서만 허용됩니다. `fixed` 키워드를 사용하여 [고정 크기 버퍼](#)를 만들 수도 있습니다.

`fixed` 문은 관리되는 변수에 대한 포인터를 설정하고 문을 실행하는 동안 해당 변수를 "고정"합니다. 이동 가능한 관리되는 변수에 대한 포인터는 `fixed` 컨텍스트에만 유용합니다. `fixed` 컨텍스트 없는 가비지 수집은 예기치 않게 변수를 재배치할 수 있습니다. C# 컴파일러만 `fixed` 문에서 관리되는 변수에 대한 포인터를 할당할 수 있습니다.

```
class Point
{
    public int x;
    public int y;
}

unsafe private static void ModifyFixedStorage()
{
    // Variable pt is a managed variable, subject to garbage collection.
    Point pt = new Point();

    // Using fixed allows the address of pt members to be taken,
    // and "pins" pt so that it is not relocated.

    fixed (int* p = &pt.x)
    {
        *p = 1;
    }
}
```

배열, 문자열, 고정 크기 버퍼 또는 변수의 주소를 사용하여 포인터를 초기화할 수 있습니다. 다음 예제에서는 변수 주소, 배열 및 문자열의 사용을 보여 줍니다.

```
Point point = new Point();
double[] arr = { 0, 1.5, 2.3, 3.4, 4.0, 5.9 };
string str = "Hello World";

// The following two assignments are equivalent. Each assigns the address
// of the first element in array arr to pointer p.

// You can initialize a pointer by using an array.
fixed (double* p = arr) { /*...*/ }

// You can initialize a pointer by using the address of a variable.
fixed (double* p = &arr[0]) { /*...*/ }

// The following assignment initializes p by using a string.
fixed (char* p = str) { /*...*/ }

// The following assignment is not valid, because str[0] is a char,
// which is a value, not a variable.
//fixed (char* p = &str[0]) { /*...*/ }
```

C# 7.3부터 `fixed` 문은 배열, 문자열, 고정 크기 버퍼 또는 비관리형 변수 외의 추가 형식에서 작동합니다.

`GetPinnableReference` 메서드를 구현하는 모든 형식을 고정할 수 있습니다. `GetPinnableReference`는 `ref` 변수

를 비관리형 형식으로 반환해야 합니다. .NET Core 2.0에 도입된 .NET 형식 `System.Span<T>` 및 `System.ReadOnlySpan<T>`은 이 패턴을 사용하여 고정될 수 있습니다. 이는 다음 예제에서 확인할 수 있습니다.

```
unsafe private static void FixedSpanExample()
{
    int[] PascalsTriangle = {
        1,
        1, 1,
        1, 2, 1,
        1, 3, 3, 1,
        1, 4, 6, 4, 1,
        1, 5, 10, 10, 5, 1
    };

    Span<int> RowFive = new Span<int>(PascalsTriangle, 10, 5);

    fixed (int* ptrToRow = RowFive)
    {
        // Sum the numbers 1,4,6,4,1
        var sum = 0;
        for (int i = 0; i < RowFive.Length; i++)
        {
            sum += *(ptrToRow + i);
        }
        Console.WriteLine(sum);
    }
}
```

이 패턴에 참여해야 하는 형식을 만드는 경우 패턴을 구현하는 예제는 `Span<T>.GetPinnableReference()`를 참조하세요.

여러 개의 포인터가 모두 동일한 형식인 경우 하나의 명령문에서 초기화할 수 있습니다.

```
fixed (byte* ps = srcarray, pd = dstarray) {...}
```

형식이 다른 포인터를 초기화하려면 다음 예제와 같이 `fixed` 문을 중첩하면 됩니다.

```
fixed (int* p1 = &point.x)
{
    fixed (double* p2 = &arr[5])
    {
        // Do something with p1 and p2.
    }
}
```

이 문의 코드를 실행하면 고정된 모든 변수가 고정 해제되고 가비지 수집됩니다. 따라서 `fixed` 문 외부에서 해당 변수를 가리키지 않습니다. `fixed` 명령문 내에서 선언된 변수는 해당 명령문에 범위가 지정되어 다음을 더 쉽게 만듭니다.

```
fixed (byte* ps = srcarray, pd = dstarray)
{
    ...
}
// ps and pd are no longer in scope here.
```

`fixed` 명령문에서 초기화된 포인터는 읽기 전용 변수입니다. 포인터 값을 수정하려는 경우 두 번째 포인터 변수를 선언하고 수정해야 합니다. `fixed` 명령문에서 선언된 변수는 수정될 수 없습니다.

```
fixed (byte* ps = srcarray, pd = dstarray)
{
    byte* pSourceCopy = ps;
    pSourceCopy++; // point to the next element.
    ps++; // invalid: cannot modify ps, as it is declared in the fixed statement.
}
```

가비지 수집의 대상이 아니므로 고정할 필요가 없는 스택에서 메모리를 할당할 수 있습니다. 이렇게 하려면 [stackalloc](#) 식을 사용합니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 [fixed 문](#) 섹션을 참조하세요.

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [unsafe](#)
- [포인터 형식](#)
- [고정 크기 버퍼](#)

# lock 문(C# 참조)

2020-11-02 • 5 minutes to read • [Edit Online](#)

`lock` 문은 지정된 개체에 대한 상호 배제 잠금을 획득하여 명령문 블록을 실행한 다음, 잠금을 해제합니다. 잠금이 유지되는 동안 잠금을 보유하는 스레드는 잠금을 다시 획득하고 해제할 수 있습니다. 다른 스레드는 잠금을 획득할 수 없도록 차단되며 잠금이 해제될 때까지 대기합니다.

`lock` 문이 형식입니다.

```
lock (x)
{
    // Your code...
}
```

여기서 `x`는 참조 형식의 식입니다. 정확히 다음과 같은 경우

```
object __lockObj = x;
bool __lockWasTaken = false;
try
{
    System.Threading.Monitor.Enter(__lockObj, ref __lockWasTaken);
    // Your code...
}
finally
{
    if (__lockWasTaken) System.Threading.Monitor.Exit(__lockObj);
}
```

코드에서 `try...finally` 블록을 사용하므로 `lock` 문의 본문 내에서 예외가 throw되더라도 잠금이 해제됩니다.

`lock` 문의 본문에서 `await` 연산자를 사용할 수 없습니다.

## 지침

공유 리소스에 대한 스레드 액세스를 동기화하는 경우 전용 개체 인스턴스(예:

`private readonly object balanceLock = new object();`) 또는 코드의 관련 없는 파트에서 잠금 개체로 사용되지 않을 가능성이 있는 다른 인스턴스를 잠금합니다. 교착 상태 또는 잠금 경합이 발생할 수 있으므로 다른 공유 리소스에 대해 동일한 잠금 개체 인스턴스를 사용하지 마세요. 특히 다음을 잠금 개체로 사용하지 마세요.

- `this` (호출자가 잠금으로 사용할 수 있음).
- `Type` 인스턴스(`typeof` 연산자 또는 리플렉션에서 획득할 수 있음).
- 문자열 인스턴스(문자열 리터럴 포함)([인터닝](#) 될 수 있음).

잠금 경합을 줄이기 위해 최대한 짧은 시간 동안 잠금을 유지하세요.

## 예제

다음 예제에서는 전용 `balanceLock` 인스턴스에 잠금을 설정하여 해당 개인 `balance` 필드에 대한 액세스를 동기화하는 `Account` 클래스를 정의합니다. 동일한 인스턴스를 잠금에 사용하면 `Debit` 또는 `Credit` 메서드를 동시에 호출하려는 두 스레드에 의해 `balance` 필드가 동시에 업데이트되지 않습니다.

```
using System;
using System.Threading.Tasks;
```

```
using System.Threading.Tasks;
```

```
public class Account
{
    private readonly object balanceLock = new object();
    private decimal balance;

    public Account(decimal initialBalance) => balance = initialBalance;

    public decimal Debit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The debit amount cannot be negative.");
        }

        decimal appliedAmount = 0;
        lock (balanceLock)
        {
            if (balance >= amount)
            {
                balance -= amount;
                appliedAmount = amount;
            }
        }
        return appliedAmount;
    }

    public void Credit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The credit amount cannot be negative.");
        }

        lock (balanceLock)
        {
            balance += amount;
        }
    }

    public decimal GetBalance()
    {
        lock (balanceLock)
        {
            return balance;
        }
    }
}

class AccountTest
{
    static async Task Main()
    {
        var account = new Account(1000);
        var tasks = new Task[100];
        for (int i = 0; i < tasks.Length; i++)
        {
            tasks[i] = Task.Run(() => Update(account));
        }
        await Task.WhenAll(tasks);
        Console.WriteLine($"Account's balance is {account.GetBalance()}");
        // Output:
        // Account's balance is 2000
    }

    static void Update(Account account)
    {
        decimal[] amounts = { 0, 2, -3, 6, -2, -1, 8, -5, 11, -6 };
        foreach (var amount in amounts)
        {
            account.Debit(amount);
        }
    }
}
```

```
foreach (var amount in amounts)
{
    if (amount >= 0)
    {
        account.Credit(amount);
    }
    else
    {
        account.Debit(Math.Abs(amount));
    }
}
```

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 `lock` 문 섹션을 참조하세요.

## 참조

- [C# 참조](#)
- [C# 키워드](#)
- [System.Threading.Monitor](#)
- [System.Threading.SpinLock](#)
- [System.Threading.Interlocked](#)
- [동기화 기본 형식 개요](#)

# 메서드 매개 변수(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

`in`, `ref` 또는 `out` 없이 메서드에 대해 선언된 매개 변수는 값으로 호출된 메서드에 전달됩니다. 메서드에서 해당 값을 변경할 수 있지만, 제어가 호출하는 프로시저로 다시 전달되면 변경된 값이 보존되지 않습니다. 메서드 매개 변수 키워드를 사용하여 이 동작을 변경할 수 있습니다.

이 섹션에서는 메서드 매개 변수를 선언할 때 사용할 수 있는 키워드를 설명합니다.

- `params`는 이 매개 변수가 가변 개수의 인수를 사용할 수 있음을 지정합니다.
- `in`은 이 매개 변수를 참조로 전달할 수 있지만 호출된 메서드로만 읽을 수 있음을 지정합니다.
- `ref`는 이 매개 변수를 참조로 전달할 수 있고 호출된 메서드로 읽거나 쓸 수 있음을 지정합니다.
- `out`는 이 매개 변수가 참조로 전달되고 호출된 메서드에 의해 기록되도록 지정합니다.

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)

# params(C# 참조)

2020-11-02 • 3 minutes to read • [Edit Online](#)

`params` 키워드를 사용하면 가변 개수의 인수를 사용하는 메서드 매개 변수를 지정할 수 있습니다. 매개 변수 배열은 1차원 배열이어야 합니다.

메서드 선언에서 `params` 키워드 뒤에는 추가 매개 변수가 허용되지 않으며, `params` 키워드 하나만 메서드 선언에 사용할 수 있습니다.

`params` 매개 변수의 선언된 형식이 1차원 배열이 아닌 경우 컴파일러 오류 [CS0225](#)가 발생합니다.

`params` 매개 변수를 사용하여 메서드를 호출하면 다음을 전달할 수 있습니다.

- 배열 요소 형식의 쉼표로 구분된 인수 목록입니다.
- 지정된 형식의 인수 배열입니다.
- 인수가 없습니다. 인수를 보내지 않는 경우 `params` 목록의 길이는 0입니다.

## 예제

다음 예제에서는 `params` 매개 변수에 인수를 보낼 수 있는 다양한 방법을 보여 줍니다.

```

public class MyClass
{
    public static void UseParams(params int[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    public static void UseParams2(params object[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    static void Main()
    {
        // You can send a comma-separated list of arguments of the
        // specified type.
        UseParams(1, 2, 3, 4);
        UseParams2(1, 'a', "test");

        // A params parameter accepts zero or more arguments.
        // The following calling statement displays only a blank line.
        UseParams2();

        // An array argument can be passed, as long as the array
        // type matches the parameter type of the method being called.
        int[] myIntArray = { 5, 6, 7, 8, 9 };
        UseParams(myIntArray);

        object[] myObjArray = { 2, 'b', "test", "again" };
        UseParams2(myObjArray);

        // The following call causes a compiler error because the object
        // array cannot be converted into an integer array.
        //UseParams(myObjArray);

        // The following call does not cause an error, but the entire
        // integer array becomes the first element of the params array.
        UseParams2(myIntArray);
    }
}
/*
Output:
1 2 3 4
1 a test

5 6 7 8 9
2 b test again
System.Int32[]
*/

```

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [메서드 매개 변수](#)

# in 매개 변수 한정자(C# 참조)

2020-11-02 • 14 minutes to read • [Edit Online](#)

`in` 키워드를 사용하면 참조를 통해 인수를 전달할 수 있습니다. 이 키워드는 정식 매개 변수를 위해 해당 인수의 별칭을 만드는데, 이는 반드시 변수여야 합니다. 즉, 매개 변수에 대한 모든 작업이 인수에서 수행됩니다. 이 키워드는 호출된 메서드에서 `in` 인수를 수정할 수 없다는 점을 제외하고 `ref` 또는 `out` 키워드와 유사합니다. `ref` 인수는 수정할 수 있지만 `out` 인수는 호출된 메서드가 수정해야 하며, 해당 수정 사항은 호출 컨텍스트에서 식별 가능합니다.

```
int readonlyArgument = 44;
InArgExample(readonlyArgument);
Console.WriteLine(readonlyArgument);      // value is still 44

void InArgExample(in int number)
{
    // Uncomment the following line to see error CS8331
    //number = 19;
}
```

앞의 예제는 호출 사이트에서 일반적으로 `in` 한정자가 필요하지 않다는 것을 설명합니다. 메서드 선언에만 필요합니다.

## NOTE

`in` 키워드는 `foreach` 명령문의 일부 또는 LINQ 쿼리에서 `join` 절의 일부로 형식 매개 변수가 반공변(contravariant)임을 지정하도록 제네릭 형식 매개 변수와 함께 사용될 수도 있습니다. 이러한 컨텍스트에서 `in` 키워드의 사용에 대한 자세한 내용은 모든 해당 사용에 대한 링크를 제공하는 [in](#)을 참조하세요.

`in` 인수로 전달되는 변수는 메서드 호출에서 전달되기 전에 초기화되어야 합니다. 그러나 호출된 메서드는 값을 할당하거나 인수를 수정하지 않을 수 있습니다.

`in` 매개 변수 한정자는 C# 7.2 이상에서 사용 가능합니다. 이전 버전은 컴파일러 오류 [CS8107](#) ("C# 7.0에서는 '일기 전용 참조' 기능을 사용할 수 없습니다. 언어 버전 7.2 이상을 사용하세요.")을 생성합니다. 컴파일러 언어 버전을 구성하려면 [C# 언어 버전 선택](#)을 참조하세요.

`in`, `ref` 및 `out` 키워드는 오버로드 해결을 위한 메서드 시그니처의 일부로 간주되지 않습니다. 따라서 메서드 하나는 `ref` 또는 `in` 인수를 사용하고 다른 하나는 `out` 인수를 사용한다는 것 외에는 차이점이 없으면 메서드를 오버로드할 수 없습니다. 예를 들어 다음 코드는 컴파일되지 않습니다.

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on in, ref and out".
    public void SampleMethod(in int i) { }
    public void SampleMethod(ref int i) { }
}
```

`in`의 존재에 기반한 오버로딩이 허용됩니다.

```

class InOverloads
{
    public void SampleMethod(in int i) { }
    public void SampleMethod(int i) { }
}

```

## 오버로드 해결 규칙

`in` 인수에 대한 동기를 이해하여 값과 `in` 인수로 메서드의 오버로드 해결 규칙을 이해할 수 있습니다. `in` 매개 변수를 사용하여 메서드를 정의하면 잠재적인 성능 최적화가 이루어집니다. 일부 `struct` 형식 인수는 크기가 클 수 있으며 긴밀한 루프 또는 중요한 코드 경로에서 메서드를 호출할 때 해당 구조를 복사하는 비용이 중요합니다. 메서드는 호출된 메서드가 인수의 상태를 수정하지 않으므로 `in` 매개 변수를 선언하여 해당 인수가 참조로 안전하게 전달될 수 있음을 지정합니다. 이러한 인수를 참조로 전달하면 (잠재적으로) 비용이 많이 드는 복사본을 방지할 수 있습니다.

호출 사이트의 인수에 `in`을 지정하는 것은 일반적으로 선택 사항입니다. 값으로 인수를 전달하고 `in` 한정자를 사용하여 인수를 전달하는 것 사이에는 의미 체계상 차이가 없습니다. 호출 사이트의 `in` 한정자는 인수 값이 변경될 수 있음을 나타내지 않아도 되므로 선택 사항입니다. 호출 사이트에서 `in` 한정자를 명시적으로 추가하여 인수가 값이 아닌 참조로 전달되도록 합니다. 명시적으로 `in`을 사용하는 경우 다음과 같은 두 가지 효과가 있습니다.

먼저 호출 사이트에서 `in`을 지정하면 컴파일러가 일치하는 `in` 매개 변수로 정의된 메서드를 선택하게 됩니다. 그렇지 않으면 두 메서드가 `in`이 있을 때만 다른 경우 by 값 오버로드가 더 적합합니다.

둘째, `in`을 지정하면 참조로 인수를 전달할 의사가 있음을 선언하는 것입니다. `in`에 사용된 인수는 직접 참조할 수 있는 위치를 나타내야 합니다. `out` 및 `ref` 인수에는 동일한 일반 규칙이 적용됩니다. 상수, 일반 속성 또는 값을 생성하는 다른 식은 사용할 수 없습니다. 그렇지 않으면 호출 사이트에서 `in`을 생략할 경우 메서드에 대한 읽기 전용 참조로 전달할 임시 변수를 만들 수 있도록 컴파일러에 알립니다. 컴파일러는 `in` 인수를 사용하여 몇 가지 제한 사항을 해결하기 위해 임시 변수를 만듭니다.

- 임시 변수는 컴파일 시간 상수를 `in` 매개 변수로 허용합니다.
- 임시 변수는 속성 또는 `in` 매개 변수에 대한 다른 식을 허용합니다.
- 임시 변수는 인수 형식에서 매개 변수 형식으로의 암시적 변환이 있는 경우 인수를 허용합니다.

앞의 모든 인스턴스에서 컴파일러는 상수, 속성 또는 다른 식의 값을 저장하는 임시 변수를 만듭니다.

다음 코드에서는 이러한 규칙을 보여줍니다.

```

static void Method(in int argument)
{
    // implementation removed
}

Method(5); // OK, temporary variable created.
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // OK, temporary int created with the value 0
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // passed by readonly reference
Method(in i); // passed by readonly reference, explicitly using `in`

```

이제 by 값 인수를 사용하는 다른 메서드를 사용할 수 있다고 가정하겠습니다. 결과는 다음 코드와 같이 변경됩니다.

```

static void Method(int argument)
{
    // implementation removed
}

static void Method(in int argument)
{
    // implementation removed
}

Method(5); // Calls overload passed by value
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // Calls overload passed by value.
Method(in s); // CS1503: cannot convert in short to in int
int i = 42;
Method(i); // Calls overload passed by value
Method(in i); // passed by readonly reference, explicitly using `in`

```

인수가 참조로 전달되는 유일한 메서드 호출이 마지막입니다.

#### NOTE

앞의 코드는 단순화를 위해 인수 형식으로 `int`를 사용합니다. `int`는 대부분의 최신 컴퓨터에서 참조보다 크지 않기 때문에 단일 `int`를 읽기 전용 참조로 전달하면 아무런 이점이 없습니다.

## `in` 매개 변수에 대한 제한 사항

다음과 같은 종류의 메서드에는 `in`, `ref` 및 `out` 키워드를 사용할 수 없습니다.

- `async` 한정자를 사용하여 정의하는 비동기 메서드
- `yield return` 또는 `yield break` 문을 포함하는 반복기 메서드
- 확장 메서드의 첫 번째 인수는 구조체인 경우가 아니면 `in` 한정자를 가질 수 없음
- 해당 인수가 제네릭 형식인 확장 메서드의 첫 번째 인수(해당 형식이 구조체로 제한되는 경우도 포함)

## 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# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [메서드 매개 변수](#)
- [안전하고 효율적인 코드 작성](#)

# ref(C# 참조)

2020-11-02 • 21 minutes to read • [Edit Online](#)

`ref` 키워드는 참조로 전달되는 값을 나타냅니다. 다음 네 가지 컨텍스트에서 사용됩니다.

- 메서드 시그니처 및 메서드 호출에서 인수를 메서드에 참조로 전달합니다. 자세한 내용은 [참조로 인수 전달](#)을 참조하세요.
- 메서드 시그니처에서 값을 호출자에게 참조로 반환합니다. 자세한 내용은 [참조 반환 값](#)을 참조하세요.
- 멤버 본문에서 참조 반환 값이 호출자가 수정하려는 참조로 로컬에 저장되거나 일반적으로 로컬 변수가 참조를 기준으로 다른 값에 액세스 함을 나타냅니다. 자세한 내용은 [ref로컬](#)을 참조하세요.
- `ref struct` 또는 `readonly ref struct`을 선언하기 위한 `struct` 선언서. 자세한 내용은 [구조체 형식](#) 문서의 [ref 구조체](#) 섹션을 참조하세요.

## 참조로 인수 전달

메서드의 매개 변수 목록에 사용되는 경우 `ref` 키워드는 인수가 값이 아니라 참조로 전달됨을 나타냅니다.

`ref` 키워드는 정식 매개 변수를 변수여야 하는 인수의 별칭으로 설정합니다. 즉, 매개 변수에 대한 모든 작업이 인수에서 수행됩니다. 예를 들어 호출자가 지역 변수 식 또는 배열 요소 액세스 식을 전달하는 경우 호출된 메서드에서 `ref` 매개 변수가 참조하는 개체를 바꾸면 메서드 반환 시 호출자의 지역 변수 또는 배열 요소가 새 개체를 참조합니다.

### NOTE

참조로 전달의 개념과 참조 형식의 개념을 혼동해서는 안 됩니다. 이 두 개념은 서로 다릅니다. 메서드 매개 변수는 값 형식이든 참조 형식이든 관계없이 `ref`를 통해 수정할 수 있으며, 참조로 전달되는 경우 값 형식은 boxing되지 않습니다.

`ref` 매개 변수를 사용하려면 다음 예제에 나와 있는 것처럼 메서드 정의와 호출 메서드가 모두 `ref` 키워드를 명시적으로 사용해야 합니다.

```
void Method(ref int refArgument)
{
    refArgument = refArgument + 44;
}

int number = 1;
Method(ref number);
Console.WriteLine(number);
// Output: 45
```

`ref` 또는 `in` 매개 변수로 전달하는 인수는 전달 전에 초기화해야 합니다. 이러한 방식은 인수를 전달하기 전에 명시적으로 초기화할 필요가 없는 `out` 매개 변수와는 다릅니다.

클래스의 멤버는 `ref`, `in` 또는 `out`만 다른 서명을 포함할 수 없습니다. 특정 형식의 두 멤버가 하나는 `ref` 매개 변수를 포함하고 다른 하나는 `out` 또는 `in` 매개 변수를 포함한다는 것 외에는 차이가 없으면 컴파일러 오류가 발생합니다. 예를 들어 다음 코드는 컴파일되지 않습니다.

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

그러나 다음 예제에 나와 있는 것처럼 메서드 하나에는 `ref`, `in` 또는 `out` 매개 변수가 포함되어 있고 다른 하나에는 값 매개 변수가 포함되어 있으면 메서드를 오버로드할 수 있습니다.

```
class RefOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(ref int i) { }
}
```

숨기기나 재정의와 같이 서명이 일치해야 하는 다른 상황에서는 `in`, `ref` 및 `out` 이 서명의 일부가 되며 서로 일치하지 않습니다.

속성은 변수가 아니라 메서드이며 `ref` 매개 변수로 전달할 수 없습니다.

다음과 같은 종류의 메서드에는 `ref`, `in` 및 `out` 키워드를 사용할 수 없습니다.

- `async` 한정자를 사용하여 정의하는 비동기 메서드
- `yield return` 또는 `yield break` 문을 포함하는 반복기 메서드

또한 [확장 메서드](#)에는 다음과 같은 제한 사항이 있습니다.

- 확장 메서드의 첫 번째 인수에는 `out` 키워드를 사용할 수 없습니다.
- 인수가 구조체가 아니거나 구조체로 제한되지 않는 제네릭 형식일 경우에는 확장 메서드의 첫 번째 인수에 `ref` 키워드를 사용할 수 없습니다.
- 첫 번째 인수가 구조체인 경우 이외에는 `in` 키워드를 사용할 수 없습니다. 구조체로 제한되는 경우에도 `in` 키워드는 제네릭 형식에 사용할 수 없습니다.

## 참조로 인수 전달: 예제

앞의 예제에서는 값 형식을 참조로 전달합니다. `ref` 키워드를 사용하여 참조 형식을 참조로 전달할 수도 있습니다. 참조 형식을 참조로 전달하는 경우 호출된 메서드는 참조 매개 변수가 호출자에서 참조하는 개체를 바꿀 수 있습니다. 개체의 스토리지 위치는 참조 매개 변수의 값으로 메서드에 전달됩니다. 매개 변수의 스토리지 위치에서 값을 변경하여 새 개체를 가리키도록 하면 호출자가 참조하는 스토리지 위치도 변경됩니다. 다음 예제에서는 참조 형식 인스턴스를 `ref` 매개 변수로 전달합니다.

```

class Product
{
    public Product(string name, int newID)
    {
        ItemName = name;
        ItemID = newID;
    }

    public string ItemName { get; set; }
    public int ItemID { get; set; }
}

private static void ChangeByReference(ref Product itemRef)
{
    // Change the address that is stored in the itemRef parameter.
    itemRef = new Product("Stapler", 99999);

    // You can change the value of one of the properties of
    // itemRef. The change happens to item in Main as well.
    itemRef.ItemID = 12345;
}

private static void ModifyProductsByReference()
{
    // Declare an instance of Product and display its initial values.
    Product item = new Product("Fasteners", 54321);
    System.Console.WriteLine("Original values in Main. Name: {0}, ID: {1}\n",
        item.ItemName, item.ItemID);

    // Pass the product instance to ChangeByReference.
    ChangeByReference(ref item);
    System.Console.WriteLine("Back in Main. Name: {0}, ID: {1}\n",
        item.ItemName, item.ItemID);
}

// This method displays the following output:
// Original values in Main. Name: Fasteners, ID: 54321
// Back in Main. Name: Stapler, ID: 12345

```

참조 형식을 참조 및 값으로 전달하는 방법에 대한 자세한 내용은 [참조-형식 매개 변수 전달](#)을 참조하세요.

## 참조 반환 값

참조 반환 값(또는 ref return)은 메서드가 호출자에게 참조로 반환하는 값입니다. 즉, 호출자는 메서드에서 반환된 값을 수정할 수 있으며 해당 변경 내용은 호출 메서드의 개체 상태에 반영됩니다.

참조 반환 값은 `ref` 키워드를 사용하여 정의됩니다.

- 메서드 시그니처에서 예를 들어 다음 메서드 시그니처는 `GetCurrentPrice` 메서드가 `Decimal` 값을 참조로 반환함을 나타냅니다.

```
public ref decimal GetCurrentPrice()
```

- 메서드의 `return` 문에서 반환된 `return` 토큰과 변수 간에 예를 들어:

```
return ref DecimalArray[0];
```

호출자가 개체 상태를 수정하려면 참조 반환 값을 [참조 로컬](#)로 명시적으로 정의된 변수에 저장해야 합니다.

다음은 메서드 시그니처와 메서드 본문을 모두 보여주는 보다 완전한 참조 반환 예제입니다.

```
public static ref int Find(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

호출된 메서드는 `ref readonly`로 반환 값을 선언하여 참조를 통해 값을 반환하고 호출 코드가 반환된 값을 수정할 수 없도록 합니다. 호출 메서드는 로컬 `ref readonly` 변수에 값을 저장하여 반환된 값이 복사되지 않도록 할 수 있습니다.

예를 들어 [ref 반환 및 ref 지역 예제](#)를 참조하세요.

## 참조 로컬

참조 지역 변수는 `return ref`을 사용하여 반환된 값을 참조하는 데 사용됩니다. 참조 지역 변수는 비참조 반환 값으로 초기화할 수 없습니다. 즉, 초기화의 오른쪽은 참조여야 합니다. 참조 로컬 값의 수정 내용은 메서드가 값을 참조로 반환하는 개체 상태에 반영됩니다.

변수 선언 앞, 값을 참조로 반환하는 메서드 호출 직전에 `ref` 키워드를 사용하여 참조 로컬을 정의합니다.

예를 들어 다음 문은 `GetEstimatedValue` 메서드에서 반환되는 참조 로컬 값을 정의합니다.

```
ref decimal estValue = ref Building.GetEstimatedValue();
```

동일한 방법으로 참조로 값에 액세스할 수 있습니다. 경우에 따라 참조로 값에 액세스하면 비용이 많이 들 수 있는 복사 작업을 피함으로써 성능이 향상됩니다. 예를 들어, 다음 명령문은 값을 참조하는 데 사용되는 참조 로컬 값을 정의하는 방법을 보여줍니다.

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
```

두 예에서 `ref` 키워드는 두 위치에 모두 사용해야 합니다. 그렇지 않으면 컴파일러 오류 CS8172, "값을 사용하여 참조 형식 변수를 초기화할 수 없습니다."가 생성됩니다.

C# 7.3부터 `foreach` 문의 반복 변수는 `ref` 지역 또는 `ref readonly` 지역 변수일 수 있습니다. 자세한 내용은 [foreach 문](#) 문서를 참조하세요.

또한 C# 7.3부터 [ref 대입 연산자](#)를 사용하여 `ref` 지역 또는 `ref readonly` 지역 변수를 다시 할당할 수 있습니다.

## Ref readonly 로컬

`Ref readonly` 로컬은 해당 시그니처에 `ref readonly` 가 있고 `return ref`를 사용하는 메서드 또는 속성을 통해 반환된 값을 참조하는 데 사용됩니다. `ref readonly` 변수는 `ref` 지역 변수의 속성을 `readonly` 변수와 결합합니다. 이는 할당된 스토리지의 별칭이고 수정할 수 없습니다.

## 참조 반환 및 참조 로컬 예제

다음 예제에서는 두 개의 `String` 필드 `Title` 및 `Author`가 있는 `Book` 클래스를 정의합니다. 또한 `Book` 개체의 `private` 배열을 포함하는 `BookCollection` 클래스를 정의합니다. 개별 책 개체는 해당 `GetBookByTitle` 메서드를 호출하여 참조로 반환됩니다.

```

public class Book
{
    public string Author;
    public string Title;
}

public class BookCollection
{
    private Book[] books = { new Book { Title = "Call of the Wild, The", Author = "Jack London" },
                            new Book { Title = "Tale of Two Cities, A", Author = "Charles Dickens" }
                           };
    private Book nobook = null;

    public ref Book GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
        {
            if (title == books[ctr].Title)
                return ref books[ctr];
        }
        return ref nobook;
    }

    public void ListBooks()
    {
        foreach (var book in books)
        {
            Console.WriteLine($"{book.Title}, by {book.Author}");
        }
        Console.WriteLine();
    }
}

```

호출자가 `GetBookByTitle` 메서드에서 참조 로컬로 반환된 값을 저장하는 경우 호출자가 반환 값을 변경하면 다음 예제와 같이 `BookCollection` 개체에 변경 내용이 반영됩니다.

```

var bc = new BookCollection();
bc.ListBooks();

ref var book = ref bc.GetBookByTitle("Call of the Wild, The");
if (book != null)
    book = new Book { Title = "Republic, The", Author = "Plato" };
bc.ListBooks();
// The example displays the following output:
//      Call of the Wild, The, by Jack London
//      Tale of Two Cities, A, by Charles Dickens
//
//      Republic, The, by Plato
//      Tale of Two Cities, A, by Charles Dickens

```

## C# 언어 사양

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

## 참조

- [안전하고 효율적인 코드 작성](#)
- [Ref return 및 ref local](#)
- [조건부 ref 식](#)

- [매개 변수 전달](#)
- [메서드 매개 변수](#)
- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)

# out 매개 변수 한정자(C# 참조)

2020-11-02 • 10 minutes to read • [Edit Online](#)

`out` 키워드를 사용하면 참조를 통해 인수를 전달할 수 있습니다. 이 키워드는 정식 매개 변수를 위해 해당 인수의 별칭을 만드는데, 이는 반드시 변수여야 합니다. 즉, 매개 변수에 대한 모든 작업이 인수에서 수행됩니다. 이러한 방식은 `ref` 키워드와 비슷합니다. 단, `ref`의 경우에는 변수를 전달하기 전에 초기화해야 합니다. `in`이 호출된 메서드에서 인수 값 수정을 허용하지 않는 것을 제외하고 `in` 키워드와도 같습니다. `out` 매개 변수를 사용하려면 메서드 정의와 호출 메서드가 모두 명시적으로 `out` 키워드를 사용해야 합니다. 다음은 그 예입니다.

```
int initializeInMethod;
OutArgExample(out initializeInMethod);
Console.WriteLine(initializeInMethod);      // value is now 44

void OutArgExample(out int number)
{
    number = 44;
}
```

## NOTE

`out` 키워드를 제네릭 형식 매개 변수와 함께 사용하여 형식 매개 변수를 공변(covariant)으로 지정할 수도 있습니다. 이 컨텍스트에서 `out` 키워드를 사용하는 방법에 대한 자세한 내용은 [out\(제네릭 한정자\)](#)를 참조하세요.

`out` 인수로 전달되는 변수는 메서드 호출에서 전달되기 전에 초기화할 필요가 없지만 호출된 메서드는 메서드가 반환되기 전에 값을 할당해야 합니다.

`in`, `ref` 및 `out` 키워드는 오버로드 해결을 위한 메서드 시그니처의 일부로 간주되지 않습니다. 따라서 메서드 하나는 `ref` 또는 `in` 인수를 사용하고 다른 하나는 `out` 인수를 사용한다는 것 외에는 차이점이 없으면 메서드를 오버로드할 수 없습니다. 예를 들어 다음 코드는 컴파일되지 않습니다.

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

그러나 다음과 같이 메서드 하나는 `ref`, `in` 또는 `out` 인수를 사용하고 다른 하나는 해당 한정자를 갖지 않는 경우에는 오버로드를 수행할 수 있습니다.

```
class OutOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(out int i) => i = 5;
}
```

컴파일러는 메서드 호출에 사용되는 매개 변수 한정자에 호출 사이트에서 매개 변수 한정자를 일치하여 적합한 오버로드를 선택합니다.

속성은 변수가 아니므로 `out` 매개 변수로 전달할 수 없습니다.

다음과 같은 종류의 메서드에는 `in`, `ref` 및 `out` 키워드를 사용할 수 없습니다.

- `async` 한정자를 사용하여 정의하는 비동기 메서드
- `yield return` 또는 `yield break` 문을 포함하는 반복기 메서드

또한 **확장 메서드**에는 다음과 같은 제한 사항이 있습니다.

- 확장 메서드의 첫 번째 인수에는 `out` 키워드를 사용할 수 없습니다.
- 인수가 구조체가 아니거나 구조체로 제한되지 않는 제네릭 형식일 경우에는 확장 메서드의 첫 번째 인수에 `ref` 키워드를 사용할 수 없습니다.
- 첫 번째 인수가 구조체인 경우 이외에는 `in` 키워드를 사용할 수 없습니다. 구조체로 제한되는 경우에도 `in` 키워드는 제네릭 형식에 사용할 수 없습니다.

## `out` 매개 변수 선언

`out` 인수를 사용하여 메서드를 선언하는 것은 여러 값을 반환하기 위한 일반적인 해결 방법입니다. C# 7.0부터 비슷한 시나리오에 대해 **값 튜플**을 고려하세요. 다음 예제에서는 `out`을 사용하여 단일 메서드 호출로 3개 변수를 반환합니다. 세 번째 인수는 `null`에 할당됩니다. 따라서 메서드가 값을 선택적으로 반환할 수 있습니다.

```
void Method(out int answer, out string message, out string stillNull)
{
    answer = 44;
    message = "I've been returned";
    stillNull = null;
}

int argNumber;
string argMessage, argDefault;
Method(out argNumber, out argMessage, out argDefault);
Console.WriteLine(argNumber);
Console.WriteLine(argMessage);
Console.WriteLine(argDefault == null);

// The example displays the following output:
//      44
//      I've been returned
//      True
```

## `out` 인수를 사용하여 메서드 호출

C# 6 및 이전 버전에서는 `out` 인수로 전달하기 전에 별도 문에서 변수를 선언해야 합니다. 다음 예제에서는 `Int32.TryParse` 메서드에 전달되기 전에 `number`라는 변수를 선언합니다. 이 메서드는 문자열을 숫자로 변환하려고 합니다.

```
string numberAsString = "1640";

int number;
if (Int32.TryParse(numberAsString, out number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//      Converted '1640' to 1640
```

C# 7.0부터 별도 변수 선언이 아니라 메서드 호출의 인수 목록에서 `out` 변수를 선언할 수 있습니다. 이렇게 하면 보다 간결하고 읽기 쉬운 코드가 생성되며 메서드 호출 전에 실수로 변수에 값이 할당되는 경우를 방지할 수

있습니다. 다음 예제는 [Int32.TryParse](#) 메서드 호출에서 `number` 변수를 정의한다는 점을 제외하고 이전 예제와 비슷합니다.

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out int number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//     Converted '1640' to 1640
```

앞의 예제에서 `number` 변수는 `int`로 강력하게 형식화됩니다. 다음 예제와 같이 암시적 형식 지역 변수를 선언할 수도 있습니다.

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out var number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//     Converted '1640' to 1640
```

## 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# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [메서드 매개 변수](#)

# namespace(C# 참조)

2020-11-02 • 3 minutes to read • [Edit Online](#)

`namespace` 키워드는 관련 개체 집합을 포함하는 범위를 선언하는데 사용됩니다. 네임스페이스를 사용하여 코드 요소를 구성하고 전역적으로 고유한 형식을 만들 수 있습니다.

```
namespace SampleNamespace
{
    class SampleClass { }

    interface ISampleInterface { }

    struct SampleStruct { }

    enum SampleEnum { a, b }

    delegate void SampleDelegate(int i);

    namespace Nested
    {
        class SampleClass2 { }
    }
}
```

## 설명

네임스페이스 내에서 다음 형식 중 0개 이상을 선언할 수 있습니다.

- 다른 네임스페이스
- `class`
- `interface`
- `struct`
- `enum`
- `delegate`

C# 소스 파일에서 네임스페이스를 명시적으로 선언하는지 여부에 관계없이 컴파일러는 기본 네임스페이스를 추가합니다. 전역 네임스페이스라고도 하는 이 명명되지 않은 네임스페이스는 모든 파일에 있습니다. 전역 네임스페이스의 모든 식별자는 명명된 네임스페이스에서 사용할 수 있습니다.

네임스페이스는 암시적으로 공용 액세스를 사용하며 이 설정은 수정할 수 없습니다. 네임스페이스의 요소에 할당할 수 있는 액세스 한정자에 대한 설명은 [액세스 한정자](#)를 참조하세요.

둘 이상의 선언에서 네임스페이스를 정의할 수 있습니다. 예를 들어 다음 예제에서는 `MyCompany` 네임스페이스의 일부로 두 클래스를 정의합니다.

```
namespace MyCompany.Proj1
{
    class MyClass
    {
    }
}

namespace MyCompany.Proj1
{
    class MyClass1
    {
    }
}
```

## 예제

다음 예제에서는 중첩된 네임스페이스에서 정적 메서드를 호출하는 방법을 보여 줍니다.

```
namespace SomeNameSpace
{
    public class MyClass
    {
        static void Main()
        {
            Nested.NestedNameSpaceClass.SayHello();
        }
    }

    // a nested namespace
    namespace Nested
    {
        public class NestedNameSpaceClass
        {
            public static void SayHello()
            {
                Console.WriteLine("Hello");
            }
        }
    }
}

// Output: Hello
```

## C# 언어 사양

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

## 참고 항목

- [C# 참조](#)
- [C# 키워드](#)
- [using](#)
- [Static 사용](#)
- [네임스페이스 별칭 한정자 ::](#)
- [네임스페이스](#)

# using(C# 참조)

2021-02-18 • 2 minutes to read • [Edit Online](#)

`using` 키워드에는 다음과 같은 세 가지 주요 사례가 있습니다.

- [using 문](#)은 개체가 삭제될 끝에서 범위를 정의합니다.
- [using 지시문](#)은 네임스페이스의 별칭을 만들거나 네임스페이스에 정의된 형식을 가져옵니다.
- [using static 지시문](#)은 단일 클래스의 멤버를 가져옵니다.

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [네임스페이스](#)
- [extern](#)

# using 지시문(C# 참조)

2021-02-18 • 7 minutes to read • [Edit Online](#)

`using` 지시문에는 다음 세 가지 용도가 있습니다.

- 네임스페이스에서 형식 사용을 한정할 필요가 없도록 해당 네임스페이스에서 형식 사용을 허용합니다.

```
using System.Text;
```

- 형식 이름을 사용하여 액세스를 한정할 필요 없이 형식의 정적 멤버 및 종첩 형식에 액세스하도록 허용합니다.

```
using static System.Math;
```

자세한 내용은 [using 정적 지시문](#)을 참조하세요.

- 네임스페이스 또는 형식에 대한 별칭을 만듭니다. 이를 *using 별칭 지시문*이라고 합니다.

```
using Project = PC.MyCompany.Project;
```

`using` 키워드는 파일 및 글꼴과 같은 [IDisposable](#) 개체가 제대로 처리될 수 있게 도와주는 *using 문*을 만드는데도 사용됩니다. 자세한 내용은 [using 문](#)을 참조하세요.

## 정적 형식 사용

형식 이름을 사용하여 액세스를 한정할 필요 없이 형식의 정적 멤버에 액세스할 수 있습니다.

```
using static System.Console;
using static System.Math;
class Program
{
    static void Main()
    {
        WriteLine(Sqrt(3*3 + 4*4));
    }
}
```

## 설명

`using` 지시문의 범위는 지시문이 나타내는 파일로 제한됩니다.

`using` 지시문이 나타날 수 있습니다.

- 모든 네임스페이스 또는 형식 정의 전에 소스 파일을 시작합니다.
- 모든 네임스페이스에 있지만 이 네임스페이스에 선언된 네임스페이스 또는 형식 앞에 있어야 합니다.

이렇게 하지 않으면 컴파일러 오류 [CS1529](#)가 생성됩니다.

`using` 별칭 지시문을 만들면 네임스페이스 또는 형식에 대한 식별자를 더 쉽게 한정할 수 있습니다. 모든 `using` 지시문에서 앞에 오는 `using` 지시문에 관계없이 정규화된 네임스페이스 또는 형식을 사용해야 합니다. `using` 별칭은 `using` 지시문 선언에 사용할 수 없습니다. 예를 들어, 다음은 컴파일러 오류를 생성합니다.

```
using s = System.Text;
using s.RegularExpressions; // Generates a compiler error.
```

`using` 지시문을 만들어서 네임스페이스를 지정할 필요 없이 네임스페이스에서 이 형식을 사용합니다. `using` 지시문은 지정한 네임스페이스에 중첩된 모든 네임스페이스에 대한 액세스 권한을 제공하지 않습니다.

네임스페이스는 두 가지 범주인 사용자 정의 및 시스템 정의로 구분됩니다. 사용자 정의 네임스페이스는 코드에서 정의된 네임스페이스입니다. 시스템 정의 네임스페이스 목록은 [.NET API 브라우저](#)를 참조하세요.

## 예 1

다음 예제에서는 `using` 네임스페이스에 대한 별칭을 정의 및 사용하는 방법을 보여 줍니다.

```
namespace PC
{
    // Define an alias for the nested namespace.
    using Project = PC.MyCompany.Project;
    class A
    {
        void M()
        {
            // Use the alias
            var mc = new Project.MyClass();
        }
    }
    namespace MyCompany
    {
        namespace Project
        {
            public class MyClass { }
        }
    }
}
```

`using alias` 지시문의 오른쪽에는 공개 제네릭 형식이 포함될 수 없습니다. 예를 들어 `List<T>`에 대한 별칭을 만들 수 없지만 `List<int>`에 대해서는 별칭을 만들 수 있습니다.

## 예제 2

다음 예제에서는 클래스에 대한 `using` 지시문 및 `using` 별칭을 정의하는 방법을 보여 줍니다.

```

using System;

// Using alias directive for a class.
using AliasToMyClass = NameSpace1.MyClass;

// Using alias directive for a generic class.
using UsingAlias = NameSpace2.MyClass<int>;

namespace NameSpace1
{
    public class MyClass
    {
        public override string ToString()
        {
            return "You are in NameSpace1.MyClass.";
        }
    }
}

namespace NameSpace2
{
    class MyClass<T>
    {
        public override string ToString()
        {
            return "You are in NameSpace2.MyClass.";
        }
    }
}

namespace NameSpace3
{
    class MainClass
    {
        static void Main()
        {
            var instance1 = new AliasToMyClass();
            Console.WriteLine(instance1);

            var instance2 = new UsingAlias();
            Console.WriteLine(instance2);
        }
    }
}
// Output:
//    You are in NameSpace1.MyClass.
//    You are in NameSpace2.MyClass.

```

## C# 언어 사양

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

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [네임스페이스 사용](#)
- [C# 키워드](#)
- [네임스페이스](#)
- [using 문](#)

# using 정적 지시문(C# 참조)

2020-11-02 • 8 minutes to read • [Edit Online](#)

`using static` 지시문은 형식 이름을 지정하지 않고 정적 멤버 및 중첩 형식에 액세스할 수 있는 형식을 지정합니다. 사용되는 구문은 다음과 같습니다.

```
using static <fully-qualified-type-name>;
```

여기서 *fully-qualified-type-name*은 형식 이름을 지정하지 않고 정적 멤버 및 중첩 형식을 참조할 수 있는 형식의 이름입니다. 정규화된 형식 이름(전체 네임스페이스 및 형식 이름)을 제공하지 않으면 C#에 컴파일러 오류 CS0246: “‘type/namespace’ 형식 또는 네임스페이스 이름을 찾을 수 없습니다(사용 중인 지시문 또는 어셈블리 참조가 없습니까?)”가 발생합니다.

`using static` 지시문은 정적 멤버(또는 중첩 형식)가 있는 모든 형식에 적용됩니다(인스턴스 멤버가 있는 경우에도). 그러나 인스턴스 멤버는 형식 인스턴스를 통해서만 호출할 수 있습니다.

`using static` 지시문은 C# 6에서 도입되었습니다.

## 설명

일반적으로 정적 멤버를 호출할 때 멤버 이름과 함께 형식 이름을 제공합니다. 형식의 멤버를 호출하기 위해 동일한 형식 이름을 반복해서 입력하면 코드가 복잡하고 난해해질 수 있습니다. 예를 들어 `Circle` 클래스에 대한 다음 정의에서는 `Math` 클래스의 많은 멤버를 참조합니다.

```
using System;

public class Circle
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Radius { get; set; }

    public double Diameter
    {
        get { return 2 * Radius; }
    }

    public double Circumference
    {
        get { return 2 * Radius * Math.PI; }
    }

    public double Area
    {
        get { return Math.PI * Math.Pow(Radius, 2); }
    }
}
```

멤버를 참조할 때마다 `Math` 클래스를 명시적으로 참조할 필요가 없으므로 `using static` 지시문은 훨씬 깔끔한 코드를 생성합니다.

```

using System;
using static System.Math;

public class Circle
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Radius { get; set; }

    public double Diameter
    {
        get { return 2 * Radius; }
    }

    public double Circumference
    {
        get { return 2 * Radius * PI; }
    }

    public double Area
    {
        get { return PI * Pow(Radius, 2); }
    }
}

```

`using static` 은 액세스 가능한 정적 멤버와 지정된 형식에 선언된 중첩된 형식만 가져옵니다. 상속된 멤버는 가져오지 않습니다. `using static` 지시문을 사용하여 Visual Basic 모듈을 포함한 모든 명명된 형식에서 가져올 수 있습니다. F# 최상위 함수가 메타데이터에서 이름이 유효한 C# 식별자인 명명된 형식의 정적 멤버로 나타나면 F# 함수를 가져올 수 있습니다.

`using static` 을 사용하면 지정된 형식에 선언된 확장 메서드를 확장 메서드 조회에 사용할 수 있습니다. 그러나 확장명 메서드의 이름은 코드의 정규화되지 않은 참조에 대한 범위로 가져오지 않습니다.

같은 컴파일 단위 또는 네임스페이스에서 여러 `using static` 지시문을 통해 다양한 형식에서 가져온 같은 이름을 사용하는 메서드는 메서드 그룹을 구성합니다. 이들 메서드 그룹 내에서 오버로드 확인은 일반 C# 규칙을 따릅니다.

## 예제

다음 예제에서는 형식 이름을 지정할 필요 없이 `using static` 지시문을 사용하여 `Console`, `Math` 및 `String` 클래스의 정적 멤버를 사용 가능하게 합니다.

```

using System;
using static System.Console;
using static System.Math;
using static System.String;

class Program
{
    static void Main()
    {
        Write("Enter a circle's radius: ");
        var input = ReadLine();
        if (!IsNullOrEmpty(input) && double.TryParse(input, out var radius)) {
            var c = new Circle(radius);

            string s = "\nInformation about the circle:\n";
            s = s + Format("    Radius: {0:N2}\n", c.Radius);
            s = s + Format("    Diameter: {0:N2}\n", c.Diameter);
            s = s + Format("    Circumference: {0:N2}\n", c.Circumference);
            s = s + Format("    Area: {0:N2}\n", c.Area);
            WriteLine(s);
        }
        else {
            WriteLine("Invalid input...");
        }
    }
}

public class Circle
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public double Radius { get; set; }

    public double Diameter
    {
        get { return 2 * Radius; }
    }

    public double Circumference
    {
        get { return 2 * Radius * PI; }
    }

    public double Area
    {
        get { return PI * Pow(Radius, 2); }
    }
}

// The example displays the following output:
//      Enter a circle's radius: 12.45
//
//      Information about the circle:
//          Radius: 12.45
//          Diameter: 24.90
//          Circumference: 78.23
//          Area: 486.95

```

이 예제에서는 `using static` 지시문을 `Double` 형식에 적용했을 수도 있습니다. 이 경우 형식 이름을 지정하지 않고도 `TryParse(String, Double)` 메서드를 호출할 수 있습니다. 그러나 이렇게 하면 코드 가독성이 떨어집니다. 어떤 숫자 형식의 `TryParse` 메서드가 호출되었는지 알아보기 위해 `using static` 지시문을 확인해야 하기 때문입니다.

## 참조

- [using 지시문](#)
- [C# 참조](#)
- [C# 키워드](#)
- [네임스페이스 사용](#)
- [네임스페이스](#)

# using 문(C# 참조)

2021-02-18 • 9 minutes to read • [Edit Online](#)

`IDisposable` 개체의 올바른 사용을 보장하는 편리한 구문을 제공합니다. C# 8.0부터 `using` 문은 `IAsyncDisposable` 개체의 올바른 사용을 보장합니다.

## 예제

다음 예제에서는 `using` 문을 사용하는 방법을 보여 줍니다.

```
string manyLines=@"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

using (var reader = new StringReader(manyLines))
{
    string? item;
    do {
        item = reader.ReadLine();
        Console.WriteLine(item);
    } while(item != null);
}
```

C# 8.0부터는 중괄호가 필요하지 않은 `using` 문에 다음 대체 구문을 사용할 수 있습니다.

```
string manyLines=@"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

using var reader = new StringReader(manyLines);
string? item;
do {
    item = reader.ReadLine();
    Console.WriteLine(item);
} while(item != null);
```

## 설명

`File` 및 `Font`는 관리되지 않는 리소스에 액세스하는 관리되는 형식의 예입니다(이 경우 파일 핸들 및 디바이스 컨텍스트). 다른 많은 종류의 관리되지 않는 리소스 및 이를 캡슐화하는 클래스 라이브러리 형식이 있습니다. 해당 형식은 모두 `IDisposable` 인터페이스 또는 `IAsyncDisposable` 인터페이스를 구현해야 합니다.

`IDisposable` 개체의 수명이 단일 메서드로 제한된 경우 `using` 문에서 선언하고 인스턴스화해야 합니다.

`using` 문은 올바른 방법으로 개체에서 `Dispose` 메서드를 호출하며, (앞서 설명한 대로 사용하는 경우)

`Dispose`가 호출되자마자 개체 자체가 벗어나도록 만듭니다. `using` 블록 내에서 개체는 읽기 전용이며 수정하거나 다시 할당할 수 없습니다. 개체가 `IDisposable` 대신 `IAsyncDisposable`을 구현하는 경우 `using` 문은 `DisposeAsync`를 호출하고 반환된 `ValueTask`를 `awaits` 합니다. `IAsyncDisposable`에 대한 자세한 내용은 `DisposeAsync` 메서드 구현을 참조하세요.

`using` 문은 `using` 블록 내에서 예외가 발생하더라도 `Dispose`(또는 `DisposeAsync`)가 호출되도록 합니다. `try`

블록 내부에 개체를 배치한 다음, `finally` 블록에서 `Dispose`(또는 `DisposeAsync`)를 호출해도 동일한 결과를 얻을 수 있습니다. 실제로 컴파일러는 이 방법으로 `using` 문을 변환합니다. 이전의 코드 예제는 컴파일 시 다음 코드로 확장됩니다(개체의 제한된 범위를 만드는 여러분의 중괄호 참고).

```
string manyLines=@"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

{
    var reader = new StringReader(manyLines);
    try {
        string? item;
        do {
            item = reader.ReadLine();
            Console.WriteLine(item);
        } while(item != null);
    } finally
    {
        reader?.Dispose();
    }
}
```

최신 `using` 문 구문은 유사한 코드로 변환됩니다. 변수가 선언된 위치에서 `try` 블록이 열립니다. `finally` 블록은 일반적으로 메서드의 끝에 있는 바깥쪽 블록의 가까이에 추가됩니다.

`try - finally` 문에 대한 자세한 내용은 [try-finally](#) 문서를 참조하세요.

다음 예제와 같이 단일 `using` 문에서 한 형식의 여러 인스턴스를 선언할 수 있습니다. 단일 문에서 여러 변수를 선언하는 경우에는 암시적으로 형식화된 변수(`var`)를 사용할 수 없습니다.

```
string numbers=@"One
Two
Three
Four.";
string letters=@"A
B
C
D.";

using (StringReader left = new StringReader(numbers),
       right = new StringReader(letters))
{
    string? item;
    do {
        item = left.ReadLine();
        Console.WriteLine(item);
        Console.Write("    ");
        item = right.ReadLine();
        Console.WriteLine(item);
    } while(item != null);
}
```

다음 예제와 같이 C# 8에 도입된 새로운 구문을 사용하여 동일한 형식의 여러 선언을 결합할 수 있습니다.

```

string numbers=@"One
Two
Three
Four.";
string letters=@"A
B
C
D.";

using StringReader left = new StringReader(numbers),
      right = new StringReader(letters);
string? item;
do {
    item = left.ReadLine();
    Console.WriteLine(item);
    Console.WriteLine("    ");
    item = right.ReadLine();
    Console.WriteLine(item);
} while(item != null);

```

리소스 개체를 인스턴스화한 후 `using` 문에 변수를 전달할 수 있지만, 이 방법이 최선은 아닙니다. 이 경우 컨트롤이 `using` 블록을 벗어난 후에도 개체가 범위 내에 있지만, 관리되지 않는 리소스에 액세스하지 못할 수 있습니다. 즉, 더 이상 완전히 초기화되지 않습니다. `using` 블록 외부에서 개체를 사용하려고 하면 예외가 `throw`될 위험이 있습니다. 따라서 `using` 문에서 개체를 인스턴스화하고 `using` 블록에서 범위를 제한하는 것이 좋습니다.

```

string manyLines=@"This is line one
This is line two
Here is line three
The penultimate line is line four
This is the final, fifth line.";

var reader = new StringReader(manyLines);
using (reader)
{
    string? item;
    do {
        item = reader.ReadLine();
        Console.WriteLine(item);
    } while(item != null);
}
// reader is in scope here, but has been disposed

```

`IDisposable` 개체를 삭제하는 방법에 대한 자세한 내용은 [IDisposable를 구현하는 개체 사용](#)을 참조하세요.

## C# 언어 사양

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

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [using 지시문](#)
- [가비지 수집](#)
- [IDisposable를 구현하는 개체 사용](#)

- IDisposable 인터페이스
- C# 8.0의 using 문

# extern alias(C# 참조)

2020-11-02 • 4 minutes to read • [Edit Online](#)

정규화된 형식 이름이 동일한 어셈블리의 두 버전을 참조해야 할 수 있습니다. 예를 들어 동일한 애플리케이션에서 어셈블리 버전을 두 개 이상 사용해야 할 수 있습니다. 외부 어셈블리 별칭을 사용하면 각 어셈블리의 네임스페이스를 별칭으로 명명된 루트 수준 네임스페이스 내에서 래핑하여 동일한 파일에서 사용하도록 할 수 있습니다.

## NOTE

`extern` 키워드는 메서드 한정자로도 사용되어 비관리 코드로 작성된 메서드를 선언합니다.

정규화된 형식 이름이 동일한 두 어셈블리를 참조하려면 다음과 같이 명령 프롬프트에서 별칭을 지정해야 합니다.

```
/r:GridV1=grid.dll
```

```
/r:GridV2=grid20.dll
```

그러면 외부 별칭 `GridV1` 및 `GridV2`가 생성됩니다. 프로그램 내에서 이러한 별칭을 사용하려면 `extern` 키워드를 사용하여 참조합니다. 예를 들어 다음과 같습니다.

```
extern alias GridV1;
```

```
extern alias GridV2;
```

각 `extern alias` 선언에서는 전역 네임스페이스와 병렬이지만 전역 네임스페이스 내에 있지 않는 추가 루트 수준 네임스페이스를 소개합니다. 따라서 각 어셈블리의 형식은 적절한 namespace-alias에서 시작되는 정규화된 이름을 사용하여 명확하게 참조할 수 있습니다.

이전 예제에서 `GridV1::Grid`는 `grid.dll`의 표 컨트롤이 되고 `GridV2::Grid`는 `grid20.dll`의 표 컨트롤이 됩니다.

## Visual Studio 사용

Visual Studio를 사용하는 경우 별칭을 비슷한 방식으로 제공할 수 있습니다.

Visual Studio의 프로젝트에 `grid.dll` 및 `grid20.dll`의 참조를 추가합니다. 속성 탭을 열고 별칭을 전역에서 `GridV1` 및 `GridV2`로 각각 변경합니다.

다음 별칭을 위와 동일한 방식으로 사용합니다.

```
extern alias GridV1;  
extern alias GridV2;
```

이제 별칭 지시문을 사용하여 네임스페이스 또는 형식에 대한 별칭을 만들 수 있습니다. 자세한 내용은 [지시문 사용](#)을 참조하세요.

```
using Class1V1 = GridV1::Namespace.Class1;  
using Class1V2 = GridV2::Namespace.Class1;
```

## 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# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [:: 연산자](#)
- [-reference\(C# 컴파일러 옵션\)](#)

# is(C# 참조)

2021-02-18 • 16 minutes to read • [Edit Online](#)

`is` 연산자는 식의 결과가 지정된 형식과 호환되는지 확인하거나, (C# 7.0부터) 패턴에 대해 식을 테스트합니다. 형식 테스트 `is` 연산자에 대한 자세한 내용은 [형식 테스트 및 캐스트 연산자](#) 문서의 `is` 연산자 섹션을 참조하세요.

## `is`를 사용한 패턴 일치

C# 7.0부터는 `is` 및 `switch` 문에서 패턴 일치를 지원합니다. `is` 키워드는 다음과 같은 패턴을 지원합니다.

- **형식 패턴** - 식을 지정된 형식으로 변환할 수 있는지를 테스트하고, 변환할 수 있으면 해당 형식의 변수로 변수를 캐스트합니다.
- **상수 패턴** - 식이 지정된 상수 값으로 평가되는지 여부를 테스트합니다.
- **var 패턴** - 항상 성공하고 식의 값을 새 로컬 변수에 바인딩하는 일치입니다.

### 형식 패턴

형식 패턴을 사용하여 패턴 일치를 수행하는 경우 `is`는 식을 지정된 형식으로 변환할 수 있는지 여부를 테스트하고, 변환할 수 있으면 해당 형식의 변수로 캐스트합니다. 간결한 형식 평가 및 변환을 사용하는 `is` 문의 간단한 확장입니다. `is` 형식 패턴의 일반적인 형식은 다음과 같습니다.

```
expr is type varname
```

여기서 `expr`은 일부 형식의 인스턴스로 평가되는 식이고 `type`은 `expr`의 결과가 변환될 형식의 이름이며 `varname`은 `is` 테스트가 `true`인 경우 `expr`의 결과변환가 되는 개체입니다.

`expr`이 `null`이 아니고 다음 조건 중 하나가 `true`일 경우 `is` 식은 `true`입니다.

- `expr`이 `type`과 동일한 형식의 인스턴스입니다.
- `expr`이 `type`에서 파생된 형식의 인스턴스입니다. 즉, `expr`의 결과를 `type`의 인스턴스로 업캐스트할 수 있습니다.
- `expr`의 컴파일 시간 형식은 `type`의 기본 클래스이고 `expr`의 런타임 형식은 `type`이거나 `type`에서 파생됩니다. 변수의 컴파일 시간 형식은 해당 선언에 정의된 변수의 형식입니다. 변수의 런타임 형식은 해당 변수에 할당된 인스턴스의 형식입니다.
- `expr`이 `type` 인터페이스를 구현하는 형식의 인스턴스입니다.

C# 7.1부터 `expr`은 제네릭 형식 매개 변수 및 해당 제약 조건을 통해 컴파일 시간 형식을 정의할 수 있습니다.

`expr`이 `true`이고 `is` 가 `if` 문에서 사용되는 경우 `varname`이 `if` 문 내에서만 할당됩니다. `varname`의 범위는 `is` 식에서부터 `if` 문을 닫는 블록의 끝까지입니다. 다른 위치에서 `varname`을 사용하면 할당되지 않은 변수 사용에 대해 컴파일 시간 오류가 생성됩니다.

다음 예제에서는 `is` 형식 패턴을 사용하여 형식의 `IComparable.CompareTo(Object)` 메서드 구현을 제공합니다.

```
using System;

public class Employee : IComparable
{
    public String Name { get; set; }
    public int Id { get; set; }

    public int CompareTo(Object o)
    {
        if (o is Employee e)
        {
            return Name.CompareTo(e.Name);
        }
        throw new ArgumentException("o is not an Employee object.");
    }
}
```

패턴 일치를 사용하지 않을 경우 이 코드를 다음과 같이 작성할 수 있습니다. 형식 패턴 일치를 사용하면 변환 결과가 `null` 인지 여부를 테스트할 필요가 없으므로 보다 간결하고 읽기 쉬운 코드가 생성됩니다.

```
using System;

public class Employee : IComparable
{
    public String Name { get; set; }
    public int Id { get; set; }

    public int CompareTo(Object o)
    {
        var e = o as Employee;
        if (e == null)
        {
            throw new ArgumentException("o is not an Employee object.");
        }
        return Name.CompareTo(e.Name);
    }
}
```

또한 `is` 형식 패턴은 값 형식의 형식을 확인할 때 보다 간결한 코드를 생성합니다. 다음 예제에서는 `is` 형식 패턴을 사용하여 개체가 `Person` 인스턴스인지 `Dog` 인스턴스인지를 확인한 후 적절한 속성의 값을 표시합니다.

```
using System;

public class Example
{
    public static void Main()
    {
        Object o = new Person("Jane");
        ShowValue(o);

        o = new Dog("Alaskan Malamute");
        ShowValue(o);
    }

    public static void ShowValue(object o)
    {
        if (o is Person p) {
            Console.WriteLine(p.Name);
        }
        else if (o is Dog d) {
            Console.WriteLine(d.Breed);
        }
    }
}

public struct Person
{
    public string Name { get; set; }

    public Person(string name) : this()
    {
        Name = name;
    }
}

public struct Dog
{
    public string Breed { get; set; }

    public Dog(string breedName) : this()
    {
        Breed = breedName;
    }
}
// The example displays the following output:
// Jane
// Alaskan Malamute
```

패턴 일치를 사용하지 않는 동일한 코드에는 명시적 캐스트를 포함하는 별도의 할당이 필요합니다.

```

using System;

public class Example
{
    public static void Main()
    {
        Object o = new Person("Jane");
        ShowValue(o);

        o = new Dog("Alaskan Malamute");
        ShowValue(o);
    }

    public static void ShowValue(object o)
    {
        if (o is Person) {
            Person p = (Person) o;
            Console.WriteLine(p.Name);
        }
        else if (o is Dog) {
            Dog d = (Dog) o;
            Console.WriteLine(d.Breed);
        }
    }
}

public struct Person
{
    public string Name { get; set; }

    public Person(string name) : this()
    {
        Name = name;
    }
}

public struct Dog
{
    public string Breed { get; set; }

    public Dog(string breedName) : this()
    {
        Breed = breedName;
    }
}

// The example displays the following output:
//      Jane
//      Alaskan Malamute

```

## 상수 패턴

상수 패턴을 사용한 패턴 일치를 수행하는 경우 `is` 는 식이 지정된 상수와 같은지 여부를 테스트합니다. C# 6 이전 버전에서는 상수 패턴이 `switch` 문에서 지원됩니다. C# 7.0부터는 `is` 문에서도 지원됩니다. 구문은 다음과 같습니다.

```
expr is constant
```

여기서 `expr`은 평가할 식이고 `constant`은 테스트할 값입니다. `constant`은 다음 상수 식 중 하나가 될 수 있습니다.

- 리터럴 값입니다.
- 선언된 `const` 변수의 이름

- 열거형 상수

상수 식은 다음과 같이 계산됩니다.

- *expr* 및 *constant*가 정수 형식인 경우 C# 같은 연산자는 식에서 `true`를 반환하는지 여부 즉, `expr == constant`인지 여부를 확인합니다.
- 정수 형식이 아니면 static `Object.Equals(expr, constant)` 메서드 호출을 통해 식의 값이 결정됩니다.

다음 예제에서는 형식 패턴과 상수 패턴을 결합하여 개체가 `Dice` 인스턴스인지 여부를 테스트하고, 그럴 경우 주사위 굴리기 값이 6인지 여부를 확인합니다.

```
using System;

public class Dice
{
    Random rnd = new Random();
    public Dice()
    {
    }
    public int Roll()
    {
        return rnd.Next(1, 7);
    }
}

class Program
{
    static void Main(string[] args)
    {
        var d1 = new Dice();
        ShowValue(d1);
    }

    private static void ShowValue(object o)
    {
        const int HIGH_ROLL = 6;

        if (o is Dice d && d.Roll() is HIGH_ROLL)
            Console.WriteLine($"The value is {HIGH_ROLL}!");
        else
            Console.WriteLine($"The dice roll is not a {HIGH_ROLL}!");
    }
}
// The example displays output like the following:
//      The value is 6!
```

상수 패턴을 사용하여 `null`을 검사할 수 있습니다. `null` 키워드는 `is` 문에서 지원됩니다. 구문은 다음과 같습니다.

```
expr is null
```

다음 예제는 `null` 검사의 비교를 보여 줍니다.

```

using System;

class Program
{
    static void Main(string[] args)
    {
        object o = null;

        if (o is null)
        {
            Console.WriteLine("o does not have a value");
        }
        else
        {
            Console.WriteLine($"o is {o}");
        }

        int? x = 10;

        if (x is null)
        {
            Console.WriteLine("x does not have a value");
        }
        else
        {
            Console.WriteLine($"x is {x.Value}");
        }

        // 'null' check comparison
        Console.WriteLine($"'is' constant pattern 'null' check result : { o is null }");
        Console.WriteLine($"object.ReferenceEquals 'null' check result : { object.ReferenceEquals(o, null) }");
        Console.WriteLine($"Equality operator (==) 'null' check result : { o == null }");
    }

    // The example displays the following output:
    // o does not have a value
    // x is 10
    // 'is' constant pattern 'null' check result : True
    // object.ReferenceEquals 'null' check result : True
    // Equality operator (==) 'null' check result : True
}

```

`x is null` 식은 참조 형식 및 null 허용 값 형식에 대해 다르게 계산됩니다. null 허용 값 형식의 경우 [Nullable<T>.HasValue](#)를 사용합니다. 참조 형식의 경우 `(object)x == null`을 사용합니다.

### var 패턴

`var` 패턴을 사용한 패턴 일치는 항상 성공합니다. 구문은 다음과 같습니다.

```
expr is var varname
```

여기서 `expr`의 값은 항상 `varname`이라는 지역 변수에 할당됩니다. `varname`은 컴파일 시간 형식의 `expr`과 동일한 형식의 변수입니다.

`expr`이 `null`로 평가되는 경우 `is` 식은 `true`를 생성하고 `varname`에 `null`을 할당합니다. `var` 패턴은 `null` 값에 대해 `true`를 생성하는 흔치 않은 `is` 용도 중 하나입니다.

다음 예와 같이 `var` 패턴을 사용하여 부울 식 내에 임시 변수를 만들 수 있습니다.

```

using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    static void Main()
    {
        int[] testSet = { 100271, 234335, 342439, 999683 };

        var primes = testSet.Where(n => Factor(n).ToList().Count == 2
                               && !primes.Contains(1)
                               && !primes.Contains(n));

        foreach (int prime in primes)
        {
            Console.WriteLine($"Found prime: {prime}");
        }
    }

    static IEnumerable<int> Factor(int number)
    {
        int max = (int)Math.Sqrt(number);
        for (int i = 1; i <= max; i++)
        {
            if (number % i == 0)
            {
                yield return i;
                if (i != number / i)
                {
                    yield return number / i;
                }
            }
        }
    }
}

// The example displays the following output:
//      Found prime: 100271
//      Found prime: 999683

```

앞의 예에서 임시 변수는 비용이 많이 드는 작업의 결과를 저장하는 데 사용됩니다. 그런 다음 변수를 여러 번 사용할 수 있습니다.

## C# 언어 사양

자세한 내용은 C# 언어 사양의 [s 연산자](#) 섹션과 다음 C# 언어 제안을 참조하세요.

- [패턴 일치](#)
- [제네릭과 패턴 일치](#)

## 참고 항목

- [C# 참조](#)
- [C# 키워드](#)
- [형식 테스트 및 캐스트 연산자](#)

# new 제약 조건(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

**new** 제약 조건은 제네릭 클래스 선언의 형식 인수에 공용 매개 변수가 없는 생성자가 있어야 함을 지정합니다.

**new** 제약 조건을 사용하기 위해 유형을 추상화할 수 없습니다.

다음 예제와 같이 제네릭 클래스가 형식의 새 인스턴스를 만드는 경우 형식 매개 변수에 **new** 제약 조건을 적용합니다.

```
class ItemFactory<T> where T : new()
{
    public T GetNewItem()
    {
        return new T();
    }
}
```

다른 제약 조건과 함께 **new()** 제약 조건을 사용하는 경우 마지막에 지정해야 합니다.

```
public class ItemFactory2<T>
    where T : IComparable, new()
{ }
```

자세한 내용은 [형식 매개 변수에 대한 제약 조건](#)을 참조하세요.

**new** 키워드를 사용하여 [형식의 인스턴스를 만들거나 멤버 선언 한정자](#)로 만들 수도 있습니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 유형 매개 변수 제약 조건](#) 섹션을 참조하세요.

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)
- [제네릭](#)

# where(제네릭 형식 제약 조건)(C# 참조)

2020-11-02 • 12 minutes to read • [Edit Online](#)

제네릭 정의의 `where` 절은 제네릭 형식, 메서드, 대리자 또는 로컬 함수의 형식 매개 변수에 대한 인수로 사용되는 형식에 대한 제약 조건을 지정합니다. 제약 조건은 인터페이스, 기본 클래스를 지정하거나 제네릭 형식을 참조, 값 또는 관리되지 않는 형식으로 요구할 수 있습니다. 형식 인수에 포함해야 하는 기능을 선언합니다.

예를 들어 형식 매개 변수 `T` 가 `IComparable<T>` 인터페이스를 구현하도록 제네릭 클래스 `MyGenericClass` 를 선언할 수 있습니다.

```
public class AGenericClass<T> where T : IComparable<T> { }
```

## NOTE

쿼리 식의 `where` 절에 대한 자세한 내용은 [where 절](#)을 참조하세요.

`where` 절에는 기본 클래스 제약 조건이 포함될 수도 있습니다. 기본 클래스 제약 조건에서는 해당 제네릭 형식에 대한 형식 인수로 사용할 형식이 지정된 클래스를 기본 클래스로 포함하거나 해당 기본 클래스임을 명시합니다. 기본 클래스 제약 조건이 사용되는 경우 해당 형식 매개 변수에 대한 다른 제약 조건 앞에 나타나야 합니다. 일부 형식은 `Object`, `Array` 및 `ValueType` 기본 클래스 제약 조건으로 허용되지 않습니다. C# 7.3 이전에는 기본 클래스 제약 조건으로 `Enum`, `Delegate` 및 `MulticastDelegate`도 허용되지 않았습니다. 다음 예제에서는 이제 기본 클래스로 지정할 수 있는 형식을 보여 줍니다.

```
public class UsingEnum<T> where T : System.Enum { }

public class UsingDelegate<T> where T : System.Delegate { }

public class Multicaster<T> where T : System.MulticastDelegate { }
```

C# 8.0 이상의 null 허용 컨텍스트에서는 기본 클래스 형식의 null 허용 여부가 적용됩니다. 기본 클래스가 `null`을 허용하지 않는 경우(예: `Base`) 형식 인수는 `null`을 허용하지 않아야 합니다. 기본 클래스가 `null`을 허용하는 경우(예: `Base?`) 형식 인수는 `null`을 허용하거나 `null`을 허용하지 않는 참조 형식일 수 있습니다. 기본 클래스가 `null`을 허용하지 않는 경우 형식 인수가 `null` 허용 참조 형식이면 컴파일러가 경고를 발생시킵니다.

`where` 절은 형식이 `class` 또는 `struct`임을 지정할 수 있습니다. `struct` 제약 조건은 `System.ValueType` 의 기본 클래스 제약 조건을 지정할 필요가 없습니다. `System.ValueType` 형식은 기본 클래스 제약 조건으로 사용할 수 없습니다. 다음 예제에서는 `class` 및 `struct` 제약 조건을 모두 보여 줍니다.

```
class MyClass<T, U>
    where T : class
    where U : struct
{ }
```

C# 8.0 이상의 null 허용 컨텍스트에서 `class` 제약 조건을 사용하려면 형식이 `null`을 허용하지 않는 참조 형식이어야 합니다. `null` 허용 참조 형식을 허용하려면 `null` 허용 참조 형식 및 `null`을 허용하지 않는 참조 형식을 둘 다 허용하는 `class?` 제약 조건을 사용합니다.

`where` 절은 `notnull` 제약 조건을 포함할 수 있습니다. `notnull` 제약 조건은 형식 매개 변수를 `Null`을 허용하지 않는 형식으로 제한합니다. 해당 형식은 `값 형식`이거나 `nullable`이 아닌 참조 형식일 수 있습니다.

`#nullable enable` 컨텍스트에서 컴파일된 코드에 대해 C# 8.0부터 `notnull` 제약 조건을 사용할 수 있습니다. 다른 제약 조건과 달리 형식 인수가 `notnull` 제약 조건을 위반하면 컴파일러는 오류 대신 경고를 생성합니다. 경고는 `#nullable enable` 컨텍스트에서만 생성됩니다.

### IMPORTANT

`notnull` 제약 조건을 포함하는 제네릭 선언은 nullable 형식을 감지하지 않는 컨텍스트에서 사용될 수 있지만 컴파일러는 제약 조건을 적용하지 않습니다.

```
#nullable enable
class NotNullContainer<T>
    where T : notnull
{
}
(nullable restore
```

`where` 절에 `unmanaged` 제약 조건이 포함될 수도 있습니다. `unmanaged` 제약 조건은 형식 매개 변수를 [관리되지 않는 형식](#)으로 알려진 형식으로 제한합니다. `unmanaged` 제약 조건을 사용하면 C#에서 하위 수준의 interop 코드를 더 쉽게 작성할 수 있습니다. 이 제약 조건을 통해 모든 관리되지 않는 형식에서 재사용 가능한 루틴을 사용할 수 있습니다. `unmanaged` 제약 조건은 `class` 또는 `struct` 제약 조건과 결합할 수 없습니다. `unmanaged` 제약 조건에 따라 형식이 `struct`이어야 합니다.

```
class UnManagedWrapper<T>
    where T : unmanaged
{ }
```

`where` 절에 `new()` 생성자 제약 조건이 포함될 수도 있습니다. 이 제약 조건을 사용하면 `new` 연산자를 사용하여 형식 매개 변수의 인스턴스를 만들 수 있습니다. [new \(\) 제약 조건](#)을 사용하면 컴파일러에서 제공된 형식 인수에 액세스 가능하고 매개 변수가 없는 생성자가 있어야 한다는 것을 알게 됩니다. 예를 들어:

```
public class MyGenericClass<T> where T : IComparable<T>, new()
{
    // The following line is not possible without new() constraint:
    T item = new T();
}
```

`new()` 제약 조건은 `where` 절 맨 끝에 나타납니다. `new()` 제약 조건은 `struct` 또는 `unmanaged` 제약 조건과 결합할 수 없습니다. 이러한 제약 조건을 충족하는 모든 형식에는 매개 변수가 없는 액세스 가능 생성자가 있어야 하므로 `new()` 제약 조건이 중복됩니다.

형식 매개 변수가 여러 개이면 각 형식 매개 변수에 하나의 `where` 절을 사용합니다. 예를 들면 다음과 같습니다.

```
public interface IMyInterface { }

namespace CodeExample
{
    class Dictionary<TKey, TValue>
        where TKey : IComparable<TKey>
        where TValue : IMyInterface
    {
        public void Add(TKey key, TValue val) { }
    }
}
```

다음 예제와 같이 제약 조건은 제네릭 메서드의 형식 매개 변수에 연결할 수도 있습니다.

```
public void MyMethod<T>(T t) where T : IMyInterface { }
```

대리자에 대한 형식 매개 변수 제약 조건을 설명하는 구문은 메서드의 구문과 동일합니다.

```
delegate T MyDelegate<T>() where T : new();
```

제네릭 대리자에 대한 자세한 내용은 [제네릭 대리자](#)를 참조하세요.

제약 조건의 구문 및 사용에 대한 자세한 내용은 [형식 매개 변수에 대한 제약 조건](#)을 참조하세요.

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [제네릭 소개](#)
- [new 제약 조건](#)
- [형식 매개 변수에 대한 제약 조건](#)

# base(C# 참조)

2020-03-18 • 4 minutes to read • [Edit Online](#)

`base` 키워드는 파생 클래스 내에서 기본 클래스의 멤버에 액세스하는 데 사용됩니다.

- 다른 메서드에 의해 재정의된 기본 클래스에서 메서드를 호출합니다.
- 파생 클래스의 인스턴스를 만들 때 호출해야 하는 기본 클래스 생성자를 지정합니다.

기본 클래스 액세스는 생성자, 인스턴스 메서드 또는 인스턴스 속성 접근자에서만 허용됩니다.

정적 메서드 내에서 `base` 키워드를 사용하는 것은 오류입니다.

액세스된 기본 클래스는 클래스 선언에 지정된 기본 클래스입니다. 예를 들어 `class ClassB : ClassA`를 지정할 경우 ClassA의 기본 클래스에 관계없이 ClassA의 멤버는 ClassB에서 액세스됩니다.

## 예제

이 예제에서 기본 클래스 `Person` 및 파생 클래스 `Employee`에는 둘 다 `GetInfo` 메서드가 있습니다. `base` 키워드를 사용하면 파생 클래스 내에서 기본 클래스에 대해 `GetInfo` 메서드를 호출할 수 있습니다.

```
public class Person
{
    protected string ssn = "444-55-6666";
    protected string name = "John L. Malgraine";

    public virtual void GetInfo()
    {
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("SSN: {0}", ssn);
    }
}

class Employee : Person
{
    public string id = "ABC567EFG";
    public override void GetInfo()
    {
        // Calling the base class GetInfo method:
        base.GetInfo();
        Console.WriteLine("Employee ID: {0}", id);
    }
}

class TestClass
{
    static void Main()
    {
        Employee E = new Employee();
        E.GetInfo();
    }
}
/*
Output
Name: John L. Malgraine
SSN: 444-55-6666
Employee ID: ABC567EFG
*/
```

추가 예제는 [new](#), [virtual](#) 및 [override](#)를 참조하세요.

## 예제

이 예제에서는 파생 클래스의 인스턴스를 만들 때 호출되는 기본 클래스 생성자를 지정하는 방법을 보여 줍니다.

```
public class BaseClass
{
    int num;

    public BaseClass()
    {
        Console.WriteLine("in BaseClass()");
    }

    public BaseClass(int i)
    {
        num = i;
        Console.WriteLine("in BaseClass(int i)");
    }

    public int GetNum()
    {
        return num;
    }
}

public class DerivedClass : BaseClass
{
    // This constructor will call BaseClass.BaseClass()
    public DerivedClass() : base()
    {
    }

    // This constructor will call BaseClass.BaseClass(int i)
    public DerivedClass(int i) : base(i)
    {
    }

    static void Main()
    {
        DerivedClass md = new DerivedClass();
        DerivedClass md1 = new DerivedClass(1);
    }
}
/*
Output:
in BaseClass()
in BaseClass(int i)
*/
```

## 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# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)

- this

# this(C# 참조)

2020-03-18 • 3 minutes to read • [Edit Online](#)

`this` 키워드는 클래스의 현재 인스턴스를 가리키며 확장 메서드의 첫 번째 매개 변수에 대한 한정자로도 사용됩니다.

## NOTE

이 문서에서는 클래스 인스턴스와 함께 `this` 를 사용하는 방법을 설명합니다. 확장 메서드에서 사용하는 방법에 대한 자세한 내용은 확장 메서드를 참조하세요.

`this`의 일반적인 사용은 다음과 같습니다.

- 비슷한 이름으로 숨겨진 멤버를 한정합니다. 예를 들면 다음과 같습니다.

```
public class Employee
{
    private string alias;
    private string name;

    public Employee(string name, string alias)
    {
        // Use this to qualify the members of the class
        // instead of the constructor parameters.
        this.name = name;
        this.alias = alias;
    }
}
```

- 개체를 다른 메서드에 매개 변수로 전달합니다. 예를 들면 다음과 같습니다.

```
CalcTax(this);
```

- 인덱서를 선언합니다. 예를 들면 다음과 같습니다.

```
public int this[int param]
{
    get { return array[param]; }
    set { array[param] = value; }
}
```

정적 멤버 함수는 개체의 일부가 아니라 클래스 수준에 있기 때문에 `this` 포인터가 없습니다. 정적 메서드에서 `this`를 참조하면 오류가 발생합니다.

## 예제

이 예제에서는 `this` 를 사용하여 유사한 이름으로 숨겨진 `Employee` 클래스 멤버 `name` 및 `alias` 를 한정합니다. 다른 클래스에 속하는 `CalcTax` 메서드에 개체를 전달하는 데에도 사용됩니다.

```

class Employee
{
    private string name;
    private string alias;
    private decimal salary = 3000.00m;

    // Constructor:
    public Employee(string name, string alias)
    {
        // Use this to qualify the fields, name and alias:
        this.name = name;
        this.alias = alias;
    }

    // Printing method:
    public void printEmployee()
    {
        Console.WriteLine("Name: {0}\nAlias: {1}", name, alias);
        // Passing the object to the CalcTax method by using this:
        Console.WriteLine("Taxes: {0:C}", Tax.CalcTax(this));
    }

    public decimal Salary
    {
        get { return salary; }
    }
}

class Tax
{
    public static decimal CalcTax(Employee E)
    {
        return 0.08m * E.Salary;
    }
}

class MainClass
{
    static void Main()
    {
        // Create objects:
        Employee E1 = new Employee("Mingda Pan", "mpan");

        // Display results:
        E1.printEmployee();
    }
}
/*
Output:
Name: Mingda Pan
Alias: mpan
Taxes: $240.00
*/

```

## 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# 참조](#)
- [C# 프로그래밍 가이드](#)

- C# 키워드
- base
- 메서드

# null(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

`null` 키워드는 개체를 참조하지 않는 `null` 참조를 나타내는 리터럴입니다. `null`은 참조 형식 변수의 기본값입니다. `nullable` 값 형식을 제외한 일반 값 형식은 `null`일 수 없습니다.

다음 예제에서는 `null` 키워드의 몇 가지 동작을 보여 줍니다.

```

class Program
{
    class MyClass
    {
        public void MyMethod() { }
    }

    static void Main(string[] args)
    {
        // Set a breakpoint here to see that mc = null.
        // However, the compiler considers it "unassigned."
        // and generates a compiler error if you try to
        // use the variable.
        MyClass mc;

        // Now the variable can be used, but...
        mc = null;

        // ... a method call on a null object raises
        // a run-time NullReferenceException.
        // Uncomment the following line to see for yourself.
        // mc.MyMethod();

        // Now mc has a value.
        mc = new MyClass();

        // You can call its method.
        mc.MyMethod();

        // Set mc to null again. The object it referenced
        // is no longer accessible and can now be garbage-collected.
        mc = null;

        // A null string is not the same as an empty string.
        string s = null;
        string t = String.Empty; // Logically the same as ""

        // Equals applied to any null object returns false.
        bool b = (t.Equals(s));
        Console.WriteLine(b);

        // Equality operator also returns false when one
        // operand is null.
        Console.WriteLine("Empty string {0} null string", s == t ? "equals": "does not equal");

        // Returns true.
        Console.WriteLine("null == null is {0}", null == null);

        // A value type cannot be null
        // int i = null; // Compiler error!

        // Use a nullable value type instead:
        int? i = null;

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}

```

## C# 언어 사양

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

## 참조

- C# 참조
- C# 키워드
- C# 형식의 기본값
- Nothing(Visual Basic)

# bool(C# 참조)

2020-11-02 • 3 minutes to read • [Edit Online](#)

`bool` 형식 키워드는 부울 값(`true` 또는 `false`)을 나타내는 .NET `System.Boolean` 구조체 형식의 별칭입니다.

`bool` 형식의 값을 사용하여 논리 연산을 수행하려면 [부울 논리](#) 연산자를 사용합니다. `bool` 형식은 [비교](#) 및 [같음](#) 연산자의 결과 형식입니다. `bool` 식은 `if`, `do`, `while` 및 `for` 문과 [조건부 연산자](#) `?:`에서 제어하는 조건식입니다.

`bool` 형식의 기본값은 `false`입니다.

## 리터럴

`true` 및 `false` 리터럴을 사용하여 `bool` 변수를 초기화하거나 `bool` 값을 전달할 수 있습니다.

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked"); // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked"); // output: Not checked
```

## 값이 세 개인 부울 논리

예를 들어 값이 세 개인 논리를 지원해야 하는 경우(예: 값이 세 개인 부울 형식을 지원하는 데이터베이스에서 작업하는 경우) nullable `bool?` 형식을 사용합니다. `bool?` 피연산자의 경우 미리 정의된 `&` 및 `|` 연산자는 값이 세 개인 논리를 지원합니다. 자세한 내용은 [부울 논리 연산자](#) 문서의 [Nullable 부울 논리 연산자](#) 섹션을 참조하세요.

nullable 값 형식에 대한 자세한 내용은 [Nullable 값 형식](#)을 참조하세요.

## 변환

C#은 `bool` 형식을 포함하는 두 개의 변환만 제공합니다. 여기에는 해당하는 nullable `bool?` 형식으로의 암시적 변환과 `bool?` 형식에서의 명시적 변환이 있습니다. 그러나 .NET에서는 `bool` 형식으로 변환하거나 해당 형식에서 변환하는 데 사용할 수 있는 추가 메서드를 제공합니다. 자세한 내용은 [System.Boolean API](#) 참조 페이지의 [부울 값 사이의 변환](#) 섹션을 참조하세요.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 `bool` 형식 섹션을 참조하세요.

## 참고 항목

- [C# 참조](#)
- [값 형식](#)
- [true 및 false 연산자](#)

# default(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

`default` 키워드는 다음 두 가지 방법으로 사용할 수 있습니다.

- `switch` 문에서 기본 레이블을 지정합니다.
- 기본 연산자 또는 리터럴로 형식의 기본값을 생성합니다.

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)

# 상황별 키워드(C# 참조)

2020-11-02 • 3 minutes to read • [Edit Online](#)

상황별 키워드는 코드에서 특정 의미를 제공하는 데 사용되지만 C#의 예약어입니다. 다음 상황별 키워드를 이 섹션에서 소개합니다.

키워드	DESCRIPTION
<code>add</code>	클라이언트 코드가 이벤트를 구독할 때 호출되는 사용자 지정 이벤트 접근자를 정의합니다.
<code>async</code>	수정된 메서드, 람다 식 또는 무명 메서드가 비동기임을 나타냅니다.
<code>await</code>	대기 작업이 완료될 때까지 비동기 메서드를 일시 중단합니다.
<code>dynamic</code>	컴파일 시간 형식 검사를 우회하기 위해 수행하는 작업을 가능하게 하는 참조 형식을 정의합니다.
<code>get</code>	속성 또는 인덱서에 대한 접근자 메서드를 정의합니다.
<code>global</code>	별도로 명명되지 않은 전역 네임스페이스의 별칭입니다.
<code>partial</code>	같은 컴파일 단위 전체에서 <code>partial</code> 클래스, 구조체 및 인터페이스를 정의합니다.
<code>remove</code>	클라이언트 코드가 이벤트를 구독 취소할 때 호출되는 사용자 지정 이벤트 접근자를 정의합니다.
<code>set</code>	속성 또는 인덱서에 대한 접근자 메서드를 정의합니다.
<code>value</code>	접근자를 설정하고 이벤트 처리기를 추가하거나 제거하는데 사용됩니다.
<code>var</code>	컴파일러가 메서드 범위에 선언된 변수의 형식을 확인하는데 사용됩니다.
<code>when</code>	<code>switch</code> 문의 <code>catch</code> 블록 또는 <code>case</code> 레이블에 대한 필터 조건을 지정합니다.
<code>where</code>	제네릭 선언에 제약 조건을 추가합니다. <code>where</code> 를 참조하세요.
<code>yield</code>	반복기 블록에서 값을 열거형 개체에 반환하거나 반복의 끝을 알리는 데 사용됩니다.

C# 3.0에 도입된 모든 쿼리 키워드도 상황별 키워드입니다. 자세한 내용은 [쿼리 키워드\(LINQ\)](#)를 참조하세요.

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)

# add(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

`add` 상황별 키워드는 클라이언트 코드가 `event`를 구독할 때 호출되는 사용자 지정 이벤트 접근자를 정의하는 데 사용됩니다. 사용자 지정 `add` 접근자를 제공하는 경우 `remove` 접근자도 제공해야 합니다.

## 예제

다음 예제에서는 사용자 지정 `add` 및 `remove` 접근자가 있는 이벤트를 보여 줍니다. 전체 예제를 보려면 [인터페이스 이벤트를 구현하는 방법](#)을 참조하세요.

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

일반적으로 고유한 사용자 지정 이벤트 접근자를 제공할 필요가 없습니다. 이벤트를 선언할 때 컴파일러에서 자동으로 생성되는 접근자만으로도 대부분의 시나리오에 충분합니다.

## 참조

- [이벤트](#)

# get(C# 참조)

2021-02-18 • 2 minutes to read • [Edit Online](#)

`get` 키워드는 속성 값 또는 인덱서 요소를 반환하는 속성 또는 인덱서의 *accessor* 메서드를 정의합니다. 자세한 내용은 [속성, 자동으로 구현된 속성 및 인덱서](#)를 참조하세요.

다음 예제에서는 `Seconds`라는 속성의 `get` 및 `set` 접근자를 둘 다 정의합니다. `_seconds`라는 private 필드를 사용하여 속성 값을 지원합니다.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get { return _seconds; }
        set { _seconds = value; }
    }
}
```

대체로 `get` 접근자는 앞의 예제와 마찬가지로 값을 반환하는 단일 문으로 구성됩니다. C# 7.0부터 `get` 접근자 를 식 본문 멤버로 구현할 수 있습니다. 다음 예제에서는 `get` 및 `set` 접근자 둘 다를 식 본문 멤버로 구현합니다.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

속성의 `get` 및 `set` 접근자가 private 필드의 값 설정 또는 검색 이외의 다른 작업을 수행하지 않는 간단한 사례의 경우 자동 구현 속성에 대한 C# 컴파일러의 지원을 활용할 수 있습니다. 다음 예제에서는 `Hours`를 자동 구현 속성으로 구현합니다.

```
class TimePeriod2
{
    public double Hours { get; set; }
```

## 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# 참조
- C# 프로그래밍 가이드
- C# 키워드
- 속성

# partial 형식(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

부분 형식(Partial Type) 정의를 사용하면 클래스, 구조체, 인터페이스 또는 레코드의 정의를 여러 파일로 분할할 수 있습니다.

*File1.cs*:

```
namespace PC
{
    partial class A
    {
        int num = 0;
        void MethodA() { }
        partial void MethodC();
    }
}
```

*File2.cs*에서 선언은 다음과 같습니다.

```
namespace PC
{
    partial class A
    {
        void MethodB() { }
        partial void MethodC() { }
    }
}
```

## 설명

클래스, 구조체 또는 인터페이스 형식을 여러 파일에 분할하면 대형 프로젝트 또는 [Windows Forms 디자이너](#)에서 제공하는 것과 같은 자동으로 생성된 코드로 작업할 때 유용할 수 있습니다. 부분 형식(Partial Type)에는 [부분 메서드\(Partial Method\)](#)가 포함될 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.

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [한정자](#)
- [제네릭 소개](#)

# partial 메서드(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

부분 메서드는 부분 형식의 한 부분에서 해당 시그니처가 정의되고 형식의 다른 부분에서 해당 구현이 정의됩니다. 클래스 디자이너는 부분 메서드를 통해 개발자가 구현 여부를 결정할 수 있는 이벤트 처리기와 유사한 메서드 후크를 제공할 수 있습니다. 개발자가 구현을 제공하지 않을 경우 컴파일러는 컴파일 시간에 시그니처를 제거합니다. 부분 메서드에는 다음 조건이 적용됩니다.

- 부분 형식의 두 부분에 있는 시그니처가 일치해야 합니다.
- 이 메서드는 void를 반환해야 합니다.
- 어떠한 액세스 한정자도 사용할 수 없습니다. 부분 메서드는 암시적으로 private입니다.

다음 예제에서는 partial 클래스의 두 부분에서 정의된 부분 메서드를 보여 줍니다.

```
namespace PM
{
    partial class A
    {
        partial void OnSomethingHappened(string s);
    }

    // This part can be in a separate file.
    partial class A
    {
        // Comment out this method and the program
        // will still compile.
        partial void OnSomethingHappened(String s)
        {
            Console.WriteLine("Something happened: {0}", s);
        }
    }
}
```

자세한 내용은 참조 [Partial 클래스 및 메서드](#)합니다.

## 참고 항목

- [C# 참조](#)
- [partial 형식](#)

# remove(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

`remove` 상황별 키워드는 클라이언트 코드가 `event`에서 구독을 취소할 때 호출되는 사용자 지정 이벤트 접근자를 정의하는 데 사용됩니다. 사용자 지정 `remove` 접근자를 제공하는 경우 `add` 접근자도 제공해야 합니다.

## 예제

다음 예제에서는 사용자 지정 `add` 및 `remove` 접근자가 있는 이벤트를 보여 줍니다. 전체 예제를 보려면 [인터페이스 이벤트를 구현하는 방법](#)을 참조하세요.

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

일반적으로 고유한 사용자 지정 이벤트 접근자를 제공할 필요가 없습니다. 이벤트를 선언할 때 컴파일러에서 자동으로 생성되는 접근자만으로도 대부분의 시나리오에 충분합니다.

## 참조

- [이벤트](#)

# set(C# 참조)

2020-11-02 • 3 minutes to read • [Edit Online](#)

`set` 키워드는 속성 또는 인덱서 요소에 값을 할당하는 속성 또는 인덱서의 *accessor* 메서드를 정의합니다. 자세한 내용 및 예제는 [속성, 자동 구현 속성 및 인덱서](#)를 참조하세요.

다음 예제에서는 `Seconds`라는 속성의 `get` 및 `set` 접근자를 둘 다 정의합니다. `_seconds`라는 private 필드를 사용하여 속성 값을 지원합니다.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get { return _seconds; }
        set { _seconds = value; }
    }
}
```

대체로 `set` 접근자는 앞의 예제와 마찬가지로 값을 할당하는 단일 명령문으로 구성됩니다. C# 7.0부터 `set` 접근자를 식 본문 멤버로 구현할 수 있습니다. 다음 예제에서는 `get` 및 `set` 접근자 둘 다를 식 본문 멤버로 구현합니다.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

속성의 `get` 및 `set` 접근자가 private 지원 필드의 값 설정 또는 검색 이외의 다른 작업을 수행하지 않는 간단한 사례의 경우 자동 구현 속성에 대한 C# 컴파일러의 지원을 활용할 수 있습니다. 다음 예제에서는 `Hours`를 자동 구현 속성으로 구현합니다.

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

## C# 언어 사양

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

## 참조

- C# 참조
- C# 프로그래밍 가이드
- C# 키워드
- 속성

# when(C# 참조)

2020-11-02 • 5 minutes to read • [Edit Online](#)

`when` 상황별 키워드를 사용하여 다음 컨텍스트에서 필터 조건을 지정할 수 있습니다.

- `try/catch` 또는 `try/catch/finally` 블록의 `catch` 문에서
- `switch` 문의 `case` 레이블에서
- `switch` 식에서

## `catch` 문의 `when`

C# 6부터, 특정 예외에 대한 처리기를 실행하기 위해 참이 되어야 하는 조건을 지정하기 위해 `catch` 문에서 `when`을 사용할 수 있습니다. 사용되는 구문은 다음과 같습니다.

```
catch (ExceptionType [e]) when (expr)
```

여기서 `expr`은 부울 값으로 계산되는 식입니다. `true`가 반환되면 예외 처리기가 실행되고, `false`가 반환되면 실행되지 않습니다.

다음 예제는 `when` 키워드를 사용하여 예외 메시지의 텍스트에 따라 `HttpRequestException`에 대한 처리기를 조건부로 실행합니다.

```

using System;
using System.Net.Http;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {
        Console.WriteLine(MakeRequest().Result);
    }

    public static async Task<string> MakeRequest()
    {
        var client = new HttpClient();
        var streamTask = client.GetStringAsync("https://localhost:10000");
        try
        {
            var responseText = await streamTask;
            return responseText;
        }
        catch (HttpRequestException e) when (e.Message.Contains("301"))
        {
            return "Site Moved";
        }
        catch (HttpRequestException e) when (e.Message.Contains("404"))
        {
            return "Page Not Found";
        }
        catch (HttpRequestException e)
        {
            return e.Message;
        }
    }
}

```

## switch 문의 when

C# 7.0부터 `case` 레이블은 더 이상 상호 배타적일 필요가 없으며, `case` 레이블이 `switch` 문에 나타나는 순서에 따라 실행되는 스위치 블록을 결정할 수 있습니다. 필터 조건도 참인 경우에만 관련 사례 레이블을 참으로 만드는 필터 조건을 지정하려면 `when` 키워드를 사용할 수 있습니다. 사용되는 구문은 다음과 같습니다.

```
case (expr) when (when-condition):
```

여기서 `expr`은 일치 식과 비교되는 상수 패턴 또는 형식 패턴이며 `when-condition`은 부울 식입니다.

다음 예제에서는 `when` 키워드를 사용하여 영역이 0인 `Shape` 객체를 테스트하고 영역이 0보다 큰 다양한 `Shape` 객체를 테스트합니다.

```

using System;

public abstract class Shape
{
    public abstract double Area { get; }
    public abstract double Circumference { get; }
}

public class Rectangle : Shape
{
    public Rectangle(double length, double width)
    {
        Length = length;
        Width = width;
    }
}
```

```

        }

    public double Length { get; set; }
    public double Width { get; set; }

    public override double Area
    {
        get { return Math.Round(Length * Width,2); }
    }

    public override double Circumference
    {
        get { return (Length + Width) * 2; }
    }
}

public class Square : Rectangle
{
    public Square(double side) : base(side, side)
    {
        Side = side;
    }

    public double Side { get; set; }
}

public class Example
{
    public static void Main()
    {
        Shape sh = null;
        Shape[] shapes = { new Square(10), new Rectangle(5, 7),
                           new Rectangle(10, 10), sh, new Square(0) };
        foreach (var shape in shapes)
            ShowShapeInfo(shape);
    }

    private static void ShowShapeInfo(Object obj)
    {
        switch (obj)
        {
            case Shape shape when shape.Area == 0:
                Console.WriteLine($"The shape: {shape.GetType().Name} with no dimensions");
                break;
            case Square sq when sq.Area > 0:
                Console.WriteLine("Information about the square:");
                Console.WriteLine($"    Length of a side: {sq.Side}");
                Console.WriteLine($"    Area: {sq.Area}");
                break;
            case Rectangle r when r.Area > 0:
                Console.WriteLine("Information about the rectangle:");
                Console.WriteLine($"    Dimensions: {r.Length} x {r.Width}");
                Console.WriteLine($"    Area: {r.Area}");
                break;
            case Shape shape:
                Console.WriteLine($"A {shape.GetType().Name} shape");
                break;
            case null:
                Console.WriteLine($"The {nameof(obj)} variable is uninitialized.");
                break;
            default:
                Console.WriteLine($"The {nameof(obj)} variable does not represent a Shape.");
                break;
        }
    }
}

// The example displays the following output:
//      Information about the square:
//          Length of a side: 10

```

```
//      Length of a side: 10
//      Area: 100
//      Information about the rectangle:
//          Dimensions: 5 x 7
//          Area: 35
//      Information about the rectangle:
//          Dimensions: 10 x 10
//          Area: 100
//      The obj variable is uninitialized.
//      The shape: Square with no dimensions
```

## 참조

- [switch 문](#)
- [try/catch 문](#)
- [try/catch/finally 문](#)

# value(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

상황별 키워드 `value`는 [속성](#) 및 [인덱서](#) 선언의 `set` 접근자에 사용됩니다. 메서드의 입력 매개 변수와 비슷합니다. `value` 단어는 클라이언트 코드에서 속성 또는 인덱서에 할당하려는 값을 참조합니다. 다음 예에서 `MyDerivedClass`에는 `Name`이라는 속성이 있습니다. 이 속성은 `value` 매개 변수를 사용하여 지원 필드 `name`에 새 문자열을 할당합니다. 클라이언트 코드의 관점에서 이 작업은 단순한 할당으로 기록됩니다.

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int num;
    public virtual int Number
    {
        get { return num; }
        set { num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string name;

    // Override auto-implemented property with ordinary property
    // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                name = value;
            }
            else
            {
                name = "Unknown";
            }
        }
    }
}
```

자세한 내용은 [속성](#) 및 [인덱서](#) 문서를 참조하세요.

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 키워드](#)

# yield(C# 참조)

2021-02-18 • 9 minutes to read • [Edit Online](#)

문에 `yield` 상황별 키워드를 사용하는 경우 해당 메서드, 연산자 또는 이 키워드가 나타나는 `get` 접근자가 반복기임을 나타냅니다. `yield`를 사용하여 반복기를 정의할 경우 사용자 지정 컬렉션 형식에 `IEnumerator<T>` 및 `IEnumerable` 패턴을 구현하면 명시적 추가 클래스(열거형의 상태를 보관하는 클래스, 예제는 `IEnumerator` 참조)를 사용하지 않아도 됩니다.

다음 예제에서는 두 가지 형태의 `yield` 문을 보여줍니다.

```
yield return <expression>;  
yield break;
```

## 설명

`yield return` 문을 사용하여 각 요소를 따로따로 반환할 수 있습니다.

반복기 메서드에서 반환된 시퀀스는 `foreach` 문 또는 LINQ 쿼리를 통해 사용할 수 있습니다. 각각의 `foreach` 루프의 반복이 반복기 메서드를 호출합니다. `yield return` 문이 반복기 메서드에 도달하면 `expression`이 반환되고 코드에서 현재 위치는 유지됩니다. 다음에 반복기 함수가 호출되면 해당 위치에서 실행이 다시 시작됩니다.

`yield break` 문을 사용하여 반복기를 종료할 수 있습니다.

반복기에 대한 자세한 내용은 [반복기](#)를 참조하세요.

## 반복기 메서드 및 Get 접근자

반복기 선언은 다음과 같은 요구 사항을 충족해야 합니다.

- 반환 형식은 `IEnumerable`, `IEnumerable<T>`, `IEnumerator`, 또는 `IEnumerator<T>`여야 합니다.
- 선언에 `in`, `ref` 또는 `out` 매개 변수가 허용되지 않습니다.

`yield` 또는 `IEnumerable`을 반환하는 반복기의 `IEnumerator` 형식은 `object`입니다. 반복기기 `IEnumerable<T>` 또는 `IEnumerator<T>`를 반환할 경우 `yield return` 문의 식 형식에서 제네릭 형식 매개 변수로 암시적 변환이 있어야 합니다.

`yield return` 또는 `yield break` 식을

- 람다 식 및 무명 메서드에 포함할 수 없습니다.
- 안전하지 않은 블록을 포함하는 메서드 자세한 내용은 [unsafe](#)를 참조하세요.

## 예외 처리

`yield return` 문은 try-catch 블록에서 찾을 수 없습니다. `yield return` 문은 try-finally 문의 try 블록에서 찾을 수 있습니다.

`yield break` 문은 try 블록이나 catch 블록에서 찾을 수 있지만 finally 블록에서는 찾을 수 없습니다.

`foreach` 본문(반복기 메서드 외부)에서 예외를 throw한 경우, 반복기 메서드의 `finally` 블록이 실행됩니다.

## 기술 구현

다음 코드는 반복기 메서드에서 `IEnumerable<string>` 을 반환하고 해당 요소를 반복합니다.

```
IEnumerable<string> elements = MyIteratorMethod();
foreach (string element in elements)
{
    ...
}
```

`MyIteratorMethod` 호출은 메서드의 본문을 실행하지 않습니다. 대신에 `IEnumerable<string>` 변수에 `elements` 을 반환합니다.

`foreach` 루프 반복에서 `MoveNext`에 대한 `elements` 메서드가 호출됩니다. 이 호출은 다음 `MyIteratorMethod` 문에 도달할 때까지 `yield return` 본문을 실행합니다. `yield return` 문에서 반환하는 식은 루프 본문에서 사용하는 `element` 변수 값뿐만 아니라 `IEnumerable<string>` 인 `elements`의 `Current` 속성도 결정합니다.

이후에 `foreach` 루프가 반복될 때마다 종지되었던 위치에서 반복기 본문 실행이 계속되고 `yield return` 문에 도달하면 다시 종지됩니다. `foreach` 루프는 반복기 메서드가 종료되거나 `yield break` 문에 도달하면 완료됩니다.

## 예제

다음 예제에는 `yield return` 루프 내에 `for` 문이 있습니다. `Main` 메서드에서 `foreach` 문의 본문을 반복할 때마다 `Power` 반복기 함수에 대한 호출이 생성됩니다. 반복기 함수를 호출할 때마다 다음에 `yield return` 루프를 반복하는 도중에 `for` 문이 실행됩니다.

반복기 메서드의 반환 형식은 반복기 인터페이스 형식인 `IEnumerable`입니다. 반복기 메서드가 호출되면 숫자의 거듭제곱이 들어 있는 열거형 개체를 반환합니다.

```
public class PowersOf2
{
    static void Main()
    {
        // Display powers of 2 up to the exponent of 8:
        foreach (int i in Power(2, 8))
        {
            Console.WriteLine("{0} ", i);
        }
    }

    public static System.Collections.Generic.IEnumerable<int> Power(int number, int exponent)
    {
        int result = 1;

        for (int i = 0; i < exponent; i++)
        {
            result = result * number;
            yield return result;
        }
    }

    // Output: 2 4 8 16 32 64 128 256
}
```

## 예제

다음 예제는 반복기인 `get` 접근자에 대해 설명합니다. 이 예제에서는 각 `yield return` 문이 사용자 정의 클래스의 인스턴스를 반환합니다.

```

public static class GalaxyClass
{
    public static void ShowGalaxies()
    {
        var theGalaxies = new Galaxies();
        foreach (Galaxy theGalaxy in theGalaxies.NextGalaxy)
        {
            Debug.WriteLine(theGalaxy.Name + " " + theGalaxy.MegaLightYears.ToString());
        }
    }

    public class Galaxies
    {

        public System.Collections.Generic.IEnumerable<Galaxy> NextGalaxy
        {
            get
            {
                yield return new Galaxy { Name = "Tadpole", MegaLightYears = 400 };
                yield return new Galaxy { Name = "Pinwheel", MegaLightYears = 25 };
                yield return new Galaxy { Name = "Milky Way", MegaLightYears = 0 };
                yield return new Galaxy { Name = "Andromeda", MegaLightYears = 3 };
            }
        }
    }

    public class Galaxy
    {
        public String Name { get; set; }
        public int MegaLightYears { get; set; }
    }
}

```

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [foreach, in](#)
- [반복기](#)

# 쿼리 키워드(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

이 섹션에는 쿼리 식에 사용되는 상황별 키워드가 포함되어 있습니다.

## 단원 내용

절	DESCRIPTION
from	데이터 소스와 범위 변수(반복 변수와 유사함)를 지정합니다.
where	논리적 AND 및 OR 연산자( <code>&amp;&amp;</code> 또는 <code>  </code> )로 구분된 하나 이상의 부울 식을 기준으로 소스 요소를 필터링합니다.
select	쿼리를 실행할 때 반환된 시퀀스의 요소에 사용할 형식 및 모양을 지정합니다.
group	지정된 키 값에 따라 쿼리 결과를 그룹화합니다.
into	<code>join</code> , <code>group</code> 또는 <code>select</code> 절의 결과에 대한 참조로 사용할 수 있는 식별자를 제공합니다.
orderby	요소 형식에 대한 기본 비교자에 따라 오름차순 또는 내림차순으로 쿼리 결과를 정렬합니다.
join	지정한 두 일치 조건 간의 같음 비교를 기반으로 하여 두 데이터 소스를 조인합니다.
let	쿼리 식에 하위 식 결과를 저장할 범위 변수를 도입합니다.
in	<code>join</code> 절의 상황별 키워드입니다.
on	<code>join</code> 절의 상황별 키워드입니다.
equals	<code>join</code> 절의 상황별 키워드입니다.
by	<code>group</code> 절의 상황별 키워드입니다.
ascending	<code>orderby</code> 절의 상황별 키워드입니다.
descending	<code>orderby</code> 절의 상황별 키워드입니다.

## 참조

- [C# 키워드](#)
- [LINQ\(Language-Integrated Query\)](#)
- [C#의 LINQ](#)

# from 절(C# 참조)

2021-02-18 • 11 minutes to read • [Edit Online](#)

쿼리 식은 `from` 절로 시작해야 합니다. 또한 쿼리 식은 `from` 절로 시작하는 하위 쿼리를 포함할 수 있습니다. `from` 절은 다음 내용을 지정합니다.

- 쿼리 또는 하위 쿼리가 실행될 데이터 소스.
- 소스 시퀀스의 각 요소를 나타내는 지역 범위 변수.

범위 변수 및 데이터 소스 모두 강력한 형식입니다. `from` 절에서 참조되는 데이터 소스는 `IEnumerable` 또는 `IEnumerable<T>` 형식이거나 `IQueryable<T>`와 같은 파생 형식이어야 합니다.

다음 예제에서 `numbers`는 데이터 소스이고 `num`은 범위 변수입니다. `var` 키워드가 사용되어도 두 변수는 모두 강력한 형식입니다.

```
class LowNums
{
    static void Main()
    {
        // A simple data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query.
        // lowNums is an IEnumerable<int>
        var lowNums = from num in numbers
                      where num < 5
                      select num;

        // Execute the query.
        foreach (int i in lowNums)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 4 1 3 2 0
```

## 범위 변수

컴파일러는 데이터 소스가 `IEnumerable<T>`을 구현할 경우 범위 변수의 형식을 유추합니다. 예를 들어, 소스의 형식이 `IEnumerable<Customer>`일 경우 범위 변수는 `Customer`로 유추됩니다. 소스가 `ArrayList`과 같이 제네릭이 아닌 `IEnumerable` 형식인 경우에만 형식을 명시적으로 지정하면 됩니다. 자세한 내용은 [LINQ를 사용하여 ArrayList를 쿼리하는 방법](#)을 참조하세요.

위의 예제에서 `num`은 `int` 형식으로 유추됩니다. 범위 변수가 강력한 형식이므로 범위 변수에서 메서드를 호출하거나 다른 작업에서 범위 변수를 사용할 수 있습니다. 예를 들어 `select num`을 작성하는 대신에 `select num.ToString()`을 작성하여 쿼리 식에서 정수 대신 문자열 시퀀스를 반환하게 할 수 있습니다. 또는 `select num + 10`을 작성하여 식에서 14, 11, 13, 12, 10 시퀀스를 반환하게 할 수 있습니다. 자세한 내용은 [select 절](#)을 참조하세요.

범위 변수가 소스의 데이터를 실제로 저장하지 않는다는 매우 중요한 한 가지 차이점을 제외하면 범위 변수는 `foreach` 문의 반복 변수와 같습니다. 범위 변수는 단순히 쿼리 실행 시에 발생하는 작업을 쿼리에서 설명할 수 있게 하는 구문상의 편리함을 제공합니다. 자세한 내용은 [LINQ 쿼리 소개\(C#\)](#)를 참조하세요.

## 복합 from 절

경우에 따라 소스 시퀀스의 각 요소 자체가 시퀀스이거나 시퀀스를 포함할 수 있습니다. 예를 들어, 시퀀스의 각 학생 개체에 테스트 점수 목록이 포함된 `IEnumerable<Student>` 가 데이터 소스일 수 있습니다. 각 `Student` 요소 내의 내부 목록에 액세스하려면 복합 `from` 절을 사용합니다. 이 기술은 중첩된 `foreach` 문을 사용하는 것과 같습니다. `where` 또는 `orderby` 절을 둘 중 하나의 `from` 절에 추가하여 결과를 필터링 할 수 있습니다. 다음 예제에서는 테스트 점수를 나타내는 정수의 내부 `List` 를 각각 포함하는 `Student` 개체의 시퀀스를 보여 줍니다. 내부 목록에 액세스하려면 복합 `from` 절을 사용합니다. 필요한 경우 두 `from` 절 사이에 다른 절을 삽입 할 수 있습니다.

```

class CompoundFrom
{
    // The element type of the data source.
    public class Student
    {
        public string LastName { get; set; }
        public List<int> Scores {get; set;}
    }

    static void Main()
    {

        // Use a collection initializer to create the data source. Note that
        // each element in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
            new Student {LastName="Omelchenko", Scores= new List<int> {97, 72, 81, 60}},
            new Student {LastName="O'Donnell", Scores= new List<int> {75, 84, 91, 39}},
            new Student {LastName="Mortensen", Scores= new List<int> {88, 94, 65, 85}},
            new Student {LastName="Garcia", Scores= new List<int> {97, 89, 85, 82}},
            new Student {LastName="Beebe", Scores= new List<int> {35, 72, 91, 70}}
        };

        // Use a compound from to access the inner sequence within each element.
        // Note the similarity to a nested foreach statement.
        var scoreQuery = from student in students
                          from score in student.Scores
                          where score > 90
                          select new { Last = student.LastName, score };

        // Execute the queries.
        Console.WriteLine("scoreQuery:");
        // Rest the mouse pointer on scoreQuery in the following line to
        // see its type. The type is IEnumerable<'a>, where 'a is an
        // anonymous type defined as new {string Last, int score}. That is,
        // each instance of this anonymous type has two members, a string
        // (Last) and an int (score).
        foreach (var student in scoreQuery)
        {
            Console.WriteLine("{0} Score: {1}", student.Last, student.score);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
scoreQuery:
Omelchenko Score: 97
O'Donnell Score: 91
Mortensen Score: 94
Garcia Score: 97
Beebe Score: 91
*/

```

## 여러 from 절을 사용하여 조인 수행

복합 `from` 절은 단일 데이터 소스의 내부 컬렉션에 액세스하는 데 사용됩니다. 그러나 독립 데이터 소스의 추가 쿼리를 생성하는 여러 `from` 절이 쿼리에 포함될 수도 있습니다. 이 기술을 사용하면 [join 절](#)로 수행할 수 없는 특정 형식의 조인 작업을 수행할 수 있습니다.

다음 예제는 두 개의 `from` 절을 사용하여 두 개의 데이터 소스의 완전한 크로스 조인을 구성하는 방법을 보여 줍니다.

```

class CompoundFrom2
{
    static void Main()
    {
        char[] upperCase = { 'A', 'B', 'C' };
        char[] lowerCase = { 'x', 'y', 'z' };

        // The type of joinQuery1 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery1 =
            from upper in upperCase
            from lower in lowerCase
            select new { upper, lower };

        // The type of joinQuery2 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery2 =
            from lower in lowerCase
            where lower != 'x'
            from upper in upperCase
            select new { lower, upper };

        // Execute the queries.
        Console.WriteLine("Cross join:");
        // Rest the mouse pointer on joinQuery1 to verify its type.
        foreach (var pair in joinQuery1)
        {
            Console.WriteLine("{0} is matched to {1}", pair.upper, pair.lower);
        }

        Console.WriteLine("Filtered non-equijoin:");
        // Rest the mouse pointer over joinQuery2 to verify its type.
        foreach (var pair in joinQuery2)
        {
            Console.WriteLine("{0} is matched to {1}", pair.lower, pair.upper);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   Cross join:
   A is matched to x
   A is matched to y
   A is matched to z
   B is matched to x
   B is matched to y
   B is matched to z
   C is matched to x
   C is matched to y
   C is matched to z
   Filtered non-equijoin:
   y is matched to A
   y is matched to B
   y is matched to C
   z is matched to A
   z is matched to B
   z is matched to C
*/

```

여러 `from` 절을 사용하는 조인 작업에 대한 자세한 내용은 [왼쪽 우선 외부 조인 수행](#)을 참조하세요.

## 참고 항목

- 쿼리 키워드(LINQ)
- LINQ(Language-Integrated Query)

# where 절(C# 참조)

2021-02-18 • 5 minutes to read • [Edit Online](#)

`where` 절은 데이터 소스의 어떤 요소가 쿼리 식에서 반환될지를 지정하기 위해 쿼리 식에서 사용됩니다. 또한 각 소스 요소(범위 변수로 참조됨)에 부울 조건(*predicate*)을 적용하고 지정된 조건이 참인 요소를 반환합니다. 단일 쿼리 식에는 여러 `where` 절을 포함할 수 있으며 단일 절에는 여러 조건부 하위 식을 포함할 수 있습니다.

## 예제

다음 예제에서 `where` 절은 5보다 작은 숫자를 제외한 모든 숫자를 필터링합니다. `where` 절을 제거하면 데이터 소스의 모든 숫자가 반환됩니다. `num < 5` 식은 각 요소에 적용되는 조건자입니다.

```
class WhereSample
{
    static void Main()
    {
        // Simple data source. Arrays support IEnumerable<T>.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Simple query with one predicate in where clause.
        var queryLowNums =
            from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (var s in queryLowNums)
        {
            Console.WriteLine(s.ToString() + " ");
        }
    }
}
//Output: 4 1 3 2 0
```

## 예제

단일 `where` 절 내에서 `&&` 및 `||` 연산자를 사용하여 조건자를 필요한 만큼 지정할 수 있습니다. 다음 예제에서 쿼리는 5 미만의 짝수만 선택하도록 두 개의 조건자를 지정합니다.

```

class WhereSample2
{
    static void Main()
    {
        // Data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with two predicates in where clause.
        var queryLowNums2 =
            from num in numbers
            where num < 5 && num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums2)
        {
            Console.WriteLine(s.ToString() + " ");
        }
        Console.WriteLine();

        // Create the query with two where clause.
        var queryLowNums3 =
            from num in numbers
            where num < 5
            where num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums3)
        {
            Console.WriteLine(s.ToString() + " ");
        }
    }
}

// Output:
// 4 2 0
// 4 2 0

```

## 예제

`where` 절에는 부울 값을 반환하는 메서드를 하나 이상 포함할 수 있습니다. 다음 예제에서 `where` 절은 메서드를 사용하여 범위 변수의 현재 값이 짝수인지 홀수인지를 확인합니다.

```

class WhereSample3
{
    static void Main()
    {
        // Data source
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with a method call in the where clause.
        // Note: This won't work in LINQ to SQL unless you have a
        // stored procedure that is mapped to a method by this name.
        var queryEvenNums =
            from num in numbers
            where IsEven(num)
            select num;

        // Execute the query.
        foreach (var s in queryEvenNums)
        {
            Console.WriteLine(s.ToString() + " ");
        }
    }

    // Method may be instance method or static method.
    static bool IsEven(int i)
    {
        return i % 2 == 0;
    }
}
//Output: 4 8 6 2 0

```

## 설명

`where` 절은 필터링 메커니즘입니다. 첫 번째 또는 마지막 절이 될 수 없다는 점을 제외하고, 쿼리 식의 거의 모든 곳에 배치할 수 있습니다. 소스 요소를 그룹화 전에 필터링할지, 그룹화 후에 필터링 할지에 따라 `where` 절은 `group` 절 앞 또는 뒤에 나타날 수 있습니다.

지정된 조건자가 데이터 소스의 요소에 대해 유효하지 않은 경우, 컴파일 시간 오류가 발생합니다. 이는 LINQ에서 제공하는 강력한 형식 검사의 이점 중 하나입니다.

컴파일 시간에 `where` 키워드는 `Where` 표준 쿼리 연산자 메서드에 대한 호출로 변환됩니다.

## 참고 항목

- [쿼리 키워드\(LINQ\)](#)
- [from 절](#)
- [select 절](#)
- [데이터 필터링](#)
- [C#의 LINQ](#)
- [LINQ\(Language-Integrated Query\)](#)

# select 절(C# 참조)

2021-02-18 • 8 minutes to read • [Edit Online](#)

쿼리 식에서 `select` 절은 쿼리를 실행할 때 생성되는 값의 형식을 지정합니다. 결과는 모든 이전 절의 평가와 `select` 절 자체의 모든 계산을 기반으로 합니다. 쿼리 식은 `select` 절이나 `group` 절로 끝나야 합니다.

다음 예제에서는 쿼리 식의 간단한 `select` 절을 보여 줍니다.

```
class SelectSample1
{
    static void Main()
    {
        //Create the data source
        List<int> Scores = new List<int>() { 97, 92, 81, 60 };

        // Create the query.
        I Enumerable<int> queryHighScores =
            from score in Scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in queryHighScores)
        {
            Console.WriteLine(i + " ");
        }
    }
}
//Output: 97 92 81
```

`select` 절에서 생성된 시퀀스의 형식에 따라 쿼리 변수 `queryHighScores`의 형식이 결정됩니다. 가장 단순한 경우에서는 `select` 절이 범위 변수를 지정합니다. 이렇게 하면 반환된 시퀀스에 데이터 소스와 동일한 형식의 요소가 포함됩니다. 자세한 내용은 [LINQ 쿼리 작업의 형식 관계](#)를 참조하세요. 그러나 `select` 절은 소스 데이터를 새 형식으로 변환(또는 [프로젝션](#))하기 위한 강력한 메커니즘도 제공합니다. 자세한 내용은 [LINQ를 통한 데이터 변환\(C#\)](#)을 참조하세요.

## 예제

다음 예제에서는 `select` 절에 사용할 수 있는 모든 형식을 보여 줍니다. 각 쿼리에서 `select` 절과 쿼리 변수 형식(`studentQuery1`, `studentQuery2` 등) 간의 관계를 확인합니다.

```
class SelectSample2
{
    // Define some classes
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
        public ContactInfo GetContactInfo(SelectSample2 app, int id)
        {
            ContactInfo cInfo =
                (from ci in app.contactList
                where ci.ID == id
                select ci)
                .FirstOrDefault();
```

```

        return cInfo;
    }

    public override string ToString()
    {
        return First + " " + Last + ":" + ID;
    }
}

public class ContactInfo
{
    public int ID { get; set; }
    public string Email { get; set; }
    public string Phone { get; set; }
    public override string ToString() { return Email + "," + Phone; }
}

public class ScoreInfo
{
    public double Average { get; set; }
    public int ID { get; set; }
}

// The primary data source
List<Student> students = new List<Student>()
{
    new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int>() {97, 92, 81,
60},},
    new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int>() {75, 84, 91,
39},},
    new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int>() {88, 94, 65, 91}},
    new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int>() {97, 89, 85, 82}},
};

// Separate data source for contact info.
List<ContactInfo> contactList = new List<ContactInfo>()
{
    new ContactInfo {ID=111, Email="Svetlan0@Contoso.com", Phone="206-555-0108"},,
    new ContactInfo {ID=112, Email="Claire0@Contoso.com", Phone="206-555-0298"},,
    new ContactInfo {ID=113, Email="SvenMort@Contoso.com", Phone="206-555-1130"},,
    new ContactInfo {ID=114, Email="CesarGar@Contoso.com", Phone="206-555-0521"}};
};

static void Main(string[] args)
{
    SelectSample2 app = new SelectSample2();

    // Produce a filtered sequence of unmodified Students.
    IEnumerable<Student> studentQuery1 =
        from student in app.students
        where student.ID > 111
        select student;

    Console.WriteLine("Query1: select range_variable");
    foreach (Student s in studentQuery1)
    {
        Console.WriteLine(s.ToString());
    }

    // Produce a filtered sequence of elements that contain
    // only one property of each Student.
    IEnumerable<String> studentQuery2 =
        from student in app.students
        where student.ID > 111
        select student.Last;

    Console.WriteLine("\r\nstudentQuery2: select range_variable.Property");
    foreach (string s in studentQuery2)
}

```

```

{
    Console.WriteLine(s);
}

// Produce a filtered sequence of objects created by
// a method call on each Student.
IEnumerable<ContactInfo> studentQuery3 =
    from student in app.students
    where student.ID > 111
    select student.GetContactInfo(app, student.ID);

Console.WriteLine("\r\n studentQuery3: select range_variable.Method");
foreach (ContactInfo ci in studentQuery3)
{
    Console.WriteLine(ci.ToString());
}

// Produce a filtered sequence of ints from
// the internal array inside each Student.
IEnumerable<int> studentQuery4 =
    from student in app.students
    where student.ID > 111
    select student.Scores[0];

Console.WriteLine("\r\n studentQuery4: select range_variable[index]");
foreach (int i in studentQuery4)
{
    Console.WriteLine("First score = {0}", i);
}

// Produce a filtered sequence of doubles
// that are the result of an expression.
IEnumerable<double> studentQuery5 =
    from student in app.students
    where student.ID > 111
    select student.Scores[0] * 1.1;

Console.WriteLine("\r\n studentQuery5: select expression");
foreach (double d in studentQuery5)
{
    Console.WriteLine("Adjusted first score = {0}", d);
}

// Produce a filtered sequence of doubles that are
// the result of a method call.
IEnumerable<double> studentQuery6 =
    from student in app.students
    where student.ID > 111
    select student.Scores.Average();

Console.WriteLine("\r\n studentQuery6: select expression2");
foreach (double d in studentQuery6)
{
    Console.WriteLine("Average = {0}", d);
}

// Produce a filtered sequence of anonymous types
// that contain only two properties from each Student.
var studentQuery7 =
    from student in app.students
    where student.ID > 111
    select new { student.First, student.Last };

Console.WriteLine("\r\n studentQuery7: select new anonymous type");
foreach (var item in studentQuery7)
{
    Console.WriteLine("{0}, {1}", item.Last, item.First);
}

```

```

// Produce a filtered sequence of named objects that contain
// a method return value and a property from each Student.
// Use named types if you need to pass the query variable
// across a method boundary.
IEnumerable<ScoreInfo> studentQuery8 =
    from student in app.students
    where student.ID > 111
    select new ScoreInfo
    {
        Average = student.Scores.Average(),
        ID = student.ID
    };

Console.WriteLine("\r\n studentQuery8: select new named type");
foreach (ScoreInfo si in studentQuery8)
{
    Console.WriteLine("ID = {0}, Average = {1}", si.ID, si.Average);
}

// Produce a filtered sequence of students who appear on a contact list
// and whose average is greater than 85.
IEnumerable<ContactInfo> studentQuery9 =
    from student in app.students
    where student.Scores.Average() > 85
    join ci in app.contactList on student.ID equals ci.ID
    select ci;

Console.WriteLine("\r\n studentQuery9: select result of join clause");
foreach (ContactInfo ci in studentQuery9)
{
    Console.WriteLine("ID = {0}, Email = {1}", ci.ID, ci.Email);
}

// Keep the console window open in debug mode
Console.WriteLine("Press any key to exit.");
Console.ReadKey();
}

/*
 * Output
Query1: select range_variable
Claire O'Donnell:112
Sven Mortensen:113
Cesar Garcia:114

studentQuery2: select range_variable.Property
O'Donnell
Mortensen
Garcia

studentQuery3: select range_variable.Method
ClaireO@Contoso.com,206-555-0298
SvenMort@Contoso.com,206-555-1130
CesarGar@Contoso.com,206-555-0521

studentQuery4: select range_variable[index]
First score = 75
First score = 88
First score = 97

studentQuery5: select expression
Adjusted first score = 82.5
Adjusted first score = 96.8
Adjusted first score = 106.7

studentQuery6: select expression2
Average = 72.25
Average = 84.5
Average = 88.25

```

```
studentQuery7: select new anonymous type
O'Donnell, Claire
Mortensen, Sven
Garcia, Cesar

studentQuery8: select new named type
ID = 112, Average = 72.25
ID = 113, Average = 84.5
ID = 114, Average = 88.25

studentQuery9: select result of join clause
ID = 114, Email = CesarGar@Contoso.com
*/
```

앞의 예제에서 `studentQuery8`에 표시된 대로 반환된 시퀀스의 요소에 소스 요소의 속성 하위 집합만 포함하려는 경우도 있습니다. 반환된 시퀀스를 최대한 작게 유지하면 메모리 요구 사항을 줄이고 쿼리 실행 속도를 높일 수 있습니다. `select` 절에서 무명 형식을 만들고 개체 이니셜라이저를 사용하여 소스 요소의 적절한 속성으로 초기화하면 됩니다. 이 작업을 수행하는 방법의 예를 보려면 [개체 및 컬렉션 이니셜라이저](#)를 참조하세요.

## 설명

컴파일 시간에 `select` 절은 [Select](#) 표준 쿼리 연산자에 대한 메서드 호출로 변환됩니다.

## 참고 항목

- [C# 참조](#)
- [쿼리 키워드\(LINQ\)](#)
- [from 절](#)
- [partial\(메서드\)\(C# 참조\)](#)
- [익명 형식](#)
- [C#의 LINQ](#)
- [LINQ\(Language-Integrated Query\)](#)

# group 절(C# 참조)

2021-02-18 • 15 minutes to read • [Edit Online](#)

`group` 절은 그룹의 키 값과 일치하는 0개 이상의 항목이 포함된 `IGrouping< TKey, TElement >` 개체 시퀀스를 반환합니다. 예를 들어 각 문자열의 첫 번째 문자에 따라 문자열 시퀀스를 그룹화할 수 있습니다. 이 경우 첫 번째 문자는 키로, `char` 형식이며 각 `IGrouping< TKey, TElement >` 개체의 `Key` 속성에 저장됩니다. 컴파일러는 키의 형식을 유추합니다.

다음 예제와 같이 쿼리 식을 `group` 절로 끝낼 수 있습니다.

```
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery1 =
    from student in students
    group student by student.Last[0];
```

각 그룹에서 추가 쿼리 작업을 수행하려는 경우 `into` 상황별 키워드를 사용하여 임시 식별자를 지정할 수 있습니다. `into`를 사용하는 경우 쿼리를 계속 진행하고, 다음 예제와 같이 궁극적으로 `select` 문이나 다른 `group` 절로 끝내야 합니다.

```
// Group students by the first letter of their last name
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0] into g
    orderby g.Key
    select g;
```

`into`를 포함 및 포함하지 않고 `group`을 사용하는 보다 자세한 예제는 이 문서의 예제 섹션에 제공됩니다.

## 그룹 쿼리의 결과 열거

`group` 쿼리에 의해 생성된 `IGrouping< TKey, TElement >` 개체는 기본적으로 목록의 목록이기 때문에 중첩된 `foreach` 루프를 사용하여 각 그룹에 있는 항목에 액세스해야 합니다. 외부 루프는 그룹 키를 반복하고, 내부 루프는 그룹 자체에 있는 각 항목을 반복합니다. 그룹에 키가 있지만 요소는 없을 수도 있습니다. 다음은 앞의 코드 예제에서 쿼리를 실행하는 `foreach` 루프입니다.

```
// Iterate group items with a nested foreach. This IGrouping encapsulates
// a sequence of Student objects, and a Key of type char.
// For convenience, var can also be used in the foreach statement.
foreach (IGrouping<char, Student> studentGroup in studentQuery2)
{
    Console.WriteLine(studentGroup.Key);
    // Explicit type for student could also be used here.
    foreach (var student in studentGroup)
    {
        Console.WriteLine("{0}, {1}", student.Last, student.First);
    }
}
```

## 키 형식

그룹 키는 문자열, 기본 제공 숫자 형식, 사용자 정의 명명된 형식, 무명 형식 등 모든 형식일 수 있습니다.

## 문자열로 그룹화

앞의 코드 예제에서는 `char` 를 사용했습니다. 대신, 문자열 키를 전체 성 등으로 쉽게 지정했을 수 있습니다.

```
// Same as previous example except we use the entire last name as a key.  
// Query variable is an I Enumerable<IGrouping<string, Student>>  
var studentQuery3 =  
    from student in students  
    group student by student.Last;
```

## 부울로 그룹화

다음 예제에서는 키에 부울 값을 사용하여 결과를 두 그룹으로 나누는 방법을 보여 줍니다. 값은 `group` 절의 하위 식에서 생성됩니다.

```

class GroupSample1
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that each element
        // in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
            new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 72, 81,
60}},
            new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},
            new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {99, 89, 91, 95}},
            new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {72, 81, 65, 84}},
            new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {97, 89, 85, 82}}
        };

        return students;
    }

    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Group by true or false.
        // Query variable is an I Enumerable<IGrouping<bool, Student>>
        var booleanGroupQuery =
            from student in students
            group student by student.Scores.Average() >= 80; //pass or fail!

        // Execute the query and access items in each group
        foreach (var studentGroup in booleanGroupQuery)
        {
            Console.WriteLine(studentGroup.Key == true ? "High averages" : "Low averages");
            foreach (var student in studentGroup)
            {
                Console.WriteLine("  {0}, {1}:{2}", student.Last, student.First, student.Scores.Average());
            }
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
Low averages
Omelchenko, Svetlana:77.5
O'Donnell, Claire:72.25
Garcia, Cesar:75.5
High averages
Mortensen, Sven:93.5
Garcia, Debra:88.25
*/

```

## 숫자 범위로 그룹화

다음 예제에서는 식을 사용하여 백분위수 범위를 나타내는 숫자 그룹 키를 만듭니다. 메서드 호출 결과를 저장

할 편리한 위치로 `let`을 사용하여 `group` 절에서 메서드를 두 번 호출할 필요가 없도록 합니다. 쿼리 식에 메서드를 안전하게 사용하는 방법에 대한 자세한 내용은 [쿼리 식의 예외 처리](#)를 참조하세요.

```
class GroupSample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that each element
        // in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
            new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 72, 81,
60}},
            new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},
            new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {99, 89, 91, 95}},
            new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {72, 81, 65, 84}},
            new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {97, 89, 85, 82}}
        };

        return students;
    }

    // This method groups students into percentile ranges based on their
    // grade average. The Average method returns a double, so to produce a whole
    // number it is necessary to cast to int before dividing by 10.
    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Write the query.
        var studentQuery =
            from student in students
            let avg = (int)student.Scores.Average()
            group student by (avg / 10) into g
            orderby g.Key
            select g;

        // Execute the query.
        foreach (var studentGroup in studentQuery)
        {
            int temp = studentGroup.Key * 10;
            Console.WriteLine("Students with an average between {0} and {1}", temp, temp + 10);
            foreach (var student in studentGroup)
            {
                Console.WriteLine("  {0}, {1}:{2}", student.Last, student.First, student.Scores.Average());
            }
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
   Students with an average between 70 and 80
   Omelchenko, Svetlana:77.5
   O'Donnell, Claire:72.25
   ...
   ...
   ...

```

```
Garcia, Cesar:75.5
Students with an average between 80 and 90
Garcia, Debra:88.25
Students with an average between 90 and 100
Mortensen, Sven:93.5
*/
```

## 복합 키로 그룹화

둘 이상의 키에 따라 요소를 그룹화하려는 경우 복합 키를 사용합니다. 무명 형식이나 명명된 형식을 통해 키 요소를 저장하여 복합 키를 만듭니다. 다음 예제에서는 `Person` 클래스가 `surname` 및 `city`라는 멤버로 선언되었다고 가정합니다. `group` 절은 성과 도시가 동일한 각 개인 집합에 대해 별도 그룹이 생성되도록 합니다.

```
group person by new {name = person.surname, city = person.city};
```

쿼리 변수를 다른 메서드에 전달해야 하는 경우 명명된 형식을 사용합니다. 키에 대해 자동 구현 속성을 사용하여 특수 클래스를 만든 다음 [Equals](#) 및 [GetHashCode](#) 메서드를 재정의합니다. 구조체를 사용할 수도 있으며, 이 경우 이러한 메서드를 엄격하게 재정의하지 않아도 됩니다. 자세한 내용은 [자동으로 구현된 속성을 사용하여 간단한 클래스를 구현하는 방법](#) 및 [디렉터리 트리의 중복 파일을 쿼리하는 방법](#)을 참조하세요. 두 번째 문서에는 명명된 형식과 함께 복합 키를 사용하는 방법을 보여 주는 코드 예제가 있습니다.

## 예제

다음 예제에서는 추가 쿼리 논리가 그룹에 적용되지 않는 경우 소스 데이터를 그룹으로 정렬하기 위한 표준 패턴을 보여 줍니다. 이를 비연속 그룹화라고 합니다. 문자열 배열에 있는 요소는 첫 문자에 따라 그룹화됩니다. 쿼리 결과는 그룹에 각 항목을 포함하는 공용 `Key` 속성 형식 `char` 및 `IEnumerable<T>` 컬렉션을 포함하는 `IGrouping< TKey, TElement >` 형식입니다.

`group` 절의 결과는 시퀀스의 시퀀스입니다. 따라서 반환된 각 그룹 내의 개별 요소에 액세스하려면 다음 예제와 같이 그룹 키를 반복하는 루프 안에 중첩된 `foreach` 루프를 사용합니다.

```

class GroupExample1
{
    static void Main()
    {
        // Create a data source.
        string[] words = { "blueberry", "chimpanzee", "abacus", "banana", "apple", "cheese" };

        // Create the query.
        var wordGroups =
            from w in words
            group w by w[0];

        // Execute the query.
        foreach (var wordGroup in wordGroups)
        {
            Console.WriteLine("Words that start with the letter '{0}':", wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine(word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
   Words that start with the letter 'b':
   blueberry
   banana
   Words that start with the letter 'c':
   chimpanzee
   cheese
   Words that start with the letter 'a':
   abacus
   apple
*/

```

## 예제

이 예제에서는 `into` 와 함께 *continuation* 을 사용하여 그룹을 만든 후 그룹에서 추가 논리를 수행하는 방법을 보여 줍니다. 자세한 내용은 [into](#)를 참조하세요. 다음 예제에서는 각 그룹을 쿼리하여 키 값이 모음인 그룹만 선택합니다.

```

class GroupClauseExample2
{
    static void Main()
    {
        // Create the data source.
        string[] words2 = { "blueberry", "chimpanzee", "abacus", "banana", "apple", "cheese", "elephant",
"umbrella", "anteater" };

        // Create the query.
        var wordGroups2 =
            from w in words2
            group w by w[0] into grps
            where (grps.Key == 'a' || grps.Key == 'e' || grps.Key == 'i'
                || grps.Key == 'o' || grps.Key == 'u')
            select grps;

        // Execute the query.
        foreach (var wordGroup in wordGroups2)
        {
            Console.WriteLine("Groups that start with a vowel: {0}", wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine("    {0}", word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
Groups that start with a vowel: a
    abacus
    apple
    anteater
Groups that start with a vowel: e
    elephant
Groups that start with a vowel: u
    umbrella
*/

```

## 설명

**group** 절은 컴파일 시간에 [GroupBy](#) 메서드 호출로 변환됩니다.

## 참조

- [IGrouping< TKey, TElement >](#)
- [GroupBy](#)
- [ThenBy](#)
- [ThenByDescending](#)
- [쿼리 키워드](#)
- [LINQ\(Language-Integrated Query\)](#)
- [중첩 그룹 만들기](#)
- [쿼리 결과 그룹화](#)
- [그룹화 작업에서 하위 쿼리 수행](#)

# into(C# 참조)

2021-02-18 • 2 minutes to read • [Edit Online](#)

`into` 상황별 키워드를 사용하여 `group`, `join`, `select` 절의 결과를 새 식별자에 저장하기 위한 임시 식별자를 만들 수 있습니다. 이 식별자 자체는 추가 쿼리 명령의 생성기일 수 있습니다. `group` 또는 `select` 절에 사용할 경우 새 식별자의 사용을 연속이라고도 합니다.

## 예제

다음 예제에서는 `into` 키워드를 사용하여 임시 식별자 `fruitGroup`을 활성화하는 방법을 보여 줍니다. 이 식별자는 `IGrouping`의 유추된 형식을 갖습니다. 식별자를 사용하여 각 그룹에서 `Count` 메서드를 호출하고 둘 이상의 단어를 포함하는 그룹만 선택할 수 있습니다.

```
class IntoSample1
{
    static void Main()
    {

        // Create a data source.
        string[] words = { "apples", "blueberries", "oranges", "bananas", "apricots" };

        // Create the query.
        var wordGroups1 =
            from w in words
            group w by w[0] into fruitGroup
            where fruitGroup.Count() >= 2
            select new { FirstLetter = fruitGroup.Key, Words = fruitGroup.Count() };

        // Execute the query. Note that we only iterate over the groups,
        // not the items in each group
        foreach (var item in wordGroups1)
        {
            Console.WriteLine("{0} has {1} elements.", item.FirstLetter, item.Words);
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   a has 2 elements.
   b has 2 elements.
*/
```

각 그룹에서 추가 쿼리 작업을 수행하려는 경우에만 `group` 절에 `into`를 사용하면 됩니다. 자세한 내용은 [group 절](#)을 참조하세요.

`join` 절에 `into`를 사용하는 방법의 예는 [join 절](#)을 참조하세요.

## 참고 항목

- [쿼리 키워드\(LINQ\)](#)
- [C#의 LINQ](#)
- [group 절](#)

# orderby 절(C# 참조)

2020-11-02 • 4 minutes to read • [Edit Online](#)

쿼리 식에서 `orderby` 절은 반환된 시퀀스 또는 하위 시퀀스(그룹)가 오름차순이나 내림차순으로 정렬되도록 합니다. 하나 이상의 보조 정렬 작업을 수행하기 위해 여러 키를 지정할 수 있습니다. 정렬은 요소 형식에 대한 기본 비교자에 의해 수행됩니다. 기본 정렬 순서는 오름차순입니다. 사용자 지정 비교자를 지정할 수도 있습니다. 그러나 메서드 기반 구문을 통해서만 사용할 수 있습니다. 자세한 내용은 [데이터 정렬](#)을 참조하세요.

## 예제

다음 예제에서 첫 번째 쿼리는 A부터 시작하여 사전순으로 단어를 정렬하고, 두 번째 쿼리는 동일한 단어를 내림차순으로 정렬합니다. `ascending` 키워드는 기본 정렬 값이며 생략할 수 있습니다.

```

class OrderbySample1
{
    static void Main()
    {
        // Create a delicious data source.
        string[] fruits = { "cherry", "apple", "blueberry" };

        // Query for ascending sort.
        IEnumerable<string> sortAscendingQuery =
            from fruit in fruits
            orderby fruit //"ascending" is default
            select fruit;

        // Query for descending sort.
        IEnumerable<string> sortDescendingQuery =
            from w in fruits
            orderby w descending
            select w;

        // Execute the query.
        Console.WriteLine("Ascending:");
        foreach (string s in sortAscendingQuery)
        {
            Console.WriteLine(s);
        }

        // Execute the query.
        Console.WriteLine(Environment.NewLine + "Descending:");
        foreach (string s in sortDescendingQuery)
        {
            Console.WriteLine(s);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
Ascending:
apple
blueberry
cherry

Descending:
cherry
blueberry
apple
*/

```

## 예제

다음 예제에서는 학생의 성을 기준으로 1차 정렬을 수행한 다음 이름을 기준으로 2차 정렬을 수행합니다.

```

class OrderbySample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
    }

    public static List<Student> GetStudents()

```

```

{
    // Use a collection initializer to create the data source. Note that each element
    // in the list contains an inner sequence of scores.
    List<Student> students = new List<Student>
    {
        new Student {First="Svetlana", Last="Omelchenko", ID=111},
        new Student {First="Claire", Last="O'Donnell", ID=112},
        new Student {First="Sven", Last="Mortensen", ID=113},
        new Student {First="Cesar", Last="Garcia", ID=114},
        new Student {First="Debra", Last="Garcia", ID=115}
    };
}

return students;
}
static void Main(string[] args)
{
    // Create the data source.
    List<Student> students = GetStudents();

    // Create the query.
    IEnumerable<Student> sortedStudents =
        from student in students
        orderby student.Last ascending, student.First ascending
        select student;

    // Execute the query.
    Console.WriteLine("sortedStudents:");
    foreach (Student student in sortedStudents)
        Console.WriteLine(student.Last + " " + student.First);

    // Now create groups and sort the groups. The query first sorts the names
    // of all students so that they will be in alphabetical order after they are
    // grouped. The second orderby sorts the group keys in alpha order.
    var sortedGroups =
        from student in students
        orderby student.Last, student.First
        group student by student.Last[0] into newGroup
        orderby newGroup.Key
        select newGroup;

    // Execute the query.
    Console.WriteLine(Environment.NewLine + "sortedGroups:");
    foreach (var studentGroup in sortedGroups)
    {
        Console.WriteLine(studentGroup.Key);
        foreach (var student in studentGroup)
        {
            Console.WriteLine("  {0}, {1}", student.Last, student.First);
        }
    }

    // Keep the console window open in debug mode
    Console.WriteLine("Press any key to exit.");
    Console.ReadKey();
}
/* Output:
sortedStudents:
Garcia Cesar
Garcia Debra
Mortensen Sven
O'Donnell Claire
Omelchenko Svetlana

sortedGroups:
G
  Garcia, Cesar
  Garcia, Debra
M

```

Mortensen, Sven  
0  
O'Donnell, Claire  
Omelchenko, Svetlana  
\*/

## 설명

컴파일 시간에 `orderby` 절은 `OrderBy` 메서드 호출로 변환됩니다. `orderby` 절의 여러 키는 `ThenBy` 메서드 호출로 변환됩니다.

## 참고 항목

- [C# 참조](#)
- [쿼리 키워드\(LINQ\)](#)
- [C#의 LINQ](#)
- [group 절](#)
- [LINQ\(Language-Integrated Query\)](#)

# join 절(C# 참조)

2020-11-02 • 19 minutes to read • [Edit Online](#)

`join` 절은 개체 모델에서 직접적인 관계가 없는 서로 다른 소스 시퀀스의 요소를 연결하는 데 유용합니다. 각 소스의 요소가 같은지 비교할 수 있는 일부 값을 공유하기만 하면 됩니다. 예를 들어 식품 유통업체에는 특정 제품의 공급업체 목록과 구매자 목록이 있을 수 있습니다. 예를 들어 `join` 절은 모두 동일한 지역에 있는, 해당 제품의 공급업체 및 구매자 목록을 만드는 데 사용할 수 있습니다.

`join` 절은 두 개의 소스 시퀀스를 입력으로 사용합니다. 각 시퀀스의 요소는 다른 시퀀스의 속성과 비교할 수 있는 속성이거나 해당 속성을 포함해야 합니다. `join` 절은 특수한 `equals` 키워드를 사용하여 지정된 키가 같은지 비교합니다. `join` 절로 수행된 모든 조인은 동등 조인입니다. `join` 절의 출력 형태는 수행하는 조인의 특정 유형에 따라 달라집니다. 다음은 세 가지 가장 일반적인 조인 유형입니다.

- 내부 조인
- 그룹 조인
- 왼쪽 우선 외부 조인

## 내부 조인

다음 예제에서는 간단한 내부 동등 조인을 보여 줍니다. 이 쿼리는 "제품 이름/범주" 쌍의 기본 시퀀스를 생성합니다. 동일한 범주 문자열이 여러 요소에 나타납니다. `categories` 의 요소에 일치하는 `products` 가 없는 경우 해당 범주는 결과에 나타나지 않습니다.

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name }; //produces flat sequence
```

자세한 내용은 [내부 조인 수행](#)을 참조하세요.

## 그룹 조인

`into` 식을 포함한 `join` 절을 그룹 조인이라고 합니다.

```
var innerGroupJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into prodGroup
    select new { CategoryName = category.Name, Products = prodGroup };
```

그룹 조인은 왼쪽 소스 시퀀스의 요소를 오른쪽 소스 시퀀스에서 일치하는 하나 이상의 요소와 연결하는 계층적 결과 시퀀스를 생성합니다. 관계적인 측면에서 그룹 조인에는 해당 항목이 없으며, 그룹 조인은 기본적으로 개체 배열의 시퀀스입니다.

왼쪽 소스의 요소와 일치하는 오른쪽 소스 시퀀스의 요소가 없을 경우 `join` 절은 해당 항목에 대해 빈 배열을 생성합니다. 따라서 결과 시퀀스가 그룹으로 구성된다는 점을 제외하면 그룹 조인은 기본적으로 내부 동등 조인입니다.

그룹 조인의 결과를 선택하는 경우 항목에 액세스할 수 있지만 항목이 일치되는 키를 식별할 수는 없습니다. 따라서 이전 예제와 같이 키 이름도 포함하는 새로운 유형으로 그룹 조인의 결과를 선택하는 것이 일반적으로 더 유용합니다.

또한 그룹 조인의 결과를 또 다른 하위 쿼리의 생성기로 사용할 수도 있습니다.

```
var innerGroupJoinQuery2 =  
    from category in categories  
    join prod in products on category.ID equals prod.CategoryID into prodGroup  
    from prod2 in prodGroup  
    where prod2.UnitPrice > 2.50M  
    select prod2;
```

자세한 내용은 [그룹화 조인 수행](#)을 참조하세요.

## 왼쪽 우선 외부 조인

왼쪽 우선 외부 조인에서는 오른쪽 시퀀스에 일치하는 요소가 없는 경우에도 왼쪽 소스 시퀀스의 모든 요소가 반환됩니다. LINQ에서 왼쪽 우선 외부 조인을 수행하려면 그룹 조인과 함께 `DefaultIfEmpty` 메서드를 사용하여 왼쪽 요소에 일치하는 요소가 없을 경우 생성할 기본 오른쪽 요소를 지정합니다. `null`을 모든 참조 형식의 기본값으로 사용하거나 사용자 정의 기본 형식을 지정할 수 있습니다. 다음 예제에서는 사용자 정의 기본 형식을 보여 줍니다.

```
var leftOuterJoinQuery =  
    from category in categories  
    join prod in products on category.ID equals prod.CategoryID into prodGroup  
    from item in prodGroup.DefaultIfEmpty(new Product { Name = String.Empty, CategoryID = 0 })  
    select new { CatName = category.Name, ProdName = item.Name };
```

자세한 내용은 [왼쪽 우선 외부 조인 수행](#)을 참조하세요.

## 같음 연산자

`join` 절은 동등 조인을 수행합니다. 즉, 일치 항목만을 기준으로 두 키가 같은지 비교할 수 있습니다. "보다 큼"이나 "같지 않음"과 같은 다른 유형의 비교는 지원되지 않습니다. 모든 조인이 동등 조인인지 확인하기 위해 `join` 절은 `==` 연산자 대신 `equals` 키워드를 사용합니다. `equals` 키워드는 `join` 절에서만 사용할 수 있으며 한 가지 중요한 측면에서 `==` 연산자와 다릅니다. `equals`를 사용할 경우 왼쪽 키는 외부 소스 시퀀스를 사용하고 오른쪽 키는 내부 소스를 사용합니다. 외부 소스는 `equals`의 왼쪽 범위에만 있고 내부 소스 시퀀스는 오른쪽 범위에만 있습니다.

## 비동등 조인

여러 개의 `from` 절을 사용하여 비동등 조인, 크로스 조인 및 기타 사용자 지정 조인 작업을 수행하면 새 시퀀스를 독립적으로 쿼리에 지정할 수 있습니다. 자세한 내용은 [사용자 지정 조인 작업 수행](#)을 참조하세요.

## 개체 컬렉션 및 관계형 테이블에서 조인

LINQ 쿼리 식에서 조인 작업은 개체 컬렉션에서 수행됩니다. 개체 컬렉션은 두 개의 관계형 테이블과 정확히 동일한 방식으로 "조인"할 수 없습니다. LINQ에서는 두 소스 시퀀스가 관계로 연결되지 않는 경우에만 명시적 `join` 절이 필요합니다. LINQ to SQL로 작업할 때 외래 키 테이블은 기본 테이블의 속성으로 개체 모델에 표시됩니다. 예를 들어 Northwind 데이터베이스에서 Customer 테이블은 Orders 테이블과 외래 키 관계가 있습니다. 테이블을 개체 모델에 매핑하면 Customer 클래스는 해당 Customer와 연결된 Orders 컬렉션을 포함하는 Orders 속성을 갖습니다. 사실상 조인은 이미 수행된 것입니다.

LINQ to SQL 컨텍스트에서 관련 테이블을 쿼리하는 방법에 대한 자세한 내용은 [방법: 데이터베이스 관계 매핑](#)을 참조하세요.

## 복합 키

복합 키를 사용하여 여러 값이 같은지 여부를 테스트할 수 있습니다. 자세한 내용은 [복합 키를 사용하여 조인을 참조하세요](#). 복합 키는 `group` 절에서도 사용할 수 있습니다.

## 예제

다음 예제에서는 동일한 데이터 소스에서 동일한 일치하는 키를 사용하여 내부 조인, 그룹 조인 및 왼쪽 우선 외부 조인의 결과를 비교합니다. 콘솔 디스플레이에 결과를 명확하게 나타내기 위해 이러한 예제에 몇 가지 추가 코드를 추가합니다.

```
class JoinDemonstration
{
    #region Data

    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
    {
        new Category {Name="Beverages", ID=001},
        new Category {Name="Condiments", ID=002},
        new Category {Name="Vegetables", ID=003},
        new Category {Name="Grains", ID=004},
        new Category {Name="Fruit", ID=005}
    };

    // Specify the second data source.
    List<Product> products = new List<Product>()
    {
        new Product {Name="Cola", CategoryID=001},
        new Product {Name="Tea", CategoryID=001},
        new Product {Name="Mustard", CategoryID=002},
        new Product {Name="Pickles", CategoryID=002},
        new Product {Name="Carrots", CategoryID=003},
        new Product {Name="Bok Choy", CategoryID=003},
        new Product {Name="Peaches", CategoryID=005},
        new Product {Name="Melons", CategoryID=005},
    };
    #endregion

    static void Main(string[] args)
    {
        JoinDemonstration app = new JoinDemonstration();

        app.InnerJoin();
        app.GroupJoin();
        app.GroupInnerJoin();
        app.GroupJoin3();
        app.LeftOuterJoin();
        app.LeftOuterJoin2();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    void InnerJoin()
```

```

    }

    // Create the query that selects
    // a property from each element.
    var innerJoinQuery =
        from category in categories
        join prod in products on category.ID equals prod.CategoryID
        select new { Category = category.ID, Product = prod.Name };

    Console.WriteLine("InnerJoin:");
    // Execute the query. Access results
    // with a simple foreach statement.
    foreach (var item in innerJoinQuery)
    {
        Console.WriteLine("{0,-10}{1}", item.Product, item.Category);
    }
    Console.WriteLine("InnerJoin: {0} items in 1 group.", innerJoinQuery.Count());
    Console.WriteLine(System.Environment.NewLine);
}

void GroupJoin()
{
    // This is a demonstration query to show the output
    // of a "raw" group join. A more typical group join
    // is shown in the GroupInnerJoin method.
    var groupJoinQuery =
        from category in categories
        join prod in products on category.ID equals prod.CategoryID into prodGroup
        select prodGroup;

    // Store the count of total items (for demonstration only).
    int totalItems = 0;

    Console.WriteLine("Simple GroupJoin:");

    // A nested foreach statement is required to access group items.
    foreach (var prodGrouping in groupJoinQuery)
    {
        Console.WriteLine("Group:");
        foreach (var item in prodGrouping)
        {
            totalItems++;
            Console.WriteLine("  {0,-10}{1}", item.Name, item.CategoryID);
        }
    }
    Console.WriteLine("Unshaped GroupJoin: {0} items in {1} unnamed groups", totalItems,
groupJoinQuery.Count());
    Console.WriteLine(System.Environment.NewLine);
}

void GroupInnerJoin()
{
    var groupJoinQuery2 =
        from category in categories
        orderby category.ID
        join prod in products on category.ID equals prod.CategoryID into prodGroup
        select new
        {
            Category = category.Name,
            Products = from prod2 in prodGroup
                        orderby prod2.Name
                        select prod2
        };
}

//Console.WriteLine("GroupInnerJoin:");
int totalItems = 0;

Console.WriteLine("GroupInnerJoin:");
foreach (var productGroup in groupJoinQuery2)
{
}

```

```

        Console.WriteLine(productGroup.Category);
        foreach (var prodItem in productGroup.Products)
        {
            totalItems++;
            Console.WriteLine(" {0,-10} {1}", prodItem.Name, prodItem.CategoryID);
        }
    }
    Console.WriteLine("GroupInnerJoin: {0} items in {1} named groups", totalItems,
groupJoinQuery2.Count());
    Console.WriteLine(System.Environment.NewLine);
}

void GroupJoin3()
{
    var groupJoinQuery3 =
        from category in categories
        join product in products on category.ID equals product.CategoryID into prodGroup
        from prod in prodGroup
        orderby prod.CategoryID
        select new { Category = prod.CategoryID, ProductName = prod.Name };

//Console.WriteLine("GroupInnerJoin:");
int totalItems = 0;

Console.WriteLine("GroupJoin3:");
foreach (var item in groupJoinQuery3)
{
    totalItems++;
    Console.WriteLine(" {0}:{1}", item.ProductName, item.Category);
}

Console.WriteLine("GroupJoin3: {0} items in 1 group", totalItems);
Console.WriteLine(System.Environment.NewLine);
}

void LeftOuterJoin()
{
    // Create the query.
    var leftOuterQuery =
        from category in categories
        join prod in products on category.ID equals prod.CategoryID into prodGroup
        select prodGroup.DefaultIfEmpty(new Product() { Name = "Nothing!", CategoryID = category.ID });

    // Store the count of total items (for demonstration only).
    int totalItems = 0;

    Console.WriteLine("Left Outer Join:");

    // A nested foreach statement is required to access group items
    foreach (var prodGrouping in leftOuterQuery)
    {
        Console.WriteLine("Group:");
        foreach (var item in prodGrouping)
        {
            totalItems++;
            Console.WriteLine(" {0,-10}{1}", item.Name, item.CategoryID);
        }
    }
    Console.WriteLine("LeftOuterJoin: {0} items in {1} groups", totalItems, leftOuterQuery.Count());
    Console.WriteLine(System.Environment.NewLine);
}

void LeftOuterJoin2()
{
    // Create the query.
    var leftOuterQuery2 =
        from category in categories

```

```

        join prod in products on category.ID equals prod.CategoryID into prodGroup
        from item in prodGroup.DefaultIfEmpty()
        select new { Name = item == null ? "Nothing!" : item.Name, CategoryID = category.ID };

        Console.WriteLine("LeftOuterJoin2: {0} items in 1 group", leftOuterQuery2.Count());
        // Store the count of total items
        int totalItems = 0;

        Console.WriteLine("Left Outer Join 2:");

        // Groups have been flattened.
        foreach (var item in leftOuterQuery2)
        {
            totalItems++;
            Console.WriteLine("{0,-10}{1}", item.Name, item.CategoryID);
        }
        Console.WriteLine("LeftOuterJoin2: {0} items in 1 group", totalItems);
    }
}
/*Output:

InnerJoin:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Peaches   5
Melons    5
InnerJoin: 8 items in 1 group.

Unshaped GroupJoin:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
Group:
    Peaches   5
    Melons    5
Unshaped GroupJoin: 8 items in 5 unnamed groups

GroupInnerJoin:
Beverages
    Cola      1
    Tea       1
Condiments
    Mustard   2
    Pickles   2
Vegetables
    Bok Choy  3
    Carrots   3
Grains
Fruit
    Melons    5
    Peaches   5
GroupInnerJoin: 8 items in 5 named groups

GroupJoin3:
    Cola:1

```

```

Tea:1
Mustard:2
Pickles:2
Carrots:3
Bok Choy:3
Peaches:5
Melons:5
GroupJoin3: 8 items in 1 group

Left Outer Join:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy   3
Group:
    Nothing!  4
Group:
    Peaches   5
    Melons    5
LeftOuterJoin: 9 items in 5 groups

LeftOuterJoin2: 9 items in 1 group
Left Outer Join 2:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy   3
Nothing!  4
Peaches   5
Melons    5
LeftOuterJoin2: 9 items in 1 group
Press any key to exit.
*/

```

## 설명

뒤에 `into` 가 오지 않는 `join` 절은 [Join](#) 메서드 호출로 변환됩니다. 뒤에 `into` 가 오는 `join` 절은 [GroupJoin](#) 메서드 호출로 변환됩니다.

## 참고 항목

- [쿼리 키워드\(LINQ\)](#)
- [LINQ\(Language-Integrated Query\)](#)
- [조인 작업](#)
- [group 절](#)
- [왼쪽 우선 외부 조인 수행](#)
- [내부 조인 수행](#)
- [그룹화 조인 수행](#)
- [Join 절 결과를 서순대로 정렬](#)
- [복합 키를 사용하여 조인](#)
- [Visual Studio용 호환 데이터베이스 시스템](#)

# let 절(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

쿼리 식에서 후속 절에 사용하기 위해 하위 식의 결과를 저장하면 유용한 경우가 있습니다. 새 범위 변수를 만들고 제공한 식의 결과로 초기화하는 `let` 키워드를 사용하면 이 작업을 수행할 수 있습니다. 값으로 초기화되면 범위 변수를 사용하여 다른 값을 저장할 수 없습니다. 그러나 범위 변수가 쿼리 가능 형식을 포함할 경우 쿼리할 수 있습니다.

## 예제

다음 예제에서 `let`은 다음 두 가지 방법으로 사용됩니다.

1. 그 자체를 쿼리할 수 있는 열거 가능한 형식을 만듭니다.
2. 쿼리가 범위 변수 `word`에서 `ToLower`를 한 번만 호출할 수 있도록 합니다. `let`을 사용하지 않을 경우 `where` 절의 각 조건자에서 `ToLower`를 호출해야 합니다.

```

class LetSample1
{
    static void Main()
    {
        string[] strings =
        {
            "A penny saved is a penny earned.",
            "The early bird catches the worm.",
            "The pen is mightier than the sword."
        };

        // Split the sentence into an array of words
        // and select those whose first letter is a vowel.
        var earlyBirdQuery =
            from sentence in strings
            let words = sentence.Split(' ')
            from word in words
            let w = word.ToLower()
            where w[0] == 'a' || w[0] == 'e'
                || w[0] == 'i' || w[0] == 'o'
                || w[0] == 'u'
            select word;

        // Execute the query.
        foreach (var v in earlyBirdQuery)
        {
            Console.WriteLine("\"{0}\" starts with a vowel", v);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
   "A" starts with a vowel
   "is" starts with a vowel
   "a" starts with a vowel
   "earned." starts with a vowel
   "early" starts with a vowel
   "is" starts with a vowel
*/

```

## 참조

- [C# 참조](#)
- [쿼리 키워드\(LINQ\)](#)
- [C#의 LINQ](#)
- [LINQ\(Language-Integrated Query\)](#)
- [쿼리 식의 예외 처리](#)

# ascending(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

`ascending` 상황별 키워드는 쿼리 식의 `orderby` 절에서 사용되어 정렬 순서를 가장 작은 값에서 가장 큰 값의 오름차순으로 지정합니다. `ascending`은 기본 정렬 순서이므로 지정할 필요가 없습니다.

## 예제

다음 예제에서는 `orderby` 절에 `ascending`을 사용하는 방법을 보여 줍니다.

```
IEnumerable<string> sortAscendingQuery =  
    from vegetable in vegetables  
    orderby vegetable ascending  
    select vegetable;
```

## 참고 항목

- [C# 참조](#)
- [C#의 LINQ](#)
- [descending](#)

# descending(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

`descending` 상황별 키워드는 쿼리 식의 `orderby` 절에서 사용되어 정렬 순서를 가장 큰 값에서 가장 작은 값의 내림차순으로 지정합니다.

## 예제

다음 예제에서는 `orderby` 절에 `descending`을 사용하는 방법을 보여 줍니다.

```
IEnumerable<string> sortDescendingQuery =  
    from vegetable in vegetables  
    orderby vegetable descending  
    select vegetable;
```

## 참고 항목

- [C# 참조](#)
- [C#의 LINQ](#)
- [ascending](#)

# on(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

`on` 상황별 키워드는 쿼리 식의 `join` 절에 사용되어 조인 조건을 지정합니다.

## 예제

다음 예제에서는 `join` 절에 `on`을 사용하는 방법을 보여 줍니다.

```
var innerJoinQuery =  
    from category in categories  
    join prod in products on category.ID equals prod.CategoryID  
    select new { ProductName = prod.Name, Category = category.Name };
```

## 참고 항목

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

# equals(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

`equals` 상황별 키워드는 쿼리 식의 `join` 절에서 사용되어 두 시퀀스의 요소를 비교합니다. 자세한 내용은 [join 절](#)을 참조하세요.

## 예제

다음 예제에서는 `join` 절에 `equals` 키워드를 사용하는 방법을 보여 줍니다.

```
var innerJoinQuery =  
    from category in categories  
    join prod in products on category.ID equals prod.CategoryID  
    select new { ProductName = prod.Name, Category = category.Name };
```

## 참고 항목

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

# by(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

**by** 상황별 키워드는 쿼리 식의 **group** 절에서 사용되어 반환된 항목의 그룹화 방법을 지정합니다. 자세한 내용은 [group 절](#)을 참조하세요.

## 예제

다음 예제에서는 **group** 절에 **by** 상황별 키워드를 사용하여 각 학생의 성에서 첫 번째 문자에 따라 학생들을 그룹화하도록 지정하는 방법을 보여 줍니다.

```
var query = from student in students
            group student by student.LastName[0];
```

## 참고 항목

- [C#의 LINQ](#)

# in(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

`in` 키워드는 다음과 같은 컨텍스트에서 사용됩니다.

- 제네릭 인터페이스와 대리자의 [제네릭 형식 매개 변수](#)
- 값이 아닌 참조로 메서드에 인수를 전달할 수 있도록 하는 [매개 변수 한정자](#)로 사용
- [foreach](#) 문
- LINQ 쿼리 식의 [from 절](#).
- LINQ 쿼리 식의 [join 절](#)

## 참조

- [C# 키워드](#)
- [C# 참조](#)

# C# 연산자 및 식(C# 참조)

2021-02-18 • 12 minutes to read • [Edit Online](#)

C#은 여러 연산자를 제공합니다. 상당수의 연산자는 [기본 제공 형식](#)에서 지원되며 해당 형식의 값을 사용하여 기본 연산을 수행할 수 있습니다. 해당 연산자는 다음 그룹을 포함합니다.

- [산술 연산자](#): 숫자 피연산자를 사용하여 산술 연산을 수행함
- [비교 연산자](#): 숫자 피연산자를 비교함
- [부울 논리 연산자](#): `bool` 피연산자를 사용하여 논리 연산을 수행함
- [비트 및 시프트 연산자](#): 정수 형식의 피연산자를 사용하여 비트 또는 시프트 연산을 수행함
- [같음 연산자](#): 해당 피연산자가 같은지를 확인함

일반적으로 이 연산자들을 [오버로드](#)할 수 있습니다. 즉, 사용자 정의 형식의 피연산자에 대한 연산자 동작을 지정할 수 있습니다.

가장 간단한 C# 식은 리터럴(예: [정수](#) 및 [실수](#)) 및 변수 이름입니다. 연산자를 사용하여 이들을 복잡한 식으로 결합할 수 있습니다. 연산자 [우선 순위](#) 및 [결합성](#)은 식에서 연산이 수행되는 순서를 결정합니다. 괄호를 사용하여 연산자 우선 순위 및 결합성에 따라 주어진 계산 순서를 변경할 수 있습니다.

다음 코드에서 식의 예는 할당의 오른쪽에 있습니다.

```
int a, b, c;
a = 7;
b = a;
c = b++;
b = a + b * c;
c = a >= 100 ? b : c / 10;
a = (int)Math.Sqrt(b * b + c * c);

string s = "String literal";
char l = s[s.Length - 1];

var numbers = new List<int>(new[] { 1, 2, 3 });
b = numbers.FindLast(n => n > 1);
```

일반적으로 식은 결과를 생성하고 다른 식에 포함될 수 있습니다. `void` 메서드 호출은 결과를 생성하지 않는 식의 예입니다. 다음 예에 나와 있는 것처럼 이 메서드 호출은 [문](#)으로만 사용할 수 있습니다.

```
Console.WriteLine("Hello, world!");
```

C#에서 제공하는 몇 가지 다른 종류의 식은 다음과 같습니다.

- [보간된 문자열 식](#): 다음과 같이 형식 문자열을 만들 수 있는 편리한 구문을 제공합니다.

```
var r = 2.3;
var message = $"The area of a circle with radius {r} is {Math.PI * r * r:F3}.";
Console.WriteLine(message);
// Output:
// The area of a circle with radius 2.3 is 16.619.
```

- [람다 식](#): 다음과 같이 익명 함수를 만들 수 있습니다.

```

int[] numbers = { 2, 3, 4, 5 };
var maximumSquare = numbers.Max(x => x * x);
Console.WriteLine(maximumSquare);
// Output:
// 25

```

- **쿼리 식**: 다음과 같이 C#에서 직접 쿼리 기능을 사용할 수 있습니다.

```

var scores = new[] { 90, 97, 78, 68, 85 };
IEnumerable<int> highScoresQuery =
    from score in scores
    where score > 80
    orderby score descending
    select score;
Console.WriteLine(string.Join(" ", highScoresQuery));
// Output:
// 97 90 85

```

[식 본문 정의](#)를 사용하여 메서드, 생성자, 속성, 인덱서 또는 종료자에 대한 간결한 정의를 제공할 수 있습니다.

## 연산자 우선 순위

여러 연산자가 있는 식에서 우선 순위가 높은 연산자는 연산자가 우선 순위가 낮은 연산자보다 먼저 계산됩니다. 다음 예제에서는 곱하기가 더하기보다 높은 우선 순위를 가지므로 곱하기가 먼저 수행됩니다.

```

var a = 2 + 2 * 2;
Console.WriteLine(a); // output: 6

```

괄호를 사용하여 연산자 우선 순위에 따라 주어진 계산 순서를 변경합니다.

```

var a = (2 + 2) * 2;
Console.WriteLine(a); // output: 8

```

다음 표에서는 우선순위가 가장 높은 것부터 시작하여 순서대로 C# 연산자를 나열합니다. 각 행 내의 연산자는 우선 순위가 같습니다.

연산자	범주 또는 이름
<code>x.y, f(x), a[i], x?.y, x?[y], x++, x--, x!, new, typeof, checked, unchecked, default, nameof, delegate, sizeof, stackalloc, x-&gt;y</code>	주
<code>+x, -x, !x, ~x, ++x, --x, ^x, (T)x, await, &amp;x, *x, true 및 false</code>	단항
<code>x..y</code>	범위
<code>switch</code>	<code>switch</code> 식
<code>with</code>	<code>with</code> 식
<code>x * y, x / y, x % y</code>	곱하기
<code>x + y, x - y</code>	더하기

연산자	범주 또는 이름
<code>x &lt;&lt; y, x &gt;&gt; y</code>	Shift
<code>x &lt; y, x &gt; y, x &lt;= y, x &gt;= y, is, as</code>	관계형 및 형식 테스트
<code>x == y, x != y</code>	같음
<code>x &amp; y</code>	부울 논리 AND 또는 비트 논리 AND
<code>x ^ y</code>	부울 논리 XOR 또는 비트 논리 XOR
<code>x   y</code>	OR 또는 비트 논리 OR
<code>x &amp;&amp; y</code>	조건부 AND
<code>x    y</code>	조건부 OR
<code>x ?? y</code>	Null 병합 연산자
<code>c ?: f</code>	조건 연산자
<code>x = y, x += y, x -= y, x *= y, x /= y, x %= y, x &amp;= y, x  = y, x ^= y, x &lt;&lt;= y, x &gt;&gt;= y, x ??= y, =&gt;</code>	할당 및 람다 선언

## 연산자 결합성

연산자의 우선 순위가 같은 경우 연산자의 결합성이 연산이 수행되는 순서를 결정합니다.

- ‘왼쪽 결합성이 있는’ 연산자는 왼쪽에서 오른쪽으로 계산됩니다. 대입 연산자 및 null 병합 연산자를 제외하고, 모든 이진 연산자는 왼쪽 결합성이 있습니다. 예를 들어, `a + b - c`는 `(a + b) - c`로 계산됩니다.
- ‘오른쪽 결합성이 있는’ 연산자는 오른쪽에서 왼쪽으로 계산됩니다. 대입 연산자, null 병합 연산자 및 조건 연산자 `?:`는 오른쪽 결합성이 있습니다. 예를 들어, `x = y = z`는 `x = (y = z)`로 계산됩니다.

괄호를 사용하여 연산자 결합성에 따라 주어진 계산 순서를 변경합니다.

```
int a = 13 / 5 / 2;
int b = 13 / (5 / 2);
Console.WriteLine($"a = {a}, b = {b}"); // output: a = 1, b = 6
```

## 피연산자 계산

연산자 우선 순위 및 결합성과 관계없이 식의 피연산자는 왼쪽에서 오른쪽으로 계산됩니다. 다음 예제는 연산자와 피연산자가 계산되는 순서를 보여 줍니다.

식	계산 순서
<code>a + b</code>	a, b, +
<code>a + b * c</code>	a, b, c, *, +
<code>a / b + c * d</code>	a, b, /, c, d, *, +

식

계산 순서

a / (b + c) \* d

a, b, c, +, /, d, \*

일반적으로 모든 연산자 피연산자가 계산됩니다. 그러나 일부 연산자는 조건부로 피연산자를 계산합니다. 즉, 이와 같은 연산자의 첫 번째 피연산자 값이 다른 피연산자를 계산해야 할지 여부 또는 계산해야 할 다른 피연산자를 정의합니다. 이에 해당하는 연산자는 조건부 논리 **AND( && )** 및 **OR( || )** 연산자, **null 병합 연산자 ??** 및 **??=**, **null 조건부 연산자 ?. 및 ?[ ]**, 그리고 **조건 연산자 ?:**입니다. 자세한 내용은 각 연산자에 관한 설명을 참조하세요.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 다음 섹션을 참조하세요.

- [식](#)
- [연산자](#)

## 참조

- [C# 참조](#)
- [연산자 오버로드](#)
- [식 트리](#)

# 산술 연산자(C# 참조)

2020-11-02 • 23 minutes to read • [Edit Online](#)

다음 연산자는 숫자 형식 피연산자를 포함한 산술 작업을 수행합니다.

- 단항 `++`(증분), `--`(감소), `+(더하기)`, `-(빼기)` 연산자
- 이진 `*`(곱하기), `/`(나누기), `%`(나머지), `+(더하기)` 및 `-(빼기)` 연산자

해당 연산자는 모든 정수 및 부동 소수점 숫자 형식을 지원합니다.

정수 형식의 경우 이러한 연산자(`++` 및 `--` 연산자 제외)는 `int`, `uint`, `long` 및 `ulong` 형식에 대해 정의됩니다. 피연산자가 다른 정수 형식(`sbyte`, `byte`, `short`, `ushort` 또는 `char`)인 경우, 해당 값은 연산의 결과 형식이기도 한 `int` 형식으로 변환됩니다. 피연산자가 정수 형식 또는 부동 소수점 형식인 경우 해당 형식이 있으면 값은 가장 근사한 포함하는 형식으로 변환됩니다. 자세한 내용은 [C# 언어 사양의 숫자 승격](#) 섹션을 참조하세요.  
`++` 및 `--` 연산자는 모든 정수 형식 및 부동 소수점 숫자 형식과 `char` 형식에 대해 정의됩니다.

## 증가 연산자 `++`

단항 증가 연산자(`++`)는 피연산자를 1씩 증가합니다. 피연산자는 변수, 속성 액세스 또는 인덱서 액세스여야 합니다.

증가 연산자는 후위 증가 연산자 `x++` 및 전위 증가 연산자 `++x`라는 두 가지 양식으로 지원됩니다.

후위 증가 연산자

`x++`의 결과는 다음 예제와 같이 연산 전 `x`의 값입니다.

```
int i = 3;
Console.WriteLine(i);    // output: 3
Console.WriteLine(i++); // output: 3
Console.WriteLine(i);    // output: 4
```

후위 증가 연산자

`++x`의 결과는 다음 예제와 같이 연산 후 `x`의 값입니다.

```
double a = 1.5;
Console.WriteLine(a);    // output: 1.5
Console.WriteLine(++a); // output: 2.5
Console.WriteLine(a);    // output: 2.5
```

## 감소 연산자 `--`

단항 감소 연산자(`--`)는 피연산자를 1씩 감소합니다. 피연산자는 변수, 속성 액세스 또는 인덱서 액세스여야 합니다.

감소 연산자는 후위 감소 연산자 `x--` 및 전위 감소 연산자 `--x`라는 두 가지 양식으로 지원됩니다.

후위 감소 연산자

`x--`의 결과는 다음 예제와 같이 연산 전 `x`의 값입니다.

```
int i = 3;
Console.WriteLine(i);    // output: 3
Console.WriteLine(i--); // output: 3
Console.WriteLine(i);    // output: 2
```

### 후위 감소 연산자

--**x**의 결과는 다음 예제와 같이 연산 후 **x**의 값입니다.

```
double a = 1.5;
Console.WriteLine(a);    // output: 1.5
Console.WriteLine(--a); // output: 0.5
Console.WriteLine(a);    // output: 0.5
```

## 단항 더하기 및 빼기 연산자

단항 **+** 연산자는 피연산자의 값을 반환합니다. 단항 **-** 연산자는 피연산자의 숫자 부정을 계산합니다.

```
Console.WriteLine(+4);      // output: 4

Console.WriteLine(-4);      // output: -4
Console.WriteLine(-(-4));  // output: 4

uint a = 5;
var b = -a;
Console.WriteLine(b);        // output: -5
Console.WriteLine(b.GetType()); // output: System.Int64

Console.WriteLine(-double.NaN); // output: NaN
```

단항 **-** 연산자는 **ulong** 형식을 지원하지 않습니다.

## 곱하기 연산자 \*

곱하기 연산자 **\***는 피연산자의 곱을 계산합니다.

```
Console.WriteLine(5 * 2);      // output: 10
Console.WriteLine(0.5 * 2.5);  // output: 1.25
Console.WriteLine(0.1m * 23.4m); // output: 2.34
```

단항 **\*** 연산자는 [포인터 간접 참조 연산자](#)입니다.

## 나누기 연산자 /

나누기 연산자 **/**는 오른쪽 피연산자로 왼쪽 피연산자를 나눕니다.

정수 나누기

정수 형식의 피연산자의 경우 **/** 연산자의 결과는 정수 형식이고 두 피연산자의 둘과 동일한 값은 0으로 반올림됩니다.

```
Console.WriteLine(13 / 5);    // output: 2
Console.WriteLine(-13 / 5);   // output: -2
Console.WriteLine(13 / -5);   // output: -2
Console.WriteLine(-13 / -5); // output: 2
```

두 피연산자의 뒷을 부동 소수점 숫자로 가져오려면 `float`, `double` 또는 `decimal` 형식을 사용하세요.

```
Console.WriteLine(13 / 5.0);      // output: 2.6  
  
int a = 13;  
int b = 5;  
Console.WriteLine((double)a / b); // output: 2.6
```

## 부동 소수점 나누기

`float`, `double` 및 `decimal` 형식의 경우 `/` 연산자의 결과는 두 피연산자의 뒷입니다.

```
Console.WriteLine(16.8f / 4.1f); // output: 4.097561  
Console.WriteLine(16.8d / 4.1d); // output: 4.09756097560976  
Console.WriteLine(16.8m / 4.1m); // output: 4.09756097560975609756098
```

피연산자 중 하나가 `decimal` 이면 `float` 또는 `double` 모두 `decimal`로 암시적으로 변환할 수 없기 때문에 나머지 피연산자는 `float` 또는 `double` 일 수 없습니다. `float` 또는 `double` 피연산자를 `decimal` 형식으로 암시적으로 변환해야 합니다. 숫자 형식 간의 변환에 대한 자세한 내용은 [기본 제공 숫자 변환](#)을 참조하세요.

## 나머지 연산자 %

나머지 연산자 `%` 는 왼쪽 피연산자를 오른쪽 피연산자로 나눈 후 나머지를 계산합니다.

### 정수 나머지

정수 형식의 피연산자의 경우 `a % b`의 결과가 `a - (a / b) * b`에서 생성된 값입니다. 다음 예와 같이 0이 아닌 나머지의 부호는 왼쪽 피연산자와 동일합니다.

```
Console.WriteLine(5 % 4); // output: 1  
Console.WriteLine(5 % -4); // output: 1  
Console.WriteLine(-5 % 4); // output: -1  
Console.WriteLine(-5 % -4); // output: -1
```

[Math.DivRem](#) 메서드를 사용하여 정수 나누기 및 나머지 결과를 모두 컴퓨팅합니다.

### 부동 소수점 나머지

`float` 및 `double` 피연산자의 경우 유한 `x` 및 `y`에 대한 `x % y`의 결과는 다음과 같은 `z` 값입니다.

- 0이 아닌 경우 `z`의 부호는 `x`의 부호와 동일합니다.
- `z`의 절대 값은  $|x| - n * |y|$ 에서 생성된 값입니다. 여기서 `n`은  $|x| / |y|$ 보다 작거나 같은 가능한 최대 정수이고 `|x|` 및 `|y|`는 각각 `x` 및 `y`의 절대 값입니다.

### NOTE

나머지 계산 방법은 정수 피연산자에 사용되는 것과 유사하지만 IEEE 754 사양과는 다릅니다. IEEE 754 사양을 준수하는 나머지 작업이 필요한 경우 [Math.IEEEremainder](#) 메서드를 사용합니다.

무한 피연산자가 있는 `%` 연산자의 동작에 대한 자세한 내용은 C# 언어 사양의 [나머지 연산자](#) 섹션을 참조하세요.

`decimal` 피연산자의 경우 나머지 연산자 `%`는 [System.Decimal](#) 형식의 [나머지 연산자](#)와 동일합니다.

다음 예제에서는 부동 소수점 피연산자를 포함한 나머지 연산자의 동작을 보여줍니다.

```
Console.WriteLine(-5.2f % 2.0f); // output: -1.2
Console.WriteLine(5.9 % 3.1);    // output: 2.8
Console.WriteLine(5.9m % 3.1m); // output: 2.8
```

## 더하기 연산자 +

더하기 연산자 `[+]`는 피연산자의 합계를 계산합니다.

```
Console.WriteLine(5 + 4);      // output: 9
Console.WriteLine(5 + 4.3);    // output: 9.3
Console.WriteLine(5.1m + 4.2m); // output: 9.3
```

문자열 연결 및 대리자 조합의 경우 `[+]` 연산자를 사용할 수도 있습니다. 자세한 내용은 참조는 `[+]` 및 `[+=]` 연산자 문서를 참조하세요.

## 빼기 연산자 -

빼기 연산자 `[-]`는 왼쪽 피연산자에서 오른쪽 피연산자를 뺍니다.

```
Console.WriteLine(47 - 3);      // output: 44
Console.WriteLine(5 - 4.3);    // output: 0.7
Console.WriteLine(7.5m - 2.3m); // output: 5.2
```

대리자 제거를 위해 `[-]` 연산자를 사용할 수도 있습니다. 자세한 내용은 참조는 `[-]` 및 `[-=]` 연산자 문서를 참조하세요.

## 복합 할당

이진 연산자(`op`)의 경우 양식의 복합 할당식

```
x op= y
```

위의 식은 아래의 식과 동일합니다.

```
x = x op y
```

단, `x`가 한 번만 계산됩니다.

다음 예제에서는 산술 연산자를 사용하는 복합 할당의 사용법을 보여줍니다.

```

int a = 5;
a += 9;
Console.WriteLine(a); // output: 14

a -= 4;
Console.WriteLine(a); // output: 10

a *= 2;
Console.WriteLine(a); // output: 20

a /= 4;
Console.WriteLine(a); // output: 5

a %= 3;
Console.WriteLine(a); // output: 2

```

숫자 승격으로 인해 `op` 연산의 결과가 암시적으로 `x`의 `T` 형식으로 변환되지 못할 수 있습니다. 이 경우 `op` 가 미리 정의된 연산자이고 연산의 결과가 명시적으로 `x`의 `T` 형식으로 변환 가능하다면 `x op= y` 양식의 복합 할당 식이 `x = (T)(x op y)`에 해당합니다. 단 `x`는 한 번만 평가됩니다. 다음 예제에서는 해당 동작을 보여줍니다.

```

byte a = 200;
byte b = 100;

var c = a + b;
Console.WriteLine(c.GetType()); // output: System.Int32
Console.WriteLine(c); // output: 300

a += b;
Console.WriteLine(a); // output: 44

```

각각 `+=` 및 `-=` 연산자를 사용하여 [이벤트](#)에서 구독하거나 구독을 취소할 수도 있습니다. 자세한 내용은 [이벤트를 구독 및 구독 취소하는 방법](#)을 참조하세요.

## 연산자 우선 순위 및 결합성

다음 목록에서는 산술 연산자를 가장 높은 우선 순위부터 가장 낮은 것으로 정렬합니다.

- 후위 증가 `x++` 및 감소 `x--` 연산자
- 전위 증가 `++x` 및 감소 `--x` 및 단항 `+` 및 `-` 연산자
- 곱하기 `*`, `/` 및 `%` 연산자
- 加減 `+` 및 `-` 연산자

이진 산술 연산자는 왼쪽 결합형입니다. 즉, 우선 순위 수준이 같은 연산자는 왼쪽에서 오른쪽으로 계산됩니다.

괄호(`()`)를 사용하여 연산자 우선 순위와 연결에 따라 주어진 계산 순서를 변경할 수 있습니다.

```

Console.WriteLine(2 + 2 * 2); // output: 6
Console.WriteLine((2 + 2) * 2); // output: 8

Console.WriteLine(9 / 5 / 2); // output: 0
Console.WriteLine(9 / (5 / 2)); // output: 4

```

우선 순위 수준에 따라 정렬된 전체 연산자 목록은 [C# 연산자](#) 문서의 [연산자 우선 순위](#) 섹션을 참조하세요.

## 산술 오버플로 및 0으로 나누기

산술 연산의 결과가 관련된 숫자 형식의 가능한 유한 값 범위를 벗어나는 경우 산술 연산자의 동작은 해당하는 피연산자의 형식에 따라 달라집니다.

### 정수 산술 오버플로

정수를 0으로 나누면 항상 [DivideByZeroException](#)이 throw됩니다.

정수 산술 오버플로의 경우 [checked](#) 또는 [unchecked](#)일 수 있는 오버플로 검사 컨텍스트는 결과 동작을 제어합니다.

- checked 컨텍스트인 경우 상수 식에서 오버플로가 일어나면 컴파일 시간 오류가 발생합니다. 그렇지 않으면, 런타임 시 작업을 수행하는 경우 [OverflowException](#)이 throw됩니다.
- unchecked 컨텍스트에서는 대상 형식에 맞지 않는 상위 비트를 버려서 해당 결과가 잘려집니다.

[checked](#) 및 [unchecked](#) 문과 함께 [checked](#) 및 [unchecked](#) 연산자를 사용하여 식을 계산하는 오버플로 검사 컨텍스트를 제어합니다.

```
int a = int.MaxValue;
int b = 3;

Console.WriteLine(unchecked(a + b)); // output: -2147483646
try
{
    int d = checked(a + b);
}
catch(OverflowException)
{
    Console.WriteLine($"Overflow occurred when adding {a} to {b}.");
}
```

기본적으로 산술 연산은 [unchecked](#) 컨텍스트에서 발생합니다.

### 부동 소수점 산술 오버플로

[float](#) 및 [double](#) 형식을 포함한 산술 연산은 예외를 throw하지 않습니다. 이러한 형식의 산술 연산 결과는 무한대를 나타내거나 숫자가 아닌 특수 값 중 하나일 수 있습니다.

```
double a = 1.0 / 0.0;
Console.WriteLine(a); // output: Infinity
Console.WriteLine(double.IsInfinity(a)); // output: True

Console.WriteLine(double.MaxValue + double.MaxValue); // output: Infinity

double b = 0.0 / 0.0;
Console.WriteLine(b); // output: NaN
Console.WriteLine(double.IsNaN(b)); // output: True
```

[decimal](#) 형식의 피연산자의 경우 산술 오버플로는 항상 [OverflowException](#)을 throw하고, 0으로 나누기는 항상 [DivideByZeroException](#)을 throw합니다.

## 반올림 오류

실수 및 부동 소수점 산술의 부동 소수점 표현을 일반적으로 제한함으로 인해 부동 소수점 형식을 사용하는 계산에서 반올림 오류가 발생할 수 있습니다. 즉, 생성된 식의 결과는 예상되는 수학적 결과와 다를 수 있습니다. 다음 예제에서는 이러한 몇몇 사례에 대해 설명합니다.

```
Console.WriteLine(.41f % .2f); // output: 0.00999999  
  
double a = 0.1;  
double b = 3 * a;  
Console.WriteLine(b == 0.3); // output: False  
Console.WriteLine(b - 0.3); // output: 5.55111512312578E-17  
  
decimal c = 1 / 3.0m;  
decimal d = 3 * c;  
Console.WriteLine(d == 1.0m); // output: False  
Console.WriteLine(d); // output: 0.999999999999999999999999999999999
```

자세한 내용은 [System.Double](#), [System.Single](#) 또는 [System.Decimal](#) 참조 페이지에서 참조하세요.

## 연산자 오버로드 가능성

사용자 정의 형식은 단항( `++`, `--`, `+` 및 `-` ) 및 이진( `*`, `/`, `%`, `+` 및 `-` ) 산술 연산자를 [오버로드](#)할 수 있습니다. 이항 연산자가 오버로드되면 해당하는 복합 대입 연산자도 암시적으로 오버로드됩니다. 사용자 정의 형식은 복합 대입 연산자를 명시적으로 오버로드할 수 없습니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 다음 섹션을 참조하세요.

- [후위 증가 및 감소 연산자](#)
- [전위 증가 및 감소 연산자](#)
- [단항 더하기 연산자](#)
- [단항 빼기 연산자](#)
- [곱하기 연산자](#)
- [나누기 연산자](#)
- [나머지 연산자](#)
- [더하기 연산자](#)
- [빼기 연산자](#)
- [복합 할당](#)
- [Checked 및 Unchecked 연산자](#)
- [숫자 승격](#)

## 참조

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [System.Math](#)
- [System.MathF](#)
- [.NET의 숫자](#)

# 부울 논리 연산자(C# 참조)

2020-11-02 • 20 minutes to read • [Edit Online](#)

다음 연산자는 **부울** 피연산자를 사용하여 논리 작업을 수행합니다.

- 단항 **!** (논리 부정) 연산자.
- 이진 **&** (논리 AND), **|** (논리 OR) 및 **^** (논리 배타적 OR) 연산자. 해당 연산자는 항상 두 피연산자를 모두 평가합니다.
- 이진 **&&** (조건부 논리 AND) 및 **||** (조건부 논리 OR) 연산자. 해당 연산자는 필요한 경우에만 오른쪽 피연산자를 평가합니다.

정수 형식 피연산자의 경우 **&**, **|** 및 **^** 연산자는 비트 논리 작업을 수행합니다. 자세한 내용은 [비트 및 시프트 연산자](#)를 참조하세요.

## 논리 부정 연산자 !

단항 접두사 **!** 연산자는 해당 피연산자의 논리 부정을 계산합니다. 즉, 피연산자가 **false**로 평가되는 경우 **true**를 생성하고, 피연산자가 **true**로 평가되는 경우 **false**를 생성합니다.

```
bool passed = false;
Console.WriteLine(!passed); // output: True
Console.WriteLine(!true); // output: False
```

C# 8.0부터 단항 후위 **!** 연산자는 [null 허용 연산자](#)입니다.

## 논리 AND 연산자 &

**&** 연산자는 해당 피연산자의 논리 AND를 컴퓨팅합니다. **x** 및 **y** 가 모두 **true**로 평가되면 **x & y**의 결과는 **true**입니다. 그렇지 않으면 결과는 **false**입니다.

왼쪽 피연산자가 **false**로 평가되더라도 **&** 연산자는 두 피연산자를 평가하여 오른쪽 피연산자의 값에 관계없이 **false**이어야 합니다.

다음 예제에서 **&** 연산자의 오른쪽 피연산자는 왼쪽 피연산자의 값과 관계없이 수행되는 메서드 호출입니다.

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false & SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// False

bool b = true & SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

조건부 논리 AND 연산자 `&&`도 해당 피연산자의 논리 AND를 계산하지만, 왼쪽 피연산자가 `false`로 평가되는 경우에는 오른쪽 피연산자를 평가하지 않습니다.

정수 형식 피연산자의 경우 `&` 연산자는 해당 피연산자의 **비트 논리 AND**를 컴퓨팅합니다. 단항 `&` 연산자는 **address-of 연산자**입니다.

## 논리 배타적 OR 연산자 ^

`^` 연산자는 해당 피연산자의 논리 XOR이라고도 하는 논리 배타적 OR을 컴퓨팅합니다. `x` 가 `true`로 평가되고 `y` 가 `false`로 평가되거나, `x` 가 `false`로 평가되고 `y` 가 `true`로 평가되는 경우 `x ^ y` 의 결과는 `true`입니다. 그렇지 않으면 결과는 `false`입니다. 즉, `bool` 피연산자의 경우 `^` 연산자는 같지 않은 연산자 `!=`과 같은 결과를 컴퓨팅합니다.

```
Console.WriteLine(true ^ true);    // output: False
Console.WriteLine(true ^ false);   // output: True
Console.WriteLine(false ^ true);   // output: True
Console.WriteLine(false ^ false);  // output: False
```

정수 숫자 형식 피연산자의 경우 `^` 연산자는 해당 피연산자의 **비트 논리 배타적 OR**를 컴퓨팅합니다.

## 논리 OR 연산자 |

`|` 연산자는 해당 피연산자의 논리 OR을 컴퓨팅합니다. `x` 또는 `y` 가 `true`로 평가되면 `x | y`의 결과는 `true`입니다. 그렇지 않으면 결과는 `false`입니다.

왼쪽 피연산자가 `true`로 평가되더라도 `|` 연산자는 두 피연산자를 평가하여 오른쪽 피연산자의 값에 관계없이 `true`이어야 합니다.

다음 예제에서 `|` 연산자의 오른쪽 피연산자는 왼쪽 피연산자의 값과 관계없이 수행되는 메서드 호출입니다.

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true | SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// True

bool b = false | SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

조건부 논리 OR 연산자 `||`도 해당 피연산자의 논리 OR을 계산하지만, 왼쪽 피연산자가 `true`로 평가되는 경우에는 오른쪽 피연산자를 평가하지 않습니다.

정수 숫자 형식 피연산자의 경우 `|` 연산자는 해당 피연산자의 **비트 논리 OR**를 컴퓨팅합니다.

## 조건부 논리 AND 연산자 &&

"단락(short-circuiting)" 논리 AND 연산자로도 알려진 조건부 논리 AND 연산자 `&&`는 해당 피연산자의 논리 AND를 컴퓨팅합니다. `x` 및 `y` 가 모두 `true`로 평가되면 `x && y`의 결과는 `true`입니다. 그렇지 않으면 결과

는 `false`입니다. `x`가 `false`이면 `y`는 계산되지 않습니다.

다음 예제에서 `&&` 연산자의 오른쪽 피연산자는 왼쪽 피연산자가 `false`로 평가되는 경우 수행되지 않는 메서드 호출입니다.

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false && SecondOperand();
Console.WriteLine(a);
// Output:
// False

bool b = true && SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

논리 AND 연산자 `&`도 해당 피연산자의 논리 AND를 컴퓨팅하지만 항상 두 피연산자를 모두 평가합니다.

## 조건부 논리 OR 연산자 `||`

“단락(short-circuiting)” 논리 OR 연산자로도 알려진 조건부 논리 OR 연산자 `||`은 해당 피연산자의 논리 OR을 컴퓨팅합니다. `x` 또는 `y` 가 `true`로 평가되면 `x || y`의 결과는 `true`입니다. 그렇지 않으면 결과는 `false`입니다. `x` 가 `true`이면 `y`는 계산되지 않습니다.

다음 예제에서 `||` 연산자의 오른쪽 피연산자는 왼쪽 피연산자가 `true`로 평가되는 경우 수행되지 않는 메서드 호출입니다.

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true || SecondOperand();
Console.WriteLine(a);
// Output:
// True

bool b = false || SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

논리 OR 연산자 `||`도 해당 피연산자의 논리 OR을 컴퓨팅하지만 항상 두 피연산자를 모두 평가합니다.

## Nullable 부울 논리 연산자

`bool?` 피연산자의 경우 `&`(논리적 AND) 및 `||`(논리적 OR) 연산자는 다음과 같이 값이 세 개인 논리를 지원합니다.

- `&` 연산자는 두 피연산자가 모두 `true`로 평가되는 경우에만 `true`를 생성합니다. `x` 또는 `y`가 `false`로 평가되는 경우 다른 피연산자가 `null`로 평가되더라도 `x & y`는 `false`를 생성합니다. 그러지 않으면 `x & y`의 결과는 `null`입니다.

- $|$  연산자는 두 피연산자가 모두 `false`로 평가되는 경우에만 `false`를 생성합니다. `x` 또는 `y`가 `true`로 평가되는 경우 다른 피연산자가 `null`로 평가되더라도 `x | y`는 `true`를 생성합니다. 그러지 않으면 `x | y`의 결과는 `null`입니다.

다음 표는 이 의미 체계를 보여 줍니다.

<code>x</code>	<code>y</code>	<code>x &amp; y</code>	<code>x   y</code>
<code>true</code>	<code>true</code>	<code>true</code>	<code>true</code>
<code>true</code>	<code>false</code>	<code>false</code>	<code>true</code>
<code>true</code>	<code>null</code>	<code>null</code>	<code>true</code>
<code>false</code>	<code>true</code>	<code>false</code>	<code>true</code>
<code>false</code>	<code>false</code>	<code>false</code>	<code>false</code>
<code>false</code>	<code>null</code>	<code>false</code>	<code>null</code>
<code>null</code>	<code>true</code>	<code>null</code>	<code>true</code>
<code>null</code>	<code>false</code>	<code>false</code>	<code>null</code>
<code>null</code>	<code>null</code>	<code>null</code>	<code>null</code>

해당 연산자의 동작은 `Nullable` 값 형식을 사용하는 일반 연산자 동작과 다릅니다. 일반적으로 값 형식의 피연산자에 대해 정의된 연산자도 해당 `Nullable` 값 형식의 피연산자와 함께 사용할 수 있습니다. 피연산자가 `null`로 평가되는 경우 해당 연산자는 `null`을 생성합니다. 그러나 피연산자 중 하나가 `null`로 평가되는 경우에도 `&` 및  $|$  연산자는 `null`이 아닌 값을 생성합니다. `Nullable` 값 형식을 사용한 연산자 동작에 대한 자세한 내용은 [Nullable 값 형식](#) 문서의 [리프트된 연산자](#) 섹션을 참조하세요.

다음 예제와 같이 `!` 및 `^` 연산자를 `bool?` 피연산자와 함께 사용할 수도 있습니다.

```
bool? test = null;
Display(!test);           // output: null
Display(test ^ false);    // output: null
Display(test ^ null);     // output: null
Display(true ^ null);     // output: null

void Display(bool? b) => Console.WriteLine(b is null ? "null" : b.Value.ToString());
```

조건부 논리 연산자 `&&` 및 `||`은 `bool?` 피연산자를 지원하지 않습니다.

## 복합 할당

이진 연산자(`op`)의 경우 양식의 복합 할당식

```
x op= y
```

위의 식은 아래의 식과 동일합니다.

```
x = x op y
```

단, `x` 가 한 번만 계산됩니다.

`&`, `|` 및 `^` 연산자는 다음 예제와 같이 복합 할당을 지원합니다.

```
bool test = true;
test &= false;
Console.WriteLine(test); // output: False

test |= true;
Console.WriteLine(test); // output: True

test ^= false;
Console.WriteLine(test); // output: True
```

#### NOTE

조건부 논리 연산자 `&&` 및 `||`는 복합 할당을 지원하지 않습니다.

## 연산자 우선 순위

다음 목록에서는 논리 연산자를 가장 높은 우선 순위부터 가장 낮은 것으로 정렬합니다.

- 논리 부정 연산자 `!`
- 논리 AND 연산자 `&`
- 논리 배타적 OR 연산자 `^`
- 논리 OR 연산자 `|`
- 조건부 논리 AND 연산자 `&&`
- 조건부 논리 OR 연산자 `||`

괄호(`()`)를 사용하여 연산자 우선 순위에 따라 주어진 평가 순서를 변경합니다.

```
Console.WriteLine(true | true & false); // output: True
Console.WriteLine((true | true) & false); // output: False

bool Operand(string name, bool value)
{
    Console.WriteLine($"Operand {name} is evaluated.");
    return value;
}

var byDefaultPrecedence = Operand("A", true) || Operand("B", true) && Operand("C", false);
Console.WriteLine(byDefaultPrecedence);
// Output:
// Operand A is evaluated.
// True

var changedOrder = (Operand("A", true) || Operand("B", true)) && Operand("C", false);
Console.WriteLine(changedOrder);
// Output:
// Operand A is evaluated.
// Operand C is evaluated.
// False
```

우선 순위 수준에 따라 정렬된 전체 연산자 목록은 [C# 연산자](#) 문서의 [연산자 우선 순위](#) 섹션을 참조하세요.

## 연산자 오버로드 가능성

사용자 정의 형식은 `!`, `&`, `|` 및 `^` 연산자를 [오버로드](#)할 수 있습니다. 이항 연산자가 오버로드되면 해당하는 복합 대입 연산자도 암시적으로 오버로드됩니다. 사용자 정의 형식은 복합 대입 연산자를 명시적으로 오버로드 할 수 없습니다.

사용자 정의 형식은 조건부 논리 연산자 `&&` 및 `||`을 오버로드할 수 없습니다. 그러나 사용자 정의 형식이 [true](#) 및 [false](#) [연산자](#)와 `&` 또는 `|` 연산자를 특정 방식으로 오버로드하는 경우 `&&` 또는 `||` 작업은 각각 해당 형식의 피연산자에 대해 평가될 수 있습니다. 자세한 내용은 [C# 언어 사양](#)의 [사용자 정의 조건부 논리 연산자](#) 섹션을 참조하세요.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 다음 섹션을 참조하세요.

- [논리 부정 연산자](#)
- [논리 연산자](#)
- [조건부 논리 연산자](#)
- [복합 할당](#)

## 참조

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [비트 및 시프트 연산자](#)

# 비트 및 시프트 연산자(C# 참조)

2020-11-02 • 19 minutes to read • [Edit Online](#)

다음 연산자는 정수 형식 또는 `char` 형식의 피연산자를 사용하여 비트 논리 또는 시프트 연산을 수행합니다.

- 단항 `~(비트 보수)` 연산자
- 이진 `<<(왼쪽 시프트)` 및 `>>(오른쪽 시프트)` 시프트 연산자
- 이진 `&(논리 AND)`, `|(논리 OR)` 및 `^(논리 배타적 OR)` 연산자

이러한 연산자는 `int`, `uint`, `long` 및 `ulong` 형식에 대해 정의되어 있습니다. 두 피연산자가 모두 다른 정수 형식(`sbyte`, `byte`, `short`, `ushort` 또는 `char`)인 경우, 해당 값은 작업의 결과 형식이기도 한 `int` 유형으로 변환됩니다. 피연산자가 다른 정수 형식인 경우 해당 값은 가장 가까운 정수 형식으로 변환됩니다. 자세한 내용은 [C# 언어 사양](#)의 [숫자 승격](#) 섹션을 참조하세요.

`bool` 유형의 피연산자에 대한 `&`, `|` 및 `^` 연산자도 정의되어 있습니다. 자세한 내용은 [부울 논리 연산자](#)를 참조하세요.

비트 및 시프트 작업으로 인해 오버플로가 발생하지 않고 `Checked` 및 `Unchecked` 컨텍스트에서 동일한 결과가 생성되지 않습니다.

## 비트 보수 연산자 ~

`~` 연산자는 각 비트를 반대로 하여 해당 피연산자의 비트 보수를 생성합니다.

```
uint a = 0b_0000_1111_0000_1111_0000_1111_0000_1100;
uint b = ~a;
Console.WriteLine(Convert.ToString(b, toBase: 2));
// Output:
// 11110000111100001111000011110011
```

`~` 기호를 사용하여 종료자를 선언할 수도 있습니다. 자세한 내용은 [종료자](#)를 참조하세요.

## 왼쪽 시프트 연산자 <<

`<<` 연산자는 왼쪽 피연산자를 [오른쪽 피연산자로 정의된 비트 수](#)만큼 왼쪽으로 이동합니다.

왼쪽 시프트 연산은 결과 형식의 범위를 벗어나는 상위 비트를 삭제하고 다음 예제와 같이 빈 하위 비트 위치를 0으로 설정합니다.

```
uint x = 0b_1100_1001_0000_0000_0000_0001_0001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2)}");

uint y = x << 4;
Console.WriteLine($"After: {Convert.ToString(y, toBase: 2)}");
// Output:
// Before: 1100100100000000000000000010001
// After: 100100000000000000000000100010000
```

시프트 연산자는 `int`, `uint`, `long` 및 `ulong` 유형에 대해서만 정의되므로 작업 결과에는 항상 32비트 이상이 포함됩니다. 왼쪽 피연산자가 다른 정수 형식(`sbyte`, `byte`, `short`, `ushort` 또는 `char`)인 경우, 다음 예제와 같이 해당 값이 `int` 유형으로 변환됩니다.

```
byte a = 0b_1111_0001;

var b = a << 8;
Console.WriteLine(b.GetType());
Console.WriteLine($"Shifted byte: {Convert.ToString(b, toBase: 2)}");
// Output:
// System.Int32
// Shifted byte: 1111000100000000
```

<< 연산자의 오른쪽 피연산자가 시프트 수를 정의하는 방법에 대한 자세한 내용은 [시프트 연산자의 시프트 수 섹션](#)을 참조하세요.

## 오른쪽 시프트 연산자 >>

>> 연산자는 왼쪽 피연산자를 [오른쪽 피연산자로 정의된 비트 수](#)만큼 오른쪽으로 이동합니다.

오른쪽 시프트 연산은 다음 예제와 같이 하위 비트를 삭제합니다.

```
uint x = 0b_1001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2), 4}");

uint y = x >> 2;
Console.WriteLine($"After: {Convert.ToString(y, toBase: 2), 4}");
// Output:
// Before: 1001
// After:    10
```

빈 상위 공백 비트 위치는 다음과 같이 왼쪽 피연산자 형식에 따라 설정됩니다.

- 왼쪽 피연산자가 `int` 또는 `long` 형식인 경우 오른쪽 시프트 연산자는 [산술 시프트](#)를 수행합니다. 왼쪽 피연산자의 최상위 비트(부호 비트) 값이 빈 상위 비트 위치로 전파됩니다. 즉, 빈 상위 비트 위치는 왼쪽 피연산자가 음수가 아닌 경우 0으로 설정되고, 음수인 경우 1로 설정됩니다.

```
int a = int.MinValue;
Console.WriteLine($"Before: {Convert.ToString(a, toBase: 2)}");

int b = a >> 3;
Console.WriteLine($"After: {Convert.ToString(b, toBase: 2)}");
// Output:
// Before: 10000000000000000000000000000000
// After:   11110000000000000000000000000000
```

- 왼쪽 피연산자가 `uint` 또는 `ulong` 형식이면 오른쪽 시프트 피연산자는 [논리적 시프트](#)를 수행합니다. 빈 상위 비트 위치가 항상 0으로 설정됩니다.

```
uint c = 0b_1000_0000_0000_0000_0000_0000_0000;
Console.WriteLine($"Before: {Convert.ToString(c, toBase: 2), 32}");

uint d = c >> 3;
Console.WriteLine($"After: {Convert.ToString(d, toBase: 2), 32}");
// Output:
// Before: 10000000000000000000000000000000
// After:    10000000000000000000000000000000
```

>> 연산자의 오른쪽 피연산자가 시프트 수를 정의하는 방법에 대한 자세한 내용은 [시프트 연산자의 시프트 수 섹션](#)을 참조하세요.

## 논리 AND 연산자 &

& 연산자는 해당 정수 피연산자의 비트 논리 AND를 컴퓨팅합니다.

```
uint a = 0b_1111_1000;
uint b = 0b_1001_1101;
uint c = a & b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10011000
```

bool 피연산자의 경우 & 연산자는 피연산자의 논리 AND를 컴퓨팅합니다. 단항 & 연산자는 address-of 연산자입니다.

## 논리 배타적 OR 연산자 ^

^ 연산자는 해당 정수 피연산자의 비트 논리 XOR이라고도 하는 비트 논리 배타적 OR을 컴퓨팅합니다.

```
uint a = 0b_1111_1000;
uint b = 0b_0001_1100;
uint c = a ^ b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 11100100
```

bool 피연산자의 경우 ^ 연산자는 피연산자의 논리 배타적 OR을 컴퓨팅합니다.

## 논리 OR 연산자 |

| 연산자는 해당 정수 피연산자의 비트 논리 OR을 컴퓨팅합니다.

```
uint a = 0b_1010_0000;
uint b = 0b_1001_0001;
uint c = a | b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10110001
```

bool 피연산자의 경우 | 연산자는 피연산자의 논리 OR을 컴퓨팅합니다.

## 복합 할당

이진 연산자( op )의 경우 양식의 복합 할당 식

```
x op= y
```

위의 식은 아래의 식과 동일합니다.

```
x = x op y
```

단, x 가 한 번만 계산됩니다.

다음 예제에서는 비트 및 시프트 연산자를 사용하는 복합 할당의 사용법을 보여줍니다.

```

uint a = 0b_1111_1000;
a &= 0b_1001_1101;
Display(a); // output: 10011000

a |= 0b_0011_0001;
Display(a); // output: 10111001

a ^= 0b_1000_0000;
Display(a); // output: 111001

a <<= 2;
Display(a); // output: 11100100

a >>= 4;
Display(a); // output: 1110

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase: 2), 8}");

```

**숫자 승격**으로 인해 `op` 연산의 결과가 암시적으로 `x`의 `T` 형식으로 변환되지 못할 수 있습니다. 이 경우 `op` 가 미리 정의된 연산자이고 연산의 결과가 명시적으로 `x`의 `T` 형식으로 변환 가능하다면 `x op= y` 양식의 복합 할당 식이 `x = (T)(x op y)`에 해당합니다. 단 `x`는 한 번만 평가됩니다. 다음 예제에서는 해당 동작을 보여줍니다.

```

byte x = 0b_1111_0001;

int b = x << 8;
Console.WriteLine($"{Convert.ToString(b, toBase: 2)}"); // output: 1111000100000000

x <<= 8;
Console.WriteLine(x); // output: 0

```

## 연산자 우선 순위

다음 목록에서는 비트 및 시프트 연산자를 가장 높은 우선순위부터 가장 낮은 것으로 정렬합니다.

- 비트 보수 연산자 `~`
- 시프트 연산자 `<<` 및 `>>`
- 논리 AND 연산자 `&`
- 논리 배타적 OR 연산자 `^`
- 논리 OR 연산자 `|`

괄호(`()`)를 사용하여 연산자 우선 순위에 따라 주어진 평가 순서를 변경합니다.

```

uint a = 0b_1101;
uint b = 0b_1001;
uint c = 0b_1010;

uint d1 = a | b & c;
Display(d1); // output: 1101

uint d2 = (a | b) & c;
Display(d2); // output: 1000

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase: 2), 4}");

```

우선 순위 수준에 따라 정렬된 전체 연산자 목록은 [C# 연산자](#) 문서의 [연산자 우선 순위](#) 섹션을 참조하세요.

## 시프트 연산자의 시프트 수

시트프 연산자 `<<` 및 `>>`의 경우 오른쪽 피연산자의 형식은 `int` 이거나 `int`로의 미리 정의된 암시적 숫자 변환이 있는 형식이어야 합니다.

`x << count` 및 `x >> count` 식의 경우 실제 시프트 수는 다음과 같이 `x` 형식에 따라 달라집니다.

- `x`의 형식이 `int` 또는 `uint`이면 시프트 수는 오른쪽 피연산자의 하위 5비트로 정의됩니다. 즉, 시프트 수는 `count & 0x1F` (또는 `count & 0b_1_1111`)에서 계산됩니다.
- `x`의 형식이 `long` 또는 `ulong`이면 시프트 수는 오른쪽 피연산자의 하위 6비트로 정의됩니다. 즉, 시프트 수는 `count & 0x3F` (또는 `count & 0b_11_1111`)에서 계산됩니다.

다음 예제에서는 해당 동작을 보여줍니다.

```
int count1 = 0b_0000_0001;
int count2 = 0b_1110_0001;

int a = 0b_0001;
Console.WriteLine($"{a} << {count1} is {a << count1}; {a} << {count2} is {a << count2}");
// Output:
// 1 << 1 is 2; 1 << 225 is 2

int b = 0b_0100;
Console.WriteLine($"{b} >> {count1} is {b >> count1}; {b} >> {count2} is {b >> count2}");
// Output:
// 4 >> 1 is 2; 4 >> 225 is 2
```

### NOTE

앞의 예제에서 볼 수 있듯이 오른쪽 피연산자의 값이 왼쪽 피연산자의 비트 수보다 크면 시프트 연산의 결과가 0이 아닐 수 있습니다.

## 열거형 논리 연산자

`~, &, |` 및 `^` 연산자도 열거형 형식에 대해 지원됩니다. 동일한 열거형 형식의 피연산자인 경우, 기본 정수 형식의 해당 값에 대해 논리 연산을 수행됩니다. 예를 들어 기본 형식이 `U`인 열거형 형식 `T`의 `x` 및 `y`에 대해 `x & y` 식은 `(T)((U)x & (U)y)` 식과 동일한 결과를 생성합니다.

일반적으로 `Flags` 특성으로 정의된 열거형 형식을 가진 비트 논리 연산자를 사용합니다. 자세한 내용은 [열거형 형식 문서의 비트 플래그로서 열거형 형식](#)을 참조하세요.

## 연산자 오버로드 가능성

사용자 정의 형식은 `~, <<, >>, &, |` 및 `^` 연산자를 [오버로드](#)할 수 있습니다. 이항 연산자가 오버로드되면 해당하는 복합 대입 연산자도 암시적으로 오버로드됩니다. 사용자 정의 형식은 복합 대입 연산자를 명시적으로 오버로드할 수 없습니다.

사용자 정의 형식 `T`가 `<<` 또는 `>>` 연산자를 오버로드하는 경우 첫 번째 피연산자의 형식은 `T`여야 하고, 두 번째 피연산자의 형식은 `int`여야 합니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 다음 섹션을 참조하세요.

- [비트 보수 연산자](#)

- 시프트 연산자
- 논리 연산자
- 복합 할당
- 숫자 승격

## 참고 항목

- C# 참조
- C# 연산자 및 식
- 부울 논리 연산자

# 같음 연산자(C# 참조)

2021-02-18 • 10 minutes to read • [Edit Online](#)

`==` (같음) 및 `!=` (같지 않음) 연산자는 피연산자가 같은지 여부를 확인합니다.

## 같음 연산자 `==`

같음 연산자 `==`는 피연산자가 같으면 `true`를 반환하고, 그렇지 않으면 `false`를 반환합니다.

값 형식 같음

기본 제공 값 형식의 피연산자는 해당 값이 같은 경우 동일합니다.

```
int a = 1 + 2 + 3;
int b = 6;
Console.WriteLine(a == b); // output: True

char c1 = 'a';
char c2 = 'A';
Console.WriteLine(c1 == c2); // output: False
Console.WriteLine(c1 == char.ToLower(c2)); // output: True
```

### NOTE

`==`, `<`, `>`, `<=` 및 `>=` 연산자의 경우 피연산자 중 하나가 숫자(`Double.NaN` 또는 `Single.NaN`)가 아니면 연산의 결과는 `false`입니다. 즉, `NaN` 값이 `NaN`을 포함한 다른 `double` (또는 `float`) 값보다 크거나, 작거나, 같지 않습니다. 자세한 내용과 예제는 `Double.NaN` 또는 `Single.NaN` 참조 문서를 참조하세요.

기본 정수 형식의 해당 값이 같은 경우 동일한 열거형 형식의 피연산자가 동일합니다.

사용자 정의 구조체 형식은 기본적으로 `==` 연산자를 지원하지 않습니다. `==` 연산자를 지원하려면 사용자 정의 구조체가 해당 연산자를 [오버로드](#)해야 합니다.

C# 7.3부터는 `==` 및 `!=` 연산자가 C# 템플에서 지원됩니다. 자세한 내용은 [템플 형식](#) 문서의 [템플 같은 선택](#)을 참조하세요.

참조 형식 같음

기본적으로 두 개의 레코드가 아닌 참조 형식 피연산자는 동일한 개체를 참조하는 경우 같습니다.

```

public class ReferenceTypesEquality
{
    public class MyClass
    {
        private int id;

        public MyClass(int id) => this.id = id;
    }

    public static void Main()
    {
        var a = new MyClass(1);
        var b = new MyClass(1);
        var c = a;
        Console.WriteLine(a == b); // output: False
        Console.WriteLine(a == c); // output: True
    }
}

```

이 예제에서 표시한 대로 사용자 지정 참조 형식은 기본적으로 `==` 연산자를 지원합니다. 그러나 참조 형식은 `==` 연산자를 오버로드할 수 있습니다. 참조 형식이 `==` 연산자를 오버로드하는 경우 [Object.ReferenceEquals](#) 메서드를 사용하여 해당 형식의 두 참조가 동일한 개체를 참조하는지 확인합니다.

#### 레코드 형식 같음

C# 9.0 이상에서 사용할 수 있는 [레코드 형식](#)은 기본적으로 값 같음 의미 체계를 제공하는 `==` 및 `!=` 연산자를 지원합니다. 즉, 두 개의 레코드 피연산자는 둘 다 `null` 이거나 모든 필드와 자동 구현 속성의 해당 값이 같은 경우에 같습니다.

```

public class RecordTypesEquality
{
    public record Point(int X, int Y, string Name);
    public record TaggedNumber(int Number, List<string> Tags);

    public static void Main()
    {
        var p1 = new Point(2, 3, "A");
        var p2 = new Point(1, 3, "B");
        var p3 = new Point(2, 3, "A");

        Console.WriteLine(p1 == p2); // output: False
        Console.WriteLine(p1 == p3); // output: True

        var n1 = new TaggedNumber(2, new List<string>() { "A" });
        var n2 = new TaggedNumber(2, new List<string>() { "A" });
        Console.WriteLine(n1 == n2); // output: False
    }
}

```

앞의 예제처럼 레코드가 아닌 참조 형식 멤버의 경우 참조되는 인스턴스가 아니라 해당 참조 값이 비교됩니다.

#### 문자열 같음

두 개의 [문자열](#) 피연산자가 모두 `null` 이거나 두 문자열 인스턴스가 같은 길이고 각 문자 위치에 동일한 문자가 있을 때 동일합니다.

```

string s1 = "hello!";
string s2 = "HeLlo!";
Console.WriteLine(s1 == s2.ToLower()); // output: True

string s3 = "Hello!";
Console.WriteLine(s1 == s3); // output: False

```

대/소문자 구분 서수 비교입니다. 문자열 비교에 대한 자세한 내용은 [C#에서 문자열을 비교하는 방법](#)을 참조하세요.

대리자 같음

동일한 런타임 형식의 두 대리자 피연산자가 둘 다 `null` 이거나 해당 호출 목록의 길이가 같고 각 위치에 동일한 항목이 있는 경우 두 피연산자는 같습니다.

```
Action a = () => Console.WriteLine("a");

Action b = a + a;
Action c = a + a;
Console.WriteLine(object.ReferenceEquals(b, c)); // output: False
Console.WriteLine(b == c); // output: True
```

자세한 내용은 [C# 언어 사양](#)의 대리자 같음 연산자 섹션을 참조하세요.

의미상 동일한 [람다 식](#) 평가에서 생성된 대리자는 다음 예제와 같이 동일하지 않습니다.

```
Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("a");

Console.WriteLine(a == b); // output: False
Console.WriteLine(a + b == a + b); // output: True
Console.WriteLine(b + a == a + b); // output: False
```

## 같지 않음 연산자 !=

같지 않음 연산자 `!=`는 피연산자가 같지 않으면 `true`를 반환하고, 그렇지 않으면 `false`를 반환합니다. [기본 제공 형식](#)의 피연산자의 경우 식 `x != y`는 식 `!(x == y)`와 동일한 결과를 생성합니다. 형식 같음에 대한 자세한 내용은 [같음 연산자](#) 섹션을 참조하세요.

다음 예제에서는 `!=` 연산자의 사용법을 보여 줍니다.

```
int a = 1 + 1 + 2 + 3;
int b = 6;
Console.WriteLine(a != b); // output: True

string s1 = "Hello";
string s2 = "Hello";
Console.WriteLine(s1 != s2); // output: False

object o1 = 1;
object o2 = 1;
Console.WriteLine(o1 != o2); // output: True
```

## 연산자 오버로드 가능성

사용자 정의 형식은 `==` 및 `!=` 연산자를 [오버로드](#)할 수 있습니다. 형식이 두 연산자 중 하나를 오버로드하는 경우 나머지 연산자도 오버로드해야 합니다.

레코드 형식은 `==` 및 `!=` 연산자를 명시적으로 오버로드할 수 없습니다. 레코드 형식 `T`에 대한 `==` 및 `!=` 연산자의 동작을 변경해야 하는 경우 다음 시그니처를 사용하여 `IEquatable<T>.Equals` 메서드를 구현합니다.

```
public virtual bool Equals(T? other);
```

# C# 언어 사양

자세한 내용은 [C# 언어 사양의 관계형 및 형식 테스트 연산자](#) 섹션을 참조하세요.

레코드 형식 같음에 관한 자세한 내용은 [레코드 기능 제안 노트](#)의 [같음 멤버](#) 섹션을 참조하세요.

## 참조

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [System.IEquatable<T>](#)
- [Object.Equals](#)
- [Object.ReferenceEquals](#)
- [같음 비교](#)
- [비교 연산자](#)

# 비교 연산자(C# 참조)

2020-11-02 • 5 minutes to read • [Edit Online](#)

< (보다 작음), > (보다 큼), <= (작거나 같음) 및 >= (크거나 같음) 비교는 관계형 연산자라고도 하며, 피연산자를 비교합니다. 해당 연산자는 모든 정수 및 부동 소수점 숫자 형식을 지원합니다.

## NOTE

==, <, >, <= 및 >= 연산자의 경우 피연산자 중 하나가 숫자(Double.NaN 또는 Single.NaN)가 아니면 연산의 결과는 false입니다. 즉, NaN 값이 NaN를 포함한 다른 double(또는 float) 값보다 크거나, 작거나, 같지 않습니다. 자세한 내용과 예제는 Double.NaN 또는 Single.NaN 참조 문서를 참조하세요.

char 형식은 비교 연산자도 지원합니다. char 피연산자의 경우 해당 문자 코드가 비교됩니다.

열거형 형식은 비교 연산자도 지원합니다. 동일한 열거형 형식의 피연산자의 경우 기본 정수 형식의 해당 값이 비교됩니다.

== 및 != 연산자는 피연산자가 같은지 여부를 확인합니다.

## 보다 작음 연산자 <

< 연산자는 왼쪽 피연산자가 오른쪽 피연산자보다 작으면 true를 반환하고, 그렇지 않으면 false를 반환합니다.

```
Console.WriteLine(7.0 < 5.1); // output: False
Console.WriteLine(5.1 < 5.1); // output: False
Console.WriteLine(0.0 < 5.1); // output: True

Console.WriteLine(double.NaN < 5.1); // output: False
Console.WriteLine(double.NaN >= 5.1); // output: False
```

## 보다 큼 연산자 >

> 연산자는 왼쪽 피연산자가 오른쪽 피연산자보다 크면 true를 반환하고, 그렇지 않으면 false를 반환합니다.

```
Console.WriteLine(7.0 > 5.1); // output: True
Console.WriteLine(5.1 > 5.1); // output: False
Console.WriteLine(0.0 > 5.1); // output: False

Console.WriteLine(double.NaN > 5.1); // output: False
Console.WriteLine(double.NaN <= 5.1); // output: False
```

## 작거나 같음 연산자 <=

<= 연산자는 왼쪽 피연산자가 오른쪽 피연산자보다 작거나 같으면 true를 반환하고, 그렇지 않으면 false를 반환합니다.

```
Console.WriteLine(7.0 <= 5.1);    // output: False
Console.WriteLine(5.1 <= 5.1);    // output: True
Console.WriteLine(0.0 <= 5.1);    // output: True

Console.WriteLine(double.NaN > 5.1);    // output: False
Console.WriteLine(double.NaN <= 5.1);    // output: False
```

## 크거나 같음 연산자 >=

>= 연산자는 왼쪽 피연산자가 오른쪽 피연산자보다 크거나 같으면 `true`를 반환하고, 그렇지 않으면 `false`를 반환합니다.

```
Console.WriteLine(7.0 >= 5.1);    // output: True
Console.WriteLine(5.1 >= 5.1);    // output: True
Console.WriteLine(0.0 >= 5.1);    // output: False

Console.WriteLine(double.NaN < 5.1);    // output: False
Console.WriteLine(double.NaN >= 5.1);    // output: False
```

## 연산자 오버로드 가능성

사용자 정의 형식은 <, >, <= 및 >= 연산자를 [오버로드](#)할 수 있습니다.

형식이 < 또는 > 연산자 중 하나를 오버로드하는 경우 < 및 > 모두 오버로드해야 합니다. 형식이 <= 또는 >= 연산자 중 하나를 오버로드하는 경우 <= 및 >= 모두 오버로드해야 합니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 [관계형 및 형식 테스트 연산자](#) 섹션을 참조하세요.

## 참조

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [System.IComparable<T>](#)
- [같음 연산자](#)

# 멤버 액세스 연산자 및 식(C# 참조)

2021-02-18 • 19 minutes to read • [Edit Online](#)

형식 멤버에 액세스할 때 다음 연산자 및 식을 사용할 수 있습니다.

- `.` (멤버 액세스): 네임스페이스 또는 형식의 멤버 액세스
- `[]` (배열 요소 또는 인덱서 액세스): 배열 요소 또는 형식 인덱서 액세스
- `?.` 및 `?[]` (null 조건부 연산자): 피연산자가 null이 아닌 경우에만 멤버 또는 요소 액세스 작업 수행
- `()` (호출): 액세스된 메서드나 대리자 호출
- `^` (끝부터 인덱스): 요소 위치가 시퀀스의 끝에서 시작됨을 표시
- `..` (범위): 시퀀스 요소의 범위를 가져오는 데 사용할 수 있는 인덱스 범위를 지정

## 멤버 액세스 식 .

다음 예제와 같이 `.` 토큰을 사용하여 네임스페이스 또는 형식의 멤버에 액세스합니다.

- `using` 지시문의 다음 예제와 같이 `.`을 사용하여 네임스페이스 내에 중첩된 네임스페이스에 액세스합니다.

```
using System.Collections.Generic;
```

- 다음 코드와 같이 `.`을 사용하여 '정규화된 이름'을 만들고 네임스페이스 내의 형식에 액세스합니다.

```
System.Collections.Generic.IEnumerable<int> numbers = new int[] { 1, 2, 3 };
```

`using` 지시문을 사용하여 정규화된 이름 사용을 선택 사항으로 설정합니다.

- 다음 코드와 같이 `.`을 사용하여 정적 및 비정적 형식 멤버에 액세스합니다.

```
var constants = new List<double>();
constants.Add(Math.PI);
constants.Add(Math.E);
Console.WriteLine($"{constants.Count} values to show:");
Console.WriteLine(string.Join(", ", constants));
// Output:
// 2 values to show:
// 3.14159265358979, 2.71828182845905
```

`.`을 사용하여 확장 메서드에 액세스할 수도 있습니다.

## 인덱서 연산자 []

대괄호 `[]`는 일반적으로 배열, 인덱서 또는 포인터 요소 액세스에 사용됩니다.

### 배열 액세스

다음 예제는 배열 요소에 액세스하는 방법을 보여 줍니다.

```

int[] fib = new int[10];
fib[0] = fib[1] = 1;
for (int i = 2; i < fib.Length; i++)
{
    fib[i] = fib[i - 1] + fib[i - 2];
}
Console.WriteLine(fib[fib.Length - 1]); // output: 55

double[,] matrix = new double[2,2];
matrix[0,0] = 1.0;
matrix[0,1] = 2.0;
matrix[1,0] = matrix[1,1] = 3.0;
var determinant = matrix[0,0] * matrix[1,1] - matrix[1,0] * matrix[0,1];
Console.WriteLine(determinant); // output: -3

```

배열 인덱스가 배열의 해당 차원 범위를 벗어난 경우 [IndexOutOfRangeException](#)이 throw됩니다.

앞의 예제와 같이, 배열 형식을 선언하거나 배열 인스턴스를 인스턴스화할 때도 대괄호를 사용합니다.

배열에 대한 자세한 내용은 [배열](#)을 참조하세요.

### 인덱서 액세스

다음 예제는 .NET [Dictionary< TKey, TValue >](#) 형식을 사용하여 인덱서 액세스를 보여 줍니다.

```

var dict = new Dictionary<string, double>();
dict["one"] = 1;
dict["pi"] = Math.PI;
Console.WriteLine(dict["one"] + dict["pi"]); // output: 4.14159265358979

```

인덱서를 사용하면 배열 인덱싱과 비슷한 방법으로 사용자 정의 형식의 인스턴스를 인덱싱할 수 있습니다. 정수어야 하는 배열 인덱스와 달리, 인덱서 매개 변수는 임의 형식으로 선언할 수 있습니다.

인덱서에 대한 자세한 내용은 [인덱서](#)를 참조하세요.

### 다른 대괄호 용도

포인터 요소 액세스에 대한 자세한 내용은 [포인터 관련 연산자](#) 문서의 [포인터 요소 액세스 연산자](#) 섹션을 참조하세요.

또한 대괄호를 사용하여 [특성](#)을 지정합니다.

```

[System.Diagnostics.Conditional("DEBUG")]
void TraceMethod() {}

```

## Null 조건부 연산자 ?. 및 ?[]

C# 6 이상에서 사용할 수 있는 null 조건부 연산자는 피연산자가 null이 아닌 것으로 평가되었을 때만 [멤버 액세스](#), [?.](#) 또는 [요소 액세스](#), [?\[\]](#), 연산을 피연산자에게 적용하며, 그렇지 않으면 [null](#)을 반환합니다. 즉, 다음과 같습니다.

- `a`가 `null`로 평가되면 `a?.x` 또는 `a?[x]`의 결과는 `null`입니다.
- `a`가 `null`이 아닌 것으로 평가되면 `a?.x` 또는 `a?[x]`의 결과는 각각 `a.x` 또는 `a[x]`의 결과와 같습니다.

### NOTE

`a.x` 또는 `a[x]` 가 예외를 throw하면 `a?.x` 또는 `a?[x]` 는 null이 아닌 `a` 와 동일한 예외를 throw합니다. 예를 들어 `a` 가 null이 아닌 배열 인스턴스이고 `x` 가 `a` 의 경계 밖에 있는 경우, `a?[x]` 는 `IndexOutOfRangeException`을 throw합니다.

Null 조건부 연산자는 단락 연산자입니다. 즉 조건부 멤버나 요소 액세스 작업의 한 체인의 작업에서 `null` 을 반환하면 나머지 체인은 실행되지 않습니다. 다음 예제에서 `A` 가 `null`로 평가되면 `B` 가 평가되지 않고, `A` 또는 `B` 가 `null`로 평가되면 `C` 가 평가되지 않습니다.

```
A?.B?.Do(C);  
A?.B?[C];
```

다음 예제에서는 `?` 및 `?[]` 연산자의 사용법을 보여 줍니다.

```
double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)  
{  
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;  
}  
  
var sum1 = SumNumbers(null, 0);  
Console.WriteLine(sum1); // output: NaN  
  
var numberSets = new List<double[]>  
{  
    new[] { 1.0, 2.0, 3.0 },  
    null  
};  
  
var sum2 = SumNumbers(numberSets, 0);  
Console.WriteLine(sum2); // output: 6  
  
var sum3 = SumNumbers(numberSets, 1);  
Console.WriteLine(sum3); // output: NaN
```

또한 앞의 예제에서는 **null 병합 연산자** `??` 를 사용하여 null 조건부 연산 결과가 `null` 인 경우 평가할 대체 식을 지정합니다.

`a.x` 또는 `a[x]` 가 `null`을 허용하지 않는 값 형식인 경우 `T`, `a?.x` 또는 `a?[x]` 는 해당하는 **null 허용 값 형식** `T?` 입니다. `T` 형식의 식이 필요하면 다음 예제와 같이 null 병합 연산자 `??` 를 null 조건식에 적용합니다.

```
int GetSumOfFirstTwoOrDefault(int[] numbers)  
{  
    if ((numbers?.Length ?? 0) < 2)  
    {  
        return 0;  
    }  
    return numbers[0] + numbers[1];  
}  
  
Console.WriteLine(GetSumOfFirstTwoOrDefault(null)); // output: 0  
Console.WriteLine(GetSumOfFirstTwoOrDefault(new int[0])); // output: 0  
Console.WriteLine(GetSumOfFirstTwoOrDefault(new[] { 3, 4, 5 })); // output: 7
```

앞의 예제에서 `??` 연산자를 사용하지 않는 경우 `numbers?.Length < 2` 는 `numbers` 가 `null` 일 때 `false` 로 평가 됩니다.

Null 조건부 멤버 액세스 연산자 `?` 를 Elvis 연산자라고도 합니다.

## 스레드로부터 안전한 대리자 호출

다음 코드에서처럼 `?.` 연산자를 사용하여 대리자가 `null`이 아닌지 확인하고 스레드로부터 안전한 방식으로 호출합니다(예: [이벤트 발생 시](#)).

```
PropertyChanged?.Invoke(...)
```

이 코드는 C# 5 및 그 이전에서 사용하는 다음 코드에 해당합니다.

```
var handler = this.PropertyChanged;
if (handler != null)
{
    handler(...);
}
```

이는 `null`이 아닌 `handler`만 호출되도록 하는 스레드로부터 안전한 방법입니다. 대리자 인스턴스는 변경할 수 없으므로 스레드는 `handler` 지역 변수가 참조하는 개체를 변경할 수 없습니다. 특히 다른 스레드가 실행한 코드가 `PropertyChanged` 이벤트에서 구독을 취소하고 `handler`를 호출하기 전에 `PropertyChanged`가 `null`이 되면 `handler`에서 참조하는 개체는 영향을 받지 않습니다. `?.` 연산자는 왼쪽 피연산자를 한 번만 계산하여 `null`이 아닌 것으로 확인된 후에는 `null`로 변경할 수 있도록 보장합니다.

## 호출 식 ()

괄호(`()`)를 사용하여 [메서드](#) 또는 [대리자](#)를 호출합니다.

다음 예제는 인수를 사용하거나 사용하지 않고 메서드를 호출하는 방법과 대리자를 호출하는 방법을 보여 줍니다.

```
Action<int> display = s => Console.WriteLine(s);

var numbers = new List<int>();
numbers.Add(10);
numbers.Add(17);
display(numbers.Count); // output: 2

numbers.Clear();
display(numbers.Count); // output: 0
```

`new` 연산자를 사용하여 [생성자](#)를 호출하는 경우에도 괄호를 사용합니다.

## 다른 0 용도

또한 괄호를 사용하여 식에서 연산을 계산하는 순서를 조정합니다. 자세한 내용은 [C# 연산자](#)를 참조하세요.

명시적 형식 변환을 수행하는 [캐스트 식](#)도 괄호를 사용합니다.

## 끝부터 인덱스 연산자 ^

C# 8.0 이상에서 사용할 수 있는 연산자 `^`는 요소 위치가 시퀀스의 끝에서 시작함을 나타냅니다. 시퀀스 길이 `length`의 경우 `^n`은 시퀀스의 시작에서 오프셋 `length - n`인 요소를 가리킵니다. 예를 들어 `^1`은 시퀀스의 마지막 요소를 가리키고, `^length`는 시퀀스의 첫 번째 요소를 가리킵니다.

```

int[] xs = new[] { 0, 10, 20, 30, 40 };
int last = xs[^1];
Console.WriteLine(last); // output: 40

var lines = new List<string> { "one", "two", "three", "four" };
string prelast = lines[^2];
Console.WriteLine(prelast); // output: three

string word = "Twenty";
Index toFirst = ^word.Length;
char first = word[toFirst];
Console.WriteLine(first); // output: T

```

위 예제에서와 같이 식 `^e`는 [System.Index](#) 형식입니다. 식 `^e`에서 `e`의 결과는 암시적으로 `int`으로 변환할 수 있어야 합니다.

연산자를 [범위 연산자](#)와 함께 사용하여 인덱스 범위를 만들 수도 있습니다. 자세한 내용은 [인덱스와 범위](#)를 참조하세요.

## 범위 연산자 .

C# 8.0 이상에서 사용 가능한 연산자 `..`은 인덱스 범위의 시작과 끝을 피연산자로 지정합니다. 왼쪽 피연산자는 범위의 시작(포함)입니다. 오른쪽 피연산자는 범위의 끝(제외)입니다. 다음 예제에서와 같이 피연산자 중 하나는 시퀀스의 시작부터 또는 끝부터 인덱스가 될 수 있습니다.

```

int[] numbers = new[] { 0, 10, 20, 30, 40, 50 };
int start = 1;
int amountToTake = 3;
int[] subset = numbers[start..(start + amountToTake)];
Display(subset); // output: 10 20 30

int margin = 1;
int[] inner = numbers[margin..^margin];
Display(inner); // output: 10 20 30 40

string line = "one two three";
int amountToTakeFromEnd = 5;
Range endIndices = ^amountToTakeFromEnd..^0;
string end = line[endIndices];
Console.WriteLine(end); // output: three

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ", xs));

```

위 예제에서와 같이 식 `a..b`는 [System.Range](#) 형식입니다. 식 `a..b`에서 `a` 및 `b`의 결과는 암시적으로 `int` 또는 [Index](#)로 변환할 수 있어야 합니다.

연산자의 피연산자 중 하나를 생략하여 개방형 범위를 지정할 수 있습니다.

- `a..`는 `a..^0`와 같습니다.
- `..b`는 `0..b`와 같습니다.
- `..`는 `0..^0`와 같습니다.

```
int[] numbers = new[] { 0, 10, 20, 30, 40, 50 };
int amountToDrop = numbers.Length / 2;

int[] rightHalf = numbers[amountToDrop..];
Display(rightHalf); // output: 30 40 50

int[] leftHalf = numbers[..^amountToDrop];
Display(leftHalf); // output: 0 10 20

int[] all = numbers[..];
Display(all); // output: 0 10 20 30 40 50

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ", xs));
```

자세한 내용은 [인덱스와 범위](#)를 참조하세요.

## 연산자 오버로드 가능성

. , () , ^ 및 .. 연산자는 오버로드할 수 없습니다. [] 연산자도 오버로드할 수 없는 연산자로 간주됩니다. [인덱서](#)를 사용하여 사용자 정의 형식의 인덱싱을 지원합니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 다음 섹션을 참조하세요.

- [멤버 액세스](#)
- [요소 액세스](#)
- [Null 조건부 연산자](#)
- [호출 식](#)

인덱스 및 범위에 대한 자세한 내용은 [기능 제안 노트](#)를 참조하세요.

## 참조

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [??\(Null 병합 연산자\)](#)
- [:: 연산자](#)

# 형식 테스트 연산자 및 캐스트 식(C# 참조)

2020-11-02 • 15 minutes to read • [Edit Online](#)

다음 연산자 및 식을 사용하여 형식 검사 또는 형식 변환을 수행할 수 있습니다.

- **is 연산자**: 식의 런타임 형식이 지정된 형식과 호환되는지 확인합니다.
- **as 연산자**: 런타임 형식이 지정된 형식과 호환되는 경우 식을 해당 형식으로 명시적으로 변환합니다.
- **캐스트 식**: 명시적 변환을 수행합니다.
- **typeof 연산자**: 형식의 `System.Type` 인스턴스를 가져옵니다.

## is 연산자

`is` 연산자는 식 결과의 런타임 형식이 지정된 형식과 호환되는지 확인합니다. C# 7.0부터, `is` 연산자는 식 결과에 패턴이 있는지도 테스트합니다.

형식 테스트 `is` 연산자가 포함된 식의 양식은 다음과 같습니다.

```
E is T
```

여기서 `E`는 값을 반환하는 식이고, `T`는 형식 또는 형식 매개 변수의 이름입니다. `E`은 무명 메서드 또는 람다 식일 수 없습니다.

`E is T` 식은 `E`의 결과가 `null`이 아니고 참조 변환, boxing 변환 또는 unboxing 변환을 통해 `T` 형식으로 변환될 수 있는 경우 `true`를 반환하고, 변환될 수 없으면 `false`를 반환합니다. `is` 연산자는 사용자 정의 변환을 고려하지 않습니다.

다음 예제에서는 식 결과의 런타임 형식이 지정된 형식에서 파생되는 경우, 즉 형식 간에 참조 변환이 있는 경우 `is` 연산자가 `true`를 반환하는 것을 보여 줍니다.

```
public class Base { }

public class Derived : Base { }

public static class IsOperatorExample
{
    public static void Main()
    {
        object b = new Base();
        Console.WriteLine(b is Base); // output: True
        Console.WriteLine(b is Derived); // output: False

        object d = new Derived();
        Console.WriteLine(d is Base); // output: True
        Console.WriteLine(d is Derived); // output: True
    }
}
```

다음 예제에서는 `is` 연산자가 boxing 및 unboxing 변환은 고려하지만 [숫자 변환](#)을 고려하지 않는 것을 보여 줍니다.

```
int i = 27;
Console.WriteLine(i is System.IFormattable); // output: True

object iBoxed = i;
Console.WriteLine(iBoxed is int); // output: True
Console.WriteLine(iBoxed is long); // output: False
```

C# 변환에 대한 자세한 내용은 [C# 언어 사양의 변환](#) 장을 참조하세요.

패턴 일치를 사용한 형식 테스트

C# 7.0부터, `is` 연산자는 식 결과에 패턴이 있는지도 테스트합니다. 특히, 다음 형태의 양식 패턴을 지원합니다.

```
E is T v
```

여기서 `E`는 값을 반환하는 식이고, `T`는 형식 또는 형식 매개 변수의 이름이며, `v`는 `T` 형식의 새 로컬 변수입니다. `E`의 결과가 `null`이 아니고 참조, boxing 또는 unboxing 변환을 통해 `T`로 변환될 수 있는 경우 `E is T v` 식이 `true`를 반환하고 `E` 결과의 변환된 값이 `v` 변수에 할당됩니다.

다음 예제에서는 형식 패턴과 `is` 연산자를 사용하는 방법을 보여 줍니다.

```
int i = 23;
object iBoxed = i;
int? jNullable = 7;
if (iBoxed is int a && jNullable is int b)
{
    Console.WriteLine(a + b); // output 30
}
```

형식 패턴 및 기타 지원되는 패턴에 대한 자세한 내용은 [is를 사용한 패턴 일치](#)를 참조하세요.

## as 연산자

`as` 연산자는 식의 결과를 지정된 참조 또는 nullable 값 형식으로 명시적으로 변환합니다. 변환할 수 없는 경우

`as` 연산자가 `null`을 반환합니다. [캐스트 식](#)과 달리 `as` 연산자는 예외를 throw하지 않습니다.

다음 형태의 식이 있다고 가정합니다.

```
E as T
```

여기서 `E`는 값을 반환하는 식이고, `T`는 형식 또는 형식 매개 변수의 이름입니다. 이 식은 다음과 동일한 결과를 생성합니다.

```
E is T ? (T)(E) : (T)null
```

단, `E`가 한 번만 계산됩니다.

`as` 연산자는 참조, nullable, boxing 및 unboxing 변환만 고려합니다. `as` 연산자를 사용하여 사용자 정의 변환을 수행할 수는 없습니다. 이렇게 하려면 [캐스트 식](#)을 사용합니다.

다음 예제에서는 `as` 연산자의 사용법을 보여 줍니다.

```
IEnumerable<int> numbers = new[] { 10, 20, 30 };
IList<int> indexable = numbers as IList<int>;
if (indexable != null)
{
    Console.WriteLine(indexable[0] + indexable[indexable.Count - 1]); // output: 40
}
```

#### NOTE

앞의 예제와 같이, `as` 식의 결과를 `null` 과 비교하여 변환에 성공했는지 확인해야 합니다. C# 7.0부터, [is 연산자](#)를 사용하여 변환에 성공하는지 테스트하고, 성공한 경우 해당 결과를 새 변수에 할당할 수 있습니다.

## 캐스트 식

`(T)E` 형태의 캐스트 식은 `E` 식의 결과를 `T` 형식으로 명시적으로 변환합니다. `E` 형식에서 `T` 형식으로의 명시적 변환이 없는 경우 컴파일 시간 오류가 발생합니다. 런타임에 명시적 변환이 실패하고 캐스트 식이 예외를 `throw`할 수도 있습니다.

다음 예제에서는 명시적 숫자 및 참조 변환을 보여 줍니다.

```
double x = 1234.7;
int a = (int)x;
Console.WriteLine(a); // output: 1234

IEnumerable<int> numbers = new int[] { 10, 20, 30 };
IList<int> list = (IList<int>)numbers;
Console.WriteLine(list.Count); // output: 3
Console.WriteLine(list[1]); // output: 20
```

지원되는 명시적 변환에 대한 자세한 내용은 [C# 언어 사양](#)의 [명시적 변환](#) 섹션을 참조하세요. 사용자 지정 명시적 또는 암시적 형식 변환을 정의하는 방법에 대한 자세한 내용은 [사용자 정의 변환 연산자](#)를 참조하세요.

### 다른 0 용도

[메서드 또는 대리자를 호출](#)하는 경우에도 괄호를 사용합니다.

괄호를 사용하여 식에서 연산을 계산하는 순서를 조정하기도 합니다. 자세한 내용은 [C# 연산자](#)를 참조하세요.

## typeof 연산자

`typeof` 연산자는 형식의 `System.Type` 인스턴스를 가져옵니다. `typeof` 연산자의 인수는 다음 예제와 같이 형식 또는 형식 매개 변수의 이름이어야 합니다.

```
void PrintType<T>() => Console.WriteLine(typeof(T));

Console.WriteLine(typeof(List<string>));
PrintType<int>();
PrintType<System.Int32>();
PrintType<Dictionary<int, char>>();
// Output:
// System.Collections.Generic.List`1[System.String]
// System.Int32
// System.Int32
// System.Collections.Generic.Dictionary`2[System.Int32, System.Char]
```

바인딩되지 않은 제네릭 형식과 `typeof` 연산자를 사용할 수도 있습니다. 바인딩되지 않은 제네릭 형식의 이름에는 형식 매개 변수 개수보다 하나 더 적은 적절한 개수의 쉼표가 포함되어야 합니다. 다음 예제에서는 바인딩

되지 않은 제네릭 형식과 `typeof` 연산자를 사용하는 방법을 보여 줍니다.

```
Console.WriteLine(typeof(Dictionary<, >));  
// Output:  
// System.Collections.Generic.Dictionary`2[TKey, TValue]
```

식은 `typeof` 연산자의 인수가 될 수 없습니다. 식 결과의 런타임 형식에 대한 `System.Type` 인스턴스를 가져오려면 `Object.GetType` 메서드를 사용합니다.

`typeof` 연산자를 사용한 형식 테스트

`typeof` 연산자를 사용하여 식 결과의 런타임 형식이 지정된 형식과 정확히 일치하는지 확인합니다. 다음 예제에서는 `typeof` 연산자와 `is` 연산자를 사용한 형식 검사의 차이점을 보여 줍니다.

```
public class Animal { }  
  
public class Giraffe : Animal { }  
  
public static class TypeOfExample  
{  
    public static void Main()  
    {  
        object b = new Giraffe();  
        Console.WriteLine(b is Animal); // output: True  
        Console.WriteLine(b.GetType() == typeof(Animal)); // output: False  
  
        Console.WriteLine(b is Giraffe); // output: True  
        Console.WriteLine(b.GetType() == typeof(Giraffe)); // output: True  
    }  
}
```

## 연산자 오버로드 가능성

`is`, `as` 및 `typeof` 연산자는 오버로드할 수 없습니다.

사용자 정의 형식은 `()` 연산자를 오버로드할 수 없지만, 캐스트 식에서 수행할 수 있는 사용자 지정 형식 변환을 정의할 수 있습니다. 자세한 내용은 [사용자 정의 변환 연산자](#)를 참조하세요.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 다음 섹션을 참조하세요.

- [is 연산자](#)
- [as 연산자](#)
- [캐스트 식](#)
- [typeof 연산자](#)

## 참조

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [패턴 일치와 is 및 as 연산자를 사용하여 안전하게 캐스트하는 방법](#)
- [.NET의 제네릭](#)

# 사용자 정의 전환 연산자(C# 참조)

2020-11-02 • 4 minutes to read • [Edit Online](#)

사용자 정의 형식은 다른 형식에서 또는 다른 형식으로 사용자 지정 암시적 또는 명시적 변환을 정의할 수 있습니다.

암시적 변환은 특별한 구문을 호출할 필요가 없으며, 할당과 메서드 호출과 같은 다양한 상황에서 발생할 수 있습니다. 미리 정의된 C# 암시적 변환은 항상 성공하며 예외를 throw하지 않습니다. 사용자 정의 암시적 변화도 이러한 방식으로 작동해야 합니다. 사용자 지정 변환이 예외를 throw하거나 정보가 손실될 수 있는 경우 이를 명시적 변환으로 정의합니다.

사용자 정의 변환은 `is` 및 `as` 연산자로 간주되지 않습니다. [캐스트 식](#)을 사용하여 사용자 정의 명시적 변환을 호출합니다.

`operator` 및 `implicit` 또는 `explicit` 키워드를 사용하여 각각 암시적 또는 명시적 변환을 정의합니다. 변환을 정의하는 유형은 해당 변환의 소스 유형 또는 대상 유형이어야 합니다. 두 형식 중 하나에서 두 개의 사용자 정의 형식 간의 변환을 정의할 수 있습니다.

다음 예제에서는 암시적 및 명시적 변환을 정의하는 방법을 보여줍니다.

```
using System;

public readonly struct Digit
{
    private readonly byte digit;

    public Digit(byte digit)
    {
        if (digit > 9)
        {
            throw new ArgumentOutOfRangeException(nameof(digit), "Digit cannot be greater than nine.");
        }
        this.digit = digit;
    }

    public static implicit operator byte(Digit d) => d.digit;
    public static explicit operator Digit(byte b) => new Digit(b);

    public override string ToString() => $"{digit}";
}

public static class UserDefinedConversions
{
    public static void Main()
    {
        var d = new Digit(7);

        byte number = d;
        Console.WriteLine(number); // output: 7

        Digit digit = (Digit)number;
        Console.WriteLine(digit); // output: 7
    }
}
```

또한 `operator` 키워드를 사용하여 미리 정의된 C# 연산자를 오버로드합니다. 자세한 내용은 [연산자 오버로드](#)를 참조하세요.

# C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 다음 섹션을 참조하세요.

- [변환 연산자](#)
- [사용자 정의 변환](#)
- [암시적 변환](#)
- [명시적 변환](#)

## 참조

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [연산자 오버로드](#)
- [형식 테스트 및 캐스트 연산자](#)
- [캐스팅 및 형식 변환](#)
- [디자인 지침 - 변환 연산자](#)
- [Chained user-defined explicit conversions in C#\(C#의 연결된 사용자 정의 명시적 변환\)](#)

# 포인터 관련 연산자(C# 참조)

2020-11-02 • 17 minutes to read • [Edit Online](#)

다음 연산자를 사용하여 포인터로 작업할 수 있습니다.

- 단항 `&` (address-of) 연산자: 변수의 주소를 가져오려면
- 단항 `*` (포인터 간접 참조) 연산자: 포인터가 가리키는 변수를 가져오려면
- `->` (멤버 액세스) 및 `[]` (요소 액세스) 연산자
- 산술 연산자 `+`, `-`, `++` 및 `--`
- 비교 연산자 `==`, `!=`, `<`, `>`, `<=` 및 `>=`

포인터 형식에 대한 내용은 [포인터 형식](#)을 참조하세요.

## NOTE

포인터를 사용한 작업에는 [안전하지 않은 컨텍스트](#)가 필요합니다. 안전하지 않은 블록을 포함하는 코드는 `-unsafe` 컴파일러 옵션으로 컴파일해야 합니다.

## Address-of 연산자 &

단항 `&` 연산자는 해당 피연산자의 주소를 반환합니다.

```
unsafe
{
    int number = 27;
    int* pointerToNumber = &number;

    Console.WriteLine($"Value of the variable: {number}");
    Console.WriteLine($"Address of the variable: {(long)pointerToNumber:X}");
}
```

`&` 연산자의 피연산자는 고정 변수여야 합니다. 고정 변수는 [가비지 수집기](#)의 작동에 영향을 받지 않는 스토리지 위치에 있는 변수입니다. 앞의 예제에서 로컬 변수 `number`는 스택에 있으므로 고정 변수입니다. 가비지 수집기에 의해 영향을 받을 수 있는 스토리지 위치에 상주하는 변수(예: 재배치됨)를 이동 가능한 변수라고 합니다. 개체 필드 및 배열 요소는 이동 가능한 변수의 예입니다. `fixed` 문으로 "fix" 또는 "pin"으로 할 경우 이동 가능한 변수의 주소를 가져올 수 있습니다. 가져온 주소는 `fixed` 문 블록 내에서만 유효합니다. 다음 예제에서는 `fixed` 문과 `&` 연산자를 사용하는 방법을 보여줍니다.

```
unsafe
{
    byte[] bytes = { 1, 2, 3 };
    fixed (byte* pointerToFirst = &bytes[0])
    {
        // The address stored in pointerToFirst
        // is valid only inside this fixed statement block.
    }
}
```

상수 또는 값의 주소를 가져올 수 없습니다.

고정 및 이동 가능한 변수에 대한 자세한 내용은 [C# 언어 사양의 고정 및 이동 가능 변수 섹션](#)을 참조하세요.

이진 `&` 연산자는 해당 부울 피연산자의 [논리 AND](#) 또는 해당 정수 피연산자의 [비트 논리 AND](#)를 컴퓨팅합니다.

## 포인터 간접 참조 연산자 \*

단항 포인터 간접 참조 연산자 `*`는 피연산자가 가리키는 변수를 가져옵니다. 역참조 연산자라고도 합니다. `*` 연산자의 피연산자는 포인터 형식이여야 합니다.

```
unsafe
{
    char letter = 'A';
    char* pointerToLetter = &letter;
    Console.WriteLine($"Value of the `letter` variable: {letter}");
    Console.WriteLine($"Address of the `letter` variable: {(long)pointerToLetter:X}");

    *pointerToLetter = 'Z';
    Console.WriteLine($"Value of the `letter` variable after update: {letter}");
}
// Output is similar to:
// Value of the `letter` variable: A
// Address of the `letter` variable: DCB977DDF4
// Value of the `letter` variable after update: Z
```

\* 연산자를 `void*` 유형의 식에 적용할 수 없습니다.

이진 `*` 연산자는 해당 숫자 피연산자의 [곱](#)을 컴퓨팅합니다.

## 포인터 멤버 액세스 연산자 ->

-> 연산자는 포인터 [간접 참조](#)와 [멤버 액세스](#)를 결합합니다. 즉, `x`가 `T*` 형식의 포인터이고 `y`가 `T`의 액세스 가능한 멤버인 경우 양식의 식은

```
x->y
```

위의 식은 아래의 식과 동일합니다.

```
(*x).y
```

다음 예제에서는 `->` 연산자의 사용법을 보여 줍니다.

```

public struct Coords
{
    public int X;
    public int Y;
    public override string ToString() => $"{{X}}, {{Y}}";
}

public class PointerMemberAccessExample
{
    public static unsafe void Main()
    {
        Coords coords;
        Coords* p = &coords;
        p->X = 3;
        p->Y = 4;
        Console.WriteLine(p->ToString()); // output: (3, 4)
    }
}

```

-> 연산자를 `void*` 유형의 식에 적용할 수 없습니다.

## 포인터 요소 액세스 연산자 []

포인터 형식의 식 `p`의 경우 `p[n]` 양식의 포인터 요소 액세스는 `*(p + n)`으로 평가됩니다. 여기서 `n`은 암시적으로 `int`, `uint`, `long` 또는 `ulong`으로 전환할 수 있는 형식이어야 합니다. 포인터가 있는 `+` 연산자의 동작에 대한 자세한 내용은 [포인터에 또는 포인터에서 정수 값 더하기 또는 빼기](#) 섹션을 참조하세요.

다음 예제에서는 포인터 및 `[]` 연산자를 사용하여 배열 요소에 액세스하는 방법을 보여 줍니다.

```

unsafe
{
    char* pointerToChars = stackalloc char[123];

    for (int i = 65; i < 123; i++)
    {
        pointerToChars[i] = (char)i;
    }

    Console.Write("Uppercase letters: ");
    for (int i = 65; i < 91; i++)
    {
        Console.Write(pointerToChars[i]);
    }
}
// Output:
// Uppercase letters: ABCDEFGHIJKLMNOPQRSTUVWXYZ

```

앞의 예제에서 `stackalloc` 식은 스택에 메모리 블록을 할당합니다.

### NOTE

포인터 요소 액세스 연산자는 범위 이탈 오류를 검사하지 않습니다.

`void*` 형식의 식으로 포인터 요소 액세스에 `[]`(을)를 사용할 수 없습니다.

[배열 요소 또는 인덱서 액세스](#)에 대해 `[]` 연산자를 사용할 수도 있습니다.

## 포인터 산술 연산자

포인터를 사용하여 다음과 같은 산술 연산을 수행할 수 있습니다.

- 포인터로 또는 포인터에서 정수 값 추가 또는 빼기
- 두 포인터 빼기
- 포인터 증가 또는 감소

`void*` 형식의 포인터를 사용하여 이러한 작업을 수행할 수 없습니다.

숫자 형식으로 지원되는 산술 연산에 대한 내용은 [산술 연산자](#)를 참조하세요.

포인터로 또는 포인터에서 정수 값 더하기 또는 빼기

`T*` 형식의 포인터 `p` 및 `int`, `uint`, `long` 또는 `ulong`으로 암시적으로 변환할 수 있는 형식의 `n` 식에 대한 더하기와 빼기가 다음과 같이 정의됩니다.

- `p + n` 및 `n + p` 식 모두 `p`에서 지정한 주소에 `n * sizeof(T)`를 추가한 결과 `T*` 유형의 포인터를 생성합니다.
- `p - n` 식은 `p`에 의해 지정된 주소에서 `n * sizeof(T)`를 뺀 결과 `T*` 유형의 포인터를 생성합니다.

`sizeof` 연산자는 바이트 단위의 형식 크기를 가져옵니다.

다음 예제에서는 포인터로 `+` 연산자를 사용하는 방법을 보여줍니다.

```
unsafe
{
    const int Count = 3;
    int[] numbers = new int[Count] { 10, 20, 30 };
    fixed (int* pointerToFirst = &numbers[0])
    {
        int* pointerToLast = pointerToFirst + (Count - 1);

        Console.WriteLine($"Value {*pointerToFirst} at address {(long)pointerToFirst}");
        Console.WriteLine($"Value {*pointerToLast} at address {(long)pointerToLast}");
    }
}
// Output is similar to:
// Value 10 at address 1818345918136
// Value 30 at address 1818345918144
```

## 포인터 빼기

`T*` 형식의 두 가지 포인터 `p1` 및 `p2`에 대해 `p1 - p2` 식은 `p1`과 `p2`에 의해 지정된 주소 간의 차이를 `sizeof(T)`로 나누어 생성합니다. 결과의 형식은 `long`입니다. 즉, `p1 - p2`는  $((long)(p1) - (long)(p2)) / sizeof(T)$ 로 계산됩니다.

다음 예제에서는 포인터 빼기를 보여줍니다.

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2, 3, 4, 5 };
    int* p1 = &numbers[1];
    int* p2 = &numbers[5];
    Console.WriteLine(p2 - p1); // output: 4
}
```

## 포인터 증가 및 감소

`++` 증가 연산자는 포인터 피연산자에 1을 [추가](#)합니다. `--` 감소 연산자는 포인터 피연산자에서 1을 [뺀다](#).

두 연산자는 접미사(`++p` 및 `p--`)와 접두사(`++p` 및 `--p`)의 두 가지 형태로 지원됩니다. `++p` 및 `p--`의 결과는 작업 '전'의 `p`의 값입니다. `++p` 및 `--p`의 결과는 작업 '후'의 `p`의 값입니다.

다음 예제에서는 접미사 및 접두사 증가 연산자의 동작을 보여줍니다.

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2 };
    int* p1 = &numbers[0];
    int* p2 = p1;
    Console.WriteLine($"Before operation: p1 - {(long)p1}, p2 - {(long)p2}");
    Console.WriteLine($"Postfix increment of p1: {(long)(p1++)}");
    Console.WriteLine($"Prefix increment of p2: {(long)(++p2)}");
    Console.WriteLine($"After operation: p1 - {(long)p1}, p2 - {(long)p2}");
}
// Output is similar to
// Before operation: p1 - 816489946512, p2 - 816489946512
// Postfix increment of p1: 816489946512
// Prefix increment of p2: 816489946516
// After operation: p1 - 816489946516, p2 - 816489946516
```

## 포인터 비교 연산자

`==`, `!=`, `<`, `>`, `<=` 및 `>=` 연산자를 사용하여 `void*`를 포함한 모든 포인터 형식의 피연산자를 비교할 수 있습니다. 이러한 연산자는 두 피연산자가 지정한 주소를 부호 없는 정수인 것처럼 비교합니다.

다른 형식의 피연산자에 대한 해당 연산자의 동작에 대한 정보는 [항등 연산자](#) 및 [비교 연산자](#) 문서를 참조하세요.

## 연산자 우선 순위

다음 목록에서는 포인터 관련 연산자를 가장 높은 우선순위부터 가장 낮은 것으로 정렬합니다.

- 접미사 증가 `x++` 및 감소 `x--` 연산자와 `->` 및 `[]` 연산자
- 접두사 증가 `++x` 및 감소 `--x` 연산자와 `&` 및 `*` 연산자
- 가감 `+` 및 `-` 연산자
- 비교 `<`, `>`, `<=` 및 `>=` 연산자
- 같음 `==` 및 `!=` 연산자

괄호(`()`)를 사용하여 연산자 우선순위에 따라 주어진 평가 순서를 변경합니다.

우선 순위 수준에 따라 정렬된 전체 연산자 목록은 [C# 연산자](#) 문서의 [연산자 우선 순위](#) 섹션을 참조하세요.

## 연산자 오버로드 가능성

사용자 정의 형식은 포인터 관련 연산자 `&`, `*`, `->` 및 `[]`(을)를 오버로드할 수 없습니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 다음 섹션을 참조하세요.

- [고정 및 고정되지 않은 변수](#)
- [address-of 연산자](#)
- [포인터 간접 참조](#)
- [포인터 멤버 액세스](#)
- [포인터 요소 액세스](#)
- [포인터 산술 연산](#)
- [포인터 증가 및 감소](#)
- [포인터 비교](#)

## 참조

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [포인터 형식](#)
- [unsafe 키워드](#)
- [fixed 키워드](#)
- [stackalloc](#)
- [sizeof 연산자](#)

# 대입 연산자(C# 참조)

2020-11-02 • 6 minutes to read • [Edit Online](#)

대입 연산자 `=`은 오른쪽 피연산자의 값을 왼쪽 피연산자가 제공하는 변수, 속성 또는 인덱서 요소에 할당합니다. 대입식의 결과는 왼쪽 피연산자에 할당된 값입니다. 오른쪽 피연산자의 형식은 왼쪽 피연산자의 형식과 동일하거나 왼쪽 피연산자의 형식으로 암시적으로 변환할 수 있어야 합니다.

= 대입 연산자는 오른쪽 결합성입니다. 즉, 다음 형식의 식을 가정해 보세요.

```
a = b = c
```

이 식은 다음과 같이 계산됩니다.

```
a = (b = c)
```

다음 예제에서는 로컬 변수, 속성 및 인덱서 요소를 왼쪽 피연산자로 포함해서 대입 연산자를 사용하는 방법을 보여 줍니다.

```
var numbers = new List<double>() { 1.0, 2.0, 3.0 };

Console.WriteLine(numbers.Capacity);
numbers.Capacity = 100;
Console.WriteLine(numbers.Capacity);
// Output:
// 4
// 100

int newFirstElement;
double originalFirstElement = numbers[0];
newFirstElement = 5;
numbers[0] = newFirstElement;
Console.WriteLine(originalFirstElement);
Console.WriteLine(numbers[0]);
// Output:
// 1
// 5
```

## ref 대입 연산자

C# 7.3부터 ref 대입 연산자 `= ref`를 사용하여 **ref 지역** 또는 **ref readonly 지역** 변수를 다시 할당할 수 있습니다. 다음 예제에서는 ref 대입 연산자의 사용법을 보여 줍니다.

```

void Display(double[] s) => Console.WriteLine(string.Join(" ", s));

double[] arr = { 0.0, 0.0, 0.0 };
Display(arr);

ref double arrayElement = ref arr[0];
arrayElement = 3.0;
Display(arr);

arrayElement = ref arr[arr.Length - 1];
arrayElement = 5.0;
Display(arr);
// Output:
// 0 0 0
// 3 0 0
// 3 0 5

```

`ref` 대입 연산자의 경우 양쪽 피연산자의 형식이 같아야 합니다.

## 복합 할당

이진 연산자(`op`)의 경우 양식의 복합 할당 식

```
x op= y
```

위의 식은 아래의 식과 동일합니다.

```
x = x op y
```

단, `x`가 한 번만 계산됩니다.

복합 할당은 [산술](#), [부울 논리](#), [비트 논리](#) 및 [시프트](#) 연산자를 통해 지원됩니다.

## null 병합 할당

C# 8.0부터 null 병합 할당 연산자 `??=`를 사용하여 왼쪽 피연산자가 `null`로 계산되는 경우에만 오른쪽 피연산자의 값을 왼쪽 피연산자에 대입할 수 있습니다. 자세한 내용은 [?? 및 ??= 연산자](#) 문서를 참조하세요.

## 연산자 오버로드 가능성

사용자 정의 형식은 대입 연산자를 [오버로드](#)할 수 없습니다. 그러나 사용자 정의 형식은 다른 형식으로 암시적 변환을 정의할 수 있습니다. 이렇게 하면 사용자 정의 형식의 값을 다른 형식의 변수, 속성 또는 인덱서 요소에 할당할 수 있습니다. 자세한 내용은 [사용자 정의 변환 연산자](#)를 참조하세요.

사용자 정의 형식은 복합 대입 연산자를 명시적으로 오버로드할 수 없습니다. 그러나 사용자 정의 형식이 이항 연산자 `op`을 오버로드하는 경우에는 `op=` 연산자(있는 경우)도 암시적으로 오버로드됩니다.

## C# 언어 사양

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

`ref` 대입 연산자 `= ref`에 대한 자세한 내용은 [기능 제안 노트](#)를 참조하세요.

## 참고 항목

- [C# 참조](#)

- C# 연산자 및 식
- ref 키워드

# 람다 식(C# 참조)

2021-02-18 • 28 minutes to read • [Edit Online](#)

'람다 식'을 사용하여 익명 함수를 만듭니다. 람다 선언 연산자 `=>`를 사용하여 본문에서 람다의 매개 변수 목록을 구분합니다. 람다 식은 다음과 같은 두 가지 형식 중 하나일 수 있습니다.

- 식이 본문으로 포함된 [식 람다](#):

```
(input-parameters) => expression
```

- 문 블록이 본문으로 포함된 [문 람다](#):

```
(input-parameters) => { <sequence-of-statements> }
```

람다 식을 만들려면 람다 연산자 왼쪽에 입력 매개 변수를 지정하고(있는 경우) 다른 쪽에 식이나 문 블록을 지정합니다.

람다 식은 [대리자](#) 형식으로 변환할 수 있습니다. 람다 식을 변환할 수 있는 대리자 형식은 해당 매개 변수 및 반환 값의 형식에 따라 정의됩니다. 람다 식에서 값을 반환하지 않는 경우 `Action` 대리자 형식 중 하나로 변환할 수 있습니다. 값을 반환하는 경우 `Func` 대리자 형식으로 변환할 수 있습니다. 예를 들어 매개 변수는 두 개지만 값을 반환하지 않는 람다 식은 `Action<T1,T2>` 대리자로 변환할 수 있습니다. 매개 변수가 하나이고 값을 반환하는 람다 식은 `Func<T, TResult>` 대리자로 변환할 수 있습니다. 다음 예제에서는 `x`라고 이름이 지정되고 `x` 제곱 값을 반환하는 매개 변수를 지정하는 람다 식 `x => x * x`는 대리자 형식의 변수에 할당됩니다.

```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

다음 예제에 표시된 대로 식 람다는 [식 트리](#) 형식으로 변환할 수도 있습니다.

```
System.Linq.Expressions.Expression<Func<int, int>> e = x => x * x;
Console.WriteLine(e);
// Output:
// x => (x * x)
```

대리자 형식이나 식 트리의 인스턴스가 필요한 코드에서 람다 식을 백그라운드에서 실행해야 하는 코드를 전달하는 [Task.Run\(Action\)](#) 메서드의 인수 등으로 사용할 수 있습니다. 다음 예제에 표시된 대로 C#에 LINQ를 작성할 때 람다 식을 사용할 수도 있습니다.

```
int[] numbers = { 2, 3, 4, 5 };
var squaredNumbers = numbers.Select(x => x * x);
Console.WriteLine(string.Join(" ", squaredNumbers));
// Output:
// 4 9 16 25
```

예를 들어 LINQ to Objects 및 LINQ to XML에서 메서드 기반 구문을 사용하여 `System.Linq.Enumerable` 클래스에서 `Enumerable.Select` 메서드를 호출하는 경우 매개 변수는 대리자 형식 `System.Func<T, TResult>`입니다. 예를 들어 LINQ to SQL에서 `System.Linq.Queryable` 클래스에서 `Queryable.Select` 메서드를 호출하는 경우 매개 변수 형식은 식 트리 형식 `Expression<Func<TSource, TResult>>`입니다. 두 경우 모두 동일한 람다 식을 사용하여 매개 변수 값을 지정할 수 있습니다. 그러면 두 `Select` 호출이 비슷하게 보일 수 있지만 실제로 람다 식을 통해 생성

되는 개체 형식은 다음과 같습니다.

## 식 람다

=> 연산자의 오른쪽에 식이 있는 람다 식을 식 람다라고 합니다. 식 람다는 식의 결과를 반환하며 기본 형식은 다음과 같습니다.

```
(input-parameters) => expression
```

식 람다의 본문은 메서드 호출로 구성될 수 있습니다. 하지만 SQL Server에서처럼 .NET CLR(공용 언어 런타임)의 컨텍스트 외부에서 평가되는 **식 트리**를 만드는 경우에는 람다 식에서 메서드 호출을 사용하면 안 됩니다. 메서드는 .NET CLR(공용 언어 런타임)의 컨텍스트 내에서만 의미가 있습니다.

## 문 람다

문 람다는 다음과 같이 중괄호 안에 문을 지정한다는 점을 제외하면 식 람다와 비슷합니다.

```
(input-parameters) => { <sequence-of-statements> }
```

문 람다의 본문에 지정할 수 있는 문의 개수에는 제한이 없지만 일반적으로 2-3개 정도만 지정합니다.

```
Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

문 람다를 사용하여 식 트리를 만들 수는 없습니다.

## 람다 식 입력 매개 변수

람다 식의 입력 매개 변수는 괄호로 묶습니다. 입력 매개 변수가 0개이면 다음과 같이 빈 괄호를 지정합니다.

```
Action line = () => Console.WriteLine();
```

람다 식에 입력 매개 변수가 하나만 있는 경우 괄호는 선택 사항입니다.

```
Func<double, double> cube = x => x * x * x;
```

두 개 이상의 입력 매개 변수는 쉼표로 구분합니다.

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

컴파일러가 입력 매개 변수의 형식을 유추할 수 없는 경우도 있습니다. 다음 예제와 같이 형식을 명시적으로 지정할 수 있습니다.

```
Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;
```

입력 매개 변수 형식은 모두 명시적이거나 암시적이어야 합니다. 그렇지 않으면 [CS0748](#) 컴파일러 오류가 발생합니다.

C# 9.0부터 [무시 항목](#)을 사용하여 람다 식에서 사용하지 않는 입력 매개 변수를 두 개 이상 지정할 수 있습니다.

```
Func<int, int, int> constant = (_, _) => 42;
```

람다 무시 항목 매개 변수는 람다 식을 사용하여 [이벤트 처리기를 제공](#)하는 경우에 유용할 수 있습니다.

#### NOTE

이전 버전과의 호환성을 위해 단일 입력 매개 변수만 `_`로 명명된 경우 람다 식 내에서 `_`가 해당 매개 변수의 이름으로 처리됩니다.

## 비동기 람다

`async` 및 `await` 키워드를 사용하여 비동기 처리를 통합하는 람다 식과 문을 쉽게 만들 수 있습니다. 예를 들어 다음 Windows Forms 예제에는 비동기 메서드 `ExampleMethodAsync`를 호출하고 기다리는 이벤트 처리기가 포함되어 있습니다.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += button1_Click;
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\n";
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

비동기 람다를 사용하여 동일한 이벤트 처리기를 추가할 수 있습니다. 이 처리기를 추가하려면 다음 예제에 표시된 것처럼 람다 매개 변수 목록에 `async` 한정자를 추가합니다.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event handler.\n";
        };
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

비동기 메서드를 만들고 사용하는 방법에 대한 자세한 내용은 [Async 및 Await를 사용한 비동기 프로그래밍](#)을

참조하세요.

## 람다 식 및 튜플

C# 7.0부터 C# 언어에서 [튜플](#)을 기본적으로 지원합니다. 람다 식에 인수로 튜플을 제공할 수 있으며 람다 식에서 튜플을 반환할 수도 있습니다. 경우에 따라 C# 컴파일러는 형식 유추를 사용하여 튜플 구성 요소의 형식을 확인할 수 있습니다.

쉼표로 구분된 해당 구성 요소 목록을 괄호로 묶어 튜플을 정의합니다. 다음 예제에서는 3개 구성 요소가 있는 튜플을 사용하여 숫자 시퀀스를 람다 식에 전달하고 각 값을 두 배로 늘린 후 곱하기의 결과가 포함된, 3개 구성 요소가 있는 튜플을 반환합니다.

```
Func<(int, int, int), (int, int, int)> doubleThem = ns => (2 * ns.Item1, 2 * ns.Item2, 2 * ns.Item3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
// Output:
// The set (2, 3, 4) doubled: (4, 6, 8)
```

일반적으로 튜플 필드의 이름은 `Item1`, `Item2` 등입니다. 그러나 다음 예제에서처럼 명명된 구성 요소가 있는 튜플을 정의할 수 있습니다.

```
Func<(int n1, int n2, int n3), (int, int, int)> doubleThem = ns => (2 * ns.n1, 2 * ns.n2, 2 * ns.n3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
```

C# 튜플에 관한 자세한 내용은 [튜플 형식](#)을 참조하세요.

## 표준 쿼리 연산자와 람다 식

다른 구현 중에 LINQ to Objects는 형식이 제네릭 대리자의 [Func<TResult>](#) 패밀리 중 하나인 입력 매개 변수를 사용합니다. 이러한 대리자는 형식 매개 변수를 사용하여 입력 매개 변수의 수와 형식 및 대리자의 반환 형식을 정의합니다. `Func` 대리자는 소스 데이터 집합에 있는 각 요소에 적용할 사용자 정의 식을 캡슐화하는 데 매우 유용합니다. 예를 들어 [Func<T, TResult>](#) 대리자 형식을 고려합니다.

```
public delegate TResult Func<in T, out TResult>(T arg)
```

이 경우 대리자를 `Func<int, bool>` 인스턴스로 인스턴스화할 수 있습니다. 여기서 `int`는 입력 매개 변수이고, `bool`은 반환 값입니다. 반환 값은 항상 마지막 형식 매개 변수에 지정됩니다. 예를 들어 `Func<int, string, bool>`은 두 입력 매개 변수 `int` 및 `string`과 반환 형식 `bool`을 사용하여 대리자를 정의합니다. 다음 `Func` 대리자를 호출하면 입력 매개 변수가 5인지 여부를 나타내는 부울 값이 반환됩니다.

```
Func<int, bool> equalsFive = x => x == 5;
bool result = equalsFive(4);
Console.WriteLine(result); // False
```

[Queryable](#) 형식에 정의되어 있는 표준 쿼리 연산자의 경우와 같이 인수 형식이 [Expression<TDelegate>](#)인 경우에도 람다 식을 사용할 수 있습니다. [Expression<TDelegate>](#) 인수를 지정하면 람다 식이 식 트리로 컴파일됩니다.

이 예제에서는 `Count` 표준 쿼리 연산자를 사용합니다.

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
Console.WriteLine($"There are {oddNumbers} odd numbers in {string.Join(" ", numbers)}");
```

컴파일러에서 입력 매개 변수의 형식을 유추하거나 사용자가 형식을 명시적으로 지정할 수 있습니다. 이 람다 식은 2로 나누었을 때 나머지가 1인 정수(`n`)의 수를 계산합니다.

다음 예제에서는 숫자 시퀀스에서 조건을 만족하지 않는 첫 번째 숫자가 9이기 때문에 `numbers` 배열에서 9 앞에 오는 모든 요소가 포함된 시퀀스를 생성합니다.

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstNumbersLessThanSix = numbers.TakeWhile(n => n < 6);
Console.WriteLine(string.Join(" ", firstNumbersLessThanSix));
// Output:
// 5 4 1 3
```

다음 예제에서는 입력 매개 변수를 괄호로 묶어 여러 개 지정합니다. 이 메서드는 값이 배열의 서수 위치보다 작은 숫자가 나타날 때까지 `numbers` 배열의 모든 요소를 반환합니다.

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);
Console.WriteLine(string.Join(" ", firstSmallNumbers));
// Output:
// 5 4
```

## 람다 식에서의 형식 유추

컴파일러에서는 람다 식 본문, 매개 변수 형식 및 C# 언어 사양에 설명되어 있는 기타 요소를 기준으로 형식을 유추할 수 있기 때문에 대부분의 경우에는 람다 식을 작성할 때 입력 매개 변수의 형식을 지정하지 않아도 됩니다. 대부분의 표준 쿼리 연산자에서 첫 번째 입력 형식은 소스 시퀀스 요소의 형식입니다.

`IEnumerable<Customer>` 를 쿼리할 경우 입력 변수가 `Customer` 개체로 유추됩니다. 이는 이 개체의 메서드와 속성에 액세스할 수 있음을 의미합니다.

```
customers.Where(c => c.City == "London");
```

람다 식의 형식 유추에 대한 일반적인 규칙은 다음과 같습니다.

- 람다 식과 대리자 형식에 포함된 매개 변수 수가 같아야 합니다.
- 람다 식의 각 입력 매개 변수는 해당되는 대리자 매개 변수로 암시적으로 변환될 수 있어야 합니다.
- 람다 식의 반환 값(있는 경우)은 대리자의 반환 형식으로 암시적으로 변환될 수 있어야 합니다.

공용 형식 시스템에는 “람다 식”이라는 개념이 기본적으로 포함되어 있지 않기 때문에 람다 식 자체에는 형식이 없습니다. 그러나 람다 식의 “형식”을 비공식적으로 언급해야 할 경우도 있는데 이 경우 형식은 대리자 형식 또는 람다 식이 변환되는 [Expression](#) 형식을 의미합니다.

## 람다 식에서 외부 변수 및 변수 범위 캡처

람다는 외부 변수를 참조할 수 있습니다. 이러한 변수는 람다 식을 정의하는 메서드 범위 내에 있거나 람다 식을 포함하는 형식 범위 내에 있는 변수입니다. 이러한 방식으로 캡처되는 변수는 변수가 범위를 벗어나 가비지 수집되는 경우에도 람다 식에 사용할 수 있도록 저장됩니다. 외부 변수는 명확하게 할당해야만 람다 식에 사용할 수 있습니다. 다음 예제에서는 이러한 규칙을 보여 줍니다.

```

public static class VariableScopeWithLambdas
{
    public class VariableCaptureGame
    {
        internal Action<int> updateCapturedLocalVariable;
        internal Func<int, bool> isEqualToCapturedLocalVariable;

        public void Run(int input)
        {
            int j = 0;

            updateCapturedLocalVariable = x =>
            {
                j = x;
                bool result = j > input;
                Console.WriteLine($"{{j}} is greater than {{input}}: {{result}}");
            };

            isEqualToCapturedLocalVariable = x => x == j;

            Console.WriteLine($"Local variable before lambda invocation: {{j}}");
            updateCapturedLocalVariable(10);
            Console.WriteLine($"Local variable after lambda invocation: {{j}}");
        }
    }

    public static void Main()
    {
        var game = new VariableCaptureGame();

        int gameInput = 5;
        game.Run(gameInput);

        int jTry = 10;
        bool result = game.isEqualToCapturedLocalVariable(jTry);
        Console.WriteLine($"Captured local variable is equal to {{jTry}}: {{result}}");

        int anotherJ = 3;
        game.updateCapturedLocalVariable(anotherJ);

        bool equalToAnother = game.isEqualToCapturedLocalVariable(anotherJ);
        Console.WriteLine($"Another lambda observes a new value of captured variable: {{equalToAnother}}");
    }
}

```

람다 식의 변수 범위에는 다음과 같은 규칙이 적용됩니다.

- 캡처된 변수는 해당 변수를 참조하는 대리자가 가비지 수집 대상이 될 때까지 가비지 수집되지 않습니다.
- 람다 식에 사용된 변수는 바깥쪽 메서드에 표시되지 않습니다.
- 람다 식은 바깥쪽 메서드에서 `in`, `ref` 또는 `out` 매개 변수를 직접 캡처할 수 없습니다.
- 람다 식의 `return` 문에 의해서는 바깥쪽 메서드가 반환되지 않습니다.
- 해당 점프 문의 대상이 람다 식 블록을 벗어나는 경우 람다 식에는 `goto`, `break` 또는 `continue` 문을 포함할 수 없습니다. 대상이 블록 내에 있는 경우 람다 식 블록 외부에 점프 문을 사용해도 오류가 발생합니다.

C# 9.0부터 람다 식에 `static` 한정자를 적용하여 의도치 않게 람다가 지역 변수 또는 인스턴스 상태를 캡처하

는 것을 방지할 수 있습니다.

```
Func<double, double> square = static x => x * x;
```

정적 람다는 바깥쪽 범위에서 지역 변수 또는 인스턴스 상태를 캡처할 수 없지만 정적 멤버와 상수 정의를 참조할 수 있습니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 익명 함수 식](#) 섹션을 참조하세요.

C# 9.0에 추가된 기능에 대한 자세한 내용은 다음 기능 제안 노트를 참조하세요.

- [람다 무시 항목 매개 변수](#)
- [정적 무명 함수](#)

## 참고 항목

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [LINQ\(Language-Integrated Query\)](#)
- [식 트리](#)
- [로컬 함수 및 람다 식](#)
- [Visual Studio 2008 C# 샘플\(LINQ 샘플 쿼리 파일 및 XQuery 프로그램 참조\)](#)

# + 및 += 연산자(C# 참조)

2020-11-02 • 5 minutes to read • [Edit Online](#)

+ 및 += 연산자에는 기본 제공 정수 및 부동 소수점 숫자 형식, 문자열 형식 및 대리자 형식이 지원됩니다.

산술 + 연산자에 대한 자세한 내용은 [산술 연산자](#) 문서의 [단항 더하기 및 빼기 연산자](#) 및 [더하기 연산자](#) + 섹션을 참조하세요.

## 문자열 연결

피연산자 중 하나 또는 둘 다가 [문자열](#) 형식이면 + 연산자는 피연산자의 문자열 표현을 연결합니다( `null` 의 문자열 표현은 빈 문자열임).

```
Console.WriteLine("Forgot" + "white space");
Console.WriteLine("Probably the oldest constant: " + Math.PI);
Console.WriteLine(null + "Nothing to add.");
// Output:
// Forgotwhite space
// Probably the oldest constant: 3.14159265358979
// Nothing to add.
```

C# 6부터 [문자열 보간](#)은 문자열 형식을 지정하는 더욱 편리한 방법을 제공합니다.

```
Console.WriteLine($"Probably the oldest constant: {Math.PI:F2}");
// Output:
// Probably the oldest constant: 3.14
```

## 대리자 조합

동일한 [대리자](#) 형식의 피연산자의 경우 + 연산자는 호출될 때 왼쪽 피연산자를 호출한 다음, 오른쪽 피연산자를 호출하는 새 대리자 인스턴스를 반환합니다. 피연산자 중 하나라도 `null` 이면 + 연산자는 다른 피연산자(`null` 일 수도 있음)의 값을 반환합니다. 다음 예제는 + 연산자를 사용하여 대리자를 결합하는 방법을 보여 줍니다.

```
Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
Action ab = a + b;
ab(); // output: ab
```

대리자 제거를 수행하려면 - 연산자를 사용합니다.

대리자 형식에 대한 자세한 내용은 [대리자](#)를 참조하세요.

## 더하기 할당 연산자 +=

다음과 같은 += 연산자를 사용하는 식의 경우

```
x += y
```

위의 식은 아래의 식과 동일합니다.

```
x = x + y
```

단, `x` 가 한 번만 계산됩니다.

다음 예제에서는 `+=` 연산자의 사용법을 보여 줍니다.

```
int i = 5;
i += 9;
Console.WriteLine(i);
// Output: 14

string story = "Start. ";
story += "End.";
Console.WriteLine(story);
// Output: Start. End.

Action printer = () => Console.Write("a");
printer(); // output: a

Console.WriteLine();
printer += () => Console.Write("b");
printer(); // output: ab
```

또한 [이벤트](#)를 구독할 때 `+=` 연산자를 사용하여 이벤트 처리기 메서드를 지정합니다. 자세한 내용은 [방법: 이벤트 구독 및 구독 취소](#)를 참조하세요.

## 연산자 오버로드 가능성

사용자 정의 형식은 `+` 연산자를 [오버로드](#) 할 수 있습니다. 이진 `+` 연산자가 오버로드되면 `+=` 연산자도 암시적으로 오버로드됩니다. 사용자 정의 형식에는 `+=` 연산자를 명시적으로 오버로드할 수 없습니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 [단항 더하기 연산자](#) 및 [더하기 연산자](#) 섹션을 참조하세요.

## 참조

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [여러 문자열 연결 방법](#)
- [이벤트](#)
- [산술 연산자](#)
- [- 및 -= 연산자](#)

# - 및 -= 연산자(C# 참조)

2020-11-02 • 7 minutes to read • [Edit Online](#)

- 및 -= 연산자에는 기본 제공 정수 및 부동 소수점 숫자 형식, 대리자 형식이 지원됩니다.

산술 - 연산자에 대한 자세한 내용은 [산술 연산자](#) 문서의 [단항 더하기 및 빼기 연산자](#) 및 [빼기 연산자](#) - 섹션을 참조하세요.

## 대리자 제거

동일한 대리자 형식의 피연산자에 대해 - 연산자는 다음과 같이 계산되는 대리자 인스턴스를 반환합니다.

- 두 피연산자가 모두 null이 아니고 오른쪽 피연산자의 호출 목록이 왼쪽 피연산자의 호출 목록에 적절한 연속 하위 목록인 경우, 이 연산의 결과는 왼쪽 피연산자의 호출 목록에서 오른쪽 피연산자의 항목을 제거하여 얻은 새로운 호출 목록입니다. 오른쪽 피연산자의 목록이 왼쪽 피연산자 목록의 여러 개의 연속 하위 목록과 일치하는 경우 가장 오른쪽의 일치하는 하위 목록만 제거됩니다. 제거로 인해 빈 목록이 생성되면 결과는 null입니다.

```
Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("b");

var abbaab = a + b + b + a + a + b;
abbaab(); // output: abbaab
Console.WriteLine();

var ab = a + b;
var abba = abbaab - ab;
abba(); // output: abba
Console.WriteLine();

var nihil = abbaab - abbaab;
Console.WriteLine(nihil is null); // output: True
```

- 오른쪽 피연산자의 호출 목록이 왼쪽 피연산자의 호출 목록의 적절한 연속 하위 목록이 아닌 경우, 해당 연산의 결과는 왼쪽 피연산자입니다. 예를 들어 멀티 캐스트 대리자의 일부가 아닌 대리자를 제거하면 아무 작업도 수행되지 않고 변경되지 않은 멀티 캐스트 대리자가 됩니다.

```

Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("b");

var abbaab = a + b + b + a + a + b;
var aba = a + b + a;

var first = abbaab - aba;
first(); // output: abbaab
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(abbaab, first)); // output: True

Action a2 = () => Console.WriteLine("a");
var changed = aba - a;
changed(); // output: ab
Console.WriteLine();
var unchanged = aba - a2;
unchanged(); // output: aba
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(aba, unchanged)); // output: True

```

또한 앞의 예제에서는 대리자 제거 중 대리자 인스턴스를 비교하는 것을 보여줍니다. 예를 들어 동일한 [람다식](#)의 평가에서 생성된 대리자는 동일하지 않습니다. 대리자 같음에 대한 자세한 내용은 [C# 언어 사용의 대리자 같음 연산자](#) 섹션을 참조하세요.

- 원쪽 피연산자가 `null`이면, 연산 결과는 `null`입니다. 오른쪽 피연산자가 `null`이면, 연산 결과는 원쪽 피연산자입니다.

```

Action a = () => Console.WriteLine("a");

var nothing = null - a;
Console.WriteLine(nothing is null); // output: True

var first = a - null;
a(); // output: a
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(first, a)); // output: True

```

대리자를 결합하려면 [+ 연산자](#)를 사용합니다.

대리자 형식에 대한 자세한 내용은 [대리자](#)를 참조하세요.

## 빼기 할당 연산자 `-=`

다음과 같은 `-=` 연산자를 사용하는 식의 경우

```
x -= y
```

위의 식은 아래의 식과 동일합니다.

```
x = x - y
```

단, `x`가 한 번만 계산됩니다.

다음 예제에서는 `-=` 연산자의 사용법을 보여 줍니다.

```
int i = 5;
i -= 9;
Console.WriteLine(i);
// Output: -4

Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
var printer = a + b + a;
printer(); // output: aba

Console.WriteLine();
printer -= a;
printer(); // output: ab
```

또한 [이벤트](#)에서 구독 취소할 때 `-=` 연산자를 사용하여 제거할 이벤트 처리기 메서드를 지정합니다. 자세한 내용은 [이벤트를 구독 및 구독 취소하는 방법](#)을 참조하세요.

## 연산자 오버로드 가능성

사용자 정의 형식은 `-` 연산자를 [오버로드](#)할 수 있습니다. 이진 `-` 연산자가 오버로드되면 `-=` 연산자도 암시적으로 오버로드됩니다. 사용자 정의 형식에는 `-=` 연산자를 명시적으로 오버로드할 수 없습니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 [단항 빼기 연산자](#) 및 [빼기 연산자](#) 섹션을 참조하세요.

## 참고 항목

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [이벤트](#)
- [산술 연산자](#)
- [+ 및 += 연산자](#)

# ?: 연산자(C# 참조)

2020-11-02 • 7 minutes to read • [Edit Online](#)

3개로 구성된 조건부 연산자라고도 하는 조건부 연산자 `?:`은 부울 식을 계산하고 부울 식이 `true` 또는 `false`으로 계산되는지에 따라 두 식 중 하나의 계산 결과를 반환합니다.

조건 연산자의 구문은 다음과 같습니다.

```
condition ? consequent : alternative
```

`condition` 식은 `true` 또는 `false`로 계산되어야 합니다. `condition`이 `true`로 계산되면 `consequent` 식이 계산되고 해당 결과가 연산 결과가 됩니다. `condition`이 `false`로 계산되면 `alternative` 식이 계산되고 해당 결과가 연산 결과가 됩니다. `consequent` 또는 `alternative`만 계산됩니다.

C# 9.0부터 조건식은 대상으로 유형화됩니다. 즉, 조건식의 대상 유형을 알고 있는 경우 `consequent` 및 `alternative`의 형식은 다음 예제와 같이 대상 유형으로 암시적으로 변환 가능해야 합니다.

```
var rand = new Random();
var condition = rand.NextDouble() > 0.5;

int? x = condition ? 12 : null;

IEnumerable<int> xs = x is null ? new List<int>() { 0, 1 } : new int[] { 2, 3 };
```

조건식의 대상 유형을 모르는 경우(예: `var` 키워드 사용) 또는 C# 8.0 이하에서 `consequent` 및 `alternative`의 형식이 같아야 하거나 한 형식에서 다른 형식으로 암시적인 변환이 있어야 합니다.

```
var rand = new Random();
var condition = rand.NextDouble() > 0.5;

var x = condition ? 12 : (int?)null;
```

조건부 연산자는 오른쪽 결합성입니다. 즉, 다음 형식의 식을 가정해 보세요.

```
a ? b : c ? d : e
```

이 식은 다음과 같이 계산됩니다.

```
a ? b : (c ? d : e)
```

## TIP

다음과 같은 니모닉 디바이스를 사용하여 조건부 연산자의 평가 방식을 기억할 수 있습니다.

```
is this condition true ? yes : no
```

다음 예제에서는 조건부 연산자의 사용법을 보여 줍니다.

```
double sinc(double x) => x != 0.0 ? Math.Sin(x) / x : 1;

Console.WriteLine(sinc(0.1));
Console.WriteLine(sinc(0.0));
// Output:
// 0.998334166468282
// 1
```

## 조건부 ref 식

C# 7.2부터 조건부 ref 식으로 **ref 지역** 또는 **ref readonly 지역** 변수를 조건부로 할당할 수 있습니다. 조건부 ref 식을 **참조 반환 값** 또는 **ref 메서드 인수**로 사용할 수도 있습니다.

조건부 ref 식의 구문은 다음과 같습니다.

```
condition ? ref consequent : ref alternative
```

원래 조건부 연산자와 마찬가지로 조건부 ref 식은 두 식 중 하나(**consequent** 또는 **alternative**)만 계산합니다.

조건부 ref 식의 경우 **consequent** 및 **alternative**의 형식이 동일해야 합니다. 조건부 ref 식은 대상으로 유형화되지 않습니다.

다음 예제에서는 조건부 ref 식의 사용법을 보여 줍니다.

```
var smallArray = new int[] { 1, 2, 3, 4, 5 };
var largeArray = new int[] { 10, 20, 30, 40, 50 };

int index = 7;
ref int refValue = ref ((index < 5) ? ref smallArray[index] : ref largeArray[index - 5]);
refValue = 0;

index = 2;
((index < 5) ? ref smallArray[index] : ref largeArray[index - 5]) = 100;

Console.WriteLine(string.Join(" ", smallArray));
Console.WriteLine(string.Join(" ", largeArray));
// Output:
// 1 2 100 4 5
// 10 20 0 40 50
```

## 조건부 연산자 및 **if..else** 문

**if..else** 문보다 조건부 연산자를 사용하면 조건부로 값을 컴퓨팅해야 하는 경우 코드가 보다 간결해질 수 있습니다. 다음 예제에서는 정수를 음수 또는 음수가 아닌 값으로 분류하는 두 가지 방법을 보여 줍니다.

```
int input = new Random().Next(-5, 5);

string classify;
if (input >= 0)
{
    classify = "nonnegative";
}
else
{
    classify = "negative";
}

classify = (input >= 0) ? "nonnegative" : "negative";
```

## 연산자 오버로드 가능성

사용자 정의 형식으로 조건부 연산자를 오버로드할 수 없습니다.

## C# 언어 사양

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

C# 7.2 이상에 추가된 기능에 대한 자세한 내용은 다음 기능 제안 노트를 참조하세요.

- [조건부 ref 식\(C# 7.2\)](#)
- [대상으로 유형화된 조건식\(C# 9.0\)](#)

## 참고 항목

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [if-else 문](#)
- [?. 및 ?\[\] 연산자](#)
- [및 ??= 연산자](#)
- [ref 키워드](#)

# !(null-forgiving) 연산자(C# 참조)

2021-02-18 • 6 minutes to read • [Edit Online](#)

C# 8.0 이상에서 사용할 수 있는 단항 후위 `!` 연산자는 null-forgiving 또는 null-suppression 연산자입니다. [null 허용 주석 컨텍스트](#)가 활성화된 경우에는 참조 형식의 `x` 식이 `null : x!` 가 아님을 선언하는 데 null-forgiving 연산자를 사용할 수 있습니다. 단항 접두사 `!` 연산자는 [논리 부정 연산자](#)입니다.

null-forgiving 연산자는 런타임에 영향을 주지 않습니다. 식의 null 상태를 변경하여 컴파일러의 정적 흐름 분석에만 영향을 줍니다. 런타임에서 `x!` 식은 기본 식 `x`의 결과로 계산됩니다.

nullable 참조 형식 기능에 대한 자세한 내용은 [nullable 참조 형식을 참조하세요](#).

## 예

null-forgiving 연산자의 사용 사례 중 하나는 인수 유효성 검사 논리를 테스트하는 것입니다. 예를 들어 다음 클래스를 예로 들어 볼 수 있습니다.

```
#nullable enable
public class Person
{
    public Person(string name) => Name = name ?? throw new ArgumentNullException(nameof(name));

    public string Name { get; }
}
```

[MSTest 테스트 프레임워크](#)를 사용하여 생성자에서 유효성 검사 논리에 대해 다음 테스트를 만들 수 있습니다.

```
[TestMethod, ExpectedException(typeof(ArgumentNullException))]
public void NullNameShouldThrowTest()
{
    var person = new Person(null!);
}
```

null-forgiving 연산자가 없으면 컴파일러가 이전 코드에 대해 다음 경고를 생성합니다.

`Warning CS8625: Cannot convert null literal to non-nullable reference type.` null-forgiving 연산자를 사용하여 `null`이 전달될 것이고 경고를 표시하지 않아야 함을 컴파일러에 알립니다.

식이 `null`이 될 수 없지만 컴파일러가 이를 인식할 수 없음이 확실할 경우에는 null-forgiving 연산자를 사용할 수도 있습니다. 다음 예에서는 `IsValid` 메서드가 `true`를 반환하는 경우 해당 인수는 `null`이 아니므로 안전하게 역참조할 수 있습니다.

```

public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p!.Name}");
    }
}

public static bool IsValid(Person? person)
{
    return person != null && !string.IsNullOrEmpty(person.Name);
}

```

null-forgiving 연산자가 없으면 컴파일러가 `p.Name` 코드에 대해 다음 경고를 생성합니다.

`Warning CS8602: Dereference of a possibly null reference`.

`IsValid` 메서드를 수정할 수 있는 경우 `NotNullWhen` 특성을 사용하여 메서드가 `true`를 반환할 때 `IsValid` 메서드의 인수는 `null`일 수 없다는 것을 컴파일러에 알릴 수 있습니다.

```

public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p.Name}");
    }
}

public static bool IsValid([NotNullWhen(true)] Person? person)
{
    return person != null && !string.IsNullOrEmpty(person.Name);
}

```

앞의 예제에서는 컴파일러가 `if` 문 내에서 `p` 가 `null`일 수 없다는 것을 확인할 수 있는 충분한 정보를 제공하기 때문에 null-forgiving 연산자를 사용할 필요가 없습니다. 변수의 null 상태에 대한 추가 정보를 제공하는데 사용할 수 있는 특성에 대한 자세한 내용은 [null 예상을 정의하는 특성을 포함한 API 업그레이드](#)를 참조하세요.

## C# 언어 사양

자세한 내용은 [Nullable 참조 형식 사양 초안](#)의 [null-forgiving 연산자](#) 섹션을 참조하세요.

## 참조

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [자습서: nullable 참조 형식을 사용하여 디자인](#)

# ?? 및 ??= 연산자(C# 참조)

2020-11-02 • 6 minutes to read • [Edit Online](#)

null 병합 연산자 ?? 는 null 이 아닌 경우 왼쪽 피연산자의 값을 반환합니다. 그렇지 않으면 오른쪽 피연자를 평가하고 그 결과를 반환합니다. 왼쪽 피연산자가 null이 아닌 것으로 평가되면 ?? 연산자는 오른쪽 피연산자를 평가하지 않습니다.

C# 8.0 이상에서 사용할 수 있는 null 병합 할당 연산자 ??= 는 왼쪽 피연산자가 null로 계산되는 경우에만 오른쪽 피연산자의 값을 왼쪽 피연산자에 대입합니다. 왼쪽 피연산자가 null이 아닌 것으로 평가되면 ??= 연산자는 오른쪽 피연산자를 평가하지 않습니다.

```
List<int> numbers = null;
int? a = null;

(numbers ??= new List<int>()).Add(5);
Console.WriteLine(string.Join(" ", numbers)); // output: 5

numbers.Add(a ??= 0);
Console.WriteLine(string.Join(" ", numbers)); // output: 5 0
Console.WriteLine(a); // output: 0
```

??= 연산자의 왼쪽 피연산자는 변수, 속성 또는 인덱서 요소여야 합니다.

C# 7.3 이전 버전에서 ?? 연산자의 왼쪽 피연산자 형식은 참조 형식 또는 Nullable 값 형식이어야 합니다. C# 8.0부터 이 요구 사항이 다음과 같이 바뀝니다. ?? 및 ??= 연산자의 왼쪽 피연산자 형식은 null을 허용하지 않는 값 형식일 수 없습니다. 특히 C# 8.0부터 비제한 형식 매개 변수와 함께 null 병합 연산자를 사용할 수 있습니다.

```
private static void Display<T>(T a, T backup)
{
    Console.WriteLine(a ?? backup);
}
```

null 병합 연산자는 오른쪽 결합입니다. 즉, 양식의 식이

```
a ?? b ?? c
d ??= e ??= f
```

다음과 같이 계산됩니다.

```
a ?? (b ?? c)
d ??= (e ??= f)
```

예

?? 및 ??= 연산자는 다음과 같은 시나리오에서 유용할 수 있습니다.

- null 병합 연산자 ?. 및 ?[] 가 있는 식에서 ?? 연산자를 사용하여 null 조건부 연산을 사용한 식의 결과가 null인 경우 평가하는 대체 식을 제공할 수 있습니다.

```
double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)
{
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;
}

var sum = SumNumbers(null, 0);
Console.WriteLine(sum); // output: NaN
```

- nullable 값 형식을 사용하고 기본값 유형의 값을 제공해야 할 때 `??` 연산자를 사용하여 nullable 값이 `null`인 경우 제공할 값을 지정합니다.

```
int? a = null;
int b = a ?? -1;
Console.WriteLine(b); // output: -1
```

nullable 형식 값이 `null`일 때 사용될 값이 기본 값 형식의 기본 값이어야 하는 경우 `Nullable<T>.GetValueOrDefault()` 메서드를 사용합니다.

- C# 7.0부터 `??` 연산자의 오른쪽 피연산자로 `throw` 식을 사용하여 인수 확인 코드를 보다 간결하게 만들 수 있습니다.

```
public string Name
{
    get => name;
    set => name = value ?? throw new ArgumentNullException(nameof(value), "Name cannot be null");
}
```

앞의 예제에서는 [식 본문 멤버](#)를 사용하여 속성을 정의하는 방법도 보여줍니다.

- C# 8.0부터 `??=` 연산자를 사용하여 다음 양식의 코드를

```
if (variable is null)
{
    variable = expression;
}
```

다음 코드와 바꿉니다.

```
variable ??= expression;
```

## 연산자 오버로드 가능성

`??` 및 `??=` 연산자는 오버로드할 수 없습니다.

## C# 언어 사양

`??` 연산자에 대한 자세한 내용은 [C# 언어 사양](#)의 [null 병합 연산자](#) 섹션을 참조하세요.

`??=` 연산자에 대한 자세한 내용은 [기능 제한 노트](#)를 참조하세요.

## 참고 항목

- [C# 참조](#)
- [C# 연산자 및 식](#)

- ?. 및 ?[] 연산자
- ?: 연산자

# => 연산자(C# 참조)

2020-11-02 • 5 minutes to read • [Edit Online](#)

=> 토큰은 [람다 연산자](#)와 [식 본문 정의](#)의 멤버 이름과 멤버 구현의 구분자라는 두 가지 형식으로 지원됩니다.

## 람다 연산자

람다 [식](#)에서 람다 연산자 => 은 왼쪽의 입력 매개 변수를 오른쪽의 람다 본문과 구분합니다.

다음 예제에서는 메서드 구문이 포함된 [LINQ](#) 기능을 사용하여 람다 식의 사용법을 보여줍니다.

```
string[] words = { "bot", "apple", "apricot" };
int minimalLength = words
    .Where(w => w.StartsWith("a"))
    .Min(w => w.Length);
Console.WriteLine(minimalLength); // output: 5

int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (interim, next) => interim * next);
Console.WriteLine(product); // output: 280
```

람다 식의 입력 매개 변수는 컴파일 시 강력한 형식을 지정합니다. 위의 예제와 같이 컴파일러가 입력 매개 변수의 형식을 추론하는 경우, 형식 선언을 생략할 수 있습니다. 입력 매개 변수의 형식을 지정해야 하는 경우 다음 예제와 같이 각 매개 변수에 대해 입력매개 변수를 지정해야 합니다.

```
int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (int interim, int next) => interim * next);
Console.WriteLine(product); // output: 280
```

다음 예제에서는 입력 매개 변수 없이 람다 식을 정의하는 방법을 보여줍니다.

```
Func<string> greet = () => "Hello, World!";
Console.WriteLine(greet());
```

자세한 내용은 [람다 식](#)을 참조하세요.

## 식 본문 정의

식 본문 정의의 일반 구문은 다음과 같습니다.

```
member => expression;
```

여기서 `expression` 은(는) 유효한 식입니다. `expression`의 반환 형식은 구성원의 반환 형식으로 암시적으로 변환할 수 있어야 합니다. 멤버의 반환 형식이 `void`거나 구성원이 생성자, 종료자, 속성 또는 인덱서 `set` 접근자인 경우 `expression`은 '[식 문](#)'이어야 합니다. 식의 결과는 삭제되므로 해당 식의 반환 형식은 어떤 형식도 될 수 있습니다.

다음 예제에서는 `Person.ToString` 메서드에 대한 식 본문 정의를 보여줍니다.

```
public override string ToString() => $"{fname} {lname}".Trim();
```

다음과 같은 메서드 정의의 약식 버전입니다.

```
public override string ToString()
{
    return $"{fname} {lname}".Trim();
}
```

메서드, 연산자 및 읽기 전용 속성에 대한 식 본문 정의는 C# 6부터 지원됩니다. 생성자, 종료자, 속성 및 인덱서 접근자에 대한 식 본문 정의는 C# 7.0부터 지원됩니다.

자세한 내용은 [식 본문 멤버](#)를 참조하세요.

## 연산자 오버로드 가능성

=> 연산자를 오버로드할 수 없습니다.

## C# 언어 사양

람다 연산자에 대한 자세한 내용은 [C# 언어 사양](#)의 [익명 함수 식](#) 섹션을 참조하세요.

## 참조

- [C# 참조](#)
- [C# 연산자 및 식](#)

# :: 연산자(C# 참조)

2020-11-02 • 3 minutes to read • [Edit Online](#)

네임스페이스 별칭 한정자( `::` )를 사용하여 별칭이 지정된 네임스페이스의 구성원에 액세스합니다. 두 식별자 사이에 `::` 한정사만 사용할 수 있습니다. 왼쪽 식별자는 다음 별칭 중 하나를 사용할 수 있습니다.

- **별칭 지시문을 사용하여 만든 네임스페이스 별칭:**

```
using forwinforms = System.Drawing;
using forwpf = System.Windows;

public class Converters
{
    public static forwpf::Point Convert(forwinforms::Point point) => new forwpf::Point(point.X,
    point.Y);
}
```

- **extern 별칭**

- 전역 네임스페이스 별칭인 `global` 별칭. 전역 네임스페이스는 명명된 네임스페이스 내에 선언되지 않은 네임스페이스와 형식을 포함하는 네임스페이스입니다. `::` 한정자와 함께 사용하는 경우 `global` 별칭은 사용자 정의 `global` 네임스페이스 별칭이 있더라도 항상 전역 네임스페이스를 참조합니다.

다음 예제에서는 `global` 별칭을 사용하여 전역 네임스페이스의 구성원인 .NET `System` 네임스페이스에 액세스합니다. `global` 별칭을 사용하지 않으면 `MyCompany.MyProduct` 네임스페이스의 구성원인 사용자 정의 `System` 네임스페이스에 액세스할 수 있습니다.

```
namespace MyCompany.MyProduct.System
{
    class Program
    {
        static void Main() => global::System.Console.WriteLine("Using global alias");
    }

    class Console
    {
        string Suggestion => "Consider renaming this class";
    }
}
```

## NOTE

`global` 키워드는 `::` 한정자의 왼쪽 식별자인 경우에만 전역 네임스페이스 별칭입니다.

. 토큰을 사용하여 별칭이 지정된 네임스페이스의 멤버에 액세스할 수도 있습니다. 그러나 . 토큰은 형식 멤버에 액세스하는 데도 사용됩니다. `::` 한정자는 이름이 같은 형식 또는 네임스페이스가 있는 경우에도 해당 왼쪽 식별자가 항상 네임스페이스 별칭을 참조하는지 확인합니다.

## C# 언어 사양

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

## 참고 항목

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [네임스페이스 사용](#)

# await 연산자(C# 참조)

2020-11-02 • 9 minutes to read • [Edit Online](#)

`await` 연산자는 피연산자가 나타내는 비동기 작업이 완료될 때까지 바깥쪽 [비동기](#) 메서드의 평가를 일시 중단합니다. 비동기 작업이 완료되면 `await` 연산자는 작업 결과를 반환합니다(있는 경우). 이미 완료된 작업을 나타내는 피연산자에 `await` 연산자가 적용되면 바깥쪽 메서드를 일시 중단하지 않고 작업 결과를 즉시 반환합니다. `await` 연산자는 비동기 메서드를 평가하는 스레드를 차단하지 않습니다. `await` 연산자가 바깥쪽 비동기 메서드를 일시 중단하면 제어가 메서드 호출자에게 반환됩니다.

다음 예제에서 `HttpClient.GetByteArrayAsync` 메서드는 완료 시 바이트 배열을 생성하는 비동기 작업을 나타내는 `Task<byte[]>` 인스턴스를 반환합니다. 작업이 완료될 때까지 `await` 연산자는 `DownloadDocs MainPageAsync` 메서드를 일시 중단합니다. `DownloadDocs MainPageAsync` 가 일시 중단되면 제어는 `DownloadDocs MainPageAsync` 의 호출자인 `Main` 메서드에 반환됩니다. `Main` 메서드는 `DownloadDocs MainPageAsync` 메서드가 수행한 비동기 작업의 결과가 필요할 때까지 실행됩니다. `GetByteArrayAsync`가 모든 바이트를 가져오면 나머지 `DownloadDocs MainPageAsync` 메서드가 평가됩니다. 그 후에는 나머지 `Main` 메서드가 평가됩니다.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

public class AwaitOperator
{
    public static async Task Main()
    {
        Task<int> downloading = DownloadDocs MainPageAsync();
        Console.WriteLine($"{nameof(Main)}: Launched downloading.");

        int bytesLoaded = await downloading;
        Console.WriteLine($"{nameof(Main)}: Downloaded {bytesLoaded} bytes.");
    }

    private static async Task<int> DownloadDocs MainPageAsync()
    {
        Console.WriteLine($"{nameof(DownloadDocs MainPageAsync)}: About to start downloading.");

        var client = new HttpClient();
        byte[] content = await client.GetByteArrayAsync("https://docs.microsoft.com/en-us/");

        Console.WriteLine($"{nameof(DownloadDocs MainPageAsync)}: Finished downloading.");
        return content.Length;
    }
}

// Output similar to:
// DownloadDocs MainPageAsync: About to start downloading.
// Main: Launched downloading.
// DownloadDocs MainPageAsync: Finished downloading.
// Main: Downloaded 27700 bytes.
```

앞의 예제에서는 C# 7.1부터 가능한 [비동기 Main 메서드](#)를 사용합니다. 자세한 내용은 [Main 메서드 섹션의 await 연산자](#)를 참조하세요.

## NOTE

비동기 프로그래밍에 대한 소개는 [async 및 await를 사용한 비동기 프로그래밍](#)을 참조하세요. `async` 및 `await`를 사용하는 비동기 프로그래밍은 [작업 기반 비동기 패턴](#)을 따릅니다.

`async` 키워드로 수정된 메서드, 람다 식 또는 무명 메서드에는 `await` 연산자만 사용할 수 있습니다. 비동기 메서드 내의 동기 함수 본문, `lock` 문 블록 및 `unsafe` 컨텍스트에서는 `await` 연산자를 사용할 수 없습니다.

`await` 연산자의 피연산자는 일반적으로 .NET 형식인 `Task`, `Task<TResult>`, `ValueTask` 또는 `ValueTask<TResult>` 중 하나에 해당합니다. 그러나 대기 가능한 모든 식은 `await` 연산자의 피연산자일 수 있습니다. 자세한 내용은 [C# 언어 사양의 대기 가능 식](#) 섹션을 참조하세요.

`t` 식의 형식이 `Task<TResult>` 또는 `ValueTask<TResult>`이면 `await t` 식의 형식은 `TResult`입니다. `t` 형식이 `Task` 또는 `ValueTask`이면 `await t` 형식은 `void`입니다. 두 경우 모두 `t` 가 예외를 throw하면 `await t`는 예외를 다시 throw합니다. 예외 처리에 대한 자세한 내용은 `try-catch` 문 문서에서 [비동기 메서드의 예외](#) 섹션을 참조하세요.

`async` 및 `await` 키워드는 C# 5 이상 버전에서 사용할 수 있습니다.

## 비동기 스트림 및 삭제 가능한 항목

C# 8.0부터 비동기 스트림 및 삭제 가능한 항목을 사용할 수 있습니다.

`await foreach` 문을 사용하여 비동기 데이터 스트림을 사용합니다. 자세한 내용은 [foreach 문](#) 문서 및 [C# 8.0의 새로운 기능](#) 문서의 [비동기 스트림](#) 섹션을 참조하세요.

`await using` 문을 사용하여 삭제 가능한 개체, 즉 `IAsyncDisposable` 인터페이스를 구현하는 형식의 개체를 비동기적으로 사용합니다. 자세한 내용은 [DisposeAsync 메서드 구현](#) 문서의 [삭제 가능한 비동기 항목 사용](#) 섹션을 참조하세요.

## Main 메서드의 await 연산자

C# 7.1부터 애플리케이션 진입점인 `Main` 메서드는 `Task` 또는 `Task<int>`를 반환하여 해당 본문에서 `await` 연산자를 사용할 수 있습니다. 이전 C# 버전에서는 `Main` 메서드가 비동기 작업이 완료될 때까지 대기하는지 확인하기 위해 해당 비동기 메서드에서 반환되는 `Task<TResult>` 인스턴스의 `Task<TResult>.Result` 속성 값을 검색할 수 있습니다. 값을 생성하지 않는 비동기 작업의 경우 `Task.Wait` 메서드를 호출할 수 있습니다. 언어 버전을 선택하는 방법에 대한 자세한 내용은 [C# 언어 버전](#)을 참조하세요.

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [async](#)
- [작업 비동기 프로그래밍 모델](#)
- [비동기 프로그래밍](#)
- [비동기에 대한 자세한 설명](#)
- [연습: async 및 await를 사용하여 웹에 액세스](#)
- [자습서: C# 8.0 및 .NET Core 3.0을 사용하여 비동기 스트림 생성 및 사용](#)

# 기본값 식(C# 참조)

2020-11-02 • 3 minutes to read • [Edit Online](#)

기본값 식은 형식의 [기본값](#)을 생성합니다. 기본값 식에는 두 가지가 있습니다. [기본 연산자](#) 호출 및 [기본 리터럴](#).

또한 `default` 키워드를 `switch` 문 내의 기본 사례 레이블로 사용할 수 있습니다.

## default 연산자

`default` 연산자의 인수는 다음 예제와 같이 형식 또는 형식 매개 변수의 이름이어야 합니다.

```
Console.WriteLine(default(int)); // output: 0
Console.WriteLine(default(object) is null); // output: True

void DisplayDefaultOf<T>()
{
    var val = default(T);
    Console.WriteLine($"Default value of {typeof(T)} is {(val == null ? "null" : val.ToString())}.");
}

DisplayDefaultOf<int?>();
DisplayDefaultOf<System.Numerics.Complex>();
DisplayDefaultOf<System.Collections.Generic.List<int>>();
// Output:
// Default value of System.Nullable`1[System.Int32] is null.
// Default value of System.Numerics.Complex is (0, 0).
// Default value of System.Collections.Generic.List`1[System.Int32] is null.
```

## 기본 리터럴

C# 7.1부터 `default` 리터럴을 사용하여 컴파일러가 식 형식을 유추할 수 있는 경우 형식의 기본값을 생성할 수 있습니다. `default` 리터럴 식은 `T`가 추론된 형식은 `default(T)` 식과 동일한 값을 생성합니다. 다음과 같은 경우에 `default` 리터럴 사용할 수 있습니다.

- 변수의 할당 또는 초기화에서
- [선택적 메서드 매개 변수](#)에 대한 기본값 선언에서
- 인수 값을 제공하기 위한 메서드 호출에서
- `return` 문에서 또는 [식 본문 멤버](#)의 식으로

다음 예제에서는 `default` 리터럴의 사용법을 보여 줍니다.

```
T[] InitializeArray<T>(int length, T initialValue = default)
{
    if (length < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(length), "Array length must be nonnegative.");
    }

    var array = new T[length];
    for (var i = 0; i < length; i++)
    {
        array[i] = initialValue;
    }
    return array;
}

void Display<T>(T[] values) => Console.WriteLine($"[ {string.Join(", ", values)} ]");

Display(InitializeArray<int>(3)); // output: [ 0, 0, 0 ]
Display(InitializeArray<bool>(4, default)); // output: [ False, False, False, False ]

System.Numerics.Complex fillValue = default;
Display(InitializeArray(3, fillValue)); // output: [ (0, 0), (0, 0), (0, 0) ]
```

## C# 언어 사양

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

`default` 리터럴에 대한 자세한 내용은 [기능 제한 노트](#)를 참조하세요.

## 참고 항목

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [C# 형식의 기본값](#)
- [.NET의 제네릭](#)

# delegate 연산자(C# 참조)

2020-11-02 • 4 minutes to read • [Edit Online](#)

`delegate` 연산자는 대리자 형식으로 변환될 수 있는 무명 메서드를 만듭니다.

```
Func<int, int, int> sum = delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(3, 4)); // output: 7
```

## NOTE

C# 3으로 시작하는 람다 식은 익명 함수를 만드는 더 간결하고 이해하기 쉬운 방법을 제공합니다. => [연산자](#)를 사용하여 람다 식을 구성합니다.

```
Func<int, int, int> sum = (a, b) => a + b;
Console.WriteLine(sum(3, 4)); // output: 7
```

람다 식의 기능(예: 외부 변수 캡처)에 대한 자세한 내용은 [람다 식](#)을 참조하세요.

`delegate` 연산자를 사용하는 경우 매개 변수 목록을 생략할 수 있습니다. 이렇게 하면 다음 예제와 같이 매개 변수 목록을 사용하여 생성된 무명 메서드를 대리자 형식으로 변환할 수 있습니다.

```
Action greet = delegate { Console.WriteLine("Hello!"); };
greet();

Action<int, double> introduce = delegate { Console.WriteLine("This is world!"); };
introduce(42, 2.7);

// Output:
// Hello!
// This is world!
```

이 기능은 람다 식에서 지원되지 않는 무명 메서드의 유일한 기능입니다. 다른 모든 경우 인라인 코드를 작성하는데 람다 식이 선호됩니다.

C# 9.0부터 [무시 항목](#)을 사용하여 무명 메서드에서 사용하지 않는 입력 매개 변수를 두 개 이상 지정할 수 있습니다.

```
Func<int, int, int> constant = delegate (int _, int _) { return 42; };
Console.WriteLine(constant(3, 4)); // output: 42
```

이전 버전과의 호환성을 위해, 단일 매개 변수만 `_`로 명명된 경우 `_`가 무명 메서드 내에서 해당 매개 변수의 이름으로 처리됩니다.

또한 C# 9.0부터 무명 메서드 선언에 `static` 한정자를 사용할 수 있습니다.

```
Func<int, int, int> sum = static delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(10, 4)); // output: 14
```

정적 무명 메서드는 바깥쪽 범위에서 지역 변수 또는 인스턴스 상태를 캡처할 수 없습니다.

`delegate` 키워드를 사용하여 [대리자 형식](#)을 선언할 수도 있습니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양의 익명 함수 식 섹션](#)을 참조하세요.

## 참고 항목

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [=> 연산자](#)

# nameof 식(C# 참조)

2021-02-18 • 2 minutes to read • [Edit Online](#)

nameof 식은 변수, 형식 또는 멤버의 이름을 문자열 상수로 가져옵니다.

```
Console.WriteLine(nameof(System.Collections.Generic)); // output: Generic
Console.WriteLine(nameof(List<int>)); // output: List
Console.WriteLine(nameof(List<int>.Count)); // output: Count
Console.WriteLine(nameof(List<int>.Add)); // output: Add

var numbers = new List<int> { 1, 2, 3 };
Console.WriteLine(nameof(numbers)); // output: numbers
Console.WriteLine(nameof(numbers.Count)); // output: Count
Console.WriteLine(nameof(numbers.Add)); // output: Add
```

이전 예제와 같이 형식 및 네임스페이스의 경우 생성되는 이름은 정규화된 이름이 아닙니다.

verbatim 식별자의 경우 다음 예제와 같이 @ 문자는 이름의 일부가 아닙니다.

```
var @new = 5;
Console.WriteLine(nameof(@new)); // output: new
```

nameof 식은 컴파일 시간에 계산되며 런타임에는 영향을 주지 않습니다.

nameof 식을 사용하여 인수 검사 코드를 더 쉽게 유지 관리할 수 있습니다.

```
public string Name
{
    get => name;
    set => name = value ?? throw new ArgumentNullException(nameof(value), $"{nameof(Name)} cannot be null");
}
```

nameof 식은 C# 6 이상에서 사용할 수 있습니다.

## C# 언어 사양

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

## 참조

- [C# 참조](#)
- [C# 연산자 및 식](#)

# new 연산자(C# 참조)

2020-11-02 • 6 minutes to read • [Edit Online](#)

`new` 연산자는 새 유형의 인스턴스를 만듭니다.

`new` 키워드를 [멤버 선언 한정자](#) 또는 [제네릭 형식 제약 조건](#)으로 사용할 수도 있습니다.

## 생성자 호출

새 인스턴스 유형을 만들려면 일반적으로 `new` 연산자를 사용하여 해당 유형의 [생성자](#) 중 하나를 호출합니다.

```
var dict = new Dictionary<string, int>();
dict["first"] = 10;
dict["second"] = 20;
dict["third"] = 30;

Console.WriteLine(string.Join(" ", dict.Select(entry => $"{entry.Key}: {entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

다음 예제와 같이 `new` 연산자와 함께 [개체 또는 컬렉션 이니셜라이저](#)를 사용하여 하나의 명령문에서 개체를 인스턴스화하고 초기화할 수 있습니다.

```
var dict = new Dictionary<string, int>
{
    ["first"] = 10,
    ["second"] = 20,
    ["third"] = 30
};

Console.WriteLine(string.Join(" ", dict.Select(entry => $"{entry.Key}: {entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

C# 9.0부터 생성자 호출식은 대상으로 형식화됩니다. 즉, 식의 대상 형식을 알고 있는 경우 다음 예제와 같이 형식 이름을 생략할 수 있습니다.

```
List<int> xs = new();
List<int> ys = new(capacity: 10_000);
List<int> zs = new() { Capacity = 20_000 };

Dictionary<int, List<int>> lookup = new()
{
    [1] = new() { 1, 2, 3 },
    [2] = new() { 5, 8, 3 },
    [5] = new() { 1, 0, 4 }
};
```

앞의 예제에 나온 것처럼 대상으로 형식화된 `new` 식에는 항상 괄호를 사용합니다.

`new` 식의 대상 형식을 알 수 없는 경우(예를 들어 `var` 키워드를 사용하는 경우) 형식 이름을 지정해야 합니다.

## 배열 생성

또한 다음 예제와 같이 `new` 연산자를 사용하여 배열 인스턴스를 만듭니다.

```
var numbers = new int[3];
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;

Console.WriteLine(string.Join(", ", numbers));
// Output:
// 10, 20, 30
```

배열 초기화 구문을 사용하여 배열 인스턴스를 만들고 하나의 명령문에 요소를 채웁니다. 다음 예제에서는 이를 수행하는 다양한 방법을 보여줍니다.

```
var a = new int[3] { 10, 20, 30 };
var b = new int[] { 10, 20, 30 };
var c = new[] { 10, 20, 30 };
Console.WriteLine(c.GetType()); // output: System.Int32[]
```

배열에 대한 자세한 내용은 [배열](#)을 참조하세요.

## 익명 형식의 인스턴스화

익명 형식의 인스턴스를 만들려면 `new` 연산자와 개체 이니셜라이저 구문을 사용합니다.

```
var example = new { Greeting = "Hello", Name = "World" };
Console.WriteLine($"{example.Greeting}, {example.Name}!");
// Output:
// Hello, World!
```

## 형식 인스턴스의 소멸

앞서 만든 형식 인스턴스를 제거할 필요가 없습니다. 참조 형식과 값 형식 모두의 인스턴스는 자동으로 제거됩니다. 값 형식의 인스턴스는 포함된 컨텍스트가 제거되는 즉시 제거됩니다. 참조 형식의 인스턴스는 마지막 참조가 제거된 후 일부 지정되지 않은 시간에 [가비지 수집기](#)에 의해 제거됩니다.

파일 핸들과 같이 관리되지 않은 리소스를 포함하는 형식 인스턴스의 경우에는 결정적 정리를 사용하여 포함된 리소스가 가능한 빨리 릴리스되도록 하는 것이 좋습니다. 자세한 내용은 [System.IDisposable API](#) 참조 및 [명령문 사용](#) 문서를 참조하세요.

## 연산자 오버로드 가능성

사용자 정의 형식은 `new` 연산자를 오버로드할 수 없습니다.

## C# 언어 사양

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

대상으로 형식화된 `new` 식에 대한 자세한 내용은 [기능 제한 노트](#)를 참조하세요.

## 참고 항목

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [개체 및 컬렉션 이니셜라이저](#)

# sizeof 연산자(C# 참조)

2020-11-02 • 3 minutes to read • [Edit Online](#)

`sizeof` 연산자는 지정된 형식의 변수에서 사용하는 바이트 수를 반환합니다. `sizeof` 연산자의 인수는 [비관리형 형식](#) 또는 [비관리형 형식](#)이 되기 위해 [제한된](#) 형식 매개 변수의 이름이어야 합니다.

`sizeof` 연산자에 [안전하지 않은](#) 컨텍스트가 필요합니다. 그러나 아래 표의 식은 컴파일 시간에 해당 상수 값으로 계산되며 안전하지 않은 컨텍스트가 필요하지 않습니다.

식	상수 값
<code>sizeof(sbyte)</code>	1
<code>sizeof(byte)</code>	1
<code>sizeof(short)</code>	2
<code>sizeof(ushort)</code>	2
<code>sizeof(int)</code>	4
<code>sizeof(uint)</code>	4
<code>sizeof(long)</code>	8
<code>sizeof(ulong)</code>	8
<code>sizeof(char)</code>	2
<code>sizeof(float)</code>	4
<code>sizeof(double)</code>	8
<code>sizeof(decimal)</code>	16
<code>sizeof(bool)</code>	1

`sizeof` 연산자의 피연산자가 [열거형](#) 형식의 이름인 경우에도 안전하지 않은 컨텍스트를 사용할 필요가 없습니다.

다음 예제에서는 `sizeof` 연산자의 사용법을 보여 줍니다.

```

using System;

public struct Point
{
    public Point(byte tag, double x, double y) => (Tag, X, Y) = (tag, x, y);

    public byte Tag { get; }
    public double X { get; }
    public double Y { get; }
}

public class SizeOfOperator
{
    public static void Main()
    {
        Console.WriteLine(sizeof(byte)); // output: 1
        Console.WriteLine(sizeof(double)); // output: 8

        DisplaySizeOf<Point>(); // output: Size of Point is 24
        DisplaySizeOf<decimal>(); // output: Size of System.Decimal is 16

        unsafe
        {
            Console.WriteLine(sizeof(Point*)); // output: 8
        }
    }

    static unsafe void DisplaySizeOf<T>() where T : unmanaged
    {
        Console.WriteLine($"Size of {typeof(T)} is {sizeof(T)}");
    }
}

```

`sizeof` 연산자는 관리되는 메모리의 공용 언어 런타임에서 할당하는 바이트 수를 반환합니다. 구조체 형식의 경우 앞의 예제에서 보여 주는 것처럼 해당 값에 안쪽 여백이 포함됩니다. `sizeof` 연산자의 결과는 '관리되지 않는' 메모리의 형식 크기를 반환하는 [Marshal.SizeOf](#) 메서드의 결과와 다를 수 있습니다.

## C# 언어 사양

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

## 참고 항목

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [포인터 관련 연산자](#)
- [포인터 형식](#)
- [메모리 및 범위 관련 형식](#)
- [.NET의 제네릭](#)

# stackalloc 식(C# 참조)

2020-11-02 • 7 minutes to read • [Edit Online](#)

`stackalloc` 식은 스택에 메모리 블록을 할당합니다. 메서드 실행 중에 생성된 스택 할당 메모리 블록은 해당 메서드가 반환될 때 자동으로 삭제됩니다. `stackalloc`을 사용하여 할당된 메모리를 명시적으로 해제할 수 없습니다. 스택 할당 메모리 블록에는 [가비지 수집](#)이 적용되지 않으며, `fixed` 문을 사용해서 고정하지 않아도 됩니다.

`stackalloc` 식의 결과를 다음 형식 중 하나의 변수에 할당할 수 있습니다.

- C# 7.2부터 `System.Span<T>` 또는 `System.ReadOnlySpan<T>`(다음 예제 참조):

```
int length = 3;
Span<int> numbers = stackalloc int[length];
for (var i = 0; i < length; i++)
{
    numbers[i] = i;
}
```

`Span<T>` 또는 `ReadOnlySpan<T>` 변수에 스택 할당 메모리 블록을 할당할 때 `unsafe` 컨텍스트를 사용하지 않아도 됩니다.

이러한 형식으로 작업하는 경우 다음 예제와 같이 [조건식](#) 또는 대입식에 `stackalloc` 식을 사용할 수 있습니다.

```
int length = 1000;
Span<byte> buffer = length <= 1024 ? stackalloc byte[length] : new byte[length];
```

C# 8.0부터 다음 예제와 같이 `Span<T>` 또는 `ReadOnlySpan<T>` 변수가 허용되는 경우 다른 식 내부에서 `stackalloc` 식을 사용할 수 있습니다.

```
Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };
var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6, 8 });
Console.WriteLine(ind); // output: 1
```

## NOTE

스택 할당 메모리로 작업할 때는 가능한 한, `Span<T>` 또는 `ReadOnlySpan<T>` 형식을 사용하는 것이 좋습니다.

- [포인터 형식](#)(다음 예제 참조)

```
unsafe
{
    int length = 3;
    int* numbers = stackalloc int[length];
    for (var i = 0; i < length; i++)
    {
        numbers[i] = i;
    }
}
```

포인터 형식으로 작업할 때는 위 예제와 같이 `unsafe` 컨텍스트를 사용해야 합니다.

포인터 형식의 경우 지역 변수 선언에서만 `stackalloc` 식을 사용하여 변수를 초기화할 수 있습니다.

스택에서 사용 가능한 메모리 양이 제한됩니다. 스택에 너무 많은 메모리를 할당하는 경우 `StackOverflowException`이 throw됩니다. 이를 방지하려면 다음 규칙을 따르세요.

- `stackalloc` 을 사용하여 할당하는 메모리의 양을 제한합니다.

```
const int MaxStackLimit = 1024;  
Span<byte> buffer = inputLength <= MaxStackLimit ? stackalloc byte[inputLength] : new  
byte[inputLength];
```

스택에서 사용 가능한 메모리 양은 코드를 실행하는 환경에 따라 달라지므로 실제 제한 값을 정의할 때는 신중해야 합니다.

- 루프 내부에서 `stackalloc` 을 사용하지 마세요. 루프 외부에서 메모리 블록을 할당하고 루프 내부에서 다시 사용합니다.

새로 할당된 메모리의 콘텐츠는 정의되지 않습니다. 사용하기 전에 초기화해야 합니다. 예를 들어 모든 항목을 `T` 형식의 기본값으로 설정하는 `Span<T>.Clear` 메서드를 사용할 수 있습니다.

C# 7.3부터, 배열 이니셜라이저 구문을 사용하여 새로 할당된 메모리의 콘텐츠를 정의할 수 있습니다. 다음 예제에서는 이 작업을 수행하는 다양한 방법을 보여 줍니다.

```
Span<int> first = stackalloc int[3] { 1, 2, 3 };  
Span<int> second = stackalloc int[] { 1, 2, 3 };  
ReadOnlySpan<int> third = stackalloc[] { 1, 2, 3 };
```

식 `stackalloc T[E]`에서 `T`는 관리되지 않는 형식이어야 하며 `E`는 음수가 아닌 `int` 값으로 계산되어야 합니다.

## 보안

`stackalloc` 를 사용하면 CLR(공용 언어 런타임)에서 버퍼 오버런 검색 기능이 자동으로 사용됩니다. 버퍼 오버런이 검색되면 악성 코드가 실행될 가능성을 최소화하기 위해 최대한 빨리 프로세스가 종료됩니다.

## C# 언어 사양

자세한 내용은 C# 언어 사양의 [스택 할당](#) 섹션과 중첩 컨텍스트 기능 제한 노트의 `stackalloc` 허용을 참조하세요.

## 참고 항목

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [포인터 관련 연산자](#)
- [포인터 형식](#)
- [메모리 및 범위 관련 형식](#)
- [stackalloc 관련 실행 사항 및 금지 사항](#)

# switch 식(C# 참조)

2021-02-18 • 8 minutes to read • [Edit Online](#)

이 문서에서는 C# 8.0에 도입된 `switch` 식에 대해 설명합니다. `switch` 문에 대한 자세한 내용은 [문 섹션의 `switch` 문](#) 문서를 참조하세요.

## 기본 예제

`switch` 식은 식 컨텍스트에서 `switch` 와 유사한 의미 체계를 제공합니다. 스위치 암(arm)에서 값을 생성할 때 간결한 구문을 제공합니다. 다음 예제에서는 `switch` 식의 구조를 보여 줍니다. 온라인 맵의 시각적 방향을 나타내는 `enum` 의 값을 해당하는 기본 방향으로 변환합니다.

```
public static class SwitchExample
{
    public enum Directions
    {
        Up,
        Down,
        Right,
        Left
    }

    public enum Orientation
    {
        North,
        South,
        East,
        West
    }

    public static void Main()
    {
        var direction = Directions.Right;
        Console.WriteLine($"Map view direction is {direction}");

        var orientation = direction switch
        {
            Directions.Up      => Orientation.North,
            Directions.Right   => Orientation.East,
            Directions.Down    => Orientation.South,
            Directions.Left    => Orientation.West,
        };
        Console.WriteLine($"Cardinal orientation is {orientation}");
    }
}
```

위의 샘플에서는 `switch` 식의 기본 요소를 보여 줍니다.

- *range* 식: 위의 예제에서는 `direction` 변수를 *range* 식으로 사용합니다.
- *switch* 식 암(arm): 각 `switch` 식 암(arm)에는 *패턴*, 선택적인 *케이스 가드*, `=>` 토큰 및 식이 포함되어 있습니다.

`switch` 식의 결과는 *패턴*이 *range* 식과 일치하고 *케이스 가드*(있는 경우)가 `true`로 평가되는 첫 번째 `switch` 식 암(arm)의 값입니다. `=>` 토큰 오른쪽에 있는 식은 식 문이 될 수 없습니다.

`switch` 식 암(arm)은 텍스트 순서대로 평가됩니다. 상위 `switch` 식 암(arm)이 모든 값과 일치하기 때문에 하위 `switch` 식 암(arm)을 선택할 수 없는 경우 컴파일러에서 오류를 발생시킵니다.

## 패턴 및 케이스 가드

switch 식 암(arm)에서는 많은 패턴이 지원됩니다. 앞의 예제에서는 '상수 패턴'을 사용합니다. '상수 패턴'은 range 식을 값과 비교합니다. 이 값은 컴파일 시간 상수여야 합니다. 형식 패턴은 range 식을 알려진 형식과 비교합니다. 다음 예제에서는 시퀀스에서 세 번째 요소를 검색합니다. 시퀀스의 형식에 따라 다른 메서드를 사용합니다.

```
public static T TypeExample<T>(IEnumerable<T> sequence) =>
    sequence switch
    {
        System.Array array => (T)array.GetValue(2),
        IList<T> list      => list[2],
        IEnumerable<T> seq  => seq.Skip(2).First(),
    };
```

패턴은 재귀적으로 지정할 수 있습니다. 그러면 패턴이 형식을 테스트하고 해당 형식이 일치하는 경우 패턴이 range 식의 하나 이상의 속성 값과 일치합니다. 재귀 패턴을 사용하여 앞의 예제를 확장할 수 있습니다. 요소가 3개 미만인 배열에 대해 switch 식 암(arm)을 추가합니다. 다음 예제에서는 재귀 패턴을 보여 줍니다.

```
public static T RecursiveExample<T>(IEnumerable<T> sequence) =>
    sequence switch
    {
        System.Array { Length : 0}      => default(T),
        System.Array { Length : 1} array => (T)array.GetValue(0),
        System.Array { Length : 2} array => (T)array.GetValue(1),
        System.Array array             => (T)array.GetValue(2),
        IList<T> list                 => list[2],
        IEnumerable<T> seq            => seq.Skip(2).First(),
    };
```

재귀 패턴은 range 식의 속성을 검사할 수 있지만 임의의 코드를 실행할 수는 없습니다. `when` 절에 지정된 케이스 가드를 사용하여 다른 시퀀스 형식에 비슷한 검사를 제공할 수 있습니다.

```
public static T CaseGuardExample<T>(IEnumerable<T> sequence) =>
    sequence switch
    {
        System.Array { Length : 0}      => default(T),
        System.Array { Length : 1} array => (T)array.GetValue(0),
        System.Array { Length : 2} array => (T)array.GetValue(1),
        System.Array array             => (T)array.GetValue(2),
        IEnumerable<T> list when !list.Any() => default(T),
        IEnumerable<T> list when list.Count() < 3 => list.Last(),
        IList<T> list                 => list[2],
        IEnumerable<T> seq            => seq.Skip(2).First(),
    };
```

마지막으로 `_` 패턴과 `null` 패턴을 추가하여 다른 switch 식 암(arm)에서 처리되지 않은 인수를 catch할 수 있습니다. 이렇게 하면 switch 식이 range 식의 가능한 값을 모두 처리합니다(*exhaustive*). 다음 예제에서는 이러한 식 암(arm)을 추가합니다.

```

public static T ExhaustiveExample<T>(IEnumerable<T> sequence) =>
    sequence switch
    {
        System.Array { Length : 0}      => default(T),
        System.Array { Length : 1} array => (T)array.GetValue(0),
        System.Array { Length : 2} array => (T)array.GetValue(1),
        System.Array array             => (T)array.GetValue(2),
        IEnumerable<T> list
            when !list.Any()          => default(T),
        IEnumerable<T> list
            when list.Count() < 3     => list.Last(),
        IList<T> list                => list[2],
        null                         => throw new ArgumentNullException(nameof(sequence)),
        _                            => sequence.Skip(2).First(),
    };

```

앞의 예제는 `null` 패턴을 추가하고 `IEnumerable<T>` 형식 패턴을 `_` 패턴으로 변경합니다. `null` 패턴은 `switch` 식 암(arm)으로 `null` 검사를 제공합니다. 해당 암(arm)에 대한 식에서 `ArgumentNullException`을 `throw`합니다. `_` 패턴은 이전 암(arm)과 일치하지 않은 모든 입력을 일치시킵니다. `null` 검사 후에 오거나 `null` 입력과 일치해야 합니다.

## 불완전 switch 식

`switch` 식의 패턴 중 하나라도 인수를 `catch`하지 않는 경우 런타임은 예외를 `throw`합니다. .NET Core 3.0 이상 버전에서 예외는 `System.Runtime.CompilerServices.SwitchExpressionException`입니다. .NET Framework에서 예외는 `InvalidOperationException`입니다.

## 추가 정보

- [재귀 패턴에 대한 C# 언어 사양 제안](#)
- [C# 참조](#)
- [C# 연산자 및 식](#)
- [패턴 일치](#)

# true 및 false 연산자(C# 참조)

2020-11-02 • 6 minutes to read • [Edit Online](#)

`true` 연산자는 `부울` 값을 반환하여 `true` 피연산자가 확실히 `true`임을 나타냅니다. `false` 연산자는 `bool` 값을 반환하여 `true` 피연산자가 확실히 `false`임을 나타냅니다. `true` 및 `false` 연산자는 서로를 보완한다고 보장되지 않습니다. 즉, `true` 및 `false` 연산자는 모두 `bool` 값을 동일한 `false` 피연산자에 반환할 수도 있습니다. 형식이 두 연산자 중 하나를 정의하는 경우 나머지 연산자도 정의해야 합니다.

## TIP

예를 들어 값이 세 개인 논리를 지원해야 하는 경우(예: 값이 세 개인 부울 형식을 지원하는 데이터베이스에서 작업하는 경우) `bool?` 형식을 사용합니다. C#은 `bool?` 피연산자를 사용하여 값이 세 개인 논리를 지원하는 `&` 및 `|` 연산자를 제공합니다. 자세한 내용은 [부울 논리 연산자 문서의 Nullable 부울 논리 연산자 섹션](#)을 참조하세요.

## 부울 식

정의된 `true` 연산자가 있는 형식은 `if`, `do`, `while` 및 `for` 문과 [조건부 연산자 ?:](#)에서 제어하는 조건식의 결과 형식일 수 있습니다. 자세한 내용은 [C# 언어 사양의 부울 식 섹션](#)을 참조하세요.

## 사용자 지정 조건부 논리 연산자

정의된 `true` 및 `false` 연산자가 있는 형식이 특정 방식으로 [논리적 OR 연산자 또는 논리적 AND 연산자 |](#)를 오버로드 `&`하는 경우, [조건부 논리적 OR 연산자 ||](#) 또는 [조건부 논리적 AND 연산자 &&](#)가 해당 형식의 피연산자에 대해 각각 계산될 수 있습니다. 자세한 내용은 [C# 언어 사양의 사용자 정의 조건부 논리 연산자 섹션](#)을 참조하세요.

## 예제

다음 예제는 `true` 및 `false` 연산자를 둘 다 정의하는 형식을 제공합니다. `&&` 연산자도 해당 형식의 피연산자에 대해 계산될 수 있는 방식으로 논리적 AND 연산자 `&`를 오버로드합니다.

```

using System;

public struct LaunchStatus
{
    public static readonly LaunchStatus Green = new LaunchStatus(0);
    public static readonly LaunchStatus Yellow = new LaunchStatus(1);
    public static readonly LaunchStatus Red = new LaunchStatus(2);

    private int status;

    private LaunchStatus(int status)
    {
        this.status = status;
    }

    public static bool operator true(LaunchStatus x) => x == Green || x == Yellow;
    public static bool operator false(LaunchStatus x) => x == Red;

    public static LaunchStatus operator &(LaunchStatus x, LaunchStatus y)
    {
        if (x == Red || y == Red || (x == Yellow && y == Yellow))
        {
            return Red;
        }

        if (x == Yellow || y == Yellow)
        {
            return Yellow;
        }

        return Green;
    }

    public static bool operator ==(LaunchStatus x, LaunchStatus y) => x.status == y.status;
    public static bool operator !=(LaunchStatus x, LaunchStatus y) => !(x == y);

    public override bool Equals(object obj) => obj is LaunchStatus other && this == other;
    public override int GetHashCode() => status;
}

public class LaunchStatusTest
{
    public static void Main()
    {
        LaunchStatus okToLaunch = GetFuelLaunchStatus() && GetNavigationLaunchStatus();
        Console.WriteLine(okToLaunch ? "Ready to go!" : "Wait!");
    }

    static LaunchStatus GetFuelLaunchStatus()
    {
        Console.WriteLine("Getting fuel launch status...");
        return LaunchStatus.Red;
    }

    static LaunchStatus GetNavigationLaunchStatus()
    {
        Console.WriteLine("Getting navigation launch status...");
        return LaunchStatus.Yellow;
    }
}

```

&& 연산자의 단락 동작을 확인합니다. `GetFuelLaunchStatus` 메서드가 `LaunchStatus.Red`를 반환하면 && 연산자의 오른쪽 피연산자는 계산되지 않습니다. `LaunchStatus.Red` 가 확실히 false이기 때문입니다. 따라서 논리적 AND의 결과가 오른쪽 피연산자의 값에 종속되지 않습니다. 예제 출력은 다음과 같습니다.

```
Getting fuel launch status...
Wait!
```

## 참고 항목

- [C# 참조](#)
- [C# 연산자 및 식](#)

# with 식(C# 참조)

2021-02-18 • 5 minutes to read • [Edit Online](#)

C# 9.0 이상에서 사용 가능한 `with` 식은 지정된 속성 및 필드를 수정하여 해당 [레코드](#) 피연산자의 복사본을 생성합니다.

```
using System;

public class WithExpressionBasicExample
{
    public record NamedPoint(string Name, int X, int Y);

    public static void Main()
    {
        var p1 = new NamedPoint("A", 0, 0);
        Console.WriteLine($"{nameof(p1)}: {p1}"); // output: p1: NamedPoint { Name = A, X = 0, Y = 0 }

        var p2 = p1 with { Name = "B", X = 5 };
        Console.WriteLine($"{nameof(p2)}: {p2}"); // output: p2: NamedPoint { Name = B, X = 5, Y = 0 }

        var p3 = p1 with
        {
            Name = "C",
            Y = 4
        };
        Console.WriteLine($"{nameof(p3)}: {p3}"); // output: p3: NamedPoint { Name = C, X = 0, Y = 4 }

        Console.WriteLine($"{nameof(p1)}: {p1}"); // output: p1: NamedPoint { Name = A, X = 0, Y = 0 }
    }
}
```

위 예제와 같이 [개체 이니셜라이저](#) 구문을 사용하여 수정할 멤버와 새 값을 지정합니다. `with` 식에서 왼쪽 피연산자는 레코드 형식이어야 합니다.

다음 예제와 같이 `with` 식의 결과는 식의 피연산자와 동일한 런타임 형식입니다.

```
using System;

public class InheritanceExample
{
    public record Point(int X, int Y);
    public record NamedPoint(string Name, int X, int Y) : Point(X, Y);

    public static void Main()
    {
        Point p1 = new NamedPoint("A", 0, 0);
        Point p2 = p1 with { X = 5, Y = 3 };
        Console.WriteLine(p2 is NamedPoint); // output: True
        Console.WriteLine(p2); // output: NamedPoint { X = 5, Y = 3, Name = A }

    }
}
```

참조 형식 멤버인 경우 레코드가 복사될 때 인스턴스에 대한 참조만 복사됩니다. 복사본과 원본 레코드 모두 동일한 참조 형식 인스턴스에 액세스할 수 있습니다. 다음 예제에서는 해당 동작을 보여줍니다.

```

using System;
using System.Collections.Generic;

public class ExampleWithReferenceType
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A", "B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B

        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B, C
    }
}

```

모든 레코드 형식에 '복사 생성자'가 있습니다. 이 생성자는 레코드 형식의 단일 매개 변수를 사용합니다. 또한 인수의 상태를 새 레코드 인스턴스로 복사합니다. `with` 식을 평가할 때 복사 생성자가 호출되어 원본 레코드를 기반으로 새 레코드 인스턴스를 인스턴스화합니다. 그런 다음, 지정된 수정 사항에 따라 새 인스턴스가 업데이트됩니다. 기본적으로 복사 생성자는 암시적으로, 컴파일러에서 생성합니다. 레코드 복사 의미 체계를 사용자 지정해야 하는 경우 원하는 동작으로 복사 생성자를 명시적으로 선언합니다. 다음 예제에서는 명시적 복사 생성자를 사용하여 위의 예제를 업데이트합니다. 새 복사 동작은 레코드가 복사될 때 목록 참조 대신 목록 항목을 복사하는 것입니다.

```

using System;
using System.Collections.Generic;

public class UserDefinedCopyConstructorExample
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        protected TaggedNumber(TaggedNumber original)
        {
            Number = original.Number;
            Tags = new List<string>(original.Tags);
        }

        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A", "B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B

        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B
    }
}

```

## C# 언어 사양

자세한 내용은 [레코드 기능 제안 노트](#)의 다음 섹션을 참조하세요.

- [with](#) 식
- 멤버 복사 및 복제

## 참조

- [C# 참조](#)
- [C# 연산자 및 식](#)

# 연산자 오버로드(C# 참조)

2021-02-18 • 9 minutes to read • [Edit Online](#)

사용자 정의 형식은 미리 정의된 C# 연산자를 오버로드할 수 있습니다. 즉, 피연산자 중 하나 또는 두 개가 해당 형식인 경우 형식은 작업의 사용자 정의 구현을 제공할 수 있습니다. [오버로드할 수 있는 연산자](#) 섹션에는 오버로드할 수 있는 C# 연산자가 나와 있습니다.

`operator` 키워드를 사용하여 연산자를 선언합니다. 연산자 선언은 다음 규칙을 충족해야 합니다.

- `public` 및 `static` 한정자를 모두 포함합니다.
- 단항 연산자에는 하나의 입력 매개 변수가 있습니다. 이항 연산자에는 두 개의 입력 매개 변수가 있습니다. 각각의 경우 하나 이상의 매개 변수가 `T` 또는 `T?` 유형을 가져야 하며, 여기서 `T`는 연산자 선언이 포함된 유형입니다.

다음 예제에서는 유리수를 나타내는 간단한 구조를 정의합니다. 구조체가 [산술 연산자](#) 중 일부를 오버로드합니다.

```

using System;

public readonly struct Fraction
{
    private readonly int num;
    private readonly int den;

    public Fraction(int numerator, int denominator)
    {
        if (denominator == 0)
        {
            throw new ArgumentException("Denominator cannot be zero.", nameof(denominator));
        }
        num = numerator;
        den = denominator;
    }

    public static Fraction operator +(Fraction a) => a;
    public static Fraction operator -(Fraction a) => new Fraction(-a.num, a.den);

    public static Fraction operator +(Fraction a, Fraction b)
        => new Fraction(a.num * b.den + b.num * a.den, a.den * b.den);

    public static Fraction operator -(Fraction a, Fraction b)
        => a + (-b);

    public static Fraction operator *(Fraction a, Fraction b)
        => new Fraction(a.num * b.num, a.den * b.den);

    public static Fraction operator /(Fraction a, Fraction b)
    {
        if (b.num == 0)
        {
            throw new DivideByZeroException();
        }
        return new Fraction(a.num * b.den, a.den * b.num);
    }

    public override string ToString() => $"{num} / {den}";
}

public static class OperatorOverloading
{
    public static void Main()
    {
        var a = new Fraction(5, 4);
        var b = new Fraction(1, 2);
        Console.WriteLine(-a); // output: -5 / 4
        Console.WriteLine(a + b); // output: 14 / 8
        Console.WriteLine(a - b); // output: 6 / 8
        Console.WriteLine(a * b); // output: 5 / 8
        Console.WriteLine(a / b); // output: 10 / 4
    }
}

```

암시적 변환을 `int`에서 `Fraction`으로 정의하여 앞의 예제를 확장할 수 있습니다. 그런 다음, 오버로드된 연산자는 해당 두 형식의 인수를 지원합니다. 즉, 분수에 정수를 추가하고 그 결과로 분수를 얻을 수 있습니다.

또한 `operator` 키워드를 사용하여 사용자 지정 형식 변환을 정의합니다. 자세한 내용은 [사용자 정의 변환 연산자](#)를 참조하세요.

## 오버로드 할 수 있는 연산자

다음 표는 C# 연산자의 오버로드 가능성에 대한 정보를 제공합니다.

연산자	오버로드 가능성
<code>+x, -x, !x, ~x, ++, --, true, false</code>	이러한 단항 연산자는 오버로드할 수 있습니다.
<code>x + y, x - y, x * y, x / y, x % y, x &amp; y, x   y, x ^ y, x &lt;&lt; y, x &gt;&gt; y, x == y, x != y, x &lt; y, x &gt; y, x &lt;= y, x &gt;= y</code>	이러한 이항 연산자는 오버로드할 수 있습니다. 특정 연산자는 쌍으로 오버로드되어야 합니다. 자세한 내용은 이 표 다음에 나오는 참고 사항을 참조하세요.
<code>x &amp;&amp; y, x    y</code>	조건부 논리 연산자는 오버로드할 수 없습니다. 그러나 오버로드된 <code>true</code> 및 <code>false</code> 연산자가 있는 형식도 특정 방식으로 <code>&amp;</code> 또는 <code> </code> 연산자를 오버로드하는 경우, <code>&amp;&amp;</code> 또는 <code>  </code> 연산자는 각각 해당 유형의 피연산자에 대해 평가될 수 있습니다. 자세한 내용은 <a href="#">C# 언어 사양의 사용자 정의 조건부 논리 연산자</a> 섹션을 참조하세요.
<code>a[i], a?[i]</code>	요소 액세스는 오버로드 가능한 연산자로 간주되지 않지만 <a href="#">인덱서</a> 를 정의할 수 있습니다.
<code>(T)x</code>	캐스트 연산자는 오버로드될 수 없지만, 캐스트 식에서 수행할 수 있는 사용자 지정 형식 변환을 정의할 수 있습니다. 자세한 내용은 <a href="#">사용자 정의 변환 연산자</a> 를 참조하세요.
<code>+=, -=, *=, /=, %=, &amp;=,  =, ^=, &lt;&lt;=, &gt;&gt;=</code>	복합 할당 연산자를 명시적으로 오버로드할 수 없습니다. 그러나 이항 연산자가 오버로드되면 해당 복합 할당 연산자(있는 경우)도 암시적으로 오버로드됩니다. 예를 들어 <code>+=</code> 는 오버로드될 수 있는 <code>+</code> 를 사용하여 계산됩니다.
<code>^x, x = y, x.y, x?.y, c ? t : f, x ?? y, x ??= y, x..y, x-&gt;y, =&gt;, f(x), as, await, checked, unchecked, default, delegate, is, nameof, new, sizeof, stackalloc, switch, typeof, with</code>	이러한 연산자는 오버로드할 수 없습니다.

#### NOTE

비교 연산자는 쌍으로 오버로드되어야 합니다. 즉, 쌍 중 하나의 연산자가 오버로드되면 다른 연산자도 오버로드되어야 합니다. 이러한 쌍은 다음과 같습니다.

- `==` 및 `!=` 연산자
- `<` 및 `>` 연산자
- `<=` 및 `>=` 연산자

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 다음 섹션을 참조하세요.

- [연산자 오버로드](#)
- [연산자](#)

## 참조

- [C# 참조](#)
- [C# 연산자 및 식](#)
- [사용자 정의 전환 연산자](#)
- [디자인 지침 - 연산자 오버 로드](#)
- [디자인 지침 - 같은 연산자](#)

- Why are overloaded operators always static in C#?(오버로드된 연산자가 C#에서 항상 정적인 이유)

# C# 특수 문자

2020-11-02 • 2 minutes to read • [Edit Online](#)

특수 문자는 해당 문자가 앞에 붙는 프로그램 요소(리터럴 문자열, 식별자 또는 특성 이름)를 수정하는 미리 정의된 상황에 맞는 문자입니다. C#은 다음과 같은 특수 문자를 지원합니다.

- @, 약어 식별자 문자입니다.
- \$, 보간된 문자열 문자입니다.

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)

# \$ - 문자열 보간(C# 참조)

2020-05-02 • 12 minutes to read • [Edit Online](#)

\$ 특수 문자는 문자열 리터럴을 보간된 문자열로 식별합니다. 보간된 문자열은 보간 식이 포함될 수 있는 문자열 리터럴입니다. 보간된 문자열이 결과 문자열로 해석되면 보간 식이 있는 항목이 식 결과의 문자열 표현으로 바뀝니다. 이 기능은 C# 6부터 사용할 수 있습니다.

문자열 보간은 [문자열 합성 서식 지정](#) 기능보다 읽기 쉽고 편리하게 서식이 지정된 문자열을 만들 수 있는 구문을 제공합니다. 다음 예제에서는 두 기능을 사용하여 동일한 출력을 생성합니다.

```
string name = "Mark";
var date = DateTime.Now;

// Composite formatting:
Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", name, date.DayOfWeek, date);
// String interpolation:
Console.WriteLine($"Hello, {name}! Today is {date.DayOfWeek}, it's {date:HH:mm} now.");
// Both calls produce the same output that is similar to:
// Hello, Mark! Today is Wednesday, it's 19:40 now.
```

## 보간된 문자열의 구조

문자열 리터럴을 보간된 문자열로 식별하려면 \$ 기호를 사용하여 추가합니다. 문자열 리터럴을 시작하는 \$ 및 " 사이에 공백이 없어야 합니다.

보간 식이 있는 항목의 구조는 다음과 같습니다.

```
{<interpolationExpression>[,<alignment>][:<formatString>]}
```

대괄호 안의 요소는 선택 사항입니다. 다음 표에서는 각 요소에 대해 설명합니다.

요소	설명
interpolationExpression	서식을 지정할 결과를 생성하는 식입니다. null의 문자열 표현은 <a href="#">String.Empty</a> 입니다.
alignment	식 결과의 문자열 표현에 최소 문자 수를 정의하는 값을 갖는 상수 식입니다. 양수이면 문자열 표현이 오른쪽에 맞춰지며, 음수이면 왼쪽에 맞춰집니다. 자세한 내용은 <a href="#">맞춤 구성 요소</a> 를 참조하세요.
formatString	식 결과의 형식을 기준으로 지원되는 서식 문자열입니다. 자세한 내용은 <a href="#">서식 문자열 구성 요소</a> 를 참조하세요.

다음 예제에서는 위에 설명된 선택적 서식 지정 구성 요소를 사용합니다.

```

Console.WriteLine($"|{"Left",-7}|{"Right",7}|");

const int FieldWidthRightAligned = 20;
Console.WriteLine($"{Math.PI,FieldWidthRightAligned} - default formatting of the pi number");
Console.WriteLine($"{Math.PI,FieldWidthRightAligned:F3} - display only three decimal digits of the pi
number");
// Expected output is:
// |Left    | Right|
//      3.14159265358979 - default formatting of the pi number
//                  3.142 - display only three decimal digits of the pi number

```

## 특수 문자

보간된 문자열로 생성된 문자에 중괄호("{" 또는 "}")를 포함하려면 두 개의 중괄호("{{" 또는 "}}")를 사용합니다. 자세한 내용은 [중괄호 이스케이프](#)를 참조하세요.

콜론(":")은 보간 식 항목에서 특별한 의미가 있으므로, 보간 식에서 조건부 연산자를 사용하려면 해당 식을 괄호로 묶습니다.

다음 예제에서는 중괄호를 결과 문자열에 포함하는 방법과 보간 식에서 조건부 연산자를 사용하는 방법을 보여 줍니다.

```

string name = "Horace";
int age = 34;
Console.WriteLine($"He asked, \"Is your name {name}?\", but didn't wait for a reply :-{{
Console.WriteLine($"{name} is {age} year{(age == 1 ? "" : "s")} old.");
// Expected output is:
// He asked, "Is your name Horace?", but didn't wait for a reply :-
// Horace is 34 years old.

```

보간된 약어 문자열은 @ 문자가 뒤에 오는 \$ 문자로 시작합니다. 약어 문자열에 대한 자세한 내용은 [문자열](#) 및 [약어 식별자](#) 항목을 참조하세요.

### NOTE

C# 8.0부터는 \$ 및 @ 토큰을 순서에 관계없이 사용할 수 있습니다. \${@"..."} 및 {@\$"..."} 는 모두 유효한 보간된 약어 문자열입니다. 이전 C# 버전에서는 \$ 토큰이 @ 토큰 앞에 나타나야 했습니다.

## 암시적 변환 및 IFormatProvider 구현 지정 방법

보간된 문자열에서 다음과 같은 세 가지 암시적 변환을 수행 할 수 있습니다.

- 보간된 문자열을 [String](#) 인스턴스로 변환. 보간 식 항목을 사용하여 보간된 문자열을 확인한 결과를 올바르게 서식이 지정된 문자열 표현으로 바꾼 것입니다. 이 변환에서는 [CurrentCulture](#)를 사용하여 식 결과의 형식을 지정합니다.
- 보간된 문자열을 서식을 지정 할 식 결과와 함께 복합 서식 문자열을 나타내는 [FormattableString](#) 인스턴스로 변환. 이 변수를 사용하면 단일 [FormattableString](#) 인스턴스에서 문화권별 콘텐츠가 포함된 여러 결과 문자열을 만들 수 있습니다. 이렇게 하려면 다음 메서드 중 하나를 호출합니다.
  - CurrentCulture에 대한 결과 문자열을 생성하는 [ToString\(\)](#) 오버로드.
  - InvariantCulture에 대한 결과 문자열을 생성하는 [Invariant](#) 메서드.
  - 지정된 문화에 대한 결과 문자열을 생성하는 [ToString\(IFormatProvider\)](#) 메서드.

[ToString\(IFormatProvider\)](#) 메서드를 사용하여 사용자 지정 형식을 지원하는 [IFormatProvider](#) 인터페이스의 사용자 정의 구현을 제공할 수도 있습니다. 자세한 내용은 [.NET의 형식 서식 지정](#) 문서의

[ICustomFormatter](#)를 사용하여 사용자 지정 형식 지정 섹션을 참조하세요.

- 보간된 문자열을 [IFormattable](#) 인스턴스로 변환. 또한 이 인스턴스를 사용하면 단일 [IFormattable](#) 인스턴스의 문화권별 콘텐츠로 여러 결과 문자열을 만들 수 있습니다.

다음 예제에서는 [FormattableString](#)에 대한 암시적 변환을 사용하여 문화권별 결과 문자열을 만듭니다.

```
double speedOfLight = 299792.458;
FormattableString message = $"The speed of light is {speedOfLight:N3} km/s.';

System.Globalization.CultureInfo.CurrentCulture = System.Globalization.CultureInfo.GetCultureInfo("nl-NL");
string messageInCurrentCulture = message.ToString();

var specificCulture = System.Globalization.CultureInfo.GetCultureInfo("en-IN");
string messageInSpecificCulture = message.ToString(specificCulture);

string messageInInvariantCulture = FormattableString.Invariant(message);

Console.WriteLine($"{System.Globalization.CultureInfo.CurrentCulture,-10} {messageInCurrentCulture}");
Console.WriteLine($"{specificCulture,-10} {messageInSpecificCulture}");
Console.WriteLine($"{Invariant,-10} {messageInInvariantCulture}");
// Expected output is:
// nl-NL      The speed of light is 299,792,458 km/s.
// en-IN      The speed of light is 2,99,792.458 km/s.
// Invariant   The speed of light is 299,792.458 km/s.
```

## 추가 자료

문자열 보간을 처음 접하는 경우 [C#의 문자열 보간](#) 대화형 자습서를 참조하세요. 보간된 문자열을 사용하여 서식화된 문자열을 생성하는 방법을 보여 주는 다른 [C#의 문자열 보간](#) 자습서를 확인할 수도 있습니다.

## 보간된 문자열의 컴파일

보간된 문자열이 `string` 형식이면 일반적으로 [String.Format](#) 메서드 호출로 변환됩니다. 컴파일러는 분석된 동작이 연결에 해당하는 경우 [String.Format](#)을 [String.Concat](#)으로 바꿀 수 있습니다.

보간된 문자열 형식이 [IFormattable](#) 또는 [FormattableString](#)이면 컴파일러가 [FormattableStringFactory.Create](#) 메서드에 대한 호출을 생성합니다.

## C# 언어 사양

자세한 내용은 [C# 언어 사양](#)의 [보간된 문자열](#) 섹션을 참조하세요.

## 참조

- [C# 참조](#)
- [C# 특수 문자](#)
- [문자열](#)
- [표준 숫자 형식 문자열](#)
- [복합 형식 지정](#)
- [String.Format](#)

# @(C# 참조)

2020-11-02 • 6 minutes to read • [Edit Online](#)

@ 특수 문자는 축자 식별자로 사용됩니다. 다음과 같은 방법으로 사용할 수 있습니다.

1. C# 키워드를 식별자로 사용하도록 설정합니다. @ 문자는 컴파일러가 C# 키워드가 아닌 식별자로 해석할 코드 요소의 접두사입니다. 다음 예제에서는 @ 문자를 사용하여 for 루프에서 사용되는 for라는 이름의 식별자를 정의합니다.

```
string[] @for = { "John", "James", "Joan", "Jamie" };
for (int ctr = 0; ctr < @for.Length; ctr++)
{
    Console.WriteLine($"Here is your gift, {@for[ctr]}!");
}
// The example displays the following output:
//      Here is your gift, John!
//      Here is your gift, James!
//      Here is your gift, Joan!
//      Here is your gift, Jamie!
```

2. 문자열 리터럴이 축자로 해석될 것임을 나타냅니다. 이 인스턴스의 @ 문자는 축자 문자 문자열을 정의합니다. 단순 이스케이프 시퀀스(예: 백슬래시에 "\\"), 16진수 이스케이프 시퀀스(예: 대문자 A에 "\x0041"), 유니코드 이스케이프 시퀀스(예: 대문자 A에 "\u0041")는 문자 그대로 해석됩니다. 인용 부호 이스케이프 시퀀스( "")만 문자 그대로 해석되지 않고 하나의 큰따옴표를 생성합니다. 또한 축자 보간된 문자열 중괄호 이스케이프 시퀀스( {{ } })는 그대로 해석되지 않습니다. 단일 중괄호 문자를 생성합니다. 다음 예제에서는 각각 일반 문자열 리터럴 및 축자 문자열 리터럴을 사용하여 두 개의 동일한 파일 경로를 정의합니다. 이것은 축자 문자열 리터럴의 일반적인 사용법 중 하나입니다.

```
string filename1 = @"c:\documents\files\u0066.txt";
string filename2 = "c:\\documents\\files\\u0066.txt";

Console.WriteLine(filename1);
Console.WriteLine(filename2);
// The example displays the following output:
//      c:\documents\files\u0066.txt
//      c:\documents\files\u0066.txt
```

다음 예제에서는 동일한 문자 시퀀스를 포함한 일반 문자열 리터럴 및 축자 문자열 리터럴의 효과를 보여 줍니다.

```
string s1 = "He said, \"This is the last \u0063hance\x0021\"";
string s2 = @"He said, ""This is the last \u0063hance\x0021""";

Console.WriteLine(s1);
Console.WriteLine(s2);
// The example displays the following output:
//      He said, "This is the last chance!"
//      He said, "This is the last \u0063hance\x0021"
```

3. 이름이 서로 충돌하는 경우 특성 간에 구분하기 위해 컴파일러를 사용합니다. 특성은 Attribute에서 파생되는 클래스입니다. 컴파일러는 이 규칙을 적용하지 않지만, 형식 이름에는 일반적으로 Attribute 접미사가 포함됩니다. 전체 이름(예: [InfoAttribute]) 또는 약식 이름(예: [Info])으로 코드에서 특성을 참조할 수 있습니다. 그러나 두 개의 약식 특성 유형 이름이 동일하고 한 유형 이름에만 Attribute 접미사가

포함된 경우 이름 충돌이 발생합니다. 다음 예에서는 컴파일러가 `Info` 및 `InfoAttribute` 특성 중 무엇을 `Example` 클래스에 적용할지 결정할 수 없으므로 코드가 컴파일되지 않습니다. 자세한 내용은 [CS1614](#)를 참조하세요.

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class Info : Attribute
{
    private string information;

    public Info(string info)
    {
        information = info;
    }
}

[AttributeUsage(AttributeTargets.Method)]
public class InfoAttribute : Attribute
{
    private string information;

    public InfoAttribute(string info)
    {
        information = info;
    }
}

[Info("A simple executable.")] // Generates compiler error CS1614. Ambiguous Info and InfoAttribute.
// Prepend '@' to select 'Info'. Specify the full name 'InfoAttribute' to select it.
public class Example
{
    [InfoAttribute("The entry point.")]
    public static void Main()
    {
    }
}
```

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 특수 문자](#)

# 예약된 특성: 어셈블리 수준 특성

2020-04-23 • 7 minutes to read • [Edit Online](#)

대부분의 특성은 클래스나 메서드와 같은 특정 언어 요소에 적용되지만 일부 특성은 전체 어셈블리나 모듈에 적용되는 전역 특성입니다. 예를 들어 다음과 같이 [AssemblyVersionAttribute](#) 특성을 사용하여 버전 정보를 어셈블리에 포함할 수 있습니다.

```
[assembly: AssemblyVersion("1.0.0.0")]
```

전역 특성은 소스 코드에서 최상위 `using` 지시문 뒤 그리고 형식, 모듈 또는 네임스페이스 선언 앞에 나타납니다. 전역 특성은 여러 소스 파일에 나타날 수 있지만 파일은 하나의 컴파일 패스에서 컴파일되어야 합니다. Visual Studio는 .NET Framework 프로젝트의 AssemblyInfo.cs 파일에 전역 특성을 추가합니다. 해당 특성은 .NET Core 프로젝트에 추가되지 않습니다.

어셈블리 특성은 어셈블리에 대한 정보를 제공하는 값입니다. 어셈블리 특성은 다음 범주로 구분됩니다.

- 어셈블리 ID 특성
- 정보 특성
- 어셈블리 매니페스트 특성

## 어셈블리 ID 특성

`name`, `version` 및 `culture`의 세 가지 특성(해당하는 경우 강력한 이름 포함)이 어셈블리의 ID를 결정합니다. 이러한 특성은 어셈블리의 전체 이름을 구성하며 코드에서 어셈블리를 참조할 때 필요합니다. 특성을 사용하여 어셈블리의 버전 및 문화권을 설정할 수 있습니다. 그러나 이름 값은 어셈블리가 만들어질 때 컴파일러, [어셈블리 정보 대화 상자](#)의 Visual Studio IDE 또는 어셈블리 링커(AI.exe)에서 설정됩니다. 어셈블리 이름은 어셈블리 매니페스트를 기반으로 합니다. [AssemblyFlagsAttribute](#) 특성은 어셈블리의 여러 복사본이 공존할 수 있는지 여부를 지정합니다.

다음 표에서는 ID 특성들을 보여 줍니다.

특성	용도
<a href="#">AssemblyVersionAttribute</a>	어셈블리의 버전을 지정합니다.
<a href="#">AssemblyCultureAttribute</a>	어셈블리에서 지원하는 문화권을 지정합니다.
<a href="#">AssemblyFlagsAttribute</a>	어셈블리가 같은 컴퓨터, 같은 프로세스 또는 같은 애플리케이션 도메인에서 Side-by-side 실행을 지원하는지 지정합니다.

## 정보 특성

정보 특성을 사용하여 어셈블리와 연관된 회사 또는 제품에 대한 추가적인 정보를 제공합니다. 다음 표에서는 [System.Reflection](#) 네임스페이스에 정의된 정보 특성을 보여 줍니다.

특성	용도
<a href="#">AssemblyProductAttribute</a>	어셈블리 매니페스트의 제품 이름을 지정합니다.

특성	용도
<a href="#">AssemblyTrademarkAttribute</a>	어셈블리 매니페스트의 상표를 지정합니다.
<a href="#">AssemblyInformationalVersionAttribute</a>	어셈블리 매니페스트의 정보 버전을 지정합니다.
<a href="#">AssemblyCompanyAttribute</a>	어셈블리 매니페스트의 회사 이름을 지정합니다.
<a href="#">AssemblyCopyrightAttribute</a>	어셈블리 매니페스트에 대한 저작권을 지정하는 사용자 지정 특성을 정의합니다.
<a href="#">AssemblyFileVersionAttribute</a>	Win32 파일 버전 리소스의 특정 버전 번호를 설정합니다.
<a href="#">CLSCompliantAttribute</a>	어셈블리가 CLS(공용 언어 사양)을 준수하는지 여부를 나타냅니다.

## 어셈블리 매니페스트 특성

어셈블리 매니페스트 특성을 사용하여 어셈블리 매니페스트의 정보를 제공할 수 있습니다. 특성에는 제목, 설명, 기본 별칭 및 구성이 포함됩니다. 다음 표에서는 [System.Reflection](#) 네임스페이스에 정의된 어셈블리 매니페스트 특성을 보여 줍니다.

특성	용도
<a href="#">AssemblyTitleAttribute</a>	어셈블리 매니페스트의 어셈블리 제목을 지정합니다.
<a href="#">AssemblyDescriptionAttribute</a>	어셈블리 매니페스트의 어셈블리 설명을 지정합니다.
<a href="#">AssemblyConfigurationAttribute</a>	어셈블리 매니페스트의 어셈블리 구성(예: 정품 또는 디버그)을 지정합니다.
<a href="#">AssemblyDefaultAliasAttribute</a>	어셈블리 매니페스트에 대한 친숙한 기본 별칭을 정의합니다.

# 예약된 특성: ConditionalAttribute, ObsoleteAttribute, AttributeUsageAttribute

2021-02-18 • 13 minutes to read • [Edit Online](#)

이 특성은 코드의 요소에 적용될 수 있습니다. 해당 요소에 의미 체계 의미를 추가합니다. 컴파일러는 해당 의미 체계 의미를 사용하여 출력을 변경하고 코드를 사용하여 개발자의 가능한 실수를 보고합니다.

## Conditional 특성

`Conditional` 특성을 사용하면 메서드 실행이 전처리 식별자에 따라 달라집니다. `Conditional` 특성은 `ConditionalAttribute`의 별칭이고 메서드 또는 특성 클래스에 적용할 수 있습니다.

다음 예제에서 `Conditional`은 프로그램 관련 진단 정보 표시를 사용하거나 사용하지 않도록 설정하는 메서드에 적용됩니다.

```
#define TRACE_ON
using System;
using System.Diagnostics;

namespace AttributeExamples
{
    public class Trace
    {
        [Conditional("TRACE_ON")]
        public static void Msg(string msg)
        {
            Console.WriteLine(msg);
        }
    }

    public class TraceExample
    {
        public static void Main()
        {
            Trace.Msg("Now in Main...");
            Console.WriteLine("Done.");
        }
    }
}
```

`TRACE_ON` 식별자가 정의되지 않으면 추적 출력이 표시되지 않습니다. 대화형 창에서 직접 살펴보세요.

`Conditional` 특성은 보통 `DEBUG` 식별자와 함께 사용하여 다음 예제에 표시된 대로 릴리스 빌드가 아닌 디버그 빌드의 추적 및 로깅 기능을 사용하도록 설정합니다.

```
[Conditional("DEBUG")]
static void DebugMethod()
{
}
```

조건부로 표시된 메서드를 호출하면 지정된 전처리 기호가 있는지 여부에 따라 호출을 포함 또는 생략할지 결정됩니다. 기호가 정의되면 호출이 포함되고, 정의되지 않으면 호출이 생략됩니다. 조건부 메서드는 클래스 또는 구조체 선언의 메서드여야 하며 `void` 반환 형식을 포함해야 합니다. 메서드를 `#if...#endif` 블록 내부에 포함하는 것보다 `Conditional`을 사용하는 것이 더 분명하고 더 정교하며 오류 가능성성이 더 작습니다.

한 메서드에 여러 `Conditional` 특성이 있을 경우 하나 이상의 조건부 기호가 정의되면(기호가 OR 연산자를 통해 논리적으로 함께 연결되면) 메서드 호출이 포함됩니다. 다음 예제에서는 `A` 또는 `B`가 있으면 메서드 호출이 발생합니다.

```
[Conditional("A"), Conditional("B")]
static void DoIfAorB()
{
    // ...
}
```

### 특성 클래스와 함께 `Conditional` 사용

`Conditional` 특성을 특성 클래스 정의에 적용할 수도 있습니다. 다음 예제에서 사용자 지정 특성 `Documentation`은 `DEBUG`가 정의된 경우에만 메타데이터에 정보를 추가합니다.

```
[Conditional("DEBUG")]
public class DocumentationAttribute : System.Attribute
{
    string text;

    public DocumentationAttribute(string text)
    {
        this.text = text;
    }
}

class SampleClass
{
    // This attribute will only be included if DEBUG is defined.
    [Documentation("This method displays an integer.")]
    static void DoWork(int i)
    {
        System.Console.WriteLine(i.ToString());
    }
}
```

## Obsolete 특성

`Obsolete` 특성은 코드 요소를 더 이상 사용이 권장되지 않는 항목으로 표시합니다. 사용되지 않음으로 표시된 엔터티를 사용하면 경고나 오류가 생성됩니다. `Obsolete` 특성은 단일 사용 특성이고 특성을 허용하는 모든 엔터티에 적용할 수 있습니다. `Obsolete`는 `ObsoleteAttribute`의 별칭입니다.

다음 예제에서는 `Obsolete` 특성이 `A` 클래스 및 `B.OldMethod` 메서드에 적용됩니다. `B.OldMethod`에 적용된 특성 생성자의 두 번째 인수가 `true`로 설정되므로 이 메서드는 컴파일러 오류를 일으키지만, `A` 클래스를 사용하면 경고가 생성됩니다. 그러나 `B.NewMethod`를 호출하면 경고나 오류가 생성되지 않습니다. 예를 들어 이전 정의와 함께 사용할 경우 다음 코드에서는 두 개의 경고 및 하나의 오류가 생성됩니다.

```

using System;

namespace AttributeExamples
{
    [Obsolete("use class B")]
    public class A
    {
        public void Method() { }

    }

    public class B
    {
        [Obsolete("use NewMethod", true)]
        public void OldMethod() { }

        public void NewMethod() { }
    }

    public static class ObsoleteProgram
    {
        public static void Main()
        {
            // Generates 2 warnings:
            A a = new A();

            // Generate no errors or warnings:
            B b = new B();
            b.NewMethod();

            // Generates an error, compilation fails.
            // b.OldMethod();
        }
    }
}

```

특성 생성자에 첫 번째 인수로 제공된 문자열은 경고 또는 오류의 일부로 표시됩니다. `A` 클래스에 대한 두 개의 경고가 각각 클래스 참조 선언 및 클래스 생성자에 대해 생성됩니다. `Obsolete` 특성은 인수 없이 사용할 수 있지만 대신 사용이 권장되는 항목에 대한 설명을 포함합니다.

## AttributeUsage 특성

`AttributeUsage` 특성은 사용자 지정 특성 클래스를 사용하는 방법을 결정합니다. `AttributeUsageAttribute`는 사용자 지정 특성 정의에 적용되는 특성입니다. `AttributeUsage` 특성을 사용하면 다음을 제어할 수 있습니다.

- 적용할 수 있는 프로그램 요소 특성 사용을 제한하지 않는 한, 다음과 같은 프로그램 요소 중 하나에 특성을 적용할 수 있습니다.
  - 어셈블리(assembly)
  - 모듈(module)
  - 필드(field)
  - event
  - 메서드(method)
  - 매개변수(param)
  - 속성(property)
  - 반환값(return)
  - 형식(type)
- 특성을 단일 프로그램 요소에 여러 번 적용할 수 있는지 여부
- 특성이 파생 클래스에게 상속되는지 여부

기본 설정을 명시적으로 적용할 경우 다음 예제와 같이 작성합니다.

```
[AttributeUsage(AttributeTargets.All,
    AllowMultiple = false,
    Inherited = true)]
class NewAttribute : Attribute { }
```

이 예제에서 `NewAttribute` 클래스는 모든 지원되는 프로그램 요소에 적용할 수 있습니다. 하지만 각 엔터티에 한 번만 적용할 수 있습니다. 이 특성은 기본 클래스에 적용될 때 파생 클래스에게 상속됩니다.

`AllowMultiple` 및 `Inherited` 인수는 선택 사항이므로 다음 코드는 동일한 효과를 가집니다.

```
[AttributeUsage(AttributeTargets.All)]
class NewAttribute : Attribute { }
```

첫 번째 `AttributeUsageAttribute` 인수는 `AttributeTargets` 열거형의 요소가 하나 이상이어야 합니다. 다음 예제와 같이 OR 연산자를 사용하여 여러 대상 형식을 함께 연결할 수 있습니다.

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field)]
class NewPropertyOrFieldAttribute : Attribute { }
```

C# 7.3부터 특성은 속성 또는 자동 구현 속성의 지원 필드에 적용할 수 있습니다. 특성에 `field` 지정자를 지정하지 않는 한 특성이 속성에 적용됩니다. 두 경우 모두 다음 예제에서 표시됩니다.

```
class MyClass
{
    // Attribute attached to property:
    [NewPropertyOrField]
    public string Name { get; set; }

    // Attribute attached to backing field:
    [field:NewPropertyOrField]
    public string Description { get; set; }
}
```

`AllowMultiple` 인수가 `true`인 경우 다음 예제와 같이 결과 특성을 단일 엔터티에 두 번 이상 적용할 수 있습니다.

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
class MultiUse : Attribute { }

[MultiUse]
[MultiUse]
class Class1 { }

[MultiUse, MultiUse]
class Class2 { }
```

이 경우 `AllowMultiple`이 `true`로 설정되므로 `MultiUseAttribute`를 반복적으로 적용할 수 있습니다. 여러 특성을 적용하기 위해 표시된 두 형식이 모두 유효합니다.

`Inherited`이 `false`인 경우 특성은 특성 클래스에서 파생된 클래스에서 상속하지 않습니다. 예를 들어:

```
[AttributeUsage(AttributeTargets.Class, Inherited = false)]
class NonInheritedAttribute : Attribute { }

[NonInherited]
class BClass { }

class DClass : BClass { }
```

이 경우에 `NonInheritedAttribute` 은 상속을 통해 `DClass`에 적용되지 않습니다.

## 참조

- [Attribute](#)
- [System.Reflection](#)
- [특성](#)
- [리플렉션](#)

# 예약된 특성: 호출자 정보 확인

2020-04-23 • 7 minutes to read • [Edit Online](#)

정보 특성을 사용하여 메서드 호출자에 대한 정보를 가져옵니다. 소스 코드 파일 경로, 소스 코드 줄 번호 및 호출자의 멤버 이름을 가져옵니다. 멤버 호출자 정보를 얻으려면 선택적 매개 변수에 적용되는 특성을 사용합니다. 각 선택적 매개 변수는 기본값을 지정합니다. 다음 표에서는 `System.Runtime.CompilerServices` 네임스페이스에 정의된 호출자 정보 특성을 보여줍니다.

특성	설명	형식
<code>CallerFilePathAttribute</code>	호출자를 포함한 소스 파일의 전체 경로입니다. 전체 경로는 컴파일 시간의 경로입니다.	<code>String</code>
<code>CallerLineNumberAttribute</code>	메서드가 호출되는 소스 파일의 줄 번호입니다.	<code>Integer</code>
<code>CallerMemberNameAttribute</code>	호출자의 메서드 이름 또는 속성 이름입니다.	<code>String</code>

이 정보는 추적, 디버깅을 작성하고 진단 도구를 만드는데 도움이 됩니다. 다음 예제에서는 호출자 정보 특성을 사용하는 방법을 보여 줍니다. `TraceMessage` 메서드에 대한 각 호출에서 호출자 정보는 선택적 매개 변수에 대한 인수로 대체됩니다.

```
public void DoProcessing()
{
    TraceMessage("Something happened.");
}

public void TraceMessage(string message,
    [CallerMemberName] string memberName = "",
    [CallerFilePath] string sourceFilePath = "",
    [CallerLineNumber] int sourceLineNumber = 0)
{
    Trace.WriteLine("message: " + message);
    Trace.WriteLine("member name: " + memberName);
    Trace.WriteLine("source file path: " + sourceFilePath);
    Trace.WriteLine("source line number: " + sourceLineNumber);
}

// Sample Output:
// message: Something happened.
// member name: DoProcessing
// source file path: c:\Visual Studio Projects\CallerInfoCS\CallerInfoCS\Form1.cs
// source line number: 31
```

각각의 선택적 매개 변수에 대한 명시적 기본값을 지정합니다. 선택적 매개 변수로 지정되지 않은 매개 변수에 호출자 정보 특성을 적용할 수 없습니다. 호출자 정보 특성은 매개 변수를 선택적 매개 변수로 만들지 못합니다. 대신, 이런 특성은 인수가 생략될 때 전달되는 기본값에 영향을 미칩니다. 호출자 정보 값은 컴파일 시간에 리터럴로 중간 언어(IL) 내로 내보냅니다. 예외에 대한 `StackTrace` 속성의 결과와 달리 결과가 난독화의 영향을 받지 않습니다. 선택적 인수를 명시적으로 제공하여 호출자 정보를 제어하거나 호출자 정보를 숨길 수 있습니다.

## 멤버 이름

`CallerMemberName` 특성을 사용하여 멤버 이름을 호출된 메서드에 대한 `String` 인수로 지정하는 것을 피할 수

있습니다. 이 기술을 사용하여 이름 바꾸기 리팩터링이 `String` 값을 변경하지 못하는 문제를 피합니다. 이 점은 다음 작업에 특히 유용합니다.

- 추적 및 진단 루틴 사용.
- 데이터를 바인딩 할 때 `INotifyPropertyChanged` 인터페이스 구현. 이 인터페이스에서는 컨트롤에서 업데이트된 정보를 표시할 수 있도록 바운드 컨트롤의 속성이 변경되었음을 알리는 개체의 속성을 사용할 수 있습니다. `CallerMemberName` 특성이 없으면 속성 이름을 리터럴로 지정해야 합니다.

아래 차트는 `CallerMemberName` 특성을 사용할 때 반환되는 멤버 이름을 보여줍니다.

호출 발생 범위	멤버 이름 결과
메서드, 속성 또는 이벤트	호출에서 시작한 메서드, 속성 또는 이벤트의 이름입니다.
생성자	".ctor" 문자열
정적 생성자	".cctor" 문자열
소멸자	"Finalize" 문자열
사용자 정의 연산자 또는 변환	멤버에 대해 생성되는 이름입니다(예: "op>Addition").
특성 생성자	특성이 적용되는 메서드 또는 속성의 이름입니다. 특성이 멤버 내에 있는 어떤 요소인 경우(예: 매개 변수, 반환 값 또는 제네릭 형식 매개 변수) 이 결과는 그 요소와 관련된 멤버의 이름입니다.
포함하는 멤버가 없음(예: 어셈블리 수준 또는 형식에 적용되는 특성)	선택적 매개 변수의 기본값입니다.

## 참조

- 명명된 인수 및 선택적 인수
- [System.Reflection](#)
- [Attribute](#)
- [특성](#)

# 예약된 특성은 컴파일러의 null 상태 정적 분석에 사용됩니다.

2021-02-18 • 42 minutes to read • [Edit Online](#)

null 허용 컨텍스트에서 컴파일러는 코드의 정적 분석을 수행하여 모든 참조 형식 변수의 null 상태를 확인합니다.

- 'null이 아님': 정적 분석을 통해 변수에 null이 아닌 값이 할당되었는지 확인합니다.
- 'null일 수 있음': 정적 분석에서 변수에 null이 아닌 값이 할당되었는지 확인할 수 없습니다.

API의 의미 체계에 대한 정보를 컴파일러에 제공하는 특성을 적용할 수 있습니다. 이 정보는 컴파일러에서 정적 분석을 수행하고 변수가 null이 아닌 경우를 확인하는데 도움이 됩니다. 이 문서에서는 각 특성 및 사용 방법에 대해 간단하게 설명합니다. 모든 예제에서는 C# 8.0 이상을 사용한다고 가정하며 코드는 null 허용 컨텍스트에 있습니다.

친숙한 예제부터 살펴보겠습니다. 라이브러리에 리소스 문자열을 검색하는 다음 API가 있다고 가정합니다.

```
bool TryGetMessage(string key, out string message)
```

앞의 예제는 .NET의 친숙한 `Try*` 패턴을 따릅니다. 이 API에 대한 두 개의 참조 인수인 `key` 및 `message` 매개 변수가 있습니다. 이 API에는 해당 인수의 nullness에 관련된 다음 규칙이 있습니다.

- 호출자는 `null`을 `key`의 인수로 전달하지 않아야 합니다.
- 호출자는 값이 `null`인 변수를 `message`의 인수로 전달할 수 있습니다.
- `TryGetMessage` 메서드가 `true`를 반환하면 `message` 값은 null이 아닙니다. 반환 값이 `false`, 이면 `message`의 값과 해당 null 상태는 null입니다.

`key`에 대한 규칙은 변수 형식으로 표현될 수 있습니다. `key`는 null을 허용하지 않는 참조 형식이어야 합니다. `message` 매개 변수는 더 복잡합니다. `null`을 인수로 허용하지만, 성공 시 해당 `out` 인수가 null이 아님을 보장합니다. 이 시나리오의 경우 기대치를 설명하는 더 다양한 어휘가 필요합니다.

변수의 null 상태에 대한 추가 정보를 표현하기 위해 여러 가지 특성이 추가되었습니다. C# 8에 null 허용 참조 형식을 도입하기 전에 작성한 모든 코드는 'null 인식 불가능'이었습니다. 즉, 모든 참조 형식 변수가 null일 수 있지만 null 검사가 필요하지 않습니다. 코드가 'null 허용 인식'이 되면 해당 규칙이 변경됩니다. 참조 형식은 `null` 값이 아니어야 하며 null 허용 참조 형식은 참조되기 전에 `null`에 대해 검사되어야 합니다.

API에 대한 규칙은 `TryGetValue` API 시나리오에서 살펴본 것처럼 더 복잡할 수 있습니다. 대부분의 API에는 변수가 `null`일 수 있거나 없는 경우에 대한 더 복잡한 규칙이 있습니다. 이 경우에는 다음 특성 중 하나를 사용하여 해당 규칙을 표현합니다.

- `AllowNull`: null을 허용하지 않는 입력 인수는 null일 수 있습니다.
- `DisallowNull`: null 허용 입력 인수는 null이 아니어야 합니다.
- `MaybeNull`: null을 허용하지 않는 반환 값은 null일 수 있습니다.
- `NotNull`: null 허용 반환 값은 null이 아닙니다.
- `MaybeNullWhen`: 메서드가 지정된 `bool` 값을 반환하는 경우 null을 허용하지 않는 입력 인수는 null일 수 있습니다.
- `NotNullWhen`: 메서드가 지정된 `bool` 값을 반환하는 경우 null 허용 입력 인수는 null이 아닙니다.
- `NotNullIfNotNull`: 지정된 매개 변수의 인수가 null이 아닌 경우 반환 값은 null이 아닙니다.
- `DoesNotReturn`: 메서드는 값을 반환하지 않습니다. 즉, 항상 예외를 throw합니다.

- **DoesNotReturnIf**: 연결된 `bool` 매개 변수에 지정된 값이 있는 경우 이 메서드는 값을 반환하지 않습니다.
- **MemberNotNull**: 메서드가 반환하는 경우 나열된 멤버는 `null`이 아닙니다.
- **MemberNotNullWhen**: 메서드가 지정된 `bool` 값을 반환하는 경우 나열된 멤버는 `null`이 아닙니다.

앞의 설명은 각 특성이 수행하는 작업에 대한 빠른 참조입니다. 다음 각 섹션에서는 동작과 의미에 대해 더 자세히 설명합니다.

이 특성을 추가하면 API 규칙에 대한 자세한 정보가 컴파일러에 제공됩니다. 호출 코드가 `null` 허용 사용 가능 컨텍스트에서 컴파일되는 경우 컴파일러는 해당 규칙을 위반할 때 호출자에게 경고를 표시합니다. 이러한 특성은 구현에 대한 더 많은 검사를 사용하지 않습니다.

## 전제 조건 지정: `AllowNull` 및 `DisallowNull`

읽기/쓰기 속성에는 적절한 기본값이 있으므로 `null`을 반환하지 않는 읽기/쓰기 속성을 살펴봅니다. 호출자는 속성을 해당 기본값으로 설정할 때 `set` 접근자에 `null`을 전달합니다. 예를 들어 대화방에서 화면 이름을 요청하는 메시지 시스템을 살펴봅니다. 아무것도 제공되지 않으면 시스템은 임의 이름을 생성합니다.

```
public string ScreenName
{
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName;
```

`null` 허용 인식 불가능 컨텍스트에서 이전 코드를 컴파일하면 정상적으로 작동합니다. `null` 허용 참조 형식을 사용하도록 설정하면 `ScreenName` 속성이 `null`을 허용하지 않는 참조가 됩니다. `get` 접근자의 경우도 마찬가지입니다. 이 접근자는 `null`을 반환하지 않습니다. 호출자는 `null`에 대해 반환된 속성을 검사할 필요가 없습니다. 그러나 지금 속성을 `null`로 설정하면 경고가 생성됩니다. 이 형식의 코드를 지원하려면 다음 코드와 같이 속성에 [System.Diagnostics.CodeAnalysis.AllowNullAttribute](#) 특성을 추가합니다.

```
[AllowNull]
public string ScreenName
{
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName = GenerateRandomScreenName();
```

이 특성과 이 문서에 설명된 다른 특성을 사용하려면 [System.Diagnostics.CodeAnalysis](#)에 대한 `using` 지시문을 추가해야 할 수 있습니다. 이 특성은 `set` 접근자가 아니라 속성에 적용됩니다. `AllowNull` 특성은 '전제 조건'을 지정하며 입력에만 적용됩니다. `get` 접근자에는 반환 값이 있지만 입력 인수가 없습니다. 따라서 `AllowNull` 특성은 `set` 접근자에만 적용됩니다.

앞의 예제에서는 인수에 대한 `AllowNull` 특성을 추가할 때 검색할 항목을 보여 줍니다.

1. 해당 변수에 대한 일반 계약은 `null`이 아니어야 하므로 `null`을 허용하지 않는 참조 형식이 필요합니다.
2. 가장 일반적인 사용은 아니지만 입력 변수가 `null`이 되는 시나리오가 있습니다.

일반적으로 속성이나 `in`, `out` 및 `ref` 인수의 경우 이 특성이 필요합니다. `AllowNull` 특성은 일반적으로 변수가 `null`이 아닌 경우에 가장 적합하지만, `null`을 전제 조건으로 허용해야 합니다.

`DisallowNull` 사용에 대한 시나리오와 비교해 보세요. 이 특성을 사용하여 `null` 허용 참조 형식의 입력 변수가 `null`이 아니도록 지정합니다. 기본값이 `null`이지만 클라이언트에서 `null`이 아닌 값으로만 설정할 수 있는 속성을 살펴봅니다. 다음 코드를 살펴보세요.

```

public string ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new ArgumentNullException(nameof(value), "Cannot set to null");
}
string _comment;

```

이전 코드는 `ReviewComment` 가 `null` 일 수 있지만 `null`로 설정될 수 없는 의도를 표현하는 가장 좋은 방법입니다. 이 코드가 `null` 허용 인식이 되면 `System.Diagnostics.CodeAnalysis.DisallowNullAttribute`를 사용하여 호출자에게 이 개념을 보다 명확하게 표현할 수 있습니다.

```

[DisallowNull]
public string? ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new ArgumentNullException(nameof(value), "Cannot set to null");
}
string? _comment;

```

`null` 허용 컨텍스트에서 `ReviewComment` `get` 접근자는 `null`의 기본값을 반환할 수 있습니다. 컴파일러는 액세스하기 전에 검사해야 한다는 경고를 표시합니다. 또한 `null` 일 수 있는 경우에도 호출자가 명시적으로 `null`로 설정하지 않아야 한다는 경고를 호출자에게 표시합니다. 또한 `DisallowNull` 특성은 전제 조건을 지정하며 `get` 접근자에는 영향을 주지 않습니다. 다음에 대해 이 특성을 관찰할 때 `DisallowNull` 특성을 사용합니다.

1. 변수는 처음 인스턴스화될 때 주로 핵심 시나리오에서 `null` 일 수 있습니다.
2. 변수를 명시적으로 `null`로 설정하면 안 됩니다.

이 상황은 원래 'null 인식 불가능'이었던 코드에서 일반적입니다. 개체 속성은 두 개의 개별 초기화 작업에서 설정될 수 있습니다. 일부 속성은 일부 비동기 작업이 완료된 후에만 설정될 수 있습니다.

`AllowNull` 및 `DisallowNull` 특성을 사용하여 변수에 대한 전제 조건이 해당 변수의 `null` 허용 주석과 일치하지 않을 수 있도록 지정할 수 있습니다. 이 특성은 API의 특징에 대한 자세한 정보를 제공합니다. 이 추가 정보는 호출자가 API를 올바르게 사용하는 데 도움이 됩니다. 다음 특성을 사용하여 전제 조건을 지정해야 합니다.

- `AllowNull`: `null`을 허용하지 않는 입력 인수는 `null`일 수 있습니다.
- `DisallowNull`: `null` 허용 입력 인수는 `null`이 아니어야 합니다.

## 사후 조건 지정: `MaybeNull` 및 `NotNull`

다음 시그니처를 사용하는 메서드가 있다고 가정합니다.

```

public Customer FindCustomer(string lastName, string firstName)

```

검색한 이름을 찾을 수 없는 경우 `null`을 반환하도록 이 메서드를 작성했을 수 있습니다. `null`은 분명히 레코드를 찾을 수 없음을 나타냅니다. 이 예제에서는 반환 형식을 `Customer`에서 `Customer?`로 변경할 수 있습니다. 반환 값을 `null` 허용 참조 형식으로 선언하면 이 API의 의도가 분명하게 지정됩니다.

[제네릭 정의 및 null 허용 여부](#)에서 설명하는 이유로 해당 기술은 제네릭 메서드에서 작동하지 않습니다. 비슷한 패턴을 따르는 제네릭 메서드가 있을 수 있습니다.

```

public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)

```

반환 값이 `T?`이도록 지정할 수 없습니다. 검색한 항목을 찾을 수 없는 경우 메서드는 `null`을 반환합니다. `T?` 반환 형식을 선언할 수 없으므로 `MaybeNull` 주석을 메서드 반환에 추가합니다.

```
[return: MaybeNull]
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

이전 코드는 계약이 null을 허용하지 않는 형식을 의미한다는 것을 호출자에게 알리지만, 반환 값은 실제로 null'일 수 있습니다'. API가 null을 허용하지 않는 형식(일반적으로 제네릭 형식 매개 변수)이어야 하지만 null이 반환되는 인스턴스가 있을 수 있는 경우 `MaybeNull` 특성을 사용합니다.

형식이 null 허용 참조 형식인 경우에도 반환 값이나 `out` 또는 `ref` 인수가 null이 아니도록 지정할 수도 있습니다. 배열이 많은 요소를 포함할 만큼 충분히 큰지 확인하는 메서드를 살펴봅니다. 입력 인수에 용량이 없으면 루틴은 새 배열을 할당하고 기존 요소를 모두 새 배열로 복사합니다. 입력 인수가 null이면 루틴은 새 스토리지 를 할당합니다. 충분한 용량이 있는 경우 루틴은 아무 작업도 수행하지 않습니다.

```
public void EnsureCapacity<T>(ref T[] storage, int size)
```

다음과 같이 이 루틴을 호출할 수 있습니다.

```
// messages has the default value (null) when EnsureCapacity is called:
EnsureCapacity<string>(ref messages, 10);
// messages is not null.
EnsureCapacity<string>(messages, 50);
```

null 참조 형식을 사용하도록 설정한 후 이전 코드가 경고 없이 컴파일되는지 확인하려고 합니다. 메서드가 반환하는 경우 `storage` 인수는 null이 아님이 보장됩니다. 그러나 null 참조를 사용하여 `EnsureCapacity` 를 호출할 수 있습니다. `storage` 를 null 허용 참조 형식으로 만들고 `NotNull` 사후 조건을 매개 변수 선언에 추가할 수 있습니다.

```
public void EnsureCapacity<T>([NotNull] ref T[]? storage, int size)
```

이전 코드는 기존 계약을 명확하게 표현합니다. 호출자는 null 값과 함께 변수를 전달할 수 있지만 반환 값은 null이 아님이 보장됩니다. null 이 인수로 전달될 수 있지만 메서드가 반환할 때 해당 인수가 null이 아님이 보장되는 경우 `NotNull` 특성은 `ref` 및 `out` 인수에 가장 유용합니다.

다음 특성을 사용하여 무조건 사후 조건을 지정합니다.

- `MaybeNull`: null을 허용하지 않는 반환 값은 null일 수 있습니다.
- `NotNull`: null 허용 반환 값은 null이 아닙니다.

조건부 사후 조건 지정: `NotNullWhen`, `MaybeNullWhen` 및  
`NotNullIfNotNull`

`string` 메서드 `String.IsNullOrEmpty(String)`에 대해 잘 알고 있을 수 있습니다. 인수가 null이거나 빈 문자열인 경우 이 메서드는 `true` 를 반환합니다. 이는 null 검사의 한 가지 형태입니다. 메서드가 `false` 를 반환하는 경우 호출자는 인수를 null 검사할 필요가 없습니다. 메서드를 이 null 허용 인식처럼 만들려면 인수를 null 허용 참조 형식으로 설정하고 `NotNullWhen` 특성을 추가합니다.

```
bool IsNullOrEmpty([NotNullWhen(false)] string? value);
```

이를 통해 반환 값이 `false` 인 코드는 null 검사가 필요하지 않음을 컴파일러에 알립니다. 특성을 추가하여 `IsNullOrEmpty` 가 필요한 null 검사를 수행한다는 것을 컴파일러의 정적 분석에 알립니다. `false` 를 반환하는 경우 입력 인수는 null 이 아닙니다.

```

string? userInput = GetUserInput();
if (!string.IsNullOrEmpty(userInput))
{
    int messageLength = userInput.Length; // no null check needed.
}
// null check needed on userInput here.

```

.NET Core 3.0의 경우 위에 표시된 대로 `String.IsNullOrEmpty(String)` 메서드가 주석으로 처리됩니다. `null` 값에 대한 개체의 상태를 확인하는 비슷한 메서드가 코드베이스에 있을 수 있습니다. 컴파일러는 사용자 지정 `null` 검사 메서드를 인식하지 않으며 주석을 직접 추가해야 합니다. 특성을 추가하면 컴파일러의 정적 분석은 테스트 된 변수가 `null` 검사된 시기를 알 수 있습니다.

또한 이 특성은 `Try*` 패턴에 사용됩니다. `ref` 및 `out` 변수에 대한 사후 조건은 반환 값을 통해 전달됩니다. 앞에서 설명한 이 메서드를 살펴봅니다.

```
bool TryGetMessage(string key, out string message)
```

이전 메서드는 일반적인 .NET 관용구를 따릅니다. 반환 값은 `message` 가 발견된 값으로 설정되었는지 또는 메시지가 없는 경우 기본값으로 설정되었는지 여부를 나타냅니다. 메서드가 `true` 를 반환하는 경우 `message` 의 값은 `null`이 아닙니다. 이 값을 반환하지 않는 경우 메서드는 `message` 를 `null`로 설정합니다.

`NotNullWhen` 특성을 사용하여 해당 관용구를 전달할 수 있습니다. `null` 허용 참조 형식의 시그니처를 업데이트 하는 경우 `string?` 을 `message` 로 만들고 특성을 추가합니다.

```
bool TryGetMessage(string key, [NotNullWhen(true)] out string? message)
```

이전 예제에서 `TryGetMessage` 가 `true` 를 반환하는 경우 `message` 값은 `null`이 아닌 것으로 알려져 있습니다. 동일한 방식으로 코드베이스에서 비슷한 메서드를 주석으로 처리해야 합니다. 인수는 `null` 일 수 있으며 메서드가 `true` 를 반환하는 경우 `null`이 아닌 것으로 알려져 있습니다.

한 가지 마지막 특성이 필요할 수도 있습니다. 경우에 따라 반환 값의 `null` 상태는 하나 이상 입력 인수의 `null` 상태에 따라 달라집니다. 해당 메서드는 특정 입력 인수가 `null` 이 아닐 때마다 `null`이 아닌 값을 반환합니다. 해당 메서드를 주석으로 옮바르게 처리하려면 `NotNullIfNotNull` 특성을 사용합니다. 다음 태그를 살펴봅니다.

```
string GetTopLevelDomainFromFullUrl(string url);
```

`url` 인수가 `null`이 아니면 출력은 `null` 이 아닙니다. `null` 허용 참조가 사용하도록 설정되면 API에서 `null` 입력을 허용하지 않는 경우 해당 시그니처가 제대로 작동합니다. 그러나 입력이 `null`일 수 있는 경우에는 반환 값도 `null`일 수 있습니다. 시그니처를 다음 코드로 변경할 수 있습니다.

```
string? GetTopLevelDomainFromFullUrl(string? url);
```

또한 시그니처가 작동하지만 호출자가 추가 `null` 검사를 강제로 구현하는 경우가 많습니다. 계약은 입력 인수 `url` 이 `null` 인 경우에만 반환 값이 `null` 이라는 것입니다. 해당 계약을 표현하려면 다음 코드와 같이 이 메서드를 주석으로 처리합니다.

```
[return: NotNullIfNotNull("url")]
string? GetTopLevelDomainFromFullUrl(string? url);
```

반환 값 및 인수 중 하나가 `null` 일 수 있음을 나타내는 `?` 를 사용하여 두 항목을 모두 주석으로 처리했습니다. 특성은 `url` 인수가 `null` 이 아닌 경우 반환 값이 `null`이 아님을 명확하게 설명합니다.

해당 특성을 사용하여 조건부 사후 조건을 지정합니다.

- [MaybeNullWhen](#): 메서드가 지정된 `bool` 값을 반환하는 경우 `null`을 허용하지 않는 입력 인수는 `null`일 수 있습니다.
- [NotNullWhen](#): 메서드가 지정된 `bool` 값을 반환하는 경우 `null` 허용 입력 인수는 `null`이 아닙니다.
- [NotNullIfNotNull](#): 지정된 매개 변수의 입력 인수가 `null`이 아닌 경우 반환 값은 `null`이 아닙니다.

## 생성자 도우미 메서드: `MemberNotNull` 및 `MemberNotNullWhen`

이러한 특성은 생성자에서 공용 코드를 도우미 메서드로 리팩터링할 때 의도를 지정합니다. C# 컴파일러는 생성자 및 필드 이니셜라이저를 분석하여 각 생성자를 반환하기 전에 `null`을 허용하지 않는 모든 참조 필드가 초기화되도록 합니다. 그러나 C# 컴파일러가 모든 도우미 메서드에서 필드 할당을 추적하지는 않습니다. 컴파일러는 필드가 생성자에서 직접 초기화되지 않고 도우미 메서드에서 초기화되는 경우 [cs8618](#) 경고를 생성합니다. 메서드 선언에 [MemberNotNullAttribute](#)를 추가하고 메서드에서 Null이 아닌 값으로 초기화되는 필드를 지정합니다. 예를 들어 다음 예제를 고려해 보겠습니다.

```
public class Container
{
    private string _uniqueIdentifier; // must be initialized.
    private string? _optionalMessage;

    public Container()
    {
        Helper();
    }

    public Container(string message)
    {
        Helper();
        _optionalMessage = message;
    }

    [MemberNotNull(nameof(_uniqueIdentifier))]
    private void Helper()
    {
        _uniqueIdentifier = DateTime.Now.Ticks.ToString();
    }
}
```

`MemberNotNull` 특성 생성자에 대한 인수로 여러 필드 이름을 지정할 수 있습니다.

[MemberNotNullWhenAttribute](#)에는 `bool` 인수가 있습니다. 도우미 메서드가 도우미 메서드에서 필드를 초기화했는지 여부를 나타내는 `bool`을 반환하는 경우 `MemberNotNullWhen`을 사용합니다.

## 접근할 수 없는 코드 확인

일반적으로 예외 도우미 또는 기타 유ти리티 메서드와 같은 일부 메서드는 항상 예외를 `throw`하여 종료됩니다. 또는 도우미는 부울 인수 값을 기반으로 예외를 `throw`할 수 있습니다.

첫 번째 경우에 [DoesNotReturn](#) 특성을 메서드 선언에 추가할 수 있습니다. 컴파일러는 세 가지 방법으로 도움이 됩니다. 첫째, 예외를 `throw`하지 않고 메서드가 종료될 수 있는 경로가 있는 경우 컴파일러는 경고를 생성합니다. 둘째, 컴파일러는 해당 메서드를 호출한 후 적절한 `catch` 절이 나타날 때까지 코드를 접근할 수 없음으로 표시합니다. 셋째, 접근할 수 없는 코드는 `null` 상태에 영향을 주지 않습니다. 이 메서드를 살펴봅니다.

```

[DoesNotReturn]
private void FailFast()
{
    throw new InvalidOperationException();
}

public void SetState(object containedField)
{
    if (!isInitialized)
    {
        FailFast();
    }

    // unreachable code:
    _field = containedField;
}

```

두 번째 경우에는 메서드의 부울 매개 변수에 `DoesNotReturnIf` 특성을 추가합니다. 이전 예제를 다음과 같이 수정할 수 있습니다.

```

private void FailFast([DoesNotReturnIf(false)] bool isValid)
{
    if (!isValid)
    {
        throw new InvalidOperationException();
    }
}

public void SetState(object containedField)
{
    FailFast(isInitialized);

    // unreachable code when "isInitialized" is false:
    _field = containedField;
}

```

## 요약

### IMPORTANT

공식 설명서는 최신 C# 버전을 추적합니다. 현재 C# 9.0에 대해 작성하고 있습니다. 사용 중인 C# 버전에 따라, 다양한 기능을 사용하지 못할 수도 있습니다. 프로젝트에 대한 '기본' C# 버전은 대상 프레임워크를 기반으로 합니다. 자세한 내용은 [C# 언어 버전 관리 기본값](#)을 참조하세요.

null 허용 참조 형식을 추가하면 `null` 일 수 있는 변수의 API 기대치를 설명하는 초기 어휘가 제공됩니다. 특성은 변수의 null 상태를 전제 조건 및 사후 조건으로 설명하는 다양한 어휘를 제공합니다. 해당 특성은 기대치를 명확하게 설명하며 API를 사용하는 개발자에게 더 나은 환경을 제공합니다.

null 허용 컨텍스트의 라이브러리를 업데이트할 때 해당 특성을 추가하여 API 사용자가 올바르게 사용하도록 안내합니다. 해당 특성을 통해 입력 인수 및 반환 값의 null 상태를 완벽하게 설명할 수 있습니다.

- [AllowNull](#): null을 허용하지 않는 입력 인수는 null일 수 있습니다.
- [DisallowNull](#): null 허용 입력 인수는 null이 아니어야 합니다.
- [MaybeNull](#): null을 허용하지 않는 반환 값은 null일 수 있습니다.
- [NotNull](#): null 허용 반환 값은 null이 아닙니다.
- [MaybeNullWhen](#): 메서드가 지정된 `bool` 값을 반환하는 경우 null을 허용하지 않는 입력 인수는 null일 수 있습니다.

- [NotNullWhen](#): 메서드가 지정된 `bool` 값을 반환하는 경우 `null` 허용 입력 인수는 `null`이 아닙니다.
- [NotNullIfNotNull](#): 지정된 매개 변수의 입력 인수가 `null`이 아닌 경우 반환 값은 `null`이 아닙니다.
- [DoesNotReturn](#): 메서드는 값을 반환하지 않습니다. 즉, 항상 예외를 `throw`합니다.
- [DoesNotReturnIf](#): 연결된 `bool` 매개 변수에 지정된 값이 있는 경우 이 메서드는 값을 반환하지 않습니다.

# C# 전처리기 지시문

2021-02-18 • 2 minutes to read • [Edit Online](#)

이 단원에서는 다음 C# 전처리기 지시문에 대한 정보를 제공합니다.

- [#if](#)
- [#else](#)
- [#elif](#)
- [#endif](#)
- [#define](#)
- [#undef](#)
- [#warning](#)
- [#error](#)
- [#line](#)
- [#nullable](#)
- [#region](#)
- [#endregion](#)
- [#pragma](#)
- [#pragma warning](#)
- [#pragma checksum](#)

자세한 내용과 예제는 각 항목을 참조하세요.

컴파일러에는 별도의 전처리기가 없지만, 이 단원에 설명된 지시문은 전처리기가 있는 것처럼 처리됩니다. 지시문은 조건부 컴파일에서 지원하기 위해 사용됩니다. C 및 C++ 지시문과 달리, 매크로를 만들기 위해 이러한 지시문을 사용할 수 없습니다.

전처리기 지시문은 한 줄에서 유일한 명령이어야 합니다.

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)

# #if(C# 참조)

2020-11-02 • 7 minutes to read • [Edit Online](#)

C# 컴파일러는 `#if` 지시문과 `#endif` 지시문이 차례로 확인되면 지정된 기호가 정의되어 있어야 지시문 사이의 코드를 컴파일합니다. C 및 C++와 달리, 기호에 숫자 값을 할당할 수 없습니다. C#의 `#if` 문은 부울이고, 기호가 정의되었는지 여부만 테스트합니다. 예:

```
#if DEBUG
    Console.WriteLine("Debug version");
#endif
```

`==(같음)` 및 `!=(같지 않음)` 연산자는 `bool` 값 `true` 또는 `false`를 테스트할 때만 사용할 수 있습니다. `true` 가 반환되면 기호가 정의된 것입니다. `#if DEBUG` 문의 의미는 `#if (DEBUG == true)` 와 같습니다. `&&(and), ||(or)` 및 `!(not)` 연산자를 사용하여 여러 기호가 정의되었는지 여부를 평가할 수 있습니다. 기호와 연산자를 괄호로 묶을 수도 있습니다.

## 설명

`#if` 를 `#else`, `#elif`, `#endif`, `#define` 및 `#undef` 지시문과 함께 사용하면 하나 이상의 기호 유무에 따라 코드를 포함하거나 제거할 수 있습니다. 따라서 코드를 디버그 빌드용으로 컴파일하거나 특정 구성용으로 컴파일할 때 유용할 수 있습니다.

`#if` 지시문으로 시작되는 조건부 지시문은 `#endif` 지시문을 사용하여 명시적으로 종료해야 합니다.

`#define` 을 사용하여 기호를 정의할 수 있습니다. 그러면 `#if` 지시문으로 전달되는 식으로 해당 기호를 사용하는 경우 식이 `true` 로 평가됩니다.

`-define` 컴파일러 옵션으로 기호를 정의할 수도 있습니다. `#undef`로 기호 정의를 해제할 수 있습니다.

`-define` 또는 `#define` 으로 정의하는 기호는 동일한 이름의 변수와 충돌하지 않습니다. 즉, 변수 이름이 전처리기 지시문에 전달되지 않아야 하며 전처리기 지시문을 통해서만 기호를 평가할 수 있습니다.

`#define` 을 사용하여 만든 기호의 범위는 해당 기호가 정의된 파일입니다.

빌드 시스템은 SDK 스타일 프로젝트의 여러 대상 프레임워크를 나타내는 미리 정의된 전처리기 기호도 인식합니다. 둘 이상의 .NET 버전을 대상으로 지정할 수 있는 애플리케이션을 만들 때 유용합니다.

대상 프레임워크	기호
.NET Framework	NETFRAMEWORK, NET48, NET472, NET471, NET47, NET462, NET461, NET46, NET452, NET451, NET45, NET40, NET35, NET20
.NET 표준	NETSTANDARD, NETSTANDARD2_1, NETSTANDARD2_0, NETSTANDARD1_6, NETSTANDARD1_5, NETSTANDARD1_4, NETSTANDARD1_3, NETSTANDARD1_2, NETSTANDARD1_1, NETSTANDARD1_0
.NET 5(및 .NET Core)	NET5_0, NETCOREAPP, NETCOREAPP3_1, NETCOREAPP3_0, NETCOREAPP2_2, NETCOREAPP2_1, NETCOREAPP2_0, NETCOREAPP1_1, NETCOREAPP1_0

## NOTE

기존의 비 SDK 스타일 프로젝트의 경우 프로젝트의 속성 페이지를 통해 Visual Studio의 여러 대상 프레임워크에 대한 조건부 컴파일 기호를 수동으로 구성해야 합니다.

기타 미리 정의된 기호에는 DEBUG와 TRACE 상수가 있습니다. `#define` 을 사용하여 프로젝트에 설정된 값을 재정의할 수 있습니다. 예를 들어 DEBUG 기호는 빌드 구성 특성("디버그" 또는 "릴리스" 모드)에 따라 자동으로 설정됩니다.

## 예

다음 예제에서는 파일에 MYTEST 기호를 정의한 다음 MYTEST 및 DEBUG 기호의 값을 테스트하는 방법을 보여줍니다. 이 예제의 출력은 디버그 구성 모드에서 프로젝트를 빌드했는지, 릴리스 구성 모드에서 프로젝트를 빌드했는지에 따라 다릅니다.

```
#define MYTEST
using System;
public class MyClass
{
    static void Main()
    {
#define DEBUG
#define !MYTEST
        Console.WriteLine("DEBUG is defined");
#define !DEBUG
#define MYTEST
        Console.WriteLine("MYTEST is defined");
#define DEBUG
#define MYTEST
        Console.WriteLine("DEBUG and MYTEST are defined");
#undef DEBUG
#undef MYTEST
        Console.WriteLine("DEBUG and MYTEST are not defined");
#endif
    }
}
```

다음 예제에서는 가능한 경우 새 API를 사용할 수 있도록 여러 대상 프레임워크에 대해 테스트하는 방법을 보여줍니다.

```
public class MyClass
{
    static void Main()
    {
#define NET40
        WebClient _client = new WebClient();
#define !
        HttpClient _client = new HttpClient();
#endif
    }
    //...
}
```

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 전처리기 지시문](#)
- [방법: 추적 및 디버그를 사용한 조건부 컴파일](#)

# #else(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

#else 를 사용하면 복합 조건부 지시문을 만들 수 있습니다. 이 경우 앞의 #if 또는 (선택 사항) #elif 지시문에 true 로 평가하는 식이 없으면 컴파일러는 #else 와 후속 #endif 사이에 있는 모든 코드를 평가합니다.

## 설명

#endif 는 #else 이후의 다음 전처리기 지시문이어야 합니다. #else 를 사용하는 방법에 대한 예제는 #if 를 참조하세요.

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 전처리기 지시문](#)

# #elif(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

#elif 를 사용하면 복합 조건부 지시문을 만들 수 있습니다. #elif 식이 계산되는 경우는 #if와 앞의 선택적 #elif 지시문 식이 모두 true 로 계산되지 않는 경우입니다. #elif 식이 true 이면 컴파일러는 #elif 와 다음 조건부 지시문 사이에 있는 모든 코드를 평가합니다. 예를 들면 다음과 같습니다.

```
#define VC7
//...
#if debug
    Console.WriteLine("Debug build");
#elif VC7
    Console.WriteLine("Visual Studio 7");
#endif
```

== (같음), != (같지 않음), && (AND), || (OR) 연산자를 사용하여 여러 기호를 평가할 수 있습니다. 기호와 연산자를 괄호로 묶을 수도 있습니다.

## 설명

#elif 는 다음 코드를 사용하는 것과 같습니다.

```
#else
#endif
```

#elif 를 사용하는 것이 더 간단합니다. 각 #if 에는 #endif 가 필요한 반면, #elif 는 짹이 되는 #endif 없이 사용할 수 있기 때문입니다.

#elif 를 사용하는 방법에 대한 예제는 #if 를 참조하세요.

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 전처리기 지시문](#)

# #endif(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

`#endif` 는 `#if` 지시문으로 시작한 조건부 지시문의 끝을 지정합니다. 예를 들면 다음과 같습니다.

```
#define DEBUG
// ...
#if DEBUG
    Console.WriteLine("Debug version");
#endif
```

## 설명

`#if` 지시문으로 시작되는 조건부 지시문은 `#endif` 지시문을 사용하여 명시적으로 종료해야 합니다. `#endif`를 사용하는 방법에 대한 예제는 [#if](#)를 참조하세요.

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 전처리기 지시문](#)

# #define(C# 참조)

2020-11-02 • 4 minutes to read • [Edit Online](#)

#define 을 사용하여 기호를 정의합니다. 기호를 #if 지시문에 전달되는 식으로 사용하는 경우 식이 다음 예제와 같이 true 로 평가됩니다.

```
#define DEBUG
```

## 설명

### NOTE

C 및 C++에서 일반적으로 수행하듯이 #define 지시문을 사용하여 상수 값을 선언할 수 없습니다. C#의 상수는 클래스 또는 구조체의 정적 멤버로 가장 잘 정의됩니다. 이러한 상수가 여러 개 있는 경우 상수를 포함할 별도의 "Constants" 클래스를 만드는 것이 좋습니다.

기호를 사용하여 컴파일 조건을 지정할 수 있습니다. #if 또는 #elif를 사용하여 기호를 테스트할 수 있습니다. [ConditionalAttribute](#)를 사용하여 조건부 컴파일을 수행 할 수도 있습니다.

기호를 정의할 수 있지만 기호에 값을 할당할 수는 없습니다. #define 지시문은 파일에서 전처리기 지시문이 아닌 명령을 사용하기 전에 나와야 합니다.

-define 컴파일러 옵션으로 기호를 정의할 수도 있습니다. #undef로 기호 정의를 해제 할 수 있습니다.

-define 또는 #define 으로 정의하는 기호는 동일한 이름의 변수와 충돌하지 않습니다. 즉, 변수 이름이 전처리기 지시문에 전달되지 않아야 하며 전처리기 지시문을 통해서만 기호를 평가할 수 있습니다.

#define 을 사용하여 만든 기호의 범위는 기호가 정의된 파일입니다.

다음 예제와 같이, #define 지시문을 파일 맨 위에 배치해야 합니다.

```
#define DEBUG
// #define TRACE
#undef TRACE

using System;

public class TestDefine
{
    static void Main()
    {
        #if (DEBUG)
            Console.WriteLine("Debugging is enabled.");
        #endif

        #if (TRACE)
            Console.WriteLine("Tracing is enabled.");
        #endif
    }
    // Output:
    // Debugging is enabled.
```

기호 정의를 해제하는 방법의 예는 [#undef](#)를 참조하세요.

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 전처리기 지시문](#)
- [const](#)
- [방법: 추적 및 디버그를 사용한 조건부 컴파일](#)
- [#undef](#)
- [#if](#)

# #undef(C# 참조)

2021-02-18 • 2 minutes to read • [Edit Online](#)

#**undef** 를 사용하면 기호의 정의를 해제할 수 있습니다. 그러면 #**if** 지시문에서 해당 기호를 식으로 사용하여 식이 **false**로 평가됩니다.

#**define** 지시문 또는 -**define** 컴파일러 옵션을 사용하여 기호를 정의할 수 있습니다. #**undef** 지시문은 지시문이 아닌 문을 사용하기 전에 파일에 나와야 합니다.

## 예제

```
// preprocessor_undef.cs
// compile with: /d:DEBUG
#undef DEBUG
using System;
class MyClass
{
    static void Main()
    {
#define DEBUG
        Console.WriteLine("DEBUG is defined");
#undef DEBUG
        Console.WriteLine("DEBUG is not defined");
#endif
    }
}
```

디버그가 정의되어 있지 않습니다.

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 전처리기 지시문](#)

# #warning(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

#warning 을 사용하면 코드의 특정 위치에서 CS1030 수준 1 컴파일러 경고를 생성할 수 있습니다. 다음은 그 예입니다.

```
#warning Deprecated code in this method.
```

## 설명

#warning 는 일반적으로 조건부 지시문에 사용됩니다. #error를 사용하여 사용자 정의 오류를 생성할 수도 있습니다.

## 예제

```
// preprocessor_warning.cs
// CS1030 expected
#define DEBUG
class MainClass
{
    static void Main()
    {
#if DEBUG
#warning DEBUG is defined
#endif
    }
}
```

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 전처리기 지시문](#)

# #error(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

#error 를 사용하면 코드의 특정 위치에서 CS1029 사용자 정의 오류를 생성할 수 있습니다. 예:

```
#error Deprecated code in this method.
```

## NOTE

컴파일러는 #error version 을 특수한 방식으로 처리하며 사용된 컴파일러 및 언어 버전을 포함한 메시지가 있는 컴파일러 오류 CS8304를 보고합니다.

## 설명

#error 는 일반적으로 조건부 지시문에 사용됩니다.

#warning 을 사용하여 사용자 정의 경고를 생성할 수도 있습니다.

## 예제

```
// preprocessor_error.cs
// CS1029 expected
#define DEBUG
class MainClass
{
    static void Main()
    {
#if DEBUG
#error DEBUG is defined
#endif
    }
}
```

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 전처리기 지시문](#)

# #line(C# 참조)

2020-11-02 • 6 minutes to read • [Edit Online](#)

#line을 사용하면 오류 및 경고에 대한 컴파일러의 줄 번호 매기기 및 파일 이름 출력(선택 사항)을 수정할 수 있습니다.

다음 예제에서는 줄 번호와 관련된 두 개의 경고를 보고하는 방법을 보여 줍니다. #line 200 지시문은 다음 줄 번호를 강제로 200(기본값은 #6임)으로 설정하며, 다음 #line 지시문까지 파일 이름이 "Special"로 보고됩니다. #line default 지시문은 줄 번호 매기기를 기본 번호 매기기로 되돌립니다. 이 경우 이전 지시문을 통해 번호가 다시 매겨진 줄이 계산됩니다.

```
class MainClass
{
    static void Main()
    {
#line 200 "Special"
        int i;
        int j;
#line default
        char c;
        float f;
#line hidden // numbering not affected
        string s;
        double d;
    }
}
```

컴파일은 다음 출력을 생성합니다.

```
Special(200,13): warning CS0168: The variable 'i' is declared but never used
Special(201,13): warning CS0168: The variable 'j' is declared but never used
MainClass.cs(9,14): warning CS0168: The variable 'c' is declared but never used
MainClass.cs(10,15): warning CS0168: The variable 'f' is declared but never used
MainClass.cs(12,16): warning CS0168: The variable 's' is declared but never used
MainClass.cs(13,16): warning CS0168: The variable 'd' is declared but never used
```

## 설명

#line 지시문은 빌드 프로세스의 자동화된 중간 단계에서 사용할 수 있습니다. 예를 들어 원래 소스 코드 파일에서 줄이 제거되었지만 여전히 컴파일러에서 파일의 원래 줄 번호 매기기에 따라 출력을 생성하려는 경우, 줄을 제거한 다음 #line을 사용하여 원래 줄 번호 매기기를 시뮬레이트할 수 있습니다.

#line hidden 지시문은 개발자가 코드를 단계별로 실행할 때 #line hidden과 다음 #line 지시문(다른 #line hidden 지시문이 아니라고 가정) 사이에 있는 모든 줄이 프로시저 단위로 실행되도록 디버거에서 연속되는 줄을 숨깁니다. 이 옵션을 사용하여 ASP.NET이 사용자 정의 코드와 컴퓨터에서 생성된 코드를 구분하도록 할 수도 있습니다. ASP.NET이 이 기능의 주 소비자지만 기능을 사용할 소스 생성기가 늘어날 가능성이 큽니다.

#line hidden 지시문은 오류 보고의 파일 이름이나 줄 번호에는 영향을 주지 않습니다. 즉, 숨겨진 블록에서 오류가 발생할 경우 컴파일러는 오류의 현재 파일 이름과 줄 번호를 보고합니다.

#line filename 지시문은 컴파일러 출력에 표시하려는 파일 이름을 지정합니다. 기본적으로 소스 코드 파일의 실제 이름이 사용됩니다. 파일 이름은 큰따옴표(" ")로 묶어야 하고 줄 번호 뒤에 와야 합니다.

소스 코드 파일에 #line 지시문을 개수와 관계없이 포함할 수 있습니다.

## 예제 1

다음 예제에서는 디버거가 코드의 숨겨진 줄을 어떻게 무시하는지를 보여 줍니다. 예제를 실행하면 세 줄의 텍스트가 표시됩니다. 그러나 예제와 같이 중단점을 설정하고 F10 키를 눌러 코드를 단계별로 실행할 경우 디버거는 숨겨진 줄을 무시합니다. 숨겨진 줄에 중단점을 설정한 경우에도 디버거가 무시합니다.

```
// preprocessor_linehidden.cs
using System;
class MainClass
{
    static void Main()
    {
        Console.WriteLine("Normal line #1."); // Set break point here.
#line hidden
        Console.WriteLine("Hidden line.");
#line default
        Console.WriteLine("Normal line #2.");
    }
}
```

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 전처리기 지시문](#)

# #nullable(C# 참조)

2021-02-18 • 3 minutes to read • [Edit Online](#)

#nullable 전처리기 지시문은 'null 허용 주석 컨텍스트' 및 'null 허용 경고 컨텍스트'를 설정합니다. 이 지시문은 null 허용 주석이 적용되는지와 null 허용 여부 경고가 지정되는지를 제어합니다. 각 컨텍스트는 *disabled* 또는 *enabled*입니다.

두 컨텍스트 모두 프로젝트 수준(C# 소스 코드 외부)에서 지정할 수 있습니다. #nullable 지시문은 주석 및 경고 컨텍스트를 제어하고 프로젝트 수준 설정보다 우선으로 적용됩니다. 지시문은 다른 지시문이 재정의할 때까지 제어하는 컨텍스트를 설정하거나 소스 파일의 끝까지 설정합니다.

지시문의 효과는 다음과 같습니다.

- #nullable disable : null 허용 주석 및 경고 컨텍스트를 *disabled*로 설정합니다.
- #nullable enable : null 허용 주석 및 경고 컨텍스트를 *enabled*로 설정합니다.
- #nullable restore : null 허용 주석 및 경고 컨텍스트를 프로젝트 설정으로 복원합니다.
- #nullable disable annotations : null 허용 주석 컨텍스트를 *disabled*로 설정합니다.
- #nullable enable annotations : null 허용 주석 컨텍스트를 *enabled*로 설정합니다.
- #nullable restore annotations : null 허용 주석 컨텍스트를 프로젝트 설정으로 복원합니다.
- #nullable disable warnings : null 허용 경고 컨텍스트를 *disabled*로 설정합니다.
- #nullable enable warnings : null 허용 경고 컨텍스트를 *enabled*로 설정합니다.
- #nullable restore warnings : null 허용 경고 컨텍스트를 프로젝트 설정으로 복원합니다.

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 전처리기 지시문](#)

# #region(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

#region 을 사용하면 코드 편집기의 [개요 기능](#)을 사용할 때 확장하거나 축소할 수 있는 코드 블록을 지정할 수 있습니다. 더 긴 코드 파일에서 현재 작업 중인 파일 부분에 집중할 수 있도록 하나 이상의 영역을 축소하거나 숨길 수 있습니다. 다음 예제에서는 영역을 정의하는 방법을 보여 줍니다.

```
#region MyClass definition
public class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

## 설명

#region 블록은 #endregion 지시문으로 종료해야 합니다.

#region 블록은 #if 블록과 겹칠 수 없습니다. 그러나 #region 블록을 #if 블록에 중첩할 수 있고 #if 블록을 #region 블록에 중첩할 수 있습니다.

## 참조

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 전처리기 지시문](#)

# #endregion(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

#endregion 은 #region 블록의 끝을 표시합니다. 다음은 그 예입니다.

```
#region MyClass definition
class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 전처리기 지시문](#)

# #pragma(C# 참조)

2020-11-02 • 2 minutes to read • [Edit Online](#)

#pragma 는 이 코드가 표시되는 파일의 컴파일에 대한 특수 명령을 컴파일러에 제공합니다. 컴파일러에서 명령을 지원해야 합니다. 즉, #pragma 를 사용하여 사용자 지정 전처리 명령을 만들 수 없습니다. Microsoft C# 컴파일러는 다음 두 가지 #pragma 명령을 지원합니다.

[#pragma warning](#)

[#pragma checksum](#)

## 구문

```
#pragma pragma-name pragma-arguments
```

## 매개 변수

pragma-name

인식된 pragma의 이름입니다.

pragma-arguments

pragma 관련 인수입니다.

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 전처리기 지시문](#)
- [#pragma warning](#)
- [#pragma checksum](#)

# #pragma warning(C# 참조)

2021-02-18 • 2 minutes to read • [Edit Online](#)

`#pragma warning`은 특정 경고를 사용하거나 사용하지 않도록 설정합니다.

## 구문

```
#pragma warning disable warning-list
#pragma warning restore warning-list
```

## 매개 변수

`warning-list` 쉼표로 구분된 경고 번호 목록입니다. "CS" 접두사는 선택 사항입니다.

경고 번호를 지정하지 않은 경우 `disable`은 모든 경고를 사용하지 않도록 설정하고 `restore`는 모든 경고를 사용하도록 설정합니다.

### NOTE

Visual Studio에서 경고 번호를 찾으려면 프로젝트를 빌드하고 출력 창에서 경고 번호를 찾습니다.

## 예제

```
// pragma_warning.cs
using System;

#pragma warning disable 414, CS3021
[CLSCompliant(false)]
public class C
{
    int i = 1;
    static void Main()
    {
    }
}
#pragma warning restore CS3021
[CLSCompliant(false)] // CS3021
public class D
{
    int i = 1;
    public static void F()
    {
    }
}
```

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)
- [C# 전처리기 지시문](#)
- [C# 컴파일러 오류](#)

- 코드 분석 경고를 표시하지 않는 방법

# #pragma checksum(C# 참조)

2020-11-02 • 3 minutes to read • [Edit Online](#)

ASP.NET 페이지 디버깅을 돋기 위해 소스 파일에 대한 체크섬을 생성합니다.

## 구문

```
#pragma checksum "filename" "{guid}" "checksum bytes"
```

## 매개 변수

"filename"

변경 내용이나 업데이트를 모니터링해야 하는 파일의 이름입니다.

"{guid}"

해시 알고리즘의 GUID(Globally Unique Identifier)입니다.

"checksum\_bytes"

체크섬의 바이트를 나타내는 16진수 문자열입니다. 짹수의 16진수여야 합니다. 홀수를 사용하면 컴파일 시간 경고가 발생하고 지시문이 무시됩니다.

## 설명

Visual Studio 디버거는 체크섬을 사용하여 항상 올바른 소스를 찾도록 합니다. 컴파일러는 소스 파일에 대한 체크섬을 계산한 다음 출력을 PDB(프로그램 데이터베이스) 파일로 내보냅니다. 그런 다음 디버거는 PDB를 사용하여 소스 파일에 대해 계산하는 체크섬과 비교합니다.

체크섬이 .aspx 파일이 아니라 생성된 소스 파일에 대해 계산되므로 이 솔루션은 ASP.NET 프로젝트에서 작동하지 않습니다. 이 문제를 해결하기 위해 `#pragma checksum`은 ASP.NET 페이지에 대한 체크섬 지원을 제공합니다.

Visual C#에서 ASP.NET 프로젝트를 만드는 경우 생성된 소스 파일에 소스가 생성되는 .aspx 파일에 대한 체크섬이 포함됩니다. 그런 다음 컴파일러는 PDB 파일에 이 정보를 씁니다.

컴파일러가 파일에서 `#pragma checksum` 지시문을 발견하지 못하면 체크섬을 계산하고 PDB 파일에 값을 씁니다.

## 예제

```
class TestClass
{
    static int Main()
    {
        #pragma checksum "file.cs" "{406EA660-64CF-4C82-B6F0-42D48172A799}" "ab007f1d23d9" // New checksum
    }
}
```

## 참고 항목

- [C# 참조](#)
- [C# 프로그래밍 가이드](#)

- C# 전처리기 지시문

# C# 컴파일러 옵션

2020-11-02 • 2 minutes to read • [Edit Online](#)

컴파일러는 실행 파일(.exe), 동적 연결 라이브러리(.dll) 또는 코드 모듈(.netmodule)을 생성합니다.

모든 컴파일러 옵션은 **-옵션** 및 **/옵션**의 두 가지 형태로 사용할 수 있습니다. 문서에는 **-옵션** 양식만 나와 있습니다.

Visual Studio에서는 *web.config* 파일에서 컴파일러 옵션을 설정합니다. 자세한 내용은 [\*\*<compiler> 요소\*\*](#)를 참조하세요.

## 단원 내용

- [csc.exe를 사용한 명령줄 빌드](#) 명령줄에서 Visual C# 애플리케이션을 빌드하는 방법에 대한 정보를 제공합니다.
- [Visual Studio 명령줄에 대한 환경 변수 설정 방법](#) *VsDevCmd.bat*를 실행하여 명령줄 빌드를 사용하도록 설정하는 단계를 제공합니다.
- [C# 컴파일러 옵션 범주별 목록](#) 컴파일러 옵션의 범주별 목록입니다.
- [C# 컴파일러 옵션 사전순 목록](#) 컴파일러 옵션의 사전순 목록입니다.

## 관련 단원

- [프로젝트 디자이너, 빌드 페이지](#) 프로젝트가 컴파일, 빌드 및 디버그되는 방법을 제어하는 속성 설정입니다. Visual C# 프로젝트의 사용자 지정 빌드 단계에 대한 정보를 포함합니다.
- [기본 및 사용자 지정 빌드](#) 빌드 형식 및 구성에 대한 정보입니다.
- [빌드 준비 및 관리](#) Visual Studio 개발 환경 내의 빌드 절차입니다.

# csc.exe를 사용한 명령줄 빌드

2021-02-18 • 10 minutes to read • [Edit Online](#)

명령 프롬프트에서 실행 파일(*csc.exe*)의 이름을 입력하여 C# 컴파일러를 호출할 수 있습니다.

**Visual Studio**용 개발자 명령 프롬프트 창을 사용하는 경우 필요한 환경 변수가 모두 설정됩니다. 이 도구에 액세스하는 방법에 대한 자세한 내용은 [Visual Studio용 개발자 명령 프롬프트](#) 항목을 참조하세요.

표준 명령 프롬프트 창을 사용하는 경우 컴퓨터의 하위 디렉터리에서 *csc.exe*를 호출하려면 먼저 경로를 조정해야 합니다. 또한 *VsDevCmd.bat*를 실행하여 명령줄 빌드를 지원하도록 적절한 환경 변수를 설정해야 합니다.

*VsDevCmd.bat*를 찾아서 실행하는 방법에 관한 지침을 포함하여 이 배치 파일에 관한 자세한 내용은 [Visual Studio 명령줄에 필요한 환경 변수 설정 방법](#)을 참조하세요.

Windows SDK(소프트웨어 개발 키트)만 있는 컴퓨터에서 작업하는 경우, **Microsoft .NET Framework SDK** 메뉴 옵션을 통해 여는 **SDK 명령 프롬프트**에서 C# 컴파일러를 사용할 수 있습니다.

또한 MSBuild를 사용하여 프로그래밍 방식으로 C# 프로그램을 빌드할 수도 있습니다. 자세한 내용은 [MSBuild](#)을 참조하세요.

*csc.exe* 실행 파일은 대개 Windows 디렉터리 아래의 Microsoft.NET\Framework\<Version> 폴더에 있습니다. 이 위치는 개별 컴퓨터의 구성에 따라 다를 수 있습니다. 컴퓨터에 .NET Framework 버전이 두 개 이상 설치된 경우 이 파일의 여러 버전을 찾을 수 있습니다. 해당 설치에 대한 자세한 내용은 [방법: 설치된 .NET Framework 버전 확인](#)을 참조하세요.

## TIP

Visual Studio IDE를 사용하여 프로젝트를 빌드할 때 출력 창에 **csc** 명령과 관련 컴파일러 옵션을 표시할 수 있습니다. 이 정보를 표시하려면 [방법: 빌드 로그 파일 보기, 저장 및 구성](#)의 지침에 따라 로그 데이터의 세부 정보 표시 수준을 보통 또는 자세히로 변경합니다. 프로젝트를 다시 빌드한 후 출력 창에서 **csc**를 검색하여 C# 컴파일러 호출을 찾습니다.

## 항목 내용

- [명령줄 구문에 대한 규칙](#)
- [샘플 명령줄](#)
- [C# 컴파일러 및 C++ 컴파일러 출력의 차이점](#)

## C# 컴파일러의 명령줄 구문에 대한 규칙

C# 컴파일러에서는 운영 체제 명령줄에 지정된 인수를 해석할 때 다음 규칙을 사용합니다.

- 인수를 공백이나 탭으로 구분합니다.
- 캐럿 문자(^)는 이스케이프 문자나 구분 기호로 인식되지 않습니다. 문자는 프로그램의 `argv` 배열에 전달된 후 운영 체제의 명령줄 구문 분석기에서 처리됩니다.
- 큰따옴표로 묶은 문자열("string")은 포함된 공백에 상관없이 하나의 인수로 해석됩니다. 따옴표로 묶은 문자열은 인수에 포함될 수 있습니다.
- 백슬래시 다음의 큰따옴표(\"")는 리터럴 큰따옴표 문자("")로 해석됩니다.
- 백슬래시는 큰따옴표 바로 앞에 있지 않으면 리터럴로 해석됩니다.
- 짹수 개의 백슬래시 다음에 큰따옴표가 오는 경우, 백슬래시 쌍마다 하나의 백슬래시가 `argv` 배열에 배

치되고 큰따옴표는 문자열 구분 기호로 해석됩니다.

- 훌수 개의 백슬래시 다음에 큰따옴표가 오는 경우, 백슬래시 쌍마다 하나의 백슬래시가 `argv` 배열에 배치되고 큰따옴표는 나머지 백슬래시에 의해 "이스케이프"됩니다. 이로 인해 리터럴 큰따옴표(")가 `argv`에 추가됩니다.

## C# 컴파일러에 대한 샘플 명령줄

- *File.cs* 를 컴파일하여 *File.exe* 를 생성합니다.

```
csc File.cs
```

- *File.cs* 를 컴파일하여 *File.dll* 를 생성합니다.

```
csc -target:library File.cs
```

- *File.cs* 를 컴파일하여 *My.exe* 를 만듭니다.

```
csc -out:My.exe File.cs
```

- 최적화를 사용하여 현재 디렉터리에 있는 모든 C# 파일을 컴파일하고 DEBUG 기호를 정의합니다. 출력은 *File2.exe* 입니다.

```
csc -define:DEBUG -optimize -out:File2.exe *.cs
```

- 현재 디렉터리에 있는 모든 C# 파일을 컴파일하여 *File2.dll* 의 디버그 버전을 생성합니다. 로고 및 경고가 표시되지 않습니다.

```
csc -target:library -out:File2.dll -warn:0 -nologo -debug *.cs
```

- 현재 디렉터리에 있는 모든 C# 파일을 *Something.xyz*(DLL)로 컴파일합니다.

```
csc -target:library -out:Something.xyz *.cs
```

## C# 컴파일러 및 C++ 컴파일러 출력의 차이점

C# 컴파일러를 호출하면 개체 파일(*.obj*)은 만들어지지 않고 출력 파일이 직접 만들어집니다. 따라서 C# 컴파일러에는 링커가 필요하지 않습니다.

## 참고 항목

- [C# 컴파일러 옵션](#)
- [사전순 C# 컴파일러 옵션 목록](#)
- [범주별 C# 컴파일러 옵션 목록](#)
- [Main\(\)과 명령줄 인수](#)
- [명령줄 인수](#)
- [명령줄 인수를 표시하는 방법](#)
- [Main\(\) 반환 값](#)

# Visual Studio 명령줄에 필요한 환경 변수를 설정하는 방법

2021-02-18 • 3 minutes to read • [Edit Online](#)

VsDevCmd.bat 파일은 명령줄 빌드를 사용하도록 적절한 환경 변수를 설정합니다.

## NOTE

Visual Studio 2015 및 이전 버전에서는 VsDevCmd.bat가 아닌 VSVARS32.bat가 같은 용도로 사용됩니다. 이 파일은 \Program Files\Microsoft Visual Studio\Version\Common7\Tools 또는 Program Files (x86)\Microsoft Visual Studio\Version\Common7\Tools에 저장되었습니다.

Visual Studio의 이전 버전이 설치된 컴퓨터에 Visual Studio의 최신 버전을 설치한 경우 동일한 명령 프롬프트 창에서 다른 버전의 VsDevCmd.bat 또는 VSVARS32.BAT를 실행해서는 안 됩니다. 대신 별도 창에서 각 버전에 대해 명령을 실행해야 합니다.

## VsDevCmd.BAT를 실행하려면

- 시작 메뉴에서 VS 2019용 개발자 명령 프롬프트를 엽니다. **Visual Studio 2019** 폴더에 있습니다.
- 설치 경로를 \Program Files\Microsoft Visual Studio\Version\Offering\Common7\Tools or \Program Files (x86)\Microsoft Visual Studio\Version\Offering\Common7\Tools 하위 디렉터리로 변경합니다. (*Version*은 현재 버전용인 2019입니다. *Offering*은 Enterprise, Professional 또는 Community 중 하나입니다.)
- VsDevCmd**를 입력하여 VsDevCmd.bat를 실행합니다.

## Caution

VsDevCmd.bat는 컴퓨터마다 다를 수 있습니다. 누락되거나 손상된 VsDevCmd.bat 파일을 다른 컴퓨터의 VsDevCmd.bat 파일로 바꾸지 마세요. 대신 설치 프로그램을 다시 실행하여 누락된 파일을 교체하십시오.

## VsDevCmd.BAT에 사용 가능한 옵션

VsDevCmd.BAT에 사용 가능한 옵션을 보려면 `-help` 옵션을 사용하여 명령을 실행하세요.

```
VsDevCmd.bat -help
```

## 참조

- [csc.exe를 사용한 명령줄 빌드](#)

# C# 컴파일러 옵션 범주별 목록

2020-11-02 • 10 minutes to read • [Edit Online](#)

여기에서는 컴파일러 옵션을 범주별로 정렬합니다. 사전순 목록은 [C# 컴파일러 옵션 사전순 목록](#)을 참조하세요.

## 최적화

옵션	용도
<a href="#">-filealign</a>	출력 파일에 있는 섹션의 크기를 지정합니다.
<a href="#">-optimize</a>	최적화를 사용하거나 사용하지 않도록 설정합니다.

## 출력 파일

옵션	용도
<a href="#">-deterministic</a>	입력이 동일한 경우 컴파일 간에 이진 콘텐츠가 동일한 어셈블리를 컴파일러에서 출력하도록 합니다.
<a href="#">-doc</a>	처리된 문서 주석을 작성할 XML 파일을 지정합니다.
<a href="#">-out</a>	출력 파일을 지정합니다.
<a href="#">-pathmap</a>	컴파일러에서 출력되는 소스 경로 이름에 대한 매팅을 지정합니다.
<a href="#">/pdb</a>	.pdb 파일의 이름과 위치를 지정합니다.
<a href="#">-platform</a>	출력 플랫폼을 지정합니다.
<a href="#">/preferreduilang</a>	컴파일러 출력 언어를 지정합니다.
<a href="#">/refout</a>	주 어셈블리 외에도 참조 어셈블리를 생성합니다.
<a href="#">/refonly</a>	주 어셈블리를 대신 참조 어셈블리를 생성합니다.
<a href="#">-target</a>	옵션(-target:appcontainerexe, -target:exe, -target:library, -target:module, -target:winexe 또는 -target:winmdobj) 중 하나를 사용하여 출력 파일 형식을 지정합니다.
<a href="#">-modulename:&lt;string&gt;</a>	소스 모듈의 이름을 지정합니다.

## .NET 어셈블리

옵션	용도
-addmodule	이 어셈블리의 일부가 될 모듈을 하나 이상 지정합니다.
-delaysign	공개 키를 추가하고 어셈블리는 서명되지 않은 채 두도록 컴파일러에 지시합니다.
-keycontainer	암호화 키 컨테이너의 이름을 지정합니다.
-keyfile	암호화 키를 포함하는 파일 이름을 지정합니다.
/lib	-reference를 통해 참조되는 어셈블리의 위치를 지정합니다.
-nostdlib	표준 라이브러리(mscorlib.dll)를 가져오지 않도록 컴파일러에 지시합니다.
-publicsign	어셈블리에 서명하지 않고 공개 키를 적용하지만 어셈블리가 서명되었음을 나타내는 비트를 어셈블리에 설정합니다.
-reference	어셈블리를 포함하는 파일에서 메타데이터를 가져옵니다.
-analyzer	이 어셈블리에서 분석기를 실행합니다(약식: /a).
-additionalfile	코드 생성에 직접 영향을 주지 않지만 오류 또는 경고를 생성하기 위해 분석기에서 사용할 수 있는 추가 파일에 이름을 지정합니다.
-embed	PDB에 모든 소스 파일을 포함합니다.
-embed:<file list>	PDB에 특정 파일을 포함합니다.

## 디버깅/오류 검사

옵션	용도
-bugreport	쉽게 버그를 보고할 수 있도록 정보가 포함된 파일을 만듭니다.
/checked	데이터 형식 범위를 오버플로하는 정수 연산이 있는 경우 런타임에 예외가 발생되는지 여부를 지정합니다.
-debug	디버깅 정보를 내보내도록 컴파일러에 지시합니다.
-errorreport	오류 보고 동작을 설정합니다.
/fullpaths	컴파일러 출력에서 파일의 절대 경로를 지정합니다.
-nowarn	지정한 경고가 컴파일러에서 생성되지 않도록 합니다.
-nullable	null 허용 컨텍스트 옵션을 지정합니다.
/warn	경고 수준을 설정합니다.

옵션	용도
-warnaserror	경고를 오류로 승격합니다.
-ruleset:<file>	특정 진단을 사용하지 않는 규칙 집합 파일을 지정합니다.

## 전처리기

옵션	용도
-define	전처리기 기호를 정의합니다.

## 리소스

옵션	용도
-link	지정된 어셈블리의 COM 형식 정보를 프로젝트에 사용할 수 있도록 합니다.
-linkresource	관리되는 리소스에 대한 링크를 만듭니다.
-resource	출력 파일에 .NET 리소스를 포함합니다.
-win32icon	출력 파일에 삽입할 .ico 파일을 지정합니다.
/win32res:	출력 파일에 삽입할 Win32 리소스를 지정합니다.

## 기타

옵션	용도
@	지시 파일을 지정합니다.
-?	stdout에 컴파일러 옵션을 나열합니다.
-baseaddress	DLL을 로드할 기본 설정 주소를 지정합니다.
-codepage	컴파일할 때 모든 소스 코드 파일에 사용할 코드 페이지를 지정합니다.
-help	stdout에 컴파일러 옵션을 나열합니다.
-highentropyva	실행 파일이 ASLR(주소 공간 레이아웃 불규칙화)을 지원하도록 지정합니다.
-langversion	언어 버전 지정: 기본값, ISO-1, ISO-2, 3, 4, 5, 6, 7, 7.1, 7.2, 7.3 또는 최신
-main	Main 메서드의 위치를 지정합니다.

옵션	용도
-noconfig	csc.rsp를 사용하여 컴파일하지 않도록 컴파일러에 지시합니다.
-nologo	컴파일러 배너 정보를 표시하지 않습니다.
-recurse	하위 디렉터리에서 컴파일할 소스 파일을 검색합니다.
-subsystemversion	실행 파일이 사용할 수 있는 하위 시스템의 최소 버전을 지정합니다.
/unsafe	unsafe 키워드를 사용하는 코드를 컴파일할 수 있도록 설정합니다.
-utf8output	UTF-8 인코딩을 사용하여 컴파일러 출력을 표시합니다.
-parallel[+ -]	동시 빌드(+)를 사용할지 여부를 지정합니다.
-checksumalgorithm:<alg>	PDB에 저장된 소스 파일 체크섬을 계산하기 위한 알고리즘을 지정합니다. 지원되는 값은 다음과 같습니다. SHA256(기본값) 또는 SHA1. SHA1 관련 충돌 문제로 인해 SHA256을 사용하는 것이 좋습니다.

## 사용되지 않는 옵션

옵션	용도
-incremental	증분 컴파일을 사용하도록 설정합니다.

## 참조

- [C# 컴파일러 옵션](#)
- [사전순 C# 컴파일러 옵션 목록](#)
- [Visual Studio 명령줄에 필요한 환경 변수를 설정하는 방법](#)

# C# 컴파일러 옵션 사전순 목록

2020-11-02 • 12 minutes to read • [Edit Online](#)

여기에서는 컴파일러 옵션을 사전순으로 정렬합니다. 범주별 목록을 보려면 [C# 컴파일러 옵션 범주별 목록](#)을 참조하세요.

옵션	용도
@	추가 옵션에 대한 지시 파일을 읽습니다.
-?	stdout에 사용법 메시지를 표시합니다.
-additionalfile	코드 생성에 직접 영향을 주지 않지만 오류 또는 경고를 생성하기 위해 분석기에서 사용할 수 있는 추가 파일에 이름을 지정합니다.
-addmodule	지정한 모듈을 이 어셈블리에 링크합니다.
-analyzer	이 어셈블리에서 분석기를 실행합니다(약식: -a).
/appconfig	어셈블리 바인딩 시간에 app.config의 위치를 지정합니다.
-baseaddress	빌드할 라이브러리의 기준 주소를 지정합니다.
-bugreport	'버그 보고서' 파일을 만듭니다. -errorreport:prompt 또는 -errorreport:send와 함께 사용하면 이 파일이 충돌 정보와 함께 전송됩니다.
/checked	컴파일러에서 오버플로 검사를 생성하도록 합니다.
-checksumalgorithm:<alg>	PDB에 저장된 소스 파일 체크섬을 계산하기 위한 알고리즘을 지정합니다. 지원되는 값은 다음과 같습니다. SHA256(기본값) 또는 SHA1. SHA1 관련 충돌 문제로 인해 SHA256을 사용하는 것이 좋습니다.
-codepage	소스 파일을 열 때 사용할 코드 페이지를 지정합니다.
-debug	디버깅 정보를 내보냅니다.
-define	조건부 컴파일 기호를 정의합니다.
-delaysign	강력한 이름 키의 공용 부분만 사용하여 어셈블리 서명을 연기합니다.
-deterministic	입력이 동일한 경우 컴파일 간에 이진 콘텐츠가 동일한 어셈블리를 컴파일러에서 출력하도록 합니다.
-doc	생성할 XML 문서 파일을 지정합니다.
-embed	PDB에 모든 소스 파일을 포함합니다.

옵션	용도
-embed:<file list>	PDB에 특정 파일을 포함합니다.
-errorendlocation	각 오류의 끝 위치에 해당하는 줄과 열을 출력합니다.
-errorlog:<file>	모든 컴파일러와 분석기 진단을 로그할 파일을 지정합니다.
-errorreport	내부 컴파일러 오류를 처리하는 방법을 지정합니다. prompt, send 또는 none 중에서 선택할 수 있으며 기본값은 none입니다.
-filealign	출력 파일 섹션에 사용되는 맞춤을 지정합니다.
/fullpaths	컴파일러에서 정규화된 경로를 생성하도록 합니다.
-help	stdout에 사용법 메시지를 표시합니다.
-highentropyva	높은 엔트로피 ASLR을 지원하도록 지정합니다.
-incremental	증분 컴파일을 사용하도록 설정합니다(사용되지 않음).
-keycontainer	강력한 이름의 키 컨테이너를 지정합니다.
-keyfile	강력한 이름의 키 파일을 지정합니다.
-langversion:<string>	언어 버전 지정: 기본값, ISO-1, ISO-2, 3, 4, 5, 6, 7, 7.1, 7.2, 7.3 또는 최신
/lib	참조를 검색할 추가 디렉터리를 지정합니다.
-link	지정된 어셈블리의 COM 형식 정보를 프로젝트에 사용할 수 있도록 합니다.
-linkresource	지정한 리소스를 이 어셈블리에 링크합니다.
-main	진입점을 포함하는 형식을 지정합니다. 다른 모든 가능한 진입점은 무시합니다.
-moduleassemblyname	.netmodule에서 public이 아닌 형식에 액세스할 수 있는 어셈블리를 지정합니다.
-modulename:<string>	소스 모듈의 이름을 지정합니다.
-noconfig	컴파일러에서 CSC.RSP 파일을 자동으로 포함하지 않도록 합니다.
-nologo	컴파일러 저작권 메시지를 표시하지 않습니다.
-nostdlib	컴파일러에서 표준 라이브러리(mscorlib.dll)를 참조하지 않도록 합니다.
-nowarn	특정 경고 메시지를 사용하지 않도록 설정합니다.

옵션	용도
-nowin32manifest	애플리케이션 매니페스트를 실행 파일에 포함하지 않도록 컴파일러에 지시합니다.
-nullable	null 허용 컨텍스트 옵션을 지정합니다.
-optimize	최적화를 사용하거나 사용하지 않도록 설정합니다.
-out	출력 파일 이름을 지정합니다(기본값: 주 클래스가 있는 파일의 기본 이름 또는 첫째 파일).
-parallel[+ -]	동시 빌드(+)를 사용할지 여부를 지정합니다.
-pathmap	컴파일러에서 출력되는 소스 경로 이름에 대한 매팅을 지정합니다.
/pdb	.pdb 파일의 이름과 위치를 지정합니다.
-platform	이 코드를 실행할 수 있는 플랫폼을 x86, Itanium, x64, anycpu 또는 anycpu32bitpreferred로 제한합니다. 기본값은 anycpu입니다.
/preferreduilang	컴파일러 출력에 사용할 언어를 지정합니다.
-publicsign	어셈블리에 서명하지 않고 공개 키를 적용하지만 어셈블리가 서명되었음을 나타내는 비트를 어셈블리에 설정합니다.
-recurse	와일드카드 지정에 따라 현재 디렉터리와 하위 디렉터리에 있는 모든 파일을 포함합니다.
-reference	지정한 어셈블리 파일에서 메타데이터를 참조합니다.
/refout	주 어셈블리 외에도 참조 어셈블리를 생성합니다.
/refonly	주 어셈블리를 대신 참조 어셈블리를 생성합니다.
-reportanalyzer	실행 시간과 같은 추가적인 분석 정보를 보고합니다.
-resource	지정한 리소스를 포함합니다.
-ruleset:<file>	특정 진단을 사용하지 않는 규칙 집합 파일을 지정합니다.
-subsystemversion	실행 파일이 사용할 수 있는 하위 시스템의 최소 버전을 지정합니다.
-target	옵션(-target:appcontainerexe, -target:exe, -target:library, -target:module, -target:winexe, -target:winmdobj) 중 하나를 사용하여 출력 파일의 형식을 지정합니다.
/unsafe	안전하지 않은 코드를 허용합니다.
-utf8output	컴파일러 메시지를 UTF-8 인코딩으로 출력합니다.

옵션	용도
-version	컴파일러 버전 번호를 표시하고 종료합니다.
/warn	경고 수준(0-4)을 설정합니다.
-warnaserror	특정 경고를 오류로 보고합니다.
-win32icon	이 아이콘을 사용하여 출력합니다.
-win32manifest	사용자 지정 win32 매니페스트 파일을 지정합니다.
/win32res:	win32 리소스 파일(.res)을 지정합니다.

## 참조

- [C# 컴파일러 옵션](#)
- [범주별 C# 컴파일러 옵션 목록](#)
- [Visual Studio 명령줄에 필요한 환경 변수를 설정하는 방법](#)
- [<compiler> 요소](#)

# @(C# 컴파일러 옵션)

2020-11-02 • 3 minutes to read • [Edit Online](#)

@ 옵션을 사용하면 컴파일러 옵션과 컴파일할 소스 코드 파일이 포함된 파일을 지정할 수 있습니다.

## 구문

```
@response_file
```

## 인수

```
response_file
```

컴파일러 옵션이나 컴파일할 소스 코드 파일이 나열된 파일입니다.

## 설명

컴파일러 옵션 및 소스 코드 파일은 명령줄에 지정된 것처럼 컴파일러에서 처리됩니다.

컴파일에 지시 파일을 둘 이상 지정하려면 여러 지시 파일 옵션을 지정합니다. 예를 들면 다음과 같습니다.

```
@file1.rsp @file2.rsp
```

지시 파일에서 여러 컴파일러 옵션과 소스 코드 파일이 한 줄에 나타날 수 있습니다. 단일 컴파일러 옵션 사양은 한 줄에 표시되어야 합니다(여러 줄에 걸쳐 있을 수 없음). 지시 파일은 # 기호로 시작하는 주석을 포함할 수 있습니다.

지시 파일 내에서 컴파일러 옵션을 지정하는 것은 명령줄에서 해당 명령을 실행하는 것과 같습니다. 자세한 내용은 [명령줄에서 빌드](#)를 참조하세요.

명령 옵션이 발견되면 컴파일러에서 처리합니다. 따라서, 명령줄 인수는 지시 파일에서 이전에 나열된 옵션을 재정의할 수 있습니다. 반대로 지시 파일의 옵션은 명령줄 또는 다른 지시 파일에서 이전에 나열된 옵션을 재정의합니다.

C#에서는 csc.exe 파일과 동일한 디렉터리에 있는 csc.rsp 파일을 제공합니다. csc.rsp에 대한 자세한 내용은 [-noconfig](#)를 참조하세요.

Visual Studio 개발 환경에서는 이 컴파일러 옵션을 설정할 수 없으며 프로그래밍 방식으로 변경할 수도 없습니다.

## 예제

다음은 샘플 지시 파일의 몇 줄입니다.

```
# build the first output file
-target:exe -out:MyExe.exe source1.cs source2.cs
```

## 참고 항목

- [C# 컴파일러 옵션](#)

# -addmodule(C# 컴파일러 옵션)

2020-11-02 • 3 minutes to read • [Edit Online](#)

이 옵션은 target:module 스위치로 만들어진 모듈을 현재 컴파일에 추가합니다.

## 구문

```
-addmodule:file[;file2]
```

## 인수

`file`, `file2`

메타데이터를 포함하는 출력 파일입니다. 파일에 어셈블리 매니페스트를 포함할 수 없습니다. 둘 이상의 파일을 가져오려면 파일 이름을 쉼표 또는 세미콜론으로 구분합니다.

## 설명

**-addmodule**을 사용하여 추가된 모든 모듈은 런타임에 출력 파일과 동일한 디렉터리에 있어야 합니다. 즉, 컴파일 시간에 임의 디렉터리에 있는 모듈을 지정할 수 있지만 런타임에 모듈이 애플리케이션 디렉터리에 있어야 합니다. 런타임에 모듈이 애플리케이션 디렉터리에 없는 경우 [TypeLoadException](#)이 발생합니다.

`file`에 어셈블리를 사용할 수 없습니다. 예를 들어, [-target:module](#)을 사용하여 출력 파일이 만들어진 경우 **-addmodule**을 사용하여 해당 메타데이터를 가져올 수 있습니다.

출력 파일이 [-target:module](#) 이외의 [-target](#) 옵션으로 작성된 경우 해당 메타데이터를 **-addmodule**로 가져올 수 없지만 [-reference](#)로 가져올 수 있습니다.

이 컴파일러 옵션은 Visual Studio에서 사용할 수 없습니다. 프로젝트에서 모듈을 참조할 수 없습니다. 또한 이 컴파일러 옵션은 프로그래밍 방식으로 변경할 수 없습니다.

## 예제

소스 파일 `input.cs`를 컴파일하고 `metad1.netmodule` 및 `metad2.netmodule`의 메타데이터를 추가하여 `out.exe`를 생성합니다.

```
csc -addmodule:metad1.netmodule;metad2.netmodule -out:out.exe input.cs
```

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)
- [다중 파일 어셈블리](#)
- [방법: 다중 파일 어셈블리 빌드](#)

# -appconfig(C# 컴파일러 옵션)

2020-11-02 • 4 minutes to read • [Edit Online](#)

-**appconfig** 컴파일러 옵션을 사용하면 C# 애플리케이션이 어셈블리 바인딩 시간에 어셈블리의 애플리케이션 구성(app.config) 파일 위치를 CLR(공용 언어 런타임)에 지정할 수 있습니다.

## 구문

```
-appconfig:file
```

## 인수

`file`

필수 요소. 어셈블리 바인딩 설정이 포함된 애플리케이션 구성 파일입니다.

## 설명

-**appconfig**의 한 가지 용도는 어셈블리가 특정 참조 어셈블리의 .NET Framework 버전과 .NET Framework for Silverlight 버전을 동시에 둘 다 참조해야 하는 고급 시나리오입니다. 예를 들어 WPF(Windows Presentation Foundation)로 작성된 XAML 디자이너는 디자이너의 사용자 인터페이스를 위한 WPF 데스크톱 및 Silverlight에 포함된 WPF 하위 집합을 둘 다 참조해야 할 수도 있습니다. 같은 디자이너 어셈블리가 두 어셈블리에 모두 액세스해야 합니다. 기본적으로, 어셈블리 바인딩 시 두 어셈블리가 동등한 것으로 간주되기 때문에 서로 다른 참조를 사용할 경우 컴파일러 오류가 발생합니다.

-**appconfig** 컴파일러 옵션을 사용하면 다음 예제와 같이 `<supportPortability>` 태그를 사용하여 기본 동작을 비활성화하는 app.config 파일의 위치를 지정할 수 있습니다.

```
<supportPortability PKT="7cec85d7bea7798e" enable="false"/>
```

컴파일러는 이 파일 위치를 CLR의 어셈블리 바인딩 논리에 전달합니다.

### NOTE

Microsoft Build Engine(MSBuild)을 사용하여 애플리케이션을 빌드하는 경우 .csproj 파일에 속성 태그를 추가하여 -**appconfig** 컴파일러 옵션을 설정할 수 있습니다. 프로젝트에 이미 설정된 app.config 파일을 사용하려면 속성 태그 `<UseAppConfigForCompiler>`를 .csproj 파일에 추가하고 해당 값을 `true`로 설정합니다. 다른 app.config 파일을 지정하려면 속성 태그 `< AppConfigForCompiler>`를 추가하고 해당 값을 파일 위치로 설정합니다.

## 예제

다음 예제에서는 애플리케이션이 두 구현에 모두 있는 .NET Framework 어셈블리의 .NET Framework for Silverlight 구현과 .NET Framework 구현 둘 다에 대한 참조를 사용할 수 있도록 하는 app.config 파일을 보여 줍니다. -**appconfig** 컴파일러 옵션은 이 app.config 파일의 위치를 지정합니다.

```
<configuration>
  <runtime>
    <assemblyBinding>
      <supportPortability PKT="7cec85d7bea7798e" enable="false"/>
      <supportPortability PKT="31bf3856ad364e35" enable="false"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

## 참조

- [<supportPortability> 요소](#)
- [사전순 C# 컴파일러 옵션 목록](#)

# -baseaddress(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

**-baseaddress** 옵션을 사용하면 DLL을 로드할 기본 기준 주소를 지정할 수 있습니다. 이 옵션을 사용하는 시기와 이유에 대한 자세한 내용은 [Larry Osterman's WebLog](#)를 참조하세요.

## 구문

```
-baseaddress:address
```

## 인수

address

DLL의 기준 주소입니다. 이 주소는 10진수, 16진수 또는 8진수 숫자로 지정할 수 있습니다.

## 설명

DLL의 기본 기준 주소는 .NET 공용 언어 런타임에서 설정됩니다.

이 주소의 하위 단어는 반올림됩니다. 예를 들어 0x11110001을 지정하면 0x11110000으로 반올림됩니다.

DLL에 대한 서명 프로세스를 완료하려면 SN.EXE에 -R 옵션을 사용합니다.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

1. 프로젝트 속성 페이지를 엽니다.
2. 빌드 속성 페이지를 클릭합니다.
3. 고급 단추를 클릭합니다.
4. DLL 기준 주소 속성을 수정합니다.

프로그래밍 방식으로 이 컴파일러 옵션을 설정하려면 [BaseAddress](#)를 참조하세요.

## 참고 항목

- [ProcessModule.BaseAddress](#)
- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -bugreport(C# 컴파일러 옵션)

2021-02-18 • 3 minutes to read • [Edit Online](#)

나중에 분석하기 위해 파일에 디버그 정보를 포함하도록 지정합니다.

## 구문

```
-bugreport:file
```

## 인수

`file`

버그 보고서를 포함하려는 파일의 이름입니다.

## 설명

**-bugreport** 옵션은 `file`에 다음 정보를 포함하도록 지정합니다:

- 컴파일에 포함된 모든 소스 코드 파일의 복사본입니다.
- 컴파일에 사용된 컴파일러 옵션 목록입니다.
- 컴파일러, 런타임 및 운영 체제에 대한 버전 정보입니다.
- .NET 및 .NET SDK와 함께 제공되는 어셈블리를 제외하고 16진수로 저장된 참조된 어셈블리 및 모듈입니다.
- 컴파일러 출력입니다(있는 경우).
- 메시지가 표시되는 문제에 대한 설명입니다.
- 메시지가 표시되는 문제 해결 방법에 대한 설명입니다.

**-errorreport:prompt** 또는 **-errorreport:send** 와 함께 이 옵션을 사용할 경우 파일의 정보가 Microsoft Corporation에 전송됩니다.

모든 소스 코드 파일의 복사본이 `file`에 저장되기 때문에 최대한 짧은 프로그램으로 의심스러운 코드 결함을 재현하는 것이 좋습니다.

이 컴파일러 옵션은 Visual Studio에서 사용할 수 없으며 프로그래밍 방식으로 변경할 수 없습니다.

생성된 파일 내용에 의해 소스 코드가 노출되어, 의도하지 않게 정보가 공개될 수도 있습니다.

## 참고 항목

- [C# 컴파일러 옵션](#)
- [-errorreport\(C# 컴파일러 옵션\)](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -checked(C# 컴파일러 옵션)

2021-02-18 • 4 minutes to read • [Edit Online](#)

**-checked** 옵션은 데이터 형식 범위를 벗어나고 **checked** 또는 **unchecked** 키워드의 범위 내에 없는 값을 생성하는 정수 산술 문이 런타임 예외를 일으킬지 여부를 지정합니다.

## 구문

```
-checked[+ | -]
```

## 설명

**checked** 또는 **unchecked** 키워드의 범위 내에 있는 정수 산술 문에는 **-checked** 옵션이 영향을 미치지 않습니다.

**checked** 또는 **unchecked** 키워드의 범위에 없는 정수 산술 문이 데이터 형식 범위를 벗어난 값을 생성하고 **-checked+**(또는 **-checked**)가 컴파일에서 사용될 경우 해당 명령문은 런타임에 예외를 일으킵니다. **-checked-**가 컴파일에 사용될 경우 해당 문은 런타임에 예외를 일으키지 않습니다.

이 옵션의 기본값은 **-checked-**이며, 오버플로 검사는 해제되어 있습니다.

경우에 따라 대형 애플리케이션을 빌드하는 데 사용되는 자동화된 도구가 +로 **-checked**를 설정합니다. **-checked-**를 사용하는 하나의 시나리오는 **-checked-**를 지정하여 도구의 글로벌 기본값을 재정의하는 것입니다.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

- 프로젝트 속성 페이지를 엽니다. 자세한 내용은 [프로젝트 디자이너, 빌드 페이지\(C#\)](#)를 참조하세요.
- 빌드 속성 페이지를 클릭합니다.
- 고급 단추를 클릭합니다.
- 산술 연산 오버플로 확인 속성을 수정합니다.

프로그래밍 방식으로 이 컴파일러 옵션에 액세스하려면 [CheckForOverflowUnderflow](#)를 참조하세요.

## 예제

다음 명령은 **t2.cs**를 컴파일합니다. 명령에서 **-checked**를 사용하면 **checked** 또는 **unchecked** 키워드의 범위에 없고 데이터 형식 범위를 벗어난 값을 생성하는 파일의 정수 산술 문이 런타임에 예외를 일으키도록 지정합니다.

```
csc t2.cs -checked
```

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -codepage(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

이 옵션은 필수 페이지가 시스템에 대한 현재 기본 코드 페이지가 아닌 경우 컴파일 중에 사용할 코드 페이지를 지정합니다.

## 구문

```
-codepage:id
```

## 인수

`id`

컴파일할 때 모든 소스 코드 파일에 사용할 코드 페이지의 ID입니다.

## 설명

컴파일러는 먼저 모든 소스 파일을 UTF-8로 해석하려고 합니다. 소스 코드 파일이 UTF-8 이외의 인코딩에 있고 7비트 ASCII 문자가 아닌 문자를 사용하는 경우 **-codepage** 옵션을 통해 사용할 코드 페이지를 지정합니다. **-codepage** 는 컴파일에 포함된 모든 소스 코드 파일에 적용됩니다.

시스템에서 지원되는 코드 페이지를 찾는 방법에 대한 자세한 내용은 [GetCPIInfo](#)를 참조하세요.

이 컴파일러 옵션은 Visual Studio에서 사용할 수 없으며 프로그래밍 방식으로 변경할 수 없습니다.

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -debug(C# 컴파일러 옵션)

2021-02-18 • 5 minutes to read • [Edit Online](#)

-debug 옵션을 사용하면 컴파일러에서 디버깅 정보를 생성하여 출력 파일에 넣습니다.

## 구문

```
-debug[+ | -]  
-debug:{full | pdbonly}
```

## 인수

+ | -

또는 **-debug** 를 지정하면 컴파일러에서 디버깅 정보를 생성하여 프로그램 데이터베이스(.pdb 파일)에 넣습니다.  를 지정하면 **-debug** 를 지정하지 않은 것으로 적용되어 디버그 정보가 생성되지 않습니다.

full | pdbonly

컴파일러에서 생성되는 디버깅 정보 형식을 지정합니다. **-debug:pdbonly** 를 지정하지 않을 경우 적용되는 전체 인수를 통해 실행 중인 프로그램에 디버거를 연결할 수 있습니다. **pdbonly**를 지정하면 디버거에서 프로그램이 시작되는 경우 소스 코드 디버깅이 가능하지만, 실행 중인 프로그램이 디버거에 연결되는 경우 어셈블러만 표시됩니다.

## 설명

디버그 빌드를 만들려면 이 옵션을 사용합니다. **-debug**, **-debug+** 또는 **-debug:full** 을 지정하지 않으면 프로그램의 출력 파일을 디버그할 수 없습니다.

**-debug:full** 을 사용하는 경우 JIT 최적화된 코드의 속도 및 크기와 **-debug:full** 을 사용한 코드 품질이 일부 영향을 받는 것에 유의하세요. 릴리스 코드를 생성하는 경우 **-debug:pdbonly** 를 사용하거나 PDB를 사용하지 않는 것이 좋습니다.

### NOTE

**-debug:pdbonly** 및 **-debug:full** 간의 차이점 중 하나는 **-debug:full** 을 사용하는 경우 컴파일러가

**DebuggableAttribute**를 내보낸다는 것입니다. 이 특성은 JIT 컴파일러에 디버그 정보를 사용할 수 있음을 알리는 데 사용됩니다. 따라서 **-debug:full** 을 사용하는 경우 **false**로 설정된 **DebuggableAttribute**가 코드에 포함되어 있으면 오류가 발생합니다.

애플리케이션의 디버그 성능을 구성하는 방법에 대한 자세한 내용은 [쉽게 디버그할 수 있도록 이미지 만들기](#)를 참조하세요.

.pdb 파일의 위치를 변경하려면 [-pdb\(C# 컴파일러 옵션\)](#)을 참조하세요.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

1. 프로젝트 속성 페이지를 엽니다.
2. 빌드 속성 페이지를 클릭합니다.
3. 고급 단추를 클릭합니다.
4. 디버그 정보 속성을 수정합니다.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [DebugSymbols](#)를 참조하세요.

## 예제

출력 파일 `app.pdb`에 디버깅 정보를 넣습니다.

```
csc -debug -pdb:app.pdb test.cs
```

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -define(C# 컴파일러 옵션)

2021-02-18 • 4 minutes to read • [Edit Online](#)

-define 옵션은 프로그램의 모든 소스 코드 파일에서 `name`을 기호로 정의합니다.

## 구문

```
-define:name[;name2]
```

## 인수

`name`, `name2`

정의하려는 하나 이상의 기호 이름입니다.

## 설명

-define 옵션은 컴파일러 옵션이 프로젝트의 모든 파일에 적용된다는 점을 제외하고는 `#define` 전처리기 지시문을 사용하는 것과 효과가 동일합니다. 소스 파일의 `#undef` 지시문이 정의를 제거할 때까지 기호는 소스 파일에 정의된 상태로 유지됩니다. -define 옵션을 사용하는 경우 한 파일의 `#undef` 지시문이 프로젝트의 다른 소스 코드 파일에는 영향을 주지 않습니다.

이 옵션으로 만든 기호를 `#if`, `#else`, `#elif`, `#endif`와 함께 사용하면 소스 파일을 조건부 컴파일 할 수 있습니다.

-d 는 -define 의 약식 형태입니다.

세미콜론 또는 쉼표를 사용해서 기호 이름을 구분하여 -define 으로 여러 기호를 정의할 수 있습니다. 예:

```
-define:DEBUG;TUESDAY
```

C# 컴파일러 자체는 소스 코드에서 사용할 수 있는 기호 또는 매크로를 정의하지 않습니다. 모든 기호 정의는 사용자가 정의해야 합니다.

### NOTE

C# `#define`에서는 C++와 같은 언어에서처럼 기호에 값을 지정할 수 없습니다. 예를 들어 `#define`을 사용하여 매크로를 만들거나 상수를 정의할 수 없습니다. 상수를 정의해야 하는 경우 `enum` 변수를 사용합니다. C++ 스타일 매크로를 만들려는 경우 제네릭과 같은 다른 방식을 고려해 보세요. 매크로는 오류가 발생할 가능성이 크므로 C#에서는 매크로를 사용할 수 없으며 더 안전한 방식이 사용됩니다.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

- 프로젝트 속성 페이지를 엽니다.
- 빌드 탭의 조건부 컴파일 기호 상자에 정의할 기호를 입력합니다. 예를 들어 다음 코드 예제를 사용하는 경우 텍스트 상자에 `xx`를 입력합니다.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [DefineConstants](#)를 참조하세요.

## 예제

```
// preprocessor_define.cs
// compile with: -define:xx
// or uncomment the next line
// #define xx
using System;
public class Test
{
    public static void Main()
    {
        #if (xx)
            Console.WriteLine("xx defined");
        #else
            Console.WriteLine("xx not defined");
        #endif
    }
}
```

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -delaySign(C# 컴파일러 옵션)

2021-02-18 • 3 minutes to read • [Edit Online](#)

이 옵션을 사용하면 나중에 디지털 시그니처를 추가할 수 있도록 컴파일러가 출력 파일에 공간을 예약합니다.

## 구문

```
-delaySign[ + | - ]
```

## 인수

+ | -

완전히 서명된 어셈블리가 필요하면 **-delaySign-** 를 사용합니다. 어셈블리에 공개 키만 배치하려면 **-delaySign+** 를 사용합니다. 기본값은 **-delaySign-** 입니다.

## 설명

**-delaySign** 옵션은 [-keyfile](#) 또는 [-keycontainer](#)와 함께 사용하지 않으면 효과가 없습니다.

**-delaySign** 및 **-publicsign** 옵션은 함께 사용할 수 없습니다.

완전히 서명된 어셈블리를 요청할 경우 컴파일러는 매니페스트(어셈블리 메타데이터)가 포함된 파일을 해시하고 프라이빗 키로 해당 해시에 서명합니다. 해당 작업으로 디지털 시그니처가 생성되어 매니페스트가 포함된 파일에 저장됩니다. 어셈블리 서명이 연기된 경우 컴파일러는 서명을 컴퓨팅하거나 저장하지 않고 나중에 서명을 추가할 수 있도록 파일에 공간을 예약합니다.

예를 들어 **-delaySign+** 를 사용하면 테스터를 통해 전역 캐시에 어셈블리를 넣을 수 있습니다. 테스트를 마친 후 [어셈블리 링커](#) 유ти리티를 통해 어셈블리에 프라이빗 키를 배치하여 어셈블리에 완전히 서명할 수 있습니다.

자세한 내용은 [강력한 이름의 어셈블리 만들기 및 사용 및 어셈블리 서명 지연](#)을 참조하세요.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

- 프로젝트의 속성 페이지를 엽니다.
- 서명만 연기 속성을 수정합니다.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [DelaySign](#)를 참조하세요.

## 참고 항목

- [C# -publicsign 옵션](#)
- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -deterministic

2021-02-18 • 4 minutes to read • [Edit Online](#)

컴파일러에서 동일한 입력에 대한 컴파일 간에 바이트 단위(byte-for-byte) 출력이 동일한 어셈블리를 생성합니다.

## 구문

-deterministic

## 설명

기본적으로 컴파일러에서 타임스탬프 및 난수에서 생성된 MVID를 추가하기 때문에 지정된 입력 세트의 컴파일러 출력은 고유합니다. `-deterministic` 옵션을 사용하여 결정적 어셈블리를 생성하고, 입력이 동일하게 유지되는 한 해당 이진 콘텐츠가 컴파일 간에 동일합니다. 해당 빌드에서 타임스탬프 및 MVID 필드는 모든 컴파일 입력의 해시에서 파생된 값으로 대체됩니다.

결정성의 목적을 위해 컴파일러에서 고려하는 입력은 다음과 같습니다.

- 명령줄 매개 변수의 순서
- 컴파일러의 .rsp 지시 파일의 내용
- 사용된 컴파일러의 정확한 버전 및 참조되는 어셈블리
- 현재 디렉터리 경로
- 다음을 포함하여 직접 또는 간접적으로 컴파일러에 명시적으로 전달된 모든 파일의 이진 콘텐츠
  - 소스 파일
  - 참조된 어셈블리
  - 참조된 모듈
  - 리소스
  - 강력한 이름 키 파일
  - @ 지시 파일
  - 분석기
  - 규칙 집합
  - 분석기에서 사용할 수 있는 추가 파일
- 현재 문화권(진단 및 예외 메시지가 생성되는 언어)
- 인코딩이 지정되지 않은 경우 기본 인코딩(또는 현재 코드 페이지)
- 컴파일러의 검색 경로(예: `-lib` 또는 `-recurse`로 지정)에 있는 파일의 존재 여부 및 내용
- 컴파일러가 실행되는 CLR 플랫폼
- `%LIBPATH%` 값(분석기 종속성 로드에 영향을 줄 수 있음)

원본을 공개적으로 사용할 수 있는 경우 결정적 컴파일을 사용하여 이진 파일이 신뢰할 수 있는 원본에서 컴파일되는지 여부를 설정할 수 있습니다. 또한 연속 빌드 시스템에서 이진 파일 변경에 종속된 빌드 단계를 실행해야 하는지 여부를 확인하는데 유용할 수도 있습니다.

## 참조

- [C# 컴파일러 옵션](#)

- 프로젝트 및 솔루션 속성 관리

# /doc(C# 컴파일러 옵션)

2021-02-18 • 4 minutes to read • [Edit Online](#)

-doc 옵션을 사용하면 XML 파일에 문서 주석을 삽입할 수 있습니다.

## 구문

```
-doc:file
```

## 인수

file

XML에 대한 출력 파일로, 컴파일의 소스 코드 파일에 있는 주석으로 채워집니다.

## 설명

소스 코드 파일에서 다음 항목 앞에 나오는 문서 주석을 처리하여 XML 파일에 추가할 수 있습니다.

- `class`, `delegate` 또는 `interface` 같은 사용자 정의 형식
- 필드, `이벤트`, `속성` 또는 메서드 같은 멤버

Main을 포함하는 소스 코드 파일이 먼저 XML로 출력됩니다.

IntelliSense 기능에서 사용하기 위해 생성된 .xml 파일을 사용하려면 .xml 파일의 이름을 지원하려는 어셈블리와 동일하게 지정한 다음 .xml 파일이 어셈블리와 동일한 디렉터리에 있도록 해야 합니다. 따라서 어셈블리가 Visual Studio 프로젝트에서 참조되면 .xml 파일도 검색됩니다. 자세한 내용은 [코드 주석 제공](#)을 참조하세요.

-target:module을 사용하여 컴파일하지 않으면 컴파일의 출력 파일에 대한 어셈블리 매니페스트를 포함하는 파일의 이름을 지정하는 `<assembly></assembly>` 태그가 file에 포함됩니다.

### NOTE

-doc 옵션은 모든 입력 파일에 적용됩니다. 또는 Project Settings에서 설정한 경우 프로젝트의 모든 파일에 적용됩니다. 특정 파일 또는 코드 섹션에 대한 문서 주석 관련 경고를 사용하지 않으려면 `#pragma warning`을 사용하세요.

코드의 주석에서 문서를 생성하는 방법은 [문서 주석에 대한 권장 태그](#)를 참조하세요.

**Visual Studio 2019** 개발 환경에서 이 컴파일러 옵션을 설정하려면

1. 프로젝트 속성 페이지를 엽니다.
2. 빌드 탭을 클릭합니다.
3. XML 문서 파일 속성을 수정합니다.

**Mac용 Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

1. 프로젝트의 옵션 페이지를 엽니다.
2. 컴파일러 탭을 선택합니다.
3. XML 문서 생성을 선택하고 텍스트 상자에 파일 이름을 입력합니다.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [DocumentationFile](#)를 참조하세요.

## 참고 항목

- C# 컴파일러 옵션
- 프로젝트 및 솔루션 속성 관리

# -errorreport(C# 컴파일러 옵션)

2021-02-18 • 5 minutes to read • [Edit Online](#)

이 옵션은 C# 내부 컴파일러 오류를 Microsoft에 보고하는 편리한 방법을 제공합니다.

## NOTE

Windows Vista 및 Windows Server 2008에서 Visual Studio에 대한 오류 보고 설정은 WER(Windows 오류 보고)을 통한 설정을 재정의하지 않습니다. 항상 WER 설정이 Visual Studio 오류 보고 설정보다 우선합니다.

## 구문

```
-errorreport:{ none | prompt | queue | send }
```

## 인수

### 없음

내부 컴파일러 오류에 대한 보고서를 수집하거나 Microsoft로 보내지 않습니다.

**prompt** 는 내부 컴파일러 오류가 발생하면 보고서를 보낼지 묻는 메시지를 표시합니다. **prompt** 는 개발 환경에서 애플리케이션을 컴파일할 때 기본값입니다.

**queue** 는 오류 보고서를 큐에 넣습니다. 관리자 자격 증명으로 로그온하면 마지막으로 로그온한 시간 이후에 발생한 모든 오류를 보고할 수 있습니다. 3일에 두 번 이상 오류 보고서를 보내라는 메시지가 표시되지는 않습니다. **queue** 는 명령줄에서 애플리케이션을 컴파일할 때의 기본값입니다.

**send** 는 내부 컴파일러 오류 보고서를 Microsoft에 자동으로 보냅니다. 이 옵션을 사용하려면 먼저 Microsoft 데이터 수집 정책에 동의해야 합니다. 처음으로 컴퓨터에서 **-errorreport:send** 를 지정하면 컴파일러 메시지에서 Microsoft 데이터 수집 정책이 포함된 Web 사이트를 참조합니다.

## 설명

내부 컴파일러 오류(ICE)는 컴파일러에서 소스 코드 파일을 처리할 수 없을 때 발생합니다. ICE가 발생하면 컴파일러에서 출력 파일 또는 코드를 수정하는데 사용할 수 있는 유용한 진단을 생성하지 않습니다.

이전 릴리스에서는 ICE를 수신하는 경우 Microsoft 기술 지원 서비스에 문의하여 문제를 보고하도록 했습니다. **-errorreport** 를 사용하여 Visual C#팀에 ICE 정보를 제공할 수 있습니다. 오류 보고서는 향후 컴파일러 릴리스를 개선하는 데 도움이 됩니다.

사용자가 보고서를 보내는 능력은 컴퓨터 및 사용자 정책 권한에 따라 달라집니다.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

- 프로젝트 속성 페이지를 엽니다. 자세한 내용은 [프로젝트 디자이너, 빌드 페이지\(C#\)](#)를 참조하세요.
- 빌드 속성 페이지를 클릭합니다.
- 고급 단추를 클릭합니다.
- 내부 컴파일러 오류 보고 속성을 수정합니다.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [ErrorReport](#)을 참조하십시오.

## 참고 항목

- C# 컴파일러 옵션

# -filealign(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

-filealign 옵션을 사용하여 출력 파일의 섹션 크기를 지정할 수 있습니다.

## 구문

```
-filealign:number
```

## 인수

number

출력 파일의 섹션 크기를 지하는 값입니다. 유효한 값은 512, 1024, 2048, 4096 및 8192입니다. 이러한 값은 바이트 단위입니다.

## 설명

각 섹션은 -filealign 값의 배수인 경계에 맞춰집니다. 고정된 기본값이 없습니다. -filealign 을 지정하지 않으면 공용언어 런타임에서 컴파일 시간에 기본값을 선택합니다.

섹션 크기를 지정하면 출력 파일의 크기에 영향을 줍니다. 섹션 크기를 수정하면 더 작은 디바이스에서 실행되는 프로그램에 유용할 수 있습니다.

출력 파일의 섹션에 대한 정보를 확인하려면 [DUMPBIN](#)을 사용하세요.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

1. 프로젝트 속성 페이지를 엽니다.
2. 빌드 속성 페이지를 클릭합니다.
3. 고급 단추를 클릭합니다.
4. 파일 맞춤 속성을 수정합니다.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [FileAlignment](#)을 참조하세요.

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -fullpaths(C# 컴파일러 옵션)

2020-11-02 • 2 minutes to read • [Edit Online](#)

**-fullpaths** 옵션을 사용하면 컴파일러에서는 컴파일 오류 및 경고 목록을 만들 때 파일의 전체 경로를 지정합니다.

## 구문

```
-fullpaths
```

## 설명

기본적으로 컴파일 도중 발생하는 오류와 경고에서는 오류가 발견된 파일의 이름을 지정합니다. **-fullpaths** 옵션을 사용하면 컴파일러에서는 파일의 전체 경로를 지정합니다.

이 컴파일러 옵션은 Visual Studio에서 사용할 수 없으며 프로그래밍 방식으로 변경할 수 없습니다.

## 참고 항목

- [C# 컴파일러 옵션](#)

# -help, -? (C# 컴파일러 옵션)

2020-11-02 • 2 minutes to read • [Edit Online](#)

이 옵션은 컴파일러 옵션 목록과 각 옵션에 대한 간략한 설명을 stdout에 보냅니다.

## 구문

```
-help  
-?
```

## 설명

이 옵션이 컴파일에 포함되어 있으면 출력 파일이 만들어지지 않으며 컴파일이 수행되지 않습니다.

이 컴파일러 옵션은 Visual Studio에서 사용할 수 없으며 프로그래밍 방식으로 변경할 수 없습니다.

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -highentropyva(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

**-highentropyva** 컴파일러 옵션은 특정 실행 파일이 높은 엔트로피 ASLR(Address Space Layout Randomization)을 지원하는지 여부를 Windows 커널에 알려줍니다.

## 구문

```
-highentropyva[+ | -]
```

## 인수

|

이 옵션은 [-platform:anycpu](#) 컴파일러 옵션으로 표시된 실행 파일이나 64비트 실행 파일이 높은 엔트로피 가상 주소 공간을 지원하도록 지정합니다. 이 옵션은 기본적으로 비활성화되어 있습니다. **-highentropyva+** 또는 **-highentropyva**를 사용하여 설정합니다.

## 설명

**-highentropyva** 옵션은 Windows 커널의 호환되는 버전에서 ASLR의 일부로 프로세스의 주소 공간 레이아웃을 임의로 선택할 때 보다 높은 수준의 엔트로피를 사용할 수 있게 합니다. 더 높은 수준의 엔트로피를 사용하면 스택 및 힙과 같은 메모리 영역에 더 많은 주소를 할당할 수 있습니다. 따라서 특정 메모리 영역의 위치를 추측하기 어려워집니다.

**-highentropyva** 컴파일러 옵션을 지정하면 대상 실행 파일과 이 실행 파일이 종속된 모든 모듈이 64비트 프로세스로 실행될 때 4GB(기가바이트)보다 큰 포인터 값을 처리할 수 있어야 합니다.

# -keycontainer(C# 컴파일러 옵션)

2021-02-18 • 3 minutes to read • [Edit Online](#)

암호화 키 컨테이너의 이름을 지정합니다.

## 구문

```
-keycontainer:string
```

## 인수

`string`

강력한 이름 키 컨테이너의 이름입니다.

## 설명

**-keycontainer** 옵션을 사용하면 컴파일러는 공유 가능한 구성 요소를 만듭니다. 컴파일러는 지정된 컨테이너의 퍼블릭 키를 어셈블리 매니페스트에 삽입하고 프라이빗 키를 사용하여 최종 어셈블리에 서명합니다. 키 파일을 생성하려면 명령줄에 `sn -k file` 을 입력합니다. `sn -i` 는 컨테이너에 키 쌍을 설치합니다. CoreCLR에서 컴파일러를 실행하면 이 옵션이 지원되지 않습니다. CoreCLR에서 빌드할 때 어셈블리에 서명하려면 **-keyfile** 옵션을 사용합니다.

**-target:module**로 컴파일하는 경우 키 파일의 이름이 모듈에 저장되고, **-addmodule**을 사용하여 이 모듈을 어셈블리로 컴파일할 때 어셈블리에 통합됩니다.

MSIL(Microsoft Intermediate Language) 모듈의 소스 코드에서 이 옵션을 사용자 지정 특성 (`System.Reflection.AssemblyKeyNameAttribute`)으로 지정할 수도 있습니다.

**-keyfile**을 사용하여 암호화 정보를 컴파일러에 전달할 수도 있습니다. 공개 키를 어셈블리 매니페스트에 추가하고자 하지만 테스트가 완료될 때까지 어셈블리 서명을 지연하려면 **-delaysign**을 사용합니다.

자세한 내용은 [강력한 이름의 어셈블리 만들기 및 사용 및 어셈블리 서명 지연](#)을 참조하세요.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

1. 이 컴파일러 옵션은 Visual Studio 개발 환경에서 사용할 수 없습니다.

프로그래밍 방식으로 `AssemblyKeyName`을 사용하여 이 컴파일러 옵션에 액세스할 수 있습니다.

## 참고 항목

- [C# 컴파일러 -keyfile 옵션](#)
- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -keyfile(C# 컴파일러 옵션)

2021-02-18 • 4 minutes to read • [Edit Online](#)

암호화 키를 포함하는 파일 이름을 지정합니다.

## 구문

```
-keyfile:file
```

## 인수

용어	정의
file	강력한 이름 키를 포함하는 파일의 이름입니다.

## 설명

이 옵션을 사용하면 컴파일러는 지정된 파일의 퍼블릭 키를 어셈블리 매니페스트에 삽입한 다음 프라이빗 키를 사용하여 최종 어셈블리에 서명합니다. 키 파일을 생성하려면 명령줄에 `sn -k file`을 입력합니다.

**-target:module**로 컴파일하는 경우 키 파일의 이름이 모듈에 저장되고, **-addmodule**을 사용하여 어셈블리를 컴파일할 때 생성되는 어셈블리에 통합됩니다.

**-keycontainer**를 사용하여 암호화 정보를 컴파일러에 전달할 수도 있습니다. 부분 서명된 어셈블리가 필요한 경우 **-delaysign**을 사용합니다.

동일한 컴파일에서 **-keyfile** 및 **-keycontainer**를 명령줄 옵션이나 사용자 지정 특성으로 둘 다 지정하면 컴파일러는 먼저 키 컨테이너를 찾습니다. 키 컨테이너를 찾으면 키 컨테이너의 정보를 사용하여 어셈블리가 서명됩니다. 컴파일러는 키 컨테이너를 찾지 못할 경우 **-keyfile**로 지정된 파일을 찾습니다. 해당 파일을 찾으면 키 파일의 정보를 사용하여 어셈블리가 서명되고, 키 정보가 키 컨테이너에 설치되므로(`sn -i`와 유사) 다음에 컴파일할 때 키 컨테이너가 유효해집니다.

키 파일에는 공개 키만 포함될 수 있습니다.

자세한 내용은 [강력한 이름의 어셈블리 만들기 및 사용 및 어셈블리 서명 지연](#)을 참조하세요.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

1. [프로젝트의 속성 페이지](#)를 엽니다.
2. 서명 속성 페이지를 클릭합니다.
3. 강력한 이름 키 파일 선택 속성을 수정합니다.

프로그래밍 방식으로 [AssemblyOriginatorKeyFile](#)을 사용하여 이 컴파일러 옵션에 액세스할 수 있습니다.

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -langversion(C# 컴파일러 옵션)

2020-11-02 • 9 minutes to read • [Edit Online](#)

컴파일러가 선택한 C# 언어 사양에 포함된 구문만 허용하도록 합니다.

## 구문

-langversion:option

## 인수

option

유효한 값은 다음과 같습니다.

값	의미
preview	컴파일러가 최신 미리 보기 버전의 유효한 언어 구문을 모두 허용합니다.
latest	컴파일러가 최신 릴리스 버전(부 버전 포함)의 구문을 허용합니다.
latestMajor ( default )	컴파일러가 최신 릴리스 주 버전의 구문을 허용합니다.
9.0	컴파일러는 C# 9.0 이하에 포함된 구문만 허용합니다.
8.0	컴파일러는 C# 8.0 이하에 포함된 구문만 허용합니다.
7.3	컴파일러는 C# 7.3 이하에 포함된 구문만 허용합니다.
7.2	컴파일러는 C# 7.2 이하에 포함된 구문만 허용합니다.
7.1	컴파일러는 C# 7.1 이하에 포함된 구문만 허용합니다.
7	컴파일러는 C# 7.0 이하에 포함된 구문만 허용합니다.
6	컴파일러는 C# 6.0 이하에 포함된 구문만 허용합니다.
5	컴파일러는 C# 5.0 이하에 포함된 구문만 허용합니다.
4	컴파일러는 C# 4.0 이하에 포함된 구문만 허용합니다.
3	컴파일러는 C# 3.0 이하에 포함된 구문만 허용합니다.
ISO-2 (또는 2)	컴파일러는 ISO/IEC 23270:2006 C#(2.0)에 포함된 구문만 허용합니다.

값	의미
ISO-1 (또는 1)	컴파일러는 ISO/IEC 23270:2003 C#(1.0/1.2)에 포함된 구문만 허용합니다.

기본 언어 버전은 애플리케이션의 대상 프레임워크 및 SDK 또는 Visual Studio의 버전에 따라 다릅니다. 해당 규칙은 [언어 버전 구성](#) 문서에 정의되어 있습니다.

## 설명

C# 애플리케이션에서 참조된 메타데이터에는 **-langversion** 컴파일러 옵션이 적용되지 않습니다.

각 C# 컴파일러 버전에 언어 사양의 확장이 포함되어 있으므로 **-langversion**은 이전 컴파일러 버전의 동등한 기능을 제공하지 않습니다.

또한 C# 버전 업데이트는 일반적으로 주 .NET Framework 릴리스와 일치하는 반면, 새 구문과 기능이 반드시 특정 프레임워크 버전에 연결되지는 않습니다. 새로운 기능에는 C# 버전과 함께 릴리스된 새 컴파일러 업데이트가 분명히 필요하지만, 각 특정 기능에 고유한 최소 .NET API 또는 공용 언어 런타임 요구 사항이 있으므로 NuGet 패키지 또는 다른 라이브러리를 포함하여 하위 수준 프레임워크에서 실행이 허용될 수도 있습니다.

사용하는 **-langversion** 설정과 관계없이 현재 버전의 공용 언어 런타임을 사용하여 고유한 .exe 또는 .dll을 만듭니다. 한 가지 예외는 friend 어셈블리와 [-moduleassemblyname\(C# 컴파일러 옵션\)](#)으로, **-langversion:ISO-1**에서 작동합니다.

C# 언어 버전을 지정하는 다른 방법은 [C# 언어 버전 선택](#) 문서를 참조하세요.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [LanguageVersion](#)을 참조하십시오.

## C# 언어 사양

버전	링크	설명
C# 7.0 이상		현재 사용할 수 없음
C# 6.0	<a href="#">링크</a>	C# 언어 사양 버전 6 - 비공식 초안: .NET Foundation
C# 5.0	<a href="#">PDF 다운로드</a>	표준 ECMA-334 다섯 번째 버전
C# 3.0	<a href="#">DOC 다운로드</a>	C# 언어 사양 버전 3.0: Microsoft Corporation
C# 2.0	<a href="#">PDF 다운로드</a>	표준 ECMA-334 네 번째 버전
C# 1.2	<a href="#">DOC 다운로드</a>	C# 언어 사양 버전 1.2: Microsoft Corporation
C# 1.0	<a href="#">DOC 다운로드</a>	C# 언어 사양 버전 1.0: Microsoft Corporation

## 모든 언어 기능을 지원하는 데 필요한 최소 SDK 버전

다음 표에서는 해당 언어 버전을 지원하는 C# 컴파일러가 포함된 SDK의 최소 버전을 보여줍니다.

C# 버전	최소 SDK 버전
C# 8.0	Microsoft Visual Studio/Build Tools 2019, 버전 16.3 또는 .NET Core 3.0 SDK
C# 7.3	Microsoft Visual Studio/Build Tools 2017 버전 15.7
C# 7.2	Microsoft Visual Studio/Build Tools 2017 버전 15.5
C# 7.1	Microsoft Visual Studio/Build Tools 2017 버전 15.3
C# 7.0	Microsoft Visual Studio/Build Tools 2017
C# 6	Microsoft Visual Studio/Build Tools 2015
C# 5	Microsoft Visual Studio/Build Tools 2012 또는 번들 .NET Framework 4.5 컴파일러
C# 4	Microsoft Visual Studio/Build Tools 2010 또는 번들 .NET Framework 4.0 컴파일러
C# 3	Microsoft Visual Studio/Build Tools 2008 또는 번들 .NET Framework 3.5 컴파일러
C# 2	Microsoft Visual Studio/Build Tools 2005 또는 번들 .NET Framework 2.0 컴파일러
C# 1.0/1.2	Microsoft Visual Studio/Build Tools .NET 2002 또는 번들된 .NET Framework 1.0 컴파일러

## 참조

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -lib(C# 컴파일러 옵션)

2021-02-18 • 5 minutes to read • [Edit Online](#)

-lib 옵션은 [-reference\(C# 컴파일러 옵션\)](#) 옵션을 통해 참조되는 어셈블리의 위치를 지정합니다.

## 구문

```
-lib:dir1[,dir2]
```

## 인수

dir1

참조된 어셈블리를 현재 작업 디렉터리(컴파일러를 호출하는 디렉터리) 또는 공용 언어 런타임의 시스템 디렉터리에서 찾을 수 없는 경우 컴파일러에서 확인할 디렉터리입니다.

dir2

어셈블리 참조를 검색할 하나 이상의 추가 디렉터리입니다. 이를 사이에 공백 없이 추가 디렉터리 이름을 쉼표로 구분합니다.

## 설명

컴파일러는 정규화되지 않은 어셈블리 참조를 다음 순서대로 검색합니다.

1. 현재 작업 디렉터리입니다. 컴파일러가 호출되는 디렉터리입니다.
2. 공용 언어 런타임 시스템 디렉터리입니다.
3. -lib 로 지정된 디렉터리입니다.
4. LIB 환경 변수로 지정된 디렉터리입니다.

어셈블리 참조를 지정하려면 [-reference](#) 를 사용합니다.

-lib 는 가감되므로 두 번 이상 지정하면 이전 값에 추가됩니다.

-lib 를 사용하는 대신 필요한 모든 어셈블리를 작업 디렉터리에 복사할 수도 있습니다. 이렇게 하면 단순히 어셈블리 이름을 [-reference](#) 에 전달할 수 있습니다. 그런 다음 작업 디렉터리에서 어셈블리를 삭제할 수 있습니다. 종속 어셈블리의 경로는 어셈블리 매니페스트에 지정되지 않으므로 애플리케이션이 대상 컴퓨터에서 시작될 수 있으며, 전역 어셈블리 캐시에서 어셈블리를 찾아 사용합니다.

컴파일러가 어셈블리를 참조할 수 있다고 해서 공용 언어 런타임이 런타임에 어셈블리를 찾아 로드할 수 있다는 의미는 아닙니다. 런타임에서 참조된 어셈블리를 검색하는 방법에 대한 자세한 내용은 [런타임에서 어셈블리를 찾는 방법](#)을 참조하세요.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

1. 프로젝트의 속성 페이지 대화 상자를 엽니다.
2. 참조 경로 속성 페이지를 클릭합니다.
3. 목록 상자 내용을 수정합니다.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [ReferencePath](#)를 참조하세요.

## 예제

t2.cs를 컴파일하여 .exe 파일을 만듭니다. 컴파일러는 작업 디렉터리 및 C 드라이브의 루트 디렉터리에서 어셈블리 참조를 찾습니다.

```
csc -lib:c:\ -reference:t2.dll t2.cs
```

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -link(C# 컴파일러 옵션)

2020-11-02 • 10 minutes to read • [Edit Online](#)

컴파일러에서 지정된 어셈블리의 COM 형식 정보를 현재 컴파일하고 있는 프로젝트에 사용할 수 있도록 합니다.

## 구문

```
-link:fileList
// -or-
-l:fileList
```

## 인수

### **fileList**

필수 요소. 쉼표로 구분된 어셈블리 파일 이름 목록입니다. 파일 이름에 공백이 있으면 이름을 따옴표로 묶습니다.

## 설명

**-link** 옵션을 사용하면 포함된 형식 정보가 있는 애플리케이션을 배포할 수 있습니다. 그러면 애플리케이션은 런타임 어셈블리에 대한 참조를 요구하지 않고 포함된 형식 정보를 구현하는 형식을 런타임 어셈블리에서 사용할 수 있습니다. 다양한 버전의 런타임 어셈블리가 게시된 경우 포함된 형식 정보를 포함하는 애플리케이션은 다시 컴파일하지 않아도 다양한 버전에서 사용할 수 있습니다. 예제를 보려면 [연습: 관리되는 어셈블리의 형식 포함](#)을 참조하세요.

**-link** 옵션을 사용하면 COM interop를 사용하여 작업할 때 특히 유용합니다. 애플리케이션에 대상 컴퓨터의 PIA(주 interop 어셈블리)가 더 이상 필요하지 않도록 COM 형식을 포함할 수 있습니다. **-link** 옵션은 참조된 interop 어셈블리의 COM 형식 정보를 컴파일된 결과 코드에 포함하도록 컴파일러에 지시합니다. COM 형식은 CLSID(GUID) 값으로 식별됩니다. 따라서 동일한 CLSID 값을 갖는 동일한 COM 형식이 설치된 대상 컴퓨터에서 애플리케이션을 실행할 수 있습니다. Microsoft Office를 자동화하는 애플리케이션이 좋은 예입니다. Office와 같은 애플리케이션은 일반적으로 여러 버전에서 동일한 CLSID 값을 유지하지 때문에 .NET Framework 4 이상이 대상 컴퓨터에 설치되어 있고 애플리케이션이 참조된 COM 형식에 포함된 메서드, 속성 또는 이벤트를 사용하는 한 애플리케이션에서 참조된 COM 형식을 사용할 수 있습니다.

**-link** 옵션은 인터페이스, 구조체 및 대리자만 포함합니다. COM 클래스는 포함할 수 없습니다.

### NOTE

코드에서 포함된 COM 형식의 인스턴스를 만드는 경우 적절한 인터페이스를 사용하여 인스턴스를 만들어야 합니다. CoClass를 사용하여 포함된 COM 형식의 인스턴스를 만들려고 하면 오류가 발생합니다.

Visual Studio에서 **-link** 옵션을 설정하려면 어셈블리 참조를 추가하고 **Embed Interop Types** 속성을 **true**로 설정합니다. **Embed Interop Types** 속성의 기본값은 **false**입니다.

COM 어셈블리 자체가 또 다른 COM 어셈블리(어셈블리 B)를 참조하는 COM 어셈블리(어셈블리 A)에 연결하는 경우 다음 중 하나에 해당되면 어셈블리 B에도 연결해야 합니다.

- 어셈블리 A의 형식은 형식에서 상속되거나 어셈블리 B의 인터페이스를 구현합니다.

- 어셈블리 B의 반환 형식이나 매개 변수 형식을 사용하는 필드, 속성, 이벤트 또는 메서드가 호출됩니다.

[-reference](#) 컴파일러 옵션과 마찬가지로, [-link](#) 컴파일러 옵션은 Csc.rsp 지시 파일을 사용하며, 자주 사용되는 .NET 어셈블리를 참조합니다. 컴파일러에서 Csc.rsp 파일을 사용하지 않도록 하려면 [-noconfig](#) 컴파일러 옵션을 사용하세요.

[-link](#)의 약식은 [-l](#)입니다.

## 제네릭 및 포함된 형식

다음 섹션에서는 interop 형식을 포함하는 애플리케이션에서 제네릭 형식을 사용할 경우의 제한 사항에 대해 설명합니다.

### 제네릭 인터페이스

interop 어셈블리에서 포함되는 제네릭 인터페이스는 사용할 수 없습니다. 다음 예제에서 이를 확인할 수 있습니다.

```
// The following code causes an error if ISampleInterface is an embedded interop type.
ISampleInterface<SampleType> sample;
```

### 제네릭 매개 변수가 있는 형식

형식이 interop 어셈블리에서 포함된 제네릭 매개 변수가 있는 형식은 해당 형식이 외부 어셈블리에서 제공된 경우 사용할 수 없습니다. 인터페이스에는 이 제한이 적용되지 않습니다. 예를 들어

[Microsoft.Office.Interop.Excel](#) 어셈블리에 정의된 [Range](#) 인터페이스를 살펴보세요. 다음 코드 예제와 같이 라이브러리가 [Microsoft.Office.Interop.Excel](#) 어셈블리의 interop 형식을 포함하고 형식이 [Range](#) 인터페이스인 매개 변수가 있는 제네릭 형식을 반환하는 메서드를 노출하는 경우 해당 메서드는 제네릭 인터페이스를 반환해야 합니다.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Office.Interop.Excel;

public class Utility
{
    // The following code causes an error when called by a client assembly.
    public List<Range> GetRange1()
    {
        return null;
    }

    // The following code is valid for calls from a client assembly.
    public IList<Range> GetRange2()
    {
        return null;
    }
}
```

다음 예제에서 클라이언트 코드는 [IList](#) 제네릭 인터페이스를 반환하는 메서드를 오류 없이 호출할 수 있습니다.

```
public class Client
{
    public void Main()
    {
        Utility util = new Utility();

        // The following code causes an error.
        List<Range> rangeList1 = util.GetRange1();

        // The following code is valid.
        List<Range> rangeList2 = (List<Range>)util.GetRange2();
    }
}
```

## 예제

다음 코드는 소스 파일 `OfficeApp.cs` 와 `COMData1.dll` 및 `COMData2.dll` 의 참조 어셈블리를 컴파일하여 `OfficeApp.exe` 를 생성합니다.

```
csc -link:COMData1.dll,COMData2.dll -out:OfficeApp.exe OfficeApp.cs
```

## 참고 항목

- [C# 컴파일러 옵션](#)
- [연습: 관리되는 어셈블리의 형식 포함](#)
- [-reference\(C# 컴파일러 옵션\)](#)
- [-noconfig\(C# 컴파일러 옵션\)](#)
- [csc.exe를 사용한 명령줄 빌드](#)
- [상호 운용성 개요](#)

# -linkresource(C# 컴파일러 옵션)

2021-02-18 • 5 minutes to read • [Edit Online](#)

출력 파일에 .NET 리소스에 대한 링크를 만듭니다. 리소스 파일은 출력 파일에 추가되지 않습니다. 이 점에서 출력 파일에 리소스 파일을 포함하는 [-resource](#) 옵션과 다릅니다.

## 구문

```
-linkresource:filename[,identifier[,accessibility-modifier]]
```

## 인수

`filename`

어셈블리에서 연결할 .NET 리소스 파일입니다.

`identifier` (선택 사항)

리소스의 논리적 이름으로, 리소스를 로드하는 데 사용되는 이름입니다. 기본값은 파일 이름입니다.

`accessibility-modifier` (선택 사항)

리소스의 접근성으로, `public` 또는 `private`입니다. 기본값은 `public`입니다.

## 설명

기본적으로 연결된 리소스는 C# 컴파일러로 생성될 때 어셈블리에서 `public`입니다. 리소스를 `private`로 만들려면 접근성 한정자로 `private`를 지정합니다. `public` 또는 `private` 이외의 다른 한정자는 허용되지 않습니다.

`-linkresource`에는 `-target:module` 이외의 [-target](#) 옵션 중 하나가 필요합니다.

예를 들어 `filename`이 [Resgen.exe](#) 또는 개발 환경에서 만들어진 .NET 리소스 파일인 경우에는 [System.Resources](#) 네임스페이스의 멤버를 사용하여 해당 파일에 액세스할 수 있습니다. 자세한 내용은 [System.Resources.ResourceManager](#)를 참조하세요. 다른 모든 리소스의 경우에는 런타임에 `GetManifestResource` 클래스의 [Assembly](#) 메서드를 사용하여 리소스에 액세스합니다.

`filename`에 지정된 파일은 모든 형식일 수 있습니다. 예를 들어 네이티브 DLL을 어셈블리의 일부로 설정하면 전역 어셈블리 캐시에 설치하고 어셈블리의 관리 코드에서 액세스할 수 있습니다. 다음 중 두 번째 예제에서 이 작업을 수행하는 방법을 보여 줍니다. 어셈블리 링커에서도 동일한 작업을 수행할 수 있습니다. 다음 중 세 번째 예제에서 이 작업을 수행하는 방법을 보여 줍니다. 자세한 내용은 [Al.exe\(어셈블리 링커\)](#) 및 [어셈블리 및 전역 어셈블리 캐시 사용](#)을 참조하세요.

`-linkres`는 `-linkresource`의 약식입니다.

이 컴파일러 옵션은 Visual Studio에서 사용할 수 없으며 프로그래밍 방식으로 변경할 수 없습니다.

## 예제

`in.cs`를 컴파일하고 리소스 파일 `rf.resource`에 연결합니다.

```
csc -linkresource:rf.resource in.cs
```

## 예제

A.cs 를 DLL로 컴파일하고, 네이티브 DLL N.dll에 연결한 다음 출력을 GAC(전역 어셈블리 캐시)에 넣습니다. 이 예제에서는 A.dll과 N.dll이 둘 다 GAC에 있습니다.

```
csc -linkresource:N.dll -t:library A.cs  
gacutil -i A.dll
```

## 예제

이 예제에서는 앞의 예제와 동일한 작업을 수행하지만 어셈블리 링커 옵션을 사용합니다.

```
csc -t:module A.cs  
al -out:A.dll A.netmodule -link:N.dll  
gacutil -i A.dll
```

## 참고 항목

- [C# 컴파일러 옵션](#)
- [Al.exe\(어셈블리 링커\)](#)
- [어셈블리 및 전역 어셈블리 캐시 사용](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -main(C# 컴파일러 옵션)

2020-11-02 • 3 minutes to read • [Edit Online](#)

이 옵션은 둘 이상의 클래스에 **Main** 메서드가 포함된 경우 프로그램에 대한 진입점을 포함하는 클래스를 지정합니다.

## 구문

```
-main:class
```

## 인수

`class`

**Main** 메서드를 포함하는 형식입니다.

제공된 클래스 이름은 완전히 정규화되어야 하며, 클래스를 포함하는 전체 네임스페이스와 클래스 이름을 포함해야 합니다. 예를 들어 `Main` 메서드가 `MyApplication.Core` 네임스페이스의 `Program` 클래스 내에 있는 경우 컴파일러 옵션은 `-main:MyApplication.Core.Program` 이어야 합니다.

## 설명

컴파일에 **Main** 메서드가 있는 형식이 둘 이상 포함된 경우 프로그램에 대한 진입점으로 사용할 **Main** 메서드를 포함하는 형식을 지정할 수 있습니다.

이 옵션은 `.exe` 파일을 컴파일할 때 사용됩니다.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

1. 프로젝트 속성 페이지를 엽니다.
2. 애플리케이션 속성 페이지를 클릭합니다.
3. 시작 개체 속성을 수정합니다.

프로그래밍 방식으로 이 컴파일러 옵션을 설정하려면 `StartupObject`를 참조하세요.

수동으로 `.csproj` 파일을 편집하여 이 컴파일러 옵션을 설정하려면

`.csproj` 파일을 편집하고 `PropertyGroup` 섹션 내에 `StartupObject` 요소를 추가하여 이 옵션을 설정할 수 있습니다. 예를 들어:

```
<PropertyGroup>
  ...
  <StartupObject>MyApplication.Core.Program</StartupObject>
</PropertyGroup>
```

## 예제

**Main** 메서드가 `Test2`에 있다고 지정하여 `t2.cs` 및 `t3.cs`를 컴파일합니다.

```
csc t2.cs t3.cs -main:Test2
```

## 참조

- C# 컴파일러 옵션
- 프로젝트 및 솔루션 속성 관리

# -moduleassemblyname(C# 컴파일러 옵션)

2021-02-18 • 4 minutes to read • [Edit Online](#)

.netmodule에서 public이 아닌 형식에 액세스할 수 있는 어셈블리를 지정합니다.

## 구문

```
-moduleassemblyname:assembly_name
```

## 인수

assembly\_name

.netmodule에서 public이 아닌 형식에 액세스할 수 있는 어셈블리의 이름입니다.

## 설명

**-moduleassemblyname** 은 .netmodule을 빌드할 때와 다음 조건이 충족되는 경우 사용해야 합니다.

- .netmodule은 기존 어셈블리의 public이 아닌 형식에 액세스해야 합니다.
- .netmodule을 빌드할 어셈블리의 이름을 알아야 합니다.
- 기존 어셈블리는 friend 어셈블리(friend assembly)에 .netmodule을 빌드할 어셈블리에 대한 액세스를 부여합니다.

.netmodule 빌드 방법에 대한 자세한 내용은 [-target:module\(C# 컴파일러 옵션\)](#)을 참조하세요.

friend 어셈블리에 대한 자세한 내용은 [Friend 어셈블리](#)를 참조하세요.

개발 환경 내에서는 이 옵션을 사용할 수 없습니다. 명령줄에서 컴파일하는 경우에만 사용할 수 있습니다.

이 컴파일러 옵션은 Visual Studio에서 사용할 수 없으며 프로그래밍 방식으로 변경할 수 없습니다.

## 예제

이 샘플에서는 private 형식을 사용하여 어셈블리를 빌드하므로 friend 어셈블리가 csman\_an\_assembly라는 어셈블리에 액세스할 수 있습니다.

```
// moduleassemblyname_1.cs
// compile with: -target:library
using System;
using System.Runtime.CompilerServices;

[assembly:InternalsVisibleTo ("csman_an_assembly")]

class An_Internal_Class
{
    public void Test()
    {
        Console.WriteLine("An_Internal_Class.Test called");
    }
}
```

## 예제

이 샘플은 어셈블리 moduleassemblyname\_1.dll에 public이 아닌 형식에 액세스하는 .netmodule을 빌드합니다. 이 .netmodule이 csman\_an\_assembly라는 어셈블리에 빌드될 것임을 알기 때문에 **-moduleassemblyname** 을 지정하여 .netmodule이 friend 어셈블리(friend assembly)에 csman\_an\_assembly에 대한 액세스를 부여한 어셈블리의 public이 아닌 형식에 액세스하도록 할 수 있습니다.

```
// moduleassemblyname_2.cs
// compile with: -moduleassemblyname:csman_an_assembly -target:module -reference:moduleassemblyname_1.dll
class B {
    public void Test() {
        An_Internal_Class x = new An_Internal_Class();
        x.Test();
    }
}
```

## 예제

이 코드 샘플에서는 이전에 빌드한 어셈블리와 .netmodule을 참조하여 csman\_an\_assembly 어셈블리를 빌드합니다.

```
// csman_an_assembly.cs
// compile with: -addmodule:moduleassemblyname_2.netmodule -reference:moduleassemblyname_1.dll
class A {
    public static void Main() {
        B bb = new B();
        bb.Test();
    }
}
```

An\_Internal\_Class.Test 호출됨

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -noconfig(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

**-noconfig** 옵션은 csc.exe 파일과 동일한 디렉터리에 있고 여기서 로드되는 csc.rsp 파일로 컴파일하지 않도록 컴파일러에 알립니다.

## 구문

```
-noconfig
```

## 설명

csc.rsp 파일은 .NET Framework와 함께 제공되는 모든 어셈블리를 참조합니다. Visual Studio .NET 개발 환경에 포함된 실제 참조는 프로젝트 형식에 따라 달라집니다.

csc.rsp 파일을 수정하여 **-noconfig** 옵션을 제외하고 csc.exe를 사용하여 명령줄에서 컴파일할 때마다 포함되어야 하는 추가 컴파일러 옵션을 지정할 수 있습니다.

컴파일러는 **csc** 명령에 마지막으로 전달된 옵션을 처리합니다. 따라서 명령줄의 모든 옵션은 csc.rsp 파일에 있는 동일한 옵션의 설정을 재정의합니다.

컴파일러가 csc.rsp 파일의 설정을 찾아서 사용하지 않도록 하려면 **-noconfig** 를 지정합니다.

이 컴파일러 옵션은 Visual Studio에서 사용할 수 없으며 프로그래밍 방식으로 변경할 수 없습니다.

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -nologo(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

**-nologo** 옵션은 컴파일러가 시작될 때 로그온 배너의 표시를 억제하고 컴파일하는 동안 정보 메시지를 표시합니다.

## 구문

```
-nologo
```

## 설명

개발 환경 내에서는 이 옵션을 사용할 수 없습니다. 명령줄에서 컴파일하는 경우에만 사용할 수 있습니다.

이 컴파일러 옵션은 Visual Studio에서 사용할 수 없으며 프로그래밍 방식으로 변경할 수 없습니다.

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -nostdlib(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

**-nostdlib** 를 사용하면 전체 시스템 네임스페이스를 정의하는 mscorelib.dll을 가져올 수 없습니다.

## 구문

```
-nostdlib[+ | -]
```

## 설명

고유한 시스템 네임스페이스와 개체를 정의하거나 만들려면 이 옵션을 사용합니다.

**-nostdlib** 를 지정하지 않으면 mscorelib.dll을 프로그램으로 가져옵니다(-**nostdlib-** 지정과 동일함). **-nostdlib** 를 지정하는 것은 **-nostdlib+** 를 지정하는 것과 같습니다.

**Visual Studio**에서 이 컴파일러 옵션을 설정하려면

### NOTE

다음 지침은 Visual Studio 2015(및 이전 버전)에만 적용됩니다. **mscorelib.dll**을 참조하지 않음 빌드 속성은 Visual Studio 최신 버전에 존재하지 않습니다.

1. 프로젝트의 속성 페이지를 엽니다.
2. 빌드 속성 페이지를 클릭합니다.
3. 고급 단추를 클릭합니다.
4. **mscorelib.dll**을 참조하지 않음 속성을 수정합니다.

프로그래밍 방식으로 이 컴파일러 옵션을 설정하려면

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [NoStdLib](#)를 참조하세요.

## 참고 항목

- [C# 컴파일러 옵션](#)

# -nowarn(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

**-nowarn** 옵션을 사용하면 컴파일러에서 하나 이상의 경고를 표시하지 않을 수 있습니다. 여러 경고 번호를 쉼표로 구분합니다.

## 구문

```
-nowarn:number1[,number2,...]
```

## 인수

`number1`, `number2`

컴파일러에서 표시하지 않으려는 경고 번호입니다.

## 설명

경고 식별자의 숫자 부분만 지정해야 합니다. 예를 들어 CS0028을 표시하지 않으려면 `-nowarn:28` 을 지정할 수 있습니다.

컴파일러는 이전 릴리스에서 유효했지만 현재 컴파일러에서 제거된, `-nowarn`에 전달된 경고 번호를 자동으로 무시합니다. 예를 들어 CS0679는 Visual Studio.NET 2002의 컴파일러에서 유효했지만 이후에 제거되었습니다.

다음 경고는 `-nowarn` 옵션으로 표시되지 않도록 설정할 수 없습니다.

- 컴파일러 경고(수준 1) CS2002
- 컴파일러 경고(수준 1) CS2023
- 컴파일러 경고(수준 1) CS2029

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

1. 프로젝트의 속성 페이지를 엽니다. 자세한 내용은 [프로젝트 디자이너, 빌드 페이지\(C#\)](#)를 참조하세요.
2. 빌드 속성 페이지를 클릭합니다.
3. 경고 표시 안 함 속성을 수정합니다.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [DelaySign](#)을 참조하십시오.

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)
- [C# 컴파일러 오류](#)

# -nowin32manifest(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

**-nowin32manifest** 옵션을 사용하면 실행 파일에 애플리케이션 매니페스트를 포함하지 않도록 컴파일러에 지시할 수 있습니다.

## 구문

```
-nowin32manifest
```

## 설명

이 옵션을 사용하는 경우 Win32 리소스 파일에서 또는 이후 빌드 단계 중에 애플리케이션 매니페스트를 제공하지 않으면 Windows Vista에서 애플리케이션에 가상화가 적용됩니다.

Visual Studio의 애플리케이션 속성 페이지에 있는 **매니페스트** 드롭다운 목록에서 **매니페스트 없이 애플리케이션 만들기** 옵션을 선택하여 이 옵션을 설정합니다. 자세한 내용은 [프로젝트 디자이너, 애플리케이션 페이지\(C#\)](#)를 참조하세요.

매니페스트 생성에 대한 자세한 내용은 [-win32manifest\(C# 컴파일러 옵션\)](#)을 참조하세요.

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -nullable(C# 컴파일러 옵션)

2021-02-18 • 5 minutes to read • [Edit Online](#)

**-nullable** 옵션을 사용하여 null 허용 컨텍스트를 지정할 수 있습니다.

## 구문

```
-nullable[+ | -]
-nullable:{enable | disable | warnings | annotations}
```

## 인수

+ | -

+ 또는 -nullable 을 지정하면 컴파일러에서 null 허용 컨텍스트를 사용하도록 설정합니다. -nullable 을 지정하지 않을 경우 적용되는 - 를 지정하면 null 허용 컨텍스트를 사용하지 않도록 설정합니다.

enable | disable | warnings | annotations

null 허용 컨텍스트 옵션을 지정합니다. + 또는 - 와 비슷하게, 사용하거나 사용하지 않도록 설정하지만 null 허용 컨텍스트 특정성을 더 세분화할 수 있습니다. enable 인수를 사용하면 실제로 -nullable 을 지정하는 것과 동일하며 null 허용 컨텍스트를 사용하도록 설정합니다. disable 을 지정하면 null 허용 컨텍스트를 사용하지 않도록 설정합니다. warnings 인수를 제공하면 (-nullable:warnings) null 허용 경고 컨텍스트가 사용하도록 설정됩니다. annotations 인수를 지정하면 -nullable:annotations 면 null 허용 주석 컨텍스트가 사용하도록 설정됩니다.

## 설명

흐름 분석은 실행 코드 내에서 변수의 Null 허용 여부를 유추하는데 사용됩니다. 변수의 유추된 Null 허용 여부는 변수의 선언된 Null 허용 여부와 관계가 없습니다. 메서드 호출은 조건부로 생략된 경우에도 분석됩니다. 예를 들어 릴리스 모드의 `Debug.Assert`입니다.

다음 특성으로 주석이 지정된 메서드를 호출하면 흐름 분석에도 영향을 줍니다.

- 간단한 사전 조건: `AllowNullAttribute` 및 `DisallowNullAttribute`
- 간단한 사후 조건: `MaybeNullAttribute` 및 `NotNullAttribute`
- 조건부 사후 조건: `MaybeNullWhenAttribute` 및 `NotNullWhenAttribute`
- `DoesNotReturnIfAttribute`(예: `Debug.Assert`에 대한 `doesNotReturnIf(false)`) 및 `DoesNotReturnAttribute`
- `NotNullIfNotNullAttribute`
- 멤버 사후 조건: `MemberNotNullAttribute(String)` 및 `MemberNotNullAttribute(String[])`

## IMPORTANT

전역 null 허용 컨텍스트는 생성된 코드 파일에 적용되지 않습니다. 이 설정과 관계없이 null 허용 컨텍스트는 생성됨으로 표시된 모든 소스 파일에 대해 *disabled*입니다. 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.

프로젝트에서 이 컴파일러 옵션을 설정하는 방법

다음과 같이 `.csproj` 파일을 편집하여 `Project/PropertyGroup` 계층 구조 내에 `<Nullable>` 태그를 추가합니다.

```
<Project Sdk="...">  
  
  <PropertyGroup>  
    <Nullable>enable</Nullable>  
  </PropertyGroup>  
  
</Project>
```

## 참조

- [C# 컴파일러 옵션](#)
- [#nullable 전처리기 지시문](#)
- [nullable 참조 형식](#)

# -optimize(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

**-optimize** 옵션은 컴파일러에서 더 작지만 빠르고 효율적인 출력 파일을 만들기 위해 수행하는 최적화 기능을 사용하거나 사용하지 않도록 설정합니다.

## 구문

```
-optimize[+ | -]
```

## 설명

또한 **-optimize** 는 런타임에 코드를 최적화하도록 공용 언어 런타임에 알립니다.

최적화는 기본적으로 사용되지 않습니다. 최적화를 사용하려면 **-optimize+** 를 지정합니다.

어셈블리에서 사용할 모듈을 빌드하는 경우 어셈블리의 설정과 동일한 **-optimize** 설정을 사용합니다.

**-o** 는 **-optimize** 의 약식 형태입니다.

**-optimize** 및 **-debug** 옵션을 함께 사용할 수 있습니다.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

- 프로젝트 속성 페이지를 엽니다.
- 빌드 속성 페이지를 클릭합니다.
- 코드 최적화 속성을 수정합니다.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [Optimize](#)를 참조하세요.

## 예제

`t2.cs` 를 컴파일하고 컴파일러 최적화를 사용하도록 설정합니다.

```
csc t2.cs -optimize
```

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -out(C# 컴파일러 옵션)

2021-02-18 • 5 minutes to read • [Edit Online](#)

-out 옵션은 출력 파일의 이름을 지정합니다.

## 구문

```
-out:filename
```

## 인수

filename

컴파일러에서 생성된 출력 파일의 이름입니다.

## 설명

명령줄에서 컴파일에 대해 여러 출력 파일을 지정할 수 있습니다. 컴파일러는 -out 옵션 뒤에 하나 이상의 소스 코드 파일이 있을 것으로 예상합니다. 그런 다음 -out 옵션으로 지정된 출력 파일에 모든 소스 코드 파일이 컴파일됩니다.

만들려는 파일의 전체 이름과 확장명을 지정합니다.

출력 파일의 이름을 지정하지 않을 경우

- .exe는 Main 메서드가 포함된 소스 코드 파일에서 해당 이름을 가져옵니다.
- .dll 또는 netmodule은 첫 번째 소스 코드 파일에서 해당 이름을 가져옵니다.

한 출력 파일을 컴파일하는 데 사용되는 소스 코드 파일을 동일한 컴파일에서 다른 출력 파일의 컴파일에 사용할 수 없습니다.

명령줄 컴파일에서 여러 출력 파일을 만들 때는 출력 파일 중 하나만 어셈블리가 될 수 있고, -out 을 사용하여 명시적으로 또는 암시적으로 지정된 첫 번째 출력 파일만 어셈블리가 될 수 있다는 것에 유의하세요.

컴파일의 일부로 생성된 모든 모듈은 컴파일할 때 함께 생성된 어셈블리와 연결된 파일이 됩니다. 연결된 파일을 보려면 [ildasm.exe](#)를 사용하여 어셈블리 매니페스트를 확인합니다.

exe가 friend 어셈블리의 대상이 되려면 -out 컴파일러 옵션이 필요합니다. 자세한 내용은 [Friend 어셈블리를 참조하세요.](#)

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

1. 프로젝트 속성 페이지를 엽니다.
2. 애플리케이션 속성 페이지를 클릭합니다.
3. 어셈블리 이름 속성을 수정합니다.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하려면: [OutputFileName](#)은 프로젝트 형식(exe, 라이브러리 등)과 어셈블리 이름의 조합으로 결정되는 읽기 전용 속성입니다. 출력 파일 이름을 설정하려면 이러한 속성 중 하나 또는 둘 다를 수정해야 합니다.

## 예제

`t.cs`를 컴파일하고 출력 파일 `t.exe`를 만들 뿐만 아니라 `t2.cs`를 작성하고 모듈 출력 파일 `mymodule.netmodule`을 만듭니다.

```
csc t.cs -out:mymodule.netmodule -target:module t2.cs
```

## 참고 항목

- [C# 컴파일러 옵션](#)
- [Friend 어셈블리](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -pathmap(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

-pathmap 컴파일러 옵션은 실제 경로를 컴파일러에서 출력되는 소스 경로 이름에 매핑하는 방법을 지정합니다.

## 구문

```
-pathmap:path1=sourcePath1,path2=sourcePath2
```

## 인수

`path1` 현재 환경에서 소스 파일의 전체 경로입니다.

`sourcePath1` 출력 파일에서 `path1`을 대체하는 소스 경로입니다.

매핑되는 소스 경로를 여럿 지정하려면 각각을 쉼표로 구분합니다.

## 설명

컴파일러가 출력에 소스 경로를 쓰는 이유는 다음과 같습니다.

- 선택적 매개 변수에 [CallerFilePathAttribute](#)를 적용할 때 소스 경로가 인수 대신 사용되는 경우
- 소스 경로가 PDB 파일에 포함된 경우
- PDB 파일의 경로가 PE(이식 가능한 실행 파일) 파일에 포함된 경우

이 옵션은 컴파일러가 실행되는 컴퓨터의 실제 경로 각각을 출력 파일에 써야 하는 해당 경로에 매핑합니다.

## 예제

C:\work\tests 디렉터리에 `t.cs`를 컴파일하고 출력에서 디렉터리를 \publish에 매핑합니다.

```
csc -pathmap:C:\work\tests=\publish t.cs
```

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -pdb(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

-pdb 컴파일러 옵션은 디버그 기호 파일의 이름 및 위치를 지정합니다.

## 구문

```
-pdb:filename
```

## 인수

filename

디버그 기호 파일의 이름 및 위치입니다.

## 설명

-debug(C# 컴파일러 옵션)를 지정하는 경우 컴파일러는 출력 파일(.exe 또는 .dll)을 만들 디렉터리에 출력 파일의 이름과 동일한 파일 이름으로 .pdb 파일을 만듭니다.

-pdb 를 사용하면 .pdb 파일에 대해 기본값이 아닌 파일 이름과 위치를 지정할 수 있습니다.

Visual Studio 개발 환경에서는 이 컴파일러 옵션을 설정할 수 없으며 프로그래밍 방식으로 변경할 수도 없습니다.

## 예제

t.cs 를 컴파일하고 tt.pdb라는 .pdb 파일을 만듭니다.

```
csc -debug -pdb:tt t.cs
```

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -platform(C# 컴파일러 옵션)

2021-02-18 • 6 minutes to read • [Edit Online](#)

어셈블리를 실행할 수 있는 CLR(공용 언어 런타임) 버전을 지정합니다.

## 구문

```
-platform:string
```

## 매개 변수

`string`

`anycpu`(기본값), `anycpu32bitpreferred`, `ARM`, `x64`, `x86` 또는 `Itanium`입니다.

## 설명

- **anycpu**(기본값)는 어셈블리를 모든 플랫폼에서 실행되도록 컴파일합니다. 애플리케이션은 가능할 때마다 64비트로 실행되고 해당 모드를 사용할 수 있을 때만 32비트로 전환됩니다.
- **anycpu32bitpreferred**는 어셈블리를 모든 플랫폼에서 실행되도록 컴파일합니다. 애플리케이션은 64비트와 32비트 애플리케이션을 모두 지원하는 시스템에서 32비트로 실행됩니다. .NET Framework 4.5 이상을 대상으로 하는 프로젝트에 대해서만 이 옵션을 지정할 수 있습니다.
- **ARM**은 ARM(고급 RISC 컴퓨터) 프로세서가 있는 컴퓨터에서 실행할 어셈블리를 컴파일합니다.
- **ARM64**는 A64 명령 집합을 지원하는 ARM(고급 RISC 머신) 프로세서가 있는 컴퓨터에서 64비트 CLR에 의해 실행되도록 어셈블리를 컴파일합니다.
- **x64**는 AMD64 또는 EM64T 명령 집합을 지원하는 컴퓨터에서 64비트 CLR에 의해 실행되도록 어셈블리를 컴파일합니다.
- **x86**은 32비트, x86 호환 CLR에 의해 실행되도록 어셈블리를 컴파일합니다.
- **Itanium**은 Itanium 프로세서 탑재 컴퓨터에서 64비트 CLR에 의해 실행되도록 어셈블리를 컴파일합니다.

64비트 Windows 운영 체제:

- **-platform:x86**으로 컴파일된 어셈블리는 WOW64에서 실행되는 32비트 CLR에서 실행됩니다.
- **-platform:anycpu**로 컴파일된 DLL은 이 DLL이 로드된 프로세스와 동일한 CLR에서 실행됩니다.
- **-platform:anycpu**로 컴파일된 실행 파일은 64비트 CLR에서 실행됩니다.
- **-platform:anycpu32bitpreferred**로 컴파일된 실행 파일은 32비트 CLR에서 실행됩니다.

`anycpu32bitpreferred` 설정은 실행 파일(.EXE)에 대해서만 유효하고 .NET Framework 4.5 이상이 필요합니다.

Windows 64비트 운영 체제에서 실행할 애플리케이션을 개발하는 방법에 대한 자세한 내용은 [64비트 애플리케이션](#)을 참조하세요.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

1. 프로젝트의 속성 페이지를 엽니다.

2. 빌드 속성 페이지를 클릭합니다.
3. .NET Framework 4.5 이상을 대상으로 하는 프로젝트에 대해 플랫폼 대상 속성을 수정하고 32비트 선호 확인란을 선택하거나 선택 취소합니다.

**NOTE**

-platform 은 Visual C# Express의 개발 환경에서 사용할 수 없습니다.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [PlatformTarget](#)를 참조하세요.

## 예제

다음 예제에서는 -platform 옵션을 사용하여 애플리케이션이 64비트 Windows 운영 체제의 64비트 CLR에서만 실행되도록 지정하는 방법을 보여 줍니다.

```
csc -platform:anycpu filename.cs
```

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -preferreduilang(C# 컴파일러 옵션)

2020-11-02 • 2 minutes to read • [Edit Online](#)

**-preferreduilang** 컴파일러 옵션을 사용하여 C# 컴파일러에서 오류 메시지 등의 출력을 표시하는 언어를 지정할 수 있습니다.

## 구문

```
-preferreduilang: language
```

## 인수

**language**

컴파일러 출력에 사용할 언어의 [언어 이름](#)입니다.

## 설명

**-preferreduilang** 컴파일러 옵션을 사용하여 C# 컴파일러에서 오류 메시지와 기타 명령줄 출력에 사용할 언어를 지정할 수 있습니다. 해당 언어의 언어 팩이 설치되지 않은 경우 운영 체제의 언어 설정이 대신 사용되고 오류가 보고되지 않습니다.

```
csc.exe -preferreduilang:ja-JP
```

## 요구 사항

## 참고 항목

- [C# 컴파일러 옵션](#)

# -publicsign(C# 컴파일러 옵션)

2020-11-02 • 3 minutes to read • [Edit Online](#)

이 옵션을 사용하면 컴파일러는 공개 키를 적용하지만 실제로 어셈블리에 서명하지 않습니다. 또한 **-publicsign** 옵션은 파일이 실제로 서명되었음을 런타임에 알리는 비트를 어셈블리에 설정합니다.

## 구문

```
-publicsign
```

## 인수

없음

## 설명

**-publicsign** 옵션에는 [-keyfile](#) 또는 [-keycontainer](#)를 사용해야 합니다. **keyfile** 또는 **keycontainer** 옵션은 공개 키를 지정합니다.

**-publicsign** 및 **-delaysign** 옵션은 함께 사용할 수 없습니다.

"모조 서명" 또는 "OSS 서명"이라고도 하는 퍼블릭 서명은 출력 어셈블리에 퍼블릭 키를 포함하고 "서명된" 플래그를 설정하지만 프라이빗 키를 사용하여 어셈블리에 실제로 서명하지는 않습니다. 이 서명은 릴리스된 "완전히 서명된" 어셈블리와 호환되지만 어셈블리 서명에 사용된 프라이빗 키에는 액세스할 수 없는 어셈블리를 빌드하려는 오픈 소스 프로젝트에 유용합니다. 어셈블리가 완전히 서명되었는지 실제로 확인해야 하는 소비자는 거의 없으므로 완전히 서명된 어셈블리가 사용되는 거의 모든 시나리오에서 이러한 공개적으로 빌드된 어셈블리를 사용할 수 있습니다.

**csproj** 파일에서 이 컴파일러 옵션을 설정하는 방법

프로젝트에 대한 **.csproj** 파일을 열고 다음 요소를 추가합니다.

```
<PublicSign>true</PublicSign>
```

## 참고 항목

- [C# 컴파일러 -delaysign 옵션](#)
- [C# 컴파일러 -keyfile 옵션](#)
- [C# 컴파일러 -keycontainer 옵션](#)
- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -recurse(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

-recurse 옵션을 사용하면 지정된 디렉터리(dir) 또는 프로젝트 디렉터리의 모든 자식 디렉터리에 있는 소스 코드 파일을 컴파일할 수 있습니다.

## 구문

```
-recurse:[dir\]file
```

## 인수

`dir` (선택 사항)

검색을 시작하려는 디렉터리입니다. 지정하지 않으면 프로젝트 디렉터리에서 검색이 시작됩니다.

`file`

검색 할 파일입니다. 와일드카드 문자가 허용됩니다.

## 설명

-recurse 옵션을 사용하면 지정된 디렉터리(`dir`) 또는 프로젝트 디렉터리의 모든 자식 디렉터리에 있는 소스 코드 파일을 컴파일할 수 있습니다.

파일 이름에 와일드카드를 사용하면 -recurse 를 사용하지 않고 프로젝트 디렉터리에서 일치하는 모든 파일을 컴파일할 수 있습니다.

이 컴파일러 옵션은 Visual Studio에서 사용할 수 없으며 프로그래밍 방식으로 변경할 수 없습니다.

## 예제

현재 디렉터리의 모든 C# 파일을 컴파일합니다.

```
csc *.cs
```

dir1\dir2 디렉터리와 자식 디렉터리에 있는 모든 C# 파일을 컴파일하여 dir2.dll을 생성합니다.

```
csc -target:library -out:dir2.dll -recurse:dir1\dir2\*.cs
```

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -reference(C# 컴파일러 옵션)

2021-02-18 • 8 minutes to read • [Edit Online](#)

**-reference** 옵션을 사용하면 컴파일러가 지정된 파일의 **public** 형식 정보를 현재 프로젝트로 가져오므로 지정된 어셈블리 파일의 메타데이터를 참조할 수 있습니다.

## 구문

```
-reference:[alias=]filename  
-reference:filename
```

## 인수

**filename**

어셈블리 매니페스트가 들어 있는 파일의 이름입니다. 둘 이상의 파일을 가져오려면 각 파일에 대해 별도 **-reference** 옵션을 포함합니다.

**alias**

어셈블리에 있는 모든 네임스페이스를 포함하는 루트 네임스페이스를 나타내는 유효한 C# 식별자입니다.

## 설명

둘 이상의 파일에서 가져오려면 각 파일에 대해 **-reference** 옵션을 포함합니다.

가져오는 파일에 매니페스트가 포함되어 있어야 합니다. 출력 파일이 **-target:module** 이외의 **-target** 옵션 중 하나로 컴파일된 상태여야 합니다.

**-r**은 **-reference**의 약식입니다.

**-addmodule**을 사용하여 어셈블리 매니페스트를 포함하지 않는 출력 파일에서 메타데이터를 가져옵니다.

다른 어셈블리(어셈블리 B)를 참조하는 어셈블리(어셈블리 A)를 참조하면 다음과 같은 경우 어셈블리 B를 참조해야 합니다.

- 어셈블리 A에서 사용하는 형식이 형식에서 상속되거나 어셈블리 B의 인터페이스를 구현하는 경우
- 어셈블리 B의 반환 형식이나 매개 변수 형식을 사용하는 필드, 속성, 이벤트 또는 메서드를 호출하는 경우

**-lib**을 사용하여 어셈블리 참조 중 하나 이상이 있는 디렉터리를 지정합니다. **-lib** 항목에서는 컴파일러가 어셈블리를 검색하는 디렉터리에 대해서도 설명합니다.

컴파일러가 모듈이 아니라 어셈블리의 형식을 인식하려면 강제로 형식을 확인하도록 해야 하며, 이 작업을 위해 형식의 인스턴스를 정의할 수 있습니다. 컴파일러를 위해 어셈블리의 형식 이름을 확인하는 다른 방법이 있습니다. 예를 들어 어셈블리의 형식에서 상속하는 경우 컴파일러가 형식 이름을 인식합니다.

때로는 하나의 어셈블리 내에서 동일한 구성 요소의 두 가지 버전을 참조해야 합니다. 이렇게 하려면 각 파일에 대한 **-reference** 스위치의 별칭 하위 옵션을 사용하여 두 파일을 구분합니다. 이 별칭은 구성 요소 이름에 대한 한정자로 사용되며, 파일 중 하나의 구성 요소로 확인됩니다.

자주 사용되는 .NET Framework 어셈블리를 참조하는 csc 지시 파일(.rsp)이 기본적으로 사용됩니다. 컴파일러에서 **csc.rsp**를 사용하지 않도록 하려면 **-noconfig**을 사용합니다.

## NOTE

Visual Studio에서 참조 추가 대화 상자를 사용합니다. 자세한 내용은 [방법: 참조 관리자를 사용하여 참조 추가 또는 제거를 참조하세요.](#) `-reference` 를 사용한 참조 추가와 참조 추가 대화 상자를 사용한 참조 추가의 동작이 같도록 하려면 추가하는 어셈블리에 대한 **Interop** 형식 포함 속성을 **False** 로 설정합니다. 이 속성의 기본값은 **True** 입니다.

## 예제

이 예제에서는 **extern 별칭** 기능을 사용하는 방법을 보여 줍니다.

소스 파일을 컴파일하고, 이전에 컴파일된 `grid.dll` 및 `grid20.dll`에서 메타데이터를 가져옵니다. 두 DLL에는 동일한 구성 요소의 서로 다른 버전이 포함되어 있으며, 두 `-reference` 를 별칭 옵션과 함께 사용하여 소스 파일을 컴파일합니다. 옵션은 다음과 같습니다.

```
-reference:GridV1=grid.dll -reference:GridV2=grid20.dll
```

이렇게 하면 외부 별칭 `GridV1` 및 `GridV2` 가 설정되며, 프로그램에서 `extern` 문을 통해 사용됩니다.

```
extern alias GridV1;
extern alias GridV2;
// Using statements go here.
```

이 작업이 완료되면 다음과 같이 컨트롤 이름 앞에 `GridV1` 을 추가하여 `grid.dll`에서 그리드 컨트롤을 참조할 수 있습니다.

```
GridV1::Grid
```

또한, 다음과 같이 컨트롤 이름 앞에 `GridV2` 를 추가하여 `grid20.dll`에서 그리드 컨트롤을 참조할 수 있습니다.

```
GridV2::Grid
```

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -refout(C# 컴파일러 옵션)

2020-11-02 • 2 minutes to read • [Edit Online](#)

-refout 옵션은 참조 어셈블리가 출력되어야 하는 파일 경로를 지정합니다. 이것은 Emit API에서 `metadataPeStream` 으로 전환됩니다. 이 옵션은 MSBuild의 `ProduceReferenceAssembly` 프로젝트 속성에 해당합니다.

## 구문

```
-refout:filepath
```

## 인수

`filepath` 참조 어셈블리의 파일 경로입니다. 일반적으로 주 어셈블리의 경로와 일치해야 합니다. 권장되는 규칙(MSBuild에서 사용됨)은 주 어셈블리에 상대적으로 "ref/" sub-폴더에 참조 어셈블리를 배치하는 것입니다.

## 설명

참조 어셈블리는 라이브러리의 퍼블릭 API 표면을 나타내는 데 필요한 최소한의 메타데이터만 포함하는 특수한 형식의 어셈블리입니다. 빌드 도구에서 어셈블리를 참조할 때 중요한 모든 멤버에 대한 선언을 포함하지만, 해당 API 계약에 영향을 미치지 않는 프라이빗 멤버의 선언과 모든 멤버 구현은 제외됩니다. 자세한 내용은 .NET 가이드에서 [참조 어셈블리](#)를 참조하세요.

`-refout` 및 `-refonly` 옵션은 함께 사용할 수 없습니다.

## 참조

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -refonly(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

**-refonly** 옵션은 구현 어셈블리 대신 참조 어셈블리가 기본 출력으로 출력되어야 함을 나타냅니다. **-refonly** 매개 변수는 참조 어셈블리가 실행될 수 없을 때 PDB 출력을 자동으로 사용하지 않도록 설정합니다. 이 옵션은 MSBuild의 [ProduceOnlyReferenceAssembly](#) 프로젝트 속성에 해당합니다.

## 구문

```
-refonly
```

## 설명

참조 어셈블리는 라이브러리의 퍼블릭 API 표면을 나타내는데 필요한 최소한의 메타데이터만 포함하는 특수한 형식의 어셈블리입니다. 빌드 도구에서 어셈블리를 참조할 때 중요한 모든 멤버에 대한 선언을 포함하지만, 해당 API 계약에 영향을 미치지 않는 프라이빗 멤버의 선언과 모든 멤버 구현은 제외됩니다. 자세한 내용은 .NET 가이드에서 [참조 어셈블리](#)를 참조하세요.

**-refonly** 및 **-refout** 옵션은 함께 사용할 수 없습니다.

## 참조

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -resource(C# 컴파일러 옵션)

2021-02-18 • 4 minutes to read • [Edit Online](#)

출력 파일에 지정된 리소스를 포함합니다.

## 구문

```
-resource:filename[,identifier[,accessibility-modifier]]
```

## 인수

`filename`

출력 파일에 포함하려는 .NET 리소스 파일입니다.

`identifier` (선택 사항)

리소스의 논리적 이름으로, 리소스를 로드하는 데 사용되는 이름입니다. 기본값은 파일 이름입니다.

`accessibility-modifier` (선택 사항)

리소스의 접근성으로, `public` 또는 `private`입니다. 기본값은 `public`입니다.

## 설명

리소스를 어셈블리에 연결하고 출력 파일에 리소스 파일을 추가하지 않으려면 [-linkresource](#)를 사용합니다.

기본적으로 리소스는 C# 컴파일러를 사용하여 생성될 때 어셈블리에서 `public`입니다. 리소스를 `private`로 만들려면 접근성 한정자로 `private`를 지정합니다. `public` 또는 `private` 이외의 다른 접근성은 허용되지 않습니다.

예를 들어 `filename`이 [Resgen.exe](#) 또는 개발 환경에서 만들어진 .NET 리소스 파일인 경우에는 `System.Resources` 네임스페이스의 멤버를 사용하여 해당 파일에 액세스할 수 있습니다. 자세한 내용은 `System.Resources.ResourceManager`를 참조하세요. 다른 모든 리소스의 경우에는 런타임에 `GetManifestResource` 클래스의 `Assembly` 메서드를 사용하여 리소스에 액세스합니다.

`-res` 는 `-resource`의 약식입니다.

출력 파일에 있는 리소스의 순서는 명령줄에 지정된 순서에 따라 결정됩니다.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

1. 프로젝트에 리소스 파일을 추가합니다.
2. 솔루션 탐색기에서 포함할 파일을 선택합니다.
3. 속성 창에서 파일에 대한 빌드 작업을 선택합니다.
4. 빌드 작업을 포함 리소스로 설정합니다.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [BuildAction](#)을 참조하십시오.

## 예제

`in.cs`를 컴파일하고 리소스 파일 `rf.resource`를 첨부합니다.

```
csc -resource:rf.resource in.cs
```

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -subsystemversion(C# 컴파일러 옵션)

2020-11-02 • 4 minutes to read • [Edit Online](#)

생성된 실행 파일을 실행할 수 있는 하위 시스템의 최소 버전을 지정하여 실행 파일을 실행할 수 있는 Windows 버전을 결정합니다. 가장 일반적으로, 이 옵션은 실행 파일이 이전 버전의 Windows에서 사용할 수 없는 특정 보안 기능을 활용할 수 있도록 합니다.

## NOTE

하위 시스템 자체를 지정하려면 `-target` 컴파일러 옵션을 사용합니다.

## 구문

```
-subsystemversion:major.minor
```

## 매개 변수

`major.minor`

주 버전과 부 버전의 점 표기법으로 표현된 필수 최소 버전의 하위 시스템입니다. 예를 들어 이 항목의 뒷부분에 나오는 표의 설명에 따라 이 옵션의 값을 6.01로 설정하는 경우 Windows 7 이전 운영 체제에서는 애플리케이션을 실행할 수 없도록 지정할 수 있습니다. `major` 및 `minor`의 값을 정수로 지정해야 합니다.

`minor` 버전에서 선행 0은 버전을 변경하지 않지만 후행 0은 버전을 변경합니다. 예를 들어 6.1과 6.01은 동일한 버전을 가리키지만 6.10은 다른 버전을 가리킵니다. 혼동을 피하기 위해 부 버전을 두 자리로 표현하는 것이 좋습니다.

## 설명

다음 표에는 Windows의 일반적인 하위 시스템 버전이 나와 있습니다.

WINDOWS 버전	하위 시스템 버전
Windows 2000	5.00
Windows XP	5.01
Windows Server 2003	5.02
Windows Vista	6.00
Windows 7	6.01
Windows Server 2008	6.01
Windows 8	6.02

## 기본값

-subsystemversion 컴파일러 옵션의 기본값은 다음 목록의 조건에 따라 달라집니다.

- 다음 목록의 컴파일러 옵션이 설정된 경우 기본값은 6.02입니다.
  - [/target:appcontainerexe](#)
  - [/target:winmdobj](#)
  - [-platform:arm](#)
- MSBuild를 사용하고 .NET Framework 4.5를 대상으로 하며, 이 목록의 앞에서 지정된 컴파일러 옵션 중 하나를 설정하지 않은 경우 기본값은 6.00입니다.
- 앞의 조건이 하나도 true가 아닌 경우 기본값은 4.00입니다.

## 이| 옵션 설정

Visual Studio에서 -subsystemversion 컴파일러 옵션을 설정하려면 .csproj 파일을 열고 MSBuild XML에서 `SubsystemVersion` 속성의 값을 지정해야 합니다. Visual Studio IDE에서는 이 옵션을 설정할 수 없습니다. 자세한 내용은 이 항목의 앞부분에 나오는 "기본값"이나 [일반적인 MSBuild 프로젝트 속성](#)을 참조하세요.

## 참조

- [C# 컴파일러 옵션](#)

# -target(C# 컴파일러 옵션)

2020-11-02 • 4 minutes to read • [Edit Online](#)

**-target** 컴파일러 옵션은 다음 형태 중 하나로 지정할 수 있습니다.

[/target:appcontainerexe](#)

Windows 8.x 스토어 앱에 사용할 .exe 파일을 만듭니다.

[/target:exe](#)

.exe 파일을 만듭니다.

[/target:library](#)

코드 라이브러리를 만듭니다.

[/target:module](#)

모듈을 만듭니다.

[/target:winexe](#)

Windows 프로그램을 만듭니다.

[/target:winmdobj](#)

중간 .winmdobj 파일을 만듭니다.

**-target:module**을 지정하지 않으면 **-target**은 .NET Framework 어셈블리 매니페스트가 출력 파일에 배치되도록 합니다. 자세한 내용은 [.NET의 어셈블리](#) 및 [공통 특성](#)을 참조하세요.

어셈블리 매니페스트는 컴파일의 첫 번째 .exe 출력 파일 또는 .exe 출력 파일이 없는 경우 첫 번째 DLL에 배치됩니다. 예를 들어 다음 명령줄에서 매니페스트는 **1.exe**에 배치됩니다.

```
csc -out:1.exe t1.cs -out:2.netmodule t2.cs
```

컴파일러는 컴파일당 하나의 어셈블리 매니페스트만 만듭니다. 컴파일의 모든 파일에 대한 정보가 어셈블리 매니페스트에 배치됩니다. **-target:module**을 사용하여 생성된 파일을 제외한 모든 출력 파일에 어셈블리 매니페스트가 포함될 수 있습니다. 명령줄에서 여러 출력 파일을 생성하는 경우 하나의 어셈블리 매니페스트만 만들 수 있으며, 명령줄에 지정된 첫 번째 출력 파일로 이동해야 합니다. 첫 번째 출력 파일이 **-target:exe**, **-target:winexe**, **-target:library** 또는 **-target:module** 중 무엇이든 관계없이 동일한 컴파일에서 생성된 다른 출력 파일은 모듈 (**-target:module**)이어야 합니다.

어셈블리를 만드는 경우 [CLSCompliantAttribute](#) 특성을 사용하여 코드의 전체 또는 일부를 CLS 규격으로 지정 할 수 있습니다.

```
// target_clscompliant.cs
[assembly:System.CLSCompliant(true)] // specify assembly compliance

[System.CLSCompliant(false)] // specify compliance for an element
public class TestClass
{
    public static void Main() {}
}
```

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [OutputType](#)을 참조하세요.

- C# 컴파일러 옵션
- 프로젝트 및 솔루션 속성 관리
- -subsystemversion(C# 컴파일러 옵션)

# -target:appcontainerexe(C# 컴파일러 옵션)

2021-02-18 • 3 minutes to read • [Edit Online](#)

**-target:appcontainerexe** 컴파일러 옵션을 사용하면 컴파일러는 앱 컨테이너에서 실행해야 하는 Windows 실행 파일(.exe)을 만듭니다. 이 옵션은 [-target:winexe](#)와 같지만, Windows 8.x 스토어 앱을 위해 설계되었습니다.

## 구문

```
-target:appcontainerexe
```

## 설명

이 옵션은 앱이 앱 컨테이너에서 실행되도록 하기 위해 PE([이식 가능 파일](#))에 비트를 설정합니다. 해당 비트가 설정된 경우 CreateProcess 메서드가 응용 프로그램 밖에서 실행 파일을 실행하려고 시도하면 오류가 발생합니다.

**-out** 옵션을 사용하지 않으면 **Main** 메서드가 포함된 입력 파일의 이름이 출력 파일의 이름으로 사용됩니다.

이 옵션을 명령 프롬프트에서 지정하면 실행 파일을 만드는데 다음 **-out** 또는 **-target** 옵션까지의 모든 파일이 사용됩니다.

**IDE**에서 이 컴파일러 옵션을 설정하려면

1. 솔루션 탐색기에서 프로젝트의 바로 가기 메뉴를 열고 속성을 선택합니다.
2. 애플리케이션 탭의 출력 형식 목록에서 **Windows 스토어 앱**을 선택합니다.

이 옵션은 Windows 8.x 스토어 앱 템플릿에만 사용할 수 있습니다.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [OutputType](#)을 참조하십시오.

## 예제

다음 명령은 `filename.cs`를 응용 프로그램 컨테이너에서만 실행할 수 있는 Windows 실행 파일로 컴파일합니다.

```
csc -target:appcontainerexe filename.cs
```

## 참고 항목

- [-target\(C# 컴파일러 옵션\)](#)
- [-target:winexe\(C# 컴파일러 옵션\)](#)
- [C# 컴파일러 옵션](#)

# -target:exe(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

-target:exe 옵션을 사용하면 컴파일러가 콘솔 애플리케이션 실행 파일(EXE)을 만듭니다.

## 구문

```
-target:exe
```

## 설명

기본적으로 -target:exe 옵션이 적용됩니다. 실행 파일은 .exe 확장명으로 생성됩니다.

-target:winexe를 사용하여 Windows 프로그램 실행 파일을 만드세요.

-out 옵션을 사용하여 지정하지 않으면 Main 메서드가 포함된 입력 파일의 이름이 출력 파일의 이름으로 사용됩니다.

명령줄에서 지정된 경우, 다음 -out 또는 -target:module 옵션까지의 모든 파일이 .exe 파일을 만드는데 사용됩니다.

.exe 파일로 컴파일되는 소스 코드 파일에 Main 메서드가 하나만 있어야 합니다. -main 컴파일러 옵션을 사용하면 코드에 Main 메서드를 포함하는 클래스가 둘 이상 있는 경우 Main 메서드를 포함하는 클래스를 지정할 수 있습니다.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

- 프로젝트 속성 페이지를 엽니다.
- 애플리케이션 속성 페이지를 클릭합니다.
- 출력 형식 속성을 수정합니다.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [OutputType](#)를 참조하세요.

## 예제

다음 명령줄은 각각 `in.cs`를 컴파일하고 `in.exe`를 만듭니다.

```
csc -target:exe in.cs
csc in.cs
```

## 참고 항목

- [-target\(C# 컴파일러 옵션\)](#)
- [C# 컴파일러 옵션](#)

# -target:library(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

-target:library 옵션을 사용하면 컴파일러가 실행 파일(EXE) 대신 DLL(동적 연결 라이브러리)을 만듭니다.

## 구문

```
-target:library
```

## 설명

DLL은 .dll 확장명으로 생성됩니다.

-out 옵션을 사용하여 달리 지정되지 않은 경우 첫 번째 입력 파일의 이름이 출력 파일의 이름으로 사용됩니다.

명령줄에서 지정된 경우, 다음 -out 또는 -target: module 옵션까지의 모든 파일이 .dll 파일을 만드는 데 사용됩니다.

.dll 파일을 빌드하는 경우에는 Main 메서드가 필요하지 않습니다.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

1. 프로젝트 속성 페이지를 엽니다.
2. 애플리케이션 속성 페이지를 클릭합니다.
3. 출력 형식 속성을 수정합니다.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [OutputType](#)를 참조하세요.

## 예제

in.cs 를 컴파일하고 in.dll 을 만듭니다.

```
csc -target:library in.cs
```

## 참고 항목

- [-target\(C# 컴파일러 옵션\)](#)
- [C# 컴파일러 옵션](#)

# -target:module(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

이 옵션은 컴파일러에서 어셈블리 매니페스트를 생성하지 않도록 합니다.

## 구문

```
-target:module
```

## 설명

기본적으로 이 옵션으로 컴파일하여 생성되는 출력 파일의 확장명은 .netmodule입니다.

어셈블리 매니페스트가 없는 파일은 .NET 런타임에서 로드할 수 없습니다. 그러나 이러한 파일은 [-addmodule](#)을 통해 어셈블리의 어셈블리 매니페스트에 통합할 수 있습니다.

둘 이상의 모듈이 단일 컴파일에서 생성될 경우 한 모듈의 [내부](#) 형식을 컴파일에 포함된 다른 모듈에서 사용할 수 있습니다. 한 모듈의 코드가 다른 모듈의 [internal](#) 형식을 참조하는 경우 [-addmodule](#)을 통해 두 모듈을 모두 어셈블리 매니페스트에 통합해야 합니다.

Visual Studio 개발 환경에서는 모듈을 만들 수 없습니다.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [OutputType](#)를 참조하세요.

## 예제

`in.cs`를 컴파일하고 `in.netmodule`을 만듭니다.

```
csc -target:module in.cs
```

## 참고 항목

- [-target\(C# 컴파일러 옵션\)](#)
- [C# 컴파일러 옵션](#)

# -target:winexe(C# 컴파일러 옵션)

2021-02-18 • 3 minutes to read • [Edit Online](#)

-target:winexe 옵션을 사용하면 컴파일러가 Windows 프로그램 실행 파일(EXE)을 만듭니다.

## 구문

```
-target:winexe
```

## 설명

실행 파일은 .exe 확장명으로 생성됩니다. Windows 프로그램은 .NET 라이브러리나 Windows API를 통해 사용자 인터페이스를 제공하는 프로그램입니다.

-target:exe를 사용하여 콘솔 애플리케이션을 만듭니다.

-out 옵션을 사용하여 지정하지 않으면 Main 메서드가 포함된 입력 파일의 이름이 출력 파일의 이름으로 사용됩니다.

명령줄에서 지정된 경우, 다음 -out 또는 -target 옵션까지의 모든 파일이 Windows 프로그램을 만드는 데 사용됩니다.

.exe 파일로 컴파일되는 소스 코드 파일에 Main 메서드가 하나만 있어야 합니다. -main 옵션을 사용하면 코드에 Main 메서드를 포함하는 클래스가 둘 이상 있는 경우 Main 메서드를 포함하는 클래스를 지정할 수 있습니다.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

- 프로젝트 속성 페이지를 엽니다.
- 애플리케이션 속성 페이지를 클릭합니다.
- 출력 형식 속성을 수정합니다.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [OutputType](#)를 참조하세요.

## 예제

in.cs 를 Windows 프로그램으로 컴파일합니다.

```
csc -target:winexe in.cs
```

## 참고 항목

- target(C# 컴파일러 옵션)
- C# 컴파일러 옵션

# -target:winmdobj(C# 컴파일러 옵션)

2021-02-18 • 4 minutes to read • [Edit Online](#)

**-target:winmdobj** 컴파일러 옵션을 사용하는 경우 컴파일러는 사용자가 Windows 런타임 이진(.winmd) 파일로 변환할 수 있는 중간 .winmdobj 파일을 만듭니다. 그런 다음 관리되는 언어 프로그램뿐만 아니라 JavaScript 및 C++ 프로그램에서도 .winmd 파일을 사용할 수 있습니다.

## 구문

```
-target:winmdobj
```

## 설명

**winmdobj** 설정이 컴파일러에 중간 모듈이 필요하다는 신호를 보냅니다. 이에 대한 응답으로 Visual Studio에서 C# 클래스 라이브러리를 .winmdobj 파일로 컴파일합니다. 그런 다음 [WinMDExp](#) 내보내기 도구를 통해 .winmdobj 파일을 공급하여 Windows 메타데이터(.winmd) 파일을 만들 수 있습니다. .winmd 파일에는 원본 라이브러리의 코드와 JavaScript 또는 C++ 및 Windows 런타임에서 사용하는 WinMD 메타데이터가 모두 들어 있습니다.

**-target:winmdobj** 컴파일러 옵션을 사용하여 컴파일된 파일의 출력은 WinMDExp 내보내기 도구의 입력으로만 사용하도록 설계되었습니다. .winmdobj 파일 자체는 직접 참조되지 않습니다.

**-out** 옵션을 사용하지 않으면 첫 번째 입력 파일의 이름이 출력 파일의 이름으로 사용됩니다. [Main](#) 메서드는 필요하지 않습니다.

명령 프롬프트에서 **-target:winmdobj** 옵션을 지정하면 Windows 프로그램을 만드는데 다음 **-out** 또는 **-target:module** 옵션까지의 모든 파일이 사용됩니다.

Windows 스토어 응용 프로그램용 **Visual Studio IDE**에서 이 컴파일러 옵션을 설정하려면

- 솔루션 탐색기에서 프로젝트의 바로 가기 메뉴를 열고 속성을 선택합니다.
- 애플리케이션 탭을 선택합니다.
- 출력 형식 목록에서 **WinMD 파일**을 선택합니다.

**WinMD 파일** 옵션은 Windows 8.x 스토어 앱 템플릿에만 사용할 수 있습니다.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [OutputType](#)을 참조하십시오.

## 예제

다음 명령은 `filename.cs`를 중간 .winmdobj 파일로 컴파일합니다.

```
csc -target:winmdobj filename.cs
```

## 참고 항목

- [-target\(C# 컴파일러 옵션\)](#)
- [C# 컴파일러 옵션](#)

# -unsafe(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

-unsafe 컴파일러 옵션을 사용하면 [unsafe](#) 키워드를 사용하는 코드를 컴파일할 수 있습니다.

## 구문

```
-unsafe
```

## 설명

안전하지 않은 코드에 대한 자세한 내용은 [안전하지 않은 코드 및 포인터](#)를 참조하세요.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

1. 프로젝트 속성 페이지를 엽니다.
2. 빌드 속성 페이지를 클릭합니다.
3. 안전하지 않은 코드 허용 확인란을 선택합니다.

이 옵션을 **csproj** 파일에 추가하려면

프로젝트에 대한 .csproj 파일을 열고 다음 요소를 추가합니다.

```
<PropertyGroup>
  <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [AllowUnsafeBlocks](#)을 참조하십시오.

## 예제

안전하지 않은 모드에 대해 `in.cs` 컴파일:

```
csc -unsafe in.cs
```

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -utf8output(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

**-utf8output** 옵션은 UTF-8 인코딩을 사용하여 컴파일러 출력을 표시합니다.

## 구문

```
-utf8output
```

## 설명

일부 국제 구성에서는 컴파일러 출력이 콘솔에 올바르게 표시되지 않을 수 있습니다. 이러한 구성에서는 **-utf8output** 을 사용하고 컴파일러 출력을 파일로 리디렉션합니다.

이 컴파일러 옵션은 Visual Studio에서 사용할 수 없으며 프로그래밍 방식으로 변경할 수 없습니다.

## 참고 항목

- [C# 컴파일러 옵션](#)

# -warn(C# 컴파일러 옵션)

2021-02-18 • 4 minutes to read • [Edit Online](#)

**-warn** 옵션은 컴파일러에서 표시할 경고 수준을 지정합니다.

## 구문

```
-warn:option
```

## 인수

option

컴파일에 대해 표시할 경고 수준입니다. 숫자가 낮으면 높은 심각도 경고만 표시되고, 숫자가 높으면 더 많은 경고가 표시됩니다. 값은 0 또는 양의 정수여야 합니다.

경고 수준	의미
0	모든 경고 메시지 내보내기를 끕니다.
1	심각한 경고 메시지를 표시합니다.
2	수준 1 경고와 덜 심각한 특정 경고(예: 클래스 멤버 숨기기에 대한 경고)를 표시합니다.
3	수준 2 경고와 덜 심각한 특정 경고(예: 항상 <code>true</code> 또는 <code>false</code> 로 평가되는 식에 대한 경고)를 표시합니다.
4(기본값)	모든 수준 3 경고와 정보 경고를 표시합니다.
5	수준 4 경고와 함께 C# 9.0에 제공된 컴파일러의 <a href="#">추가 경고</a> 를 표시합니다.
5보다 큼	5보다 큰 값은 5로 처리됩니다. 일반적으로 큰 임의의 값(예: <code>9999</code> )을 배치하여 컴파일러가 새 경고 수준으로 업데이트되는 경우 항상 모든 경고가 표시되는지 확인합니다.

## 설명

오류 또는 경고에 대한 정보를 가져오려면 도움말 색인에서 오류 코드를 조회할 수 있습니다. 오류 또는 경고에 대한 정보를 가져오는 다른 방법은 [C# 컴파일러 오류](#)를 참조하세요.

모든 경고를 오류로 처리하려면 `-warnaserror`를 사용합니다. 특정 경고를 사용하지 않으려면 `-nowarn`을 사용합니다.

`-w` 는 `-warn` 의 약식 형태입니다.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

- 프로젝트 속성 페이지를 엽니다.
- 빌드 속성 페이지를 클릭합니다.

### 3. 경고 수준 속성을 수정합니다.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [WarningLevel](#)를 참조하세요.

## 예제

`in.cs`를 컴파일하고 컴파일러에서 수준 1 경고만 표시하도록 합니다.

```
csc -warn:1 in.cs
```

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -warnaserror(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

-warnaserror+ 옵션은 모든 경고를 오류로 처리합니다.

## 구문

```
-warnaserror[+ | -][:warning-list]
```

## 설명

일반적으로 경고로 보고되는 메시지가 대신 오류로 보고되며, 빌드 프로세스가 중지됩니다(출력 파일이 작성되지 않음).

기본적으로 -warnaserror- 가 적용되며, 경고가 발생해도 출력 파일이 생성됩니다. -warnaserror+ 와 동일한 -warnaserror 는 경고가 오류로 처리되도록 합니다.

필요에 따라 몇 개의 특정 경고만 오류로 처리하려는 경우 오류로 처리할 경고 번호의 쉼표로 구분된 목록을 지정할 수 있습니다. 모든 Null 허용 여부 경고 집합은 **nullable** 축약형을 사용하여 지정할 수 있습니다.

컴파일러에서 표시할 경고 수준을 지정하려면 [-warn](#)을 사용합니다. 특정 경고를 사용하지 않으려면 [-nowarn](#)을 사용합니다.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

- 프로젝트 속성 페이지를 엽니다.
- 빌드 속성 페이지를 클릭합니다.
- 경고를 오류로 처리 속성을 수정합니다.

프로그래밍 방식으로 이 컴파일러 옵션을 설정하려면 [TreatWarningsAsErrors](#)를 참조하세요.

## 예제

in.cs 를 컴파일하고 컴파일러에서 경고를 표시하지 않도록 합니다.

```
csc -warnaserror in.cs
csc -warnaserror:642,649,652,nullable in.cs
```

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -win32icon(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

**-win32icon** 옵션은 .ico 파일을 출력 파일에 삽입하여 파일 탐색기에서 출력 파일을 원하는 모양으로 표시합니다.

## 구문

```
-win32icon:filename
```

## 인수

`filename`

출력 파일에 추가하려는 .ico 파일입니다.

## 설명

.ico 파일은 [리소스 컴파일러](#)로 만들 수 있습니다. 리소스 컴파일러는 Visual C++ 프로그램을 컴파일할 때 실행되며 .rc 파일에서 .iso 파일이 만들어집니다.

.NET Framework 리소스 파일을(참조하려면) [-linkresource](#)를(첨부하려면) [-resource](#)를 참조하세요. .res 파일을 가져오려면 [-win32res](#)를 참조하세요.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

- 프로젝트 속성 페이지를 엽니다.
- 애플리케이션 속성 페이지를 클릭합니다.
- 애플리케이션 아이콘 속성을 수정합니다.

이 컴파일러 옵션을 프로그래밍 방식으로 설정하는 방법에 대한 자세한 내용은 [ApplicationIcon](#)을 참조하세요.

## 예제

`in.cs`를 컴파일하고 .ico 파일 `rf.ico`를 첨부하여 `in.exe`를 생성합니다.

```
csc -win32icon:rf.ico in.cs
```

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -win32manifest(C# 컴파일러 옵션)

2021-02-18 • 7 minutes to read • [Edit Online](#)

프로젝트의 PE(포팅 가능한 실행 파일) 파일에 포함할 사용자 정의 Win32 애플리케이션 매니페스트 파일을 식별하려면 **-win32manifest** 옵션을 사용합니다.

## 구문

```
-win32manifest: filename
```

## 인수

`filename`

사용자 지정 매니페스트 파일의 이름 및 위치입니다.

## 설명

기본적으로 Visual C# 컴파일러는 "asInvoker"의 요청된 실행 수준을 지정하는 애플리케이션 매니페스트를 포함합니다. 실행 파일이 작성되는 폴더에 매니페스트가 생성됩니다. Visual Studio를 사용하는 경우 대개 bin\Debug 또는 bin\Release 폴더입니다. 예를 들어 "highestAvailable" 또는 "requireAdministrator"의 요청된 실행 수준을 지정하기 위해 사용자 지정 매니페스트를 제공하려면 이 옵션을 사용하여 파일 이름을 지정합니다.

### NOTE

이 옵션과 [-win32res\(C# 컴파일러 옵션\)](#) 옵션은 함께 사용할 수 없습니다. 같은 명령줄에서 두 옵션을 모두 사용하려고 하면 빌드 오류가 발생합니다.

요청된 실행 수준을 지정하는 애플리케이션 매니페스트가 없는 애플리케이션은 Windows의 사용자 계정 컨트롤 기능에서 파일/레지스트리 가상화의 적용을 받습니다. 자세한 내용은 [사용자 계정 컨트롤](#)을 참조하십시오.

다음 조건 중 하나라도 참인 경우 애플리케이션에 가상화가 적용됩니다.

- 사용자는 **-nowin32manifest** 옵션을 사용하며, **-win32res** 옵션을 사용하여 나중 빌드 단계 또는 Windows Resource(.res) 파일의 일부로 매니페스트를 제공하지 않습니다.
- 요청한 실행 수준을 지정하지 않는 사용자 지정 매니페스트를 제공합니다.

Visual Studio는 기본 .manifest 파일을 만들고 이를 실행 파일과 함께 debug 및 release 디렉토리에 저장합니다. 텍스트 편집기에서 파일을 만들고 프로젝트에 파일을 추가하여 사용자 지정 매니페스트를 추가할 수 있습니다. 또는 솔루션 탐색기에서 프로젝트 아이콘을 마우스 오른쪽 단추로 클릭하고, 새 항목 추가 와 애플리케이션 매니페스트 파일을 차례로 클릭합니다. 신규 또는 기존 매니페스트 파일을 추가하면 매니페스트 드롭다운 목록에 나타납니다. 자세한 내용은 [프로젝트 디자이너, 애플리케이션 페이지\(C#\)](#)를 참조하세요.

[/nowin32manifest\(C# 컴파일러 옵션\)](#) 옵션을 사용하여 애플리케이션 매니페스트를 사용자 지정 빌드 후 단계 또는 Win32 리소스 파일의 일부로 제공할 수 있습니다. 애플리케이션이 Windows Vista에서 파일 또는 레지스트리 가상화의 적용을 받도록 하려면 동일한 옵션을 사용합니다. 이렇게 하면 컴파일러가 PE(이식 가능한 실행 파일) 파일에 기본 매니페스트를 만들고 포함할 수 없습니다.

## 예제

다음 예제는 Visual C# 컴파일러가 PE에 삽입하는 기본 매니페스트를 보여 줍니다.

#### NOTE

컴파일러는 표준 애플리케이션 이름인 "MyApplication.app"을 xml에 삽입합니다. 이는 Windows Server 2003 서비스 팩 3에서 애플리케이션을 실행하기 위한 해결 방법입니다.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity version="1.0.0.0" name="MyApplication.app"/>
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-microsoft-com:asm.v3">
        <requestedExecutionLevel level="asInvoker"/>
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

## 참조

- [C# 컴파일러 옵션](#)
- [-nowin32manifest\(C# 컴파일러 옵션\)](#)
- [프로젝트 및 솔루션 속성 관리](#)

# -win32res(C# 컴파일러 옵션)

2021-02-18 • 2 minutes to read • [Edit Online](#)

-win32res 옵션은 출력 파일에 Win32 리소스를 삽입합니다.

## 구문

```
-win32res:filename
```

## 인수

filename

출력 파일에 추가하려는 리소스 파일입니다.

## 설명

Win32 리소스 파일은 [리소스 컴파일러](#)로 만들 수 있습니다. 리소스 컴파일러는 Visual C++ 프로그램을 컴파일 할 때 실행되며 .rc 파일에서 .res 파일이 만들어집니다.

Win32 리소스는 파일 탐색기에서 애플리케이션을 식별하는 데 도움이 되는 버전 정보나 비트맵 (아이콘) 정보를 포함할 수 있습니다. -win32res 를 지정하지 않으면 컴파일러에서 어셈블리 버전을 기반으로 하여 버전 정보를 생성합니다.

.NET Framework 리소스 파일을(참조하려면) [-linkresource](#)를(첨부하려면) [-resource](#)를 참조하세요.

**Visual Studio** 개발 환경에서 이 컴파일러 옵션을 설정하려면

1. [프로젝트 속성](#) 페이지를 엽니다.
2. [애플리케이션 속성](#) 페이지를 클릭합니다.
3. [리소스 파일](#) 단추를 클릭한 다음 콤보 상자를 사용하여 파일을 선택합니다.

## 예제

in.cs 를 컴파일하고 Win32 리소스 파일 rf.res 를 첨부하여 in.exe 를 생성합니다.

```
csc -win32res:rf.res in.cs
```

## 참고 항목

- [C# 컴파일러 옵션](#)
- [프로젝트 및 솔루션 속성 관리](#)

# C# 컴파일러 오류

2020-11-02 • 3 minutes to read • [Edit Online](#)

일부 C# 컴파일러 오류는 오류가 생성된 이유 및 경우에 따라 오류 수정 방법을 설명하는 항목이 있습니다. 특정 오류 메시지에 대해 도움말을 사용할 수 있는지 여부를 확인하려면 다음 단계 중 하나를 사용합니다.

- 오류 번호(예: CS0029)를 [출력 창](#)에서 찾고 Microsoft Docs에서 검색합니다.
- 오류 번호(예: CS0029)를 [출력 창](#)에서 선택하고 F1 키를 선택합니다.
- 인덱스에서 찾을 대상 상자에 오류 번호를 입력합니다.

위의 단계를 수행해도 오류에 대한 정보를 얻을 수 없으면 이 페이지 끝으로 이동하여 오류 번호나 텍스트를 포함한 피드백을 보내십시오.

C#에서 오류 및 경고 옵션을 구성하는 방법에 대한 자세한 내용은 [프로젝트 디자이너, 빌드 페이지\(C#\)](#)를 참조하세요.

## NOTE

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.

## 참조

- [C# 컴파일러 옵션](#)
- [죄송 합니다. 이 C# 오류에 대한 구체적인 정보가 없습니다.](#)
- [프로젝트 디자이너, 빌드 페이지\(C#\)](#)
- [-warn\(C# 컴파일러 옵션\)](#)
- [-nowarn\(C# 컴파일러 옵션\)](#)

# 소개

2020-11-02 • 143 minutes to read • [Edit Online](#)

C#("씨샵"이라고 발음합니다)은 간단하면서도 형식이 안전한 최신 개체 지향 프로그래밍 언어입니다. C#은 C 언어의 루트를 포함하고 있으며 C, C++ 및 Java 프로그래머에게 즉시 친숙합니다. C#은 ecma 국제에서 **ecma-334** 표준으로 표준화되며 iso/iec는 **iso/iec 23270** 표준으로 표준화됩니다. .NET Framework에 C# 대한 Microsoft의 컴파일러는 이러한 두 표준의 준수 구현입니다.

C#은 개체 지향 언어이지만 구성 요소 지향 프로그래밍도 지원합니다. 현대의 소프트웨어 설계는 독립적이고 자체 설명적인 기능 패키지 형식을 갖는 소프트웨어 구성 요소에 점점 더 많이 의존하고 있습니다. 이러한 구성 요소의 핵심은 속성, 메서드 및 이벤트를 포함하는 프로그래밍 모델을 제공한다는 데 있습니다. 이러한 구성 요소는 구성 요소에 대한 선언적 정보를 제공하는 특성을 보유하며 자체 설명서를 통합하고 있습니다. C#은 이러한 개념을 C# 직접 지원하여 소프트웨어 구성 요소를 만들고 사용하는 매우 자연스러운 언어를 제공하는 언어 구문을 제공합니다.

몇 가지 C# 기능은 강력하고 안정적인 애플리케이션을 구축하는데 도움을 줍니다. **가비지 수집**은 사용하지 않는 개체가 차지하는 메모리를 자동으로 회수합니다. **예외 처리**는 오류 검색 및 복구에 대한 구조적이고 확장 가능한 방법을 제공합니다. 또한 **형식이 안전한** 언어 디자인에서는 초기화되지 않은 변수를 읽을 수 없거나, 범위를 벗어나 배열을 인덱싱하고, 선택되지 않은 형식 캐스팅을 수행할 수 없습니다.

C#에는 **통합 형식 시스템**이 있습니다. `int` 및 `double`과 같은 기본 형식을 포함하는 모든 C# 형식은 단일 루트 `object`에서 상속됩니다. 따라서 일반적인 작업 집합을 공유하는 모든 형식과 모든 형식의 값을 일관된 방식으로 저장 및 전송하고 작업을 수행할 수 있습니다. 그뿐 아니라 C#은 사용자 정의 참조 형식 및 값 형식을 모두 지원하여 개체의 동적 할당과 더불어 간단한 구조의 인라인 스토리지도 구현합니다.

C# 프로그램 및 라이브러리가 호환되는 방식으로 시간에 따라 진화하게 하려면 디자인의 **버전 관리** C#에 많은 중점을 두어야 합니다. 다양한 프로그래밍 언어가 이 문제를 등한시하여 결과적으로 해당 언어로 작성된 프로그램이 최신 버전의 종속 라이브러리가 도입될 때 필요한 것보다 더 자주 종단되게 되었습니다. C# 버전 관리 고려 사항에 직접적으로 영향을 주는 디자인 측면에 `virtual`은 `override` 별도의 및 한정자, 메서드 오버로드 확인에 대한 규칙 및 명시적 인터페이스 멤버 선언에 대한 지원이 포함됩니다.

이 장의 나머지 부분에서는 C# 언어의 주요 기능에 대해 설명합니다. 이후 장에서는 세부 정보 지향적이고 때로는 수학적 방식으로 규칙 및 예외를 설명 하지만 장에서는 명확하게 이해하고 완전성을 위해 노력하고 있습니다. 초기 프로그램을 작성하는 데 도움이 되는 언어에 대한 소개와 이후 챕터의 읽기를 제공하는 것이 목적입니다.

## Hello World

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

```
using System;

class Hello
{
    static void Main() {
        Console.WriteLine("Hello, World");
    }
}
```

C# 소스 파일은 일반적으로 파일 확장명이 `.cs`입니다. "Hello, 세계" 프로그램이 파일 `hello.cs`에 저장되어 있는 경우 명령줄을 사용하여 Microsoft C# 컴파일러로 프로그램을 컴파일할 수 있습니다.

```
csc hello.cs
```

이라는 `hello.exe` 실행 가능한 어셈블리가 생성됩니다. 이 응용 프로그램이 실행될 때 생성되는 출력은 다음과 같습니다.

```
Hello, World
```

"Hello, World" 프로그램은 `System` 네임스페이스를 참조하는 `using` 지시문으로 시작합니다. 네임스페이스는 계층적으로 C# 프로그램 및 라이브러리를 구성하는 방법을 제공합니다. 네임스페이스에는 형식 및 다른 네임스페이스가 포함됩니다. 예를 들어 `System` 네임스페이스에는 많은 형식(예: 프로그램에 참조되는 `Console` 클래스) 및 많은 다른 네임스페이스(예: `IO` 및 `Collections`)가 포함되어 있습니다. 지정된 네임스페이스를 참조

하는 `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.

"Hello, World" 프로그램에서 선언된 `Hello` 클래스에는 단일 멤버인 `Main` 메서드가 있습니다. 메서드는 `static` 한정자를 사용하여 선언됩니다. `Main` 인스턴스 메서드는 키워드 `this`를 사용하여 특정 바깥쪽 개체 인스턴스를 참조할 수 있지만 정적 메서드는 특정 개체에 대한 참조 없이 작동합니다. 관례상 `Main`이라는 정적 메서드가 프로그램의 진입점으로 사용됩니다.

프로그램의 출력은 `System` 네임스페이스에 있는 `Console` 클래스의 `WriteLine` 메서드에 의해 생성됩니다. 이 클래스는 기본적으로 Microsoft C# 컴파일러에서 자동으로 참조하는 .NET Framework 클래스 라이브러리에서 제공됩니다. C# 자체에는 별도의 런타임 라이브러리가 없습니다. 대신 .NET Framework은의 C# 런타임 라이브러리입니다.

## 프로그램 구조

C#의 핵심적인 조직 개념은 **프로그램**, **네임스페이스**, **형식**, **멤버** 및 **어셈블리**입니다. C# 프로그램은 하나 이상의 소스 파일로 구성됩니다. 프로그램은 멤버를 포함하고 네임스페이스로 구성될 수 있는 형식을 선언합니다. 클래스와 인터페이스는 형식의 예입니다. 필드, 메서드, 속성 및 이벤트는 멤버의 예입니다. C# 프로그램을 컴파일하면 실제로 어셈블리로 패키지됩니다. 어셈블리는 일반적으로 **응용 프로그램** 또는 `.dll` **라이브러리**를 구현하는지에 따라 파일 확장명이 `.exe` 또는입니다.

예제

```
using System;

namespace Acme.Collections
{
    public class Stack
    {
        Entry top;

        public void Push(object data) {
            top = new Entry(top, data);
        }

        public object Pop() {
            if (top == null) throw new InvalidOperationException();
            object result = top.data;
            top = top.next;
            return result;
        }

        class Entry
        {
            public Entry next;
            public object data;

            public Entry(Entry next, object data) {
                this.next = next;
                this.data = data;
            }
        }
    }
}
```

`Stack`라는 `Acme.Collections` 네임스페이스의 이라는 클래스를 선언 합니다. 이 클래스의 정규화된 이름은 `Acme.Collections.Stack`입니다. 클래스에는 필드 `top`, 2개의 메서드 `Push` 및 `Pop`, 중첩된 클래스 `Entry` 등의 여러 멤버가 포함됩니다. `Entry` 클래스는 필드 `next` 및 필드 `data`, 생성자의 세 멤버가 포함됩니다. 예제의 소스 코드가 파일 `acme.cs`에 포함된다고 가정할 경우 다음 명령줄은

```
csc /t:library acme.cs
```

예제를 라이브러리(`Main` 진입점이 없는 코드)를 컴파일하고 `acme.dll`이라는 어셈블리를 생성합니다.

어셈블리에는 IL(중간 언어) 명령 형식의 실행 코드와 메타데이터 형식의 기호화된 정보가 포함되어 있습니다. 실행되기 전에 어셈블리의 IL 코드는 .NET 공용 언어 런타임의 JIT(Just-In-Time) 컴파일러에 의해 자동으로 프로세서 특정 코드로 변환됩니다.

어셈블리는 코드와 메타데이터를 모두 포함하는 기능의 자체 설명 단위이므로 C#에 `#include` 지시문 및 헤더 파일이 필요하지 않습니다. 특정 어셈블리에 포함된 공용 형식 및 멤버는 프로그램을 컴파일할 때 해당 어셈블리를 참조하는 것만으로 C# 프로그램에서 사용 가능해집니다. 예를 들어 이 프로그램에서는 `acme.dll` 어셈블

리의 `Acme.Collections.Stack` 클래스를 사용합니다.

```
using System;
using Acme.Collections;

class Test
{
    static void Main()
    {
        Stack s = new Stack();
        s.Push(1);
        s.Push(10);
        s.Push(100);
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
    }
}
```

`test.cs` 프로그램이 파일 `test.cs`에 저장 되어 있는 경우를 컴파일하면 `acme.dll` 컴파일러의 `/r` 옵션을 사용 하여 어셈블리를 참조할 수 있습니다.

```
csc /r:acme.dll test.cs
```

이를 통해 실행 시 출력을 생성하는 `test.exe`라는 실행 가능한 어셈블리가 만들어집니다.

```
100
10
1
```

C#은 프로그램의 소스 텍스트가 여러 소스 파일에 저장되도록 허용합니다. 다중 파일 C# 프로그램이 컴파일되면 모든 소스 파일이 함께 처리되고 소스 파일은 서로 자유롭게 참조될 수 있습니다. 개념적으로는 마치 모든 소스 파일이 처리되기 전에 하나의 큰 파일로 연결되는 것처럼 보입니다. 극소수의 경우를 제외하고 선언 순서는 중요하지 않으므로 C#에서는 정방향 선언이 절대 필요하지 않습니다. C#은 소스 파일을 하나의 공용 형식만 선언하도록 제한하거나 소스 파일 이름이 소스 파일에 선언된 형식과 일치하도록 요구하지 않습니다.

## 형식 및 변수

C#에는 두 가지 종류의 형식, 즉 **값 형식**과 **참조 형식**이 있습니다. 값 형식의 변수에는 해당 데이터가 직접 포함되지만 참조 형식의 변수에는 데이터(객체라고도 함)에 대한 참조가 저장됩니다. 참조 형식에서는 두 가지 변수가 같은 개체를 참조할 수 있으므로 한 변수에 대한 작업이 다른 변수에서 참조하는 개체에 영향을 미칠 수 있습니다. 값 형식에서는 변수에 데이터의 자체 사본이 들어 있으며 한 변수의 작업이 다른 변수에 영향을 미칠 수 없습니다(`ref` 및 `out` `ref` 및 `out` 매개 변수 제외).

C#의 값 형식은 **단순 형식**, **열거형 형식**, **구조체 형식** 및 **nullable 형식**으로 세분화 되며, C#의 참조 형식은 **클래스 형식**, **인터페이스 형식**, 배열로 추가로 나뉩니다. **형식 및 대리자 형식**.

다음 표에서는 형식 시스템에 C#대 한 개요를 제공 합니다.

범주		설명
값 형식	단순 형식	부호 있는 정수: <code>sbyte</code> , <code>short</code> , <code>int</code> , <code>long</code>
		부호 없는 정수: <code>byte</code> , <code>ushort</code> , <code>uint</code> , <code>ulong</code>
		유니코드 문자: <code>char</code>
		IEEE 부동 소수점: <code>float</code> , <code>double</code>
		High-Precision 10진수: <code>decimal</code>
		부울: <code>bool</code>
열거형		<code>enum E { ... }</code> 양식의 사용자 정의 형식

범주		설명
	구조체 형식	<code>struct S {...}</code> 양식의 사용자 정의 형식
	Nullable 형식	<code>null</code> 값을 갖는 다른 모든 값 형식의 확장
참조 형식	클래스 형식	다른 모든 형식의 기본 클래스: <code>object</code>
		유니코드 문자열: <code>string</code>
		<code>class C {...}</code> 양식의 사용자 정의 형식
	인터페이스 형식	<code>interface I {...}</code> 양식의 사용자 정의 형식
	배열 형식	단일 차원 및 다차원(예: <code>int[]</code> 및 <code>int[,]</code> )
	대리자 형식	사용자 정의 형식 (예: <code>delegate int D(...)</code> )

8가지 정수 형식은 부호 있는 형식 또는 부호 없는 형식으로 8비트, 16비트, 32비트 및 64비트 값에 대한 지원을 제공합니다.

두 부동 소수점 형식 `float` 및 `double`는 32 비트 단 정밀도 및 64 비트 배정밀도 IEEE 754 형식을 사용하여 표현됩니다.

`decimal` 형식은 재무 및 통화 계산에 적합한 128비트 데이터 형식입니다.

C#의 `bool` 형식은 부울 값을 나타내는데 사용되며, 값 `true` 은 또는 `false` 입니다.

C#의 문자 및 문자열 처리에서는 유니코드 인코딩이 사용됩니다. `char` 형식은 UTF-16 코드 단위를 나타내고, `string` 형식은 UTF-16 코드 단위 시퀀스를 나타냅니다.

다음 표에서는 숫자 C#형식을 요약 합니다.

범주	비트씩	형식	범위/전체 자릿수
부호 있는 정수	8	<code>sbyte</code>	-128...127
	16	<code>short</code>	-32,768...32,767
	32	<code>int</code>	-2,147,483,648...2,147,483,647
	64	<code>long</code>	-9,223,372,036,854,775,808...9,223,372,036,854,775,807
부호 없는 정수	8	<code>byte</code>	0...255
	16	<code>ushort</code>	0 ... 65,535
	32	<code>uint</code>	0 ... 4,294 개, 967,295
	64	<code>ulong</code>	0...18,446,744,073,709,551,615
부동 소수점	32	<code>float</code>	$1.5 \times 10^{-45} \sim 3.4 \times 10^{38}$ , 7 자리 전체 자릿수

범주	비트 쪽	형식	범위/전체 자릿수
	64	double	$5.0 \times 10^{-324} \sim 1.7 \times 10^{308}$ , 15 자리 전체 자릿수
Decimal	128	decimal	$1.0 \times 10^{-28} \sim 7.9 \times 10^{28}$ , 28 자리 전체 자릿수

C# 프로그램에서는 **형식 선언**을 사용하여 새 형식을 만듭니다. 형식 선언은 새 형식의 이름과 멤버를 지정합니다. 의 형식 C# 범주 중 5 개는 클래스 형식, 구조체 형식, 인터페이스 형식, 열거형 형식 및 대리자 형식과 같이 사용자가 정의할 수 있습니다.

클래스 형식은 데이터 멤버 (필드) 및 함수 멤버 (메서드, 속성 및 기타)를 포함하는 데이터 구조를 정의 합니다. 클래스 형식은 단일 상속 및 다형성과 파생된 클래스가 기본 클래스를 확장하고 특수화할 수 있는 메커니즘을 지원합니다.

구조체 형식은 데이터 멤버 및 함수 멤버를 포함하는 구조체를 나타내므로의 클래스 형식과 비슷합니다. 그러나 클래스와 달리 구조체는 값 형식이며 힘 할당이 필요하지 않습니다. 구조체 형식은 사용자 지정 상속을 지원하지 않으며 모든 구조체 형식은 `object` 형식에서 암시적으로 상속됩니다.

인터페이스 형식은 계약을 `public` 함수 멤버의 명명된 집합으로 정의 합니다. 인터페이스를 구현하는 클래스 또는 구조체는 인터페이스의 함수 멤버에 대한 구현을 제공해야 합니다. 인터페이스는 여러 기본 인터페이스에서 상속할 수 있으며, 클래스 또는 구조체는 여러 인터페이스를 구현할 수 있습니다.

대리자 형식은 특정 매개 변수 목록 및 반환 형식이 있는 메서드에 대한 참조를 나타냅니다. 대리자는 메서드를 변수에 할당되고 매개 변수로 전달될 수 있는 엔터티로 취급할 수 있도록 합니다. 또한 대리자는 다른 언어에 나오는 함수 포인터의 개념과 비슷하지만 함수 포인터와 달리 대리자는 개체 지향적이며 형식 안전 방식입니다.

클래스, 구조체, 인터페이스 및 대리자 형식은 모두 제네릭을 지원 하므로 다른 형식으로 매개 변수화 할 수 있습니다.

열거형 형식은 명명된 상수가 있는 고유 형식입니다. 모든 열거형 형식에는 8 개의 정수 계열 형식 중 하나 여야 하는 기본 형식이 있습니다. 열거형 형식의 값 집합은 내부 형식의 값 집합과 동일 합니다.

C#은 형식의 단일 차원 및 다차원 배열을 지원합니다. 위에 나열된 형식과 달리, 배열 형식은 사용하기 전에 먼저 선언할 필요가 없습니다. 대신, 배열 형식은 형식 이름을 대괄호로 묶어 생성합니다. `int[]` 예를 들어은의 1 차원 `int` `int` 배열이 `int[,]` 고, 는의 2 차원 배열이 고, 은의 `int[1]` `int[][]` 차원 배열에 해당하는 1 차원 배열입니다.

또한 `Nullable` 형식을 사용 하려면 먼저 선언 하지 않아도 됩니다. `Null`을 허용하지 않는 각 값 `T` 형식에 대해 추가 값 `null`을 `T?` 보유할 수 있는 해당 `nullable` 형식이 있습니다. 예를 들어 `int?` 는 32 비트 정수 또는 값 `null`을 보유할 수 있는 형식입니다.

C#의 형식 시스템은 통합 되어 모든 형식의 값을 개체로 처리할 수 있습니다. C#의 모든 형식은 `object` 클래스 형식에서 직접 또는 간접적으로 파생되고 `object`는 모든 형식의 기본 클래스입니다. 참조 형식의 값은 `object`로 인식함으로써 간단히 개체로 처리됩니다. 값 형식의 값은 **boxing** 및 **unboxing** 작업을 수행 하여 개체로 처리 됩니다. 다음 예제에서 `int` 값은 `object`로 변환되었다가 다시 `int`로 변환됩니다.

```
using System;

class Test
{
    static void Main()
    {
        int i = 123;
        object o = i;           // Boxing
        int j = (int)o;         // Unboxing
    }
}
```

값 형식의 값이 형식 `object`으로 변환 되면 "box" 라고도 하는 개체 인스턴스가 값을 보유하기 위해 할당 되고 값이 해당 상자에 복사 됩니다. 반대로 `object` 참조가 값 형식으로 캐스팅 되는 경우 참조 되는 개체가 올바른 값 형식의 상자인지 확인 하고, 검사에 성공 하면 상자의 값이 복사 됩니다.

C#의 통합 형식 시스템은 값 형식이 "요청 시" 개체가 될 수 있음을 의미 합니다. 통합 때문에 `object` 형식을 사용하는 범용 라이브러리는 참조 형식 및 값 형식 둘 다에 사용될 수 있습니다.

C#에는 필드, 배열 요소, 지역 변수 및 매개 변수를 포함하는 여러 종류의 **변수**가 있습니다. 변수는 저장소 위치를 나타내며 모든 변수에는 다음 표와 같이 변수에 저장할 수 있는 값을 결정하는 형식이 있습니다.

변수의 형식입니다.	가능한 내용
Null을 허용하지 않는 값 형식	정확한 해당 형식의 값
Null 허용 값 형식	Null 값 이거나 정확한 형식의 값입니다.
<code>object</code>	Null 참조, 참조 형식의 개체에 대한 참조 또는 모든 값 형식의 boxed 값에 대한 참조
클래스 형식	Null 참조, 해당 클래스 형식의 인스턴스에 대한 참조 또는 해당 클래스 형식에서 파생된 클래스의 인스턴스에 대한 참조입니다.
인터페이스 유형	Null 참조, 해당 인터페이스 형식을 구현하는 클래스 형식의 인스턴스에 대한 참조 또는 해당 인터페이스 형식을 구현하는 값 형식의 boxed 값에 대한 참조입니다.
배열 형식	Null 참조, 해당 배열 형식의 인스턴스에 대한 참조 또는 반환되는 배열 형식의 인스턴스에 대한 참조입니다.
대리자 형식	Null 참조 또는 해당 대리자 형식의 인스턴스에 대한 참조입니다.

## 표현식

식은 **피연산자** 및 **연산자**로 생성됩니다. 식의 연산자는 피연산자에 적용할 연산을 나타냅니다. 연산자의 예로 `+`, `-`, `*`, `/` 및 `new`가 있습니다. 피연산자의 예로는 리터럴, 필드, 지역 변수 및 식이 있습니다.

식에 여러 연산자가 포함되어 있으면 연산자의 **우선 순위**에 따라 개별 연산자가 계산되는 순서가 제어됩니다. 예를 들어 `*` 연산자는 `+` 연산자보다 우선 순위가 더 높기 때문에 식 `x + y * z`는 `x + (y * z)`로 계산됩니다.

대부분의 연산자는 **오버로드**할 수 있습니다. 연산자 오버로드는 피연산자 중 하나 또는 둘 다가 사용자 정의 클래스 또는 구조체 형식인 연산에 대해 사용자 정의 연산자 구현을 지정할 수 있도록 허용합니다.

다음 표에서는 연산자 C#를 요약하여 가장 높은 우선 순위 순으로 연산자 범주를 나열합니다. 동일한 범주의 연산자는 우선 순위가 같습니다.

범주	식	설명
주	<code>x.m</code>	멤버 액세스
	<code>x(...)</code>	메서드 및 대리자 호출
	<code>x[...]</code>	배열 및 인덱서 액세스
	<code>x++</code>	후위 증가
	<code>x--</code>	후위 감소
	<code>new T(...)</code>	개체 및 대리자 생성
	<code>new T(...){...}</code>	이니셜라이저를 사용한 개체 생성
	<code>new {...}</code>	익명 개체 이니셜라이저
	<code>new T[...]</code>	배열 생성
	<code>typeof(T)</code>	<code>T</code> 에 대한 <code>System.Type</code> 개체 가져오기
	<code>checked(x)</code>	<code>checked</code> 컨텍스트에서 식 계산
	<code>unchecked(x)</code>	<code>unchecked</code> 컨텍스트에서 식 계산
	<code>default(T)</code>	<code>T</code> 형식의 기본값 가져오기

범주	식	설명
	<code>delegate {...}</code>	익명 함수(무명 메서드)
단항	<code>+x</code>	ID
	<code>-x</code>	부정
	<code>!x</code>	논리 부정
	<code>~x</code>	비트 부정 연산
	<code>++x</code>	전위 증가
	<code>--x</code>	전위 감소
	<code>(T)x</code>	<code>x</code> 를 <code>T</code> 형식으로 명시적 변환
	<code>await x</code>	비동기적으로 <code>x</code> 완료 대기
곱하기	<code>x * y</code>	곱하기
	<code>x / y</code>	나눗셈 기호
	<code>x % y</code>	나머지
더하기	<code>x + y</code>	더하기, 문자열 연결, 대리자 결합
	<code>x - y</code>	빼기, 대리자 제거
Shift	<code>x &lt;&lt; y</code>	왼쪽 시프트
	<code>x &gt;&gt; y</code>	오른쪽 시프트
관계형 및 형식 테스트	<code>x &lt; y</code>	보다 작음
	<code>x &gt; y</code>	보다 큼
	<code>x &lt;= y</code>	다음보다 적거나 같음
	<code>x &gt;= y</code>	크거나 같음
	<code>x is T</code>	<code>x</code> 가 <code>T</code> 이면 <code>true</code> 반환, 그렇지 않으면 <code>false</code> 반환
	<code>x as T</code>	<code>T</code> 로 형식이 지정된 <code>x</code> 반환, <code>x</code> 가 <code>T</code> 가 아닌 경우 <code>null</code> 반환
같음	<code>x == y</code>	같음
	<code>x != y</code>	다음과 같지 않음
논리적 AND	<code>x &amp; y</code>	정수 비트 AND, 부울 논리곱 AND
논리 XOR	<code>x ^ y</code>	정수 비트 XOR, 부울 논리곱 XOR
논리적 OR	<code>x   y</code>	정수 비트 OR, 부울 논리곱 OR
조건부 AND	<code>x &amp;&amp; y</code>	가 <code>y</code> 인 경우 <code>x</code> 예만 평가 됩니다. <code>true</code>

범주	식	설명
조건부 OR	<code>x    y</code>	가 <code>y</code> 인 경우 <code>x</code> 에만 평가 됩니다. <code>false</code>
Null 결합	<code>x ?? y</code>	가 이면이 고 <code>null</code> , <code>x</code> 그렇지 않으면로 평가 됩니다. <code>y</code> <code>x</code>
조건	<code>x ? y : z</code>	<code>x</code> 가 <code>true</code> 이면 <code>y</code> , <code>x</code> 가 <code>false</code> 0면 <code>z</code> 로 평가
대입 또는 익명 함수	<code>x = y</code>	할당
	<code>x op= y</code>	복합 할당; 지원 되는 <code>*=</code> 연산자 <code>-=</code> <code>/=</code> <code>%=</code> <code>+=</code> <code>&lt;&lt;=</code> <code>&gt;&gt;=</code> <code>&amp;=</code> <code>^=</code> <code> =</code>
	<code>(T x) =&gt; y</code>	익명 함수(람다 식)

## 문

프로그램의 동작은 문을 사용하여 표현됩니다. C#은 여러 다른 종류의 문을 지원하며 이중 많은 문이 포함 문에 대해 정의됩니다.

**블록**은 단일 문이 허용되는 컨텍스트에서 여러 문을 쓸 수 있도록 허용합니다. 블록은 구분 기호 `{` 와 `}` 사이에 쓴 문 목록으로 구성됩니다.

**선언 문**은 지역 변수 및 상수를 선언하는 데 사용됩니다.

**식 문**은 식을 평가하는 데 사용됩니다. 문으로 사용할 수 있는 식에는 메서드 호출, `new` 연산자를 사용 하는 개체 할당, `=` 및 복합 할당 연산자를 사용 하는 할당,를 사용 하 `++`는 증가 및 감소 작업 등이 있습니다. `and` `--` 연산자 및 `wait` 식.

**선택 문**은 일부 식 값에 따라 실행할 수 있는 다양한 문 중에서 하나를 선택하는 데 사용됩니다. 이 그룹에는 `if` 및 `switch` 문이 포함됩니다.

**반복 문**은 포함 문을 반복 해 서 실행 하는 데 사용 됩니다. 이 그룹에는 `while`, `do`, `for` 및 `foreach` 문이 포함 됩니다.

**점프 문**은 제어를 전달하는 데 사용됩니다. 이 그룹에는 `break`, `continue`, `goto`, `throw`, `return` 및 `yield` 문이 포함됩니다.

`try ... catch` 문은 블록 실행 중에 발생하는 예외를 `catch`하는 데 사용되고 `try ... finally` 문은 예외 발생 여부에 관계 없이 항상 실행되는 종료 코드를 지정하는 데 사용됩니다.

`checked` 및 `unchecked` 문은 정수 계열 형식 산술 연산 및 변환에 대한 오버플로 검사 컨텍스트를 제어 하는 데 사용 됩니다.

`lock` 문은 지정된 개체에 대한 상호 배타적 잠금을 획득하고, 문을 실행한 후 잠금을 해제하는 데 사용됩니다.

`using` 문은 리소스를 획득하고, 문을 실행한 후 해당 리소스를 삭제하는 데 사용됩니다.

다음은 각 종류의 문 예입니다.

### 지역 변수 선언

```
static void Main() {
    int a;
    int b = 2, c = 3;
    a = 1;
    Console.WriteLine(a + b + c);
}
```

### 지역 상수 선언

```
static void Main() {
    const float pi = 3.1415927f;
    const int r = 25;
    Console.WriteLine(pi * r * r);
}
```

## 식 문

```
static void Main() {
    int i;
    i = 123;           // Expression statement
    Console.WriteLine(i); // Expression statement
    i++;              // Expression statement
    Console.WriteLine(i); // Expression statement
}
```

## if statement

```
static void Main(string[] args) {
    if (args.Length == 0) {
        Console.WriteLine("No arguments");
    }
    else {
        Console.WriteLine("One or more arguments");
    }
}
```

## switch statement

```
static void Main(string[] args) {
    int n = args.Length;
    switch (n) {
        case 0:
            Console.WriteLine("No arguments");
            break;
        case 1:
            Console.WriteLine("One argument");
            break;
        default:
            Console.WriteLine("{0} arguments", n);
            break;
    }
}
```

## while statement

```
static void Main(string[] args) {
    int i = 0;
    while (i < args.Length) {
        Console.WriteLine(args[i]);
        i++;
    }
}
```

## do statement

```
static void Main() {
    string s;
    do {
        s = Console.ReadLine();
        if (s != null) Console.WriteLine(s);
    } while (s != null);
}
```

## for statement

```
static void Main(string[] args) {
    for (int i = 0; i < args.Length; i++) {
        Console.WriteLine(args[i]);
    }
}
```

**foreach statement**

```
static void Main(string[] args) {
    foreach (string s in args) {
        Console.WriteLine(s);
    }
}
```

**break statement**

```
static void Main() {
    while (true) {
        string s = Console.ReadLine();
        if (s == null) break;
        Console.WriteLine(s);
    }
}
```

**continue statement**

```
static void Main(string[] args) {
    for (int i = 0; i < args.Length; i++) {
        if (args[i].StartsWith("/")) continue;
        Console.WriteLine(args[i]);
    }
}
```

**goto statement**

```
static void Main(string[] args) {
    int i = 0;
    goto check;
    loop:
    Console.WriteLine(args[i++]);
    check:
    if (i < args.Length) goto loop;
}
```

**return statement**

```
static int Add(int a, int b) {
    return a + b;
}

static void Main() {
    Console.WriteLine(Add(1, 2));
    return;
}
```

**yield statement**

```
static IEnumerable<int> Range(int from, int to) {
    for (int i = from; i < to; i++) {
        yield return i;
    }
    yield break;
}

static void Main() {
    foreach (int x in Range(-10, 10)) {
        Console.WriteLine(x);
    }
}
```

**throw 및 try 문**

```

static double Divide(double x, double y) {
    if (y == 0) throw new DivideByZeroException();
    return x / y;
}

static void Main(string[] args) {
    try {
        if (args.Length != 2) {
            throw new Exception("Two numbers required");
        }
        double x = double.Parse(args[0]);
        double y = double.Parse(args[1]);
        Console.WriteLine(Divide(x, y));
    }
    catch (Exception e) {
        Console.WriteLine(e.Message);
    }
    finally {
        Console.WriteLine("Good bye!");
    }
}

```

`checked` 및 `unchecked` 문

```

static void Main() {
    int i = int.MaxValue;
    checked {
        Console.WriteLine(i + 1);      // Exception
    }
    unchecked {
        Console.WriteLine(i + 1);      // Overflow
    }
}

```

`lock` statement

```

class Account
{
    decimal balance;
    public void Withdraw(decimal amount) {
        lock (this) {
            if (amount > balance) {
                throw new Exception("Insufficient funds");
            }
            balance -= amount;
        }
    }
}

```

`using` statement

```

static void Main() {
    using (TextWriter w = File.CreateText("test.txt")) {
        w.WriteLine("Line one");
        w.WriteLine("Line two");
        w.WriteLine("Line three");
    }
}

```

## 클래스 및 개체

클래스는 C#의 가장 기본적인 형식입니다. 클래스는 상태(필드)와 작업(메서드 및 기타 함수 멤버)을 하나의 단위로 결합하는 데이터 구조입니다. 클래스는 해당 클래스의 동적으로 생성된 인스턴스(개체라고도 함)에 대한 정의를 제공합니다. 클래스는 상속 및 다형성과 파생된 클래스가 기본 클래스를 확장하고 특수화할 수 있는 메커니즘을 지원합니다.

새 클래스는 클래스 선언을 사용하여 만들어집니다. 클래스 선언은 클래스의 특성 및 한정자, 클래스의 이름, 기본 클래스(제공된 경우), 클래스로 구현되는 인터페이스를 지정하는 헤더로 시작합니다. 헤더 다음에는 구분 기호 `{` 및 `}` 간에 작성되는 멤버 선언 목록으로 구성되는 클래스 본문이 나옵니다.

다음은 `Point`라는 간단한 클래스 선언입니다.

```

public class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

```

클래스의 인스턴스는 새 인스턴스에 대한 메모리를 할당하고, 인스턴스를 초기화하는 생성자를 호출하고, 인스턴스에 대한 참조를 반환하는 `new` 연산자를 사용하여 만들어집니다. 다음 문은 두 개의 `Point` 객체를 만들고 해당 객체에 대한 참조를 두 변수에 저장합니다.

```

Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);

```

개체가 차지하는 메모리는 개체가 더 이상 사용되지 않을 때 자동으로 회수 됩니다. C#에서 개체를 명시적으로 할당 취소할 필요도 없으며 가능하지도 않습니다.

### 멤버

클래스의 멤버는 정적 멤버 또는 인스턴스 멤버입니다. 정적 멤버는 클래스에 속하며 인스턴스 멤버는 개체(클래스의 인스턴스)에 속합니다.

다음 표에서는 클래스가 포함할 수 있는 멤버의 종류에 대한 개요를 제공합니다.

멤버	설명
상수	클래스와 연결된 상수 값
필드	클래스의 변수
메서드	클래스가 수행할 수 있는 계산 및 작업
속성	클래스의 명명된 속성에 대한 읽기 및 쓰기와 관련된 작업
인덱서	클래스 인스턴스를 배열처럼 인덱싱하는 것과 관련된 작업
이벤트	클래스에 의해 생성될 수 있는 알림
연산자	클래스가 지원하는 변환 및 식 연산자
생성자	클래스의 인스턴스 또는 클래스 자체를 초기화하는 데 필요한 작업
소멸자	클래스의 인스턴스가 영구적으로 삭제되기 전에 수행 작업
유형	클래스에 의해 선언된 중첩 형식

### 액세스 가능성

클래스의 각 멤버에는 멤버에 액세스할 수 있는 프로그램 텍스트의 영역을 제어하는 액세스 가능성성이 연결되어 있습니다. 액세스 가능성은 5가지 형태로 제공됩니다. 각각은 다음 표에 요약되어 있습니다.

액세스 가능성	임을
<code>public</code>	액세스가 제한되지 않음
<code>protected</code>	이 클래스 또는 이 클래스에서 파생된 클래스로만 액세스가 제한됨
<code>internal</code>	이 프로그램으로만 액세스가 제한됨
<code>protected internal</code>	이 프로그램 또는 이 클래스에서 파생된 클래스로만 액세스가 제한됨
<code>private</code>	이 클래스로만 액세스가 제한됨

## 형식 매개 변수

클래스 정의는 클래스 이름 다음에 대괄호로 묶은 형식 매개 변수 이름 목록을 지정하여 형식 매개 변수 집합을 지정할 수 있습니다. 형식 매개 변수는 클래스 선언 본문에서 클래스의 멤버를 정의 하는 데 사용할 수 있습니다. 다음 예제에서 `Pair`의 형식 매개 변수는 `TFirst` 및 `TSecond`입니다.

```
public class Pair<TFirst,TSecond>
{
    public TFirst First;
    public TSecond Second;
}
```

형식 매개 변수를 사용 하도록 선언 된 클래스 형식을 제네릭 클래스 형식이라고 합니다. 구조체, 인터페이스 및 대리자 형식도 제네릭일 수 있습니다.

제네릭 클래스를 사용하는 경우 각 형식 매개 변수에 대해 다음과 같은 형식 인수가 제공되어야 합니다.

```
Pair<int,string> pair = new Pair<int,string> { First = 1, Second = "two" };
int i = pair.First;      // TFirst is int
string s = pair.Second; // TSecond is string
```

위와 같이 `Pair<int,string>` 형식 인수를 제공 하는 제네릭 형식을 생성 된 형식이라고 합니다.

## 기본 클래스

클래스 선언은 클래스 이름 및 형식 매개 변수 뒤에 콜론과 기본 클래스의 이름을 사용하여 기본 클래스를 지정 할 수 있습니다. 기본 클래스 지정을 생략하면 `object` 형식에서 파생되는 클래스와 같습니다. 다음 예제에서 `Point3D`의 기본 클래스는 `Point`이고 `Point`의 기본 클래스는 `object`입니다.

```
public class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

public class Point3D: Point
{
    public int z;

    public Point3D(int x, int y, int z): base(x, y) {
        this.z = z;
    }
}
```

클래스는 기본 클래스의 멤버를 상속합니다. 상속은 인스턴스 및 정적 생성자와 기본 클래스의 소멸자를 제외 하고, 클래스에 기본 클래스의 모든 멤버를 암시적으로 포함 하는 것을 의미 합니다. 파생된 클래스를 상속하는 대상에 새 멤버를 추가할 수 있지만 상속된 멤버의 정의를 제거할 수 없습니다. 앞의 예제에서 `Point3D`는 `Point`에서 `x` 및 `y` 필드를 상속하고 모든 `Point3D` 인스턴스는 세 개의 필드, 즉 `x`, `y` 및 `z`를 포함합니다.

클래스 형식에서 해당 기본 클래스 형식 간에 암시적 변환이 존재합니다. 따라서 클래스 형식의 변수는 해당 클래스의 인스턴스 또는 파생된 모든 클래스의 인스턴스를 참조할 수 있습니다. 예를 들어 이전 클래스 선언에서 형식 `Point`의 변수는 `Point` 또는 `Point3D`를 참조할 수 있습니다.

```
Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);
```

## 필드

필드는 클래스 또는 클래스의 인스턴스와 연결 된 변수입니다.

`static` 한정자를 사용 하여 선언 된 필드는 정적 필드를 정의 합니다. 정적 필드는 정확히 하나의 스토리지 위치를 식별합니다. 생성된 클래스 인스턴스 수에 관계없이 정적 필드의 복사본은 하나뿐입니다.

한정자를 `static` 사용 하지 않고 선언 된 필드는 인스턴스 필드를 정의 합니다. 클래스의 모든 인스턴스는 해당 클래스의 모든 인스턴스 필드의 별도 복사본을 포함합니다.

다음 예제에서 `Color` 클래스의 각 인스턴스는 `r`, `g` 및 `b` 인스턴스 필드의 별도 복사본을 갖지만 `Black`, `White`, `Red`, `Green` 및 `Blue` 정적 필드의 복사본은 하나뿐입니다.

```

public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);
    private byte r, g, b;

    public Color(byte r, byte g, byte b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}

```

앞의 예제와 같이 **읽기 전용 필드**는 `readonly` 한정자를 사용하여 선언될 수 있습니다. 필드에 대 `readonly` 한 할당은 필드의 선언이 나 같은 클래스의 생성자에서만 발생할 수 있습니다.

### 메서드

**메서드**는 개체 또는 클래스에서 수행할 수 있는 계산이나 작업을 구현하는 멤버입니다. **정적 메서드**는 클래스를 통해 액세스됩니다. **인스턴스 메서드**는 클래스의 인스턴스를 통해 액세스됩니다.

메서드에는 메서드에 전달 되는 값 또는 변수 참조를 나타내는 **매개 변수의 목록** (비어 있을 수 있음) 및 메서드에서 계산 되고 반환 되는 값의 형식을 지정 하는 **반환 형식이 있습니다**. 값을 반환 하지 않는 메서드의 `void` 반환 형식은입니다.

형식과 마찬가지로 메서드에는 메서드가 호출될 때 형식 인수가 지정되어야 하는 형식 매개 변수 집합도 있을 수 있습니다. 형식과 달리 형식 인수는 종종 메서드 호출의 인수에서 유추될 수 있으므로 명시적으로 지정할 필요가 없습니다.

메서드의 **시그니처**는 메서드가 선언되는 클래스에서 고유해야 합니다. 메서드 시그니처는 메서드의 이름, 형식 매개 변수의 수, 해당 매개 변수의 수, 한정자 및 형식으로 구성됩니다. 메서드 시그니처는 반환 형식을 포함하지 않습니다.

### 매개 변수

매개 변수는 메서드에 값 또는 변수 참조를 전달하는 데 사용됩니다. 메서드의 매개 변수는 메서드가 호출될 때 지정된 **인수**에서 실제 값을 가져옵니다. 매개 변수에는 값 매개 변수, 참조 매개 변수, 출력 매개 변수 및 매개 변수 배열의 네 가지 종류가 있습니다.

**값 매개 변수**는 입력 매개 변수 전달에 사용됩니다. 값 매개 변수는 매개 변수에 전달된 인수에서 초기 값을 가져오는 지역 변수에 해당합니다. 값 매개 변수를 수정해도 매개 변수에 전달된 인수에는 영향을 주지 않습니다.

해당 인수를 생략할 수 있도록 기본값을 지정하면 값 매개 변수는 선택적일 수 있습니다.

**참조 매개 변수**는 입력 및 출력 매개 변수 전달 둘 다에 사용됩니다. 참조 매개 변수에 전달되는 인수는 변수어야 하며, 메서드를 실행하는 동안 참조 매개 변수는 인수 변수와 동일한 스토리지 위치를 나타냅니다. 참조 매개 변수는 `ref` 한정자를 사용하여 선언됩니다. 다음 예제에서는 `ref` 매개 변수를 사용하는 방법을 보여 줍니다.

```

using System;

class Test
{
    static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
    }

    static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("{0} {1}", i, j);           // Outputs "2 1"
    }
}

```

**출력 매개 변수**는 출력 매개 변수 전달에 사용됩니다. 출력 매개 변수는 호출자가 제공한 인수의 초기 값이 중요하지 않다는 점을 제외하고 참조 매개 변수와 비슷합니다. 출력 매개 변수는 `out` 한정자를 사용하여 선언됩니다. 다음 예제에서는 `out` 매개 변수를 사용하는 방법을 보여 줍니다.

```

using System;

class Test
{
    static void Divide(int x, int y, out int result, out int remainder) {
        result = x / y;
        remainder = x % y;
    }

    static void Main() {
        int res, rem;
        Divide(10, 3, out res, out rem);
        Console.WriteLine("{0} {1}", res, rem); // Outputs "3 1"
    }
}

```

**매개 변수 배열**은 다양한 개수의 인수가 메서드에 전달되도록 허용합니다. 매개 변수 배열은 `params` 한정자를 사용하여 선언됩니다. 메서드의 마지막 매개 변수만 매개 변수 배열일 수 있으며 매개 변수 배열의 형식은 1차원 배열 형식이어야 합니다. 클래스의 `WriteLine` `Write` 및 메서드는 매개변수 배열 사용의 좋은 예입니다.

`System.Console`. 이러한 메서드는 다음과 같이 선언됩니다.

```

public class Console
{
    public static void Write(string fmt, params object[] args) {...}
    public static void WriteLine(string fmt, params object[] args) {...}
    ...
}

```

매개 변수 배열을 사용하는 메서드 내에서 매개 변수 배열은 배열 형식의 일반 매개 변수와 정확히 동일하게 동작합니다. 그러나 매개 변수 배열을 사용한 메서드 호출에서 매개 변수 배열 형식의 단일 인수 또는 매개 변수 배열에 있는 임의 개수의 요소 형식 인수를 전달할 수 있습니다. 후자의 경우 지정된 인수를 사용하여 배열 인스턴스가 자동으로 만들어지고 초기화됩니다. 다음 예제는

```
Console.WriteLine("x={0} y={1} z={2}", x, y, z);
```

다음을 작성하는 것과 같습니다.

```

string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);

```

메서드 본문 및 지역 변수

메서드 본문은 메서드가 호출될 때 실행할 문을 지정합니다.

메서드 본문은 메서드 호출과 관련된 변수를 선언할 수 있습니다. 이러한 변수를 **지역 변수**라고 합니다. 지역 변수 선언은 형식 이름, 변수 이름을 지정하며 초기 값을 지정할 수도 있습니다. 다음 예제에서는 초기 값이 0인 지역 변수 `i`와 초기 값이 없는 지역 변수 `j`를 선언합니다.

```

using System;

class Squares
{
    static void Main() {
        int i = 0;
        int j;
        while (i < 10) {
            j = i * i;
            Console.WriteLine("{0} x {0} = {1}", i, j);
            i = i + 1;
        }
    }
}

```

C#에서는 해당 값을 얻기 위해 먼저 로컬 변수를 **명확하게 할당**해야 합니다. 예를 들어 이전 `i`의 선언에 초기 값이 포함되지 않으면 컴파일러는 `i`의 후속 사용에 대해 오류를 보고합니다. `i`는 프로그램에서 해당 시점에 명확하게 할당되지 않은 것이기 때문입니다.

메서드는 `return` 문을 사용하여 해당 호출자에게 컨트롤을 반환할 수 있습니다. `void`를 반환하는 메서드에서

`return` 문은 식을 지정할 수 없습니다. `void` 이 아닌 `return` 반환하는 메서드에서는 반환 값을 계산하는 식을 포함해야 합니다.

정적 및 인스턴스 메서드

`static` 한정자를 사용 하여 선언 된 메서드는 정적 메서드입니다. 정적 메서드는 특정 인스턴스에 작동하지 않고 정적 멤버에 직접적으로만 액세스할 수 있습니다.

`static` 한정자 없이 선언 된 메서드는 인스턴스 메서드입니다. 인스턴스 메서드는 특정 인스턴스에 작동하며 정적 및 인스턴스 멤버 둘 다에 액세스할 수 있습니다. 인스턴스 메서드가 호출된 인스턴스는 `this`로 명시적으로 액세스할 수 있습니다. 정적 메서드에서 `this`를 참조하면 오류가 발생합니다.

다음 `Entity` 클래스에는 정적 멤버와 인스턴스 멤버가 모두 있습니다.

```
class Entity
{
    static int nextSerialNo;
    int serialNo;

    public Entity() {
        serialNo = nextSerialNo++;
    }

    public int GetSerialNo() {
        return serialNo;
    }

    public static int GetNextSerialNo() {
        return nextSerialNo;
    }

    public static void SetNextSerialNo(int value) {
        nextSerialNo = value;
    }
}
```

각 `Entity` 인스턴스에는 일련 번호(및 여기에 표시되지 않는 일부 정보)가 포함되어 있습니다. `Entity` 생성자(인스턴스 메서드와 유사함)는 사용 가능한 다음 일련 번호를 사용하여 새 인스턴스를 초기화합니다. 생성자가 인스턴스 멤버이기 때문에 `serialNo` 인스턴스 필드 및 `nextSerialNo` 정적 필드 둘 다에 액세스하도록 허용됩니다.

`GetNextSerialNo` 및 `SetNextSerialNo` 정적 메서드는 `nextSerialNo` 정적 필드에 액세스할 수 있지만 `serialNo` 인스턴스 필드에 직접 액세스하면 오류가 발생합니다.

다음 예제에서는 사용 된 `Entity` 클래스입니다.

```
using System;

class Test
{
    static void Main() {
        Entity.SetNextSerialNo(1000);
        Entity e1 = new Entity();
        Entity e2 = new Entity();
        Console.WriteLine(e1.GetSerialNo());           // Outputs "1000"
        Console.WriteLine(e2.GetSerialNo());           // Outputs "1001"
        Console.WriteLine(Entity.GetNextSerialNo());   // Outputs "1002"
    }
}
```

`SetNextSerialNo` 및 `GetNextSerialNo` 정적 메서드는 클래스에 대해 호출되지만 `GetSerialNo` 인스턴스 메서드는 클래스의 인스턴스에 대해 호출됩니다.

가상, 재정의 및 추상 메서드

인스턴스 메서드 선언에 `virtual` 한정자가 포함되면 해당 메서드를 가상 메서드라고 합니다. 한정자가 `virtual` 없는 경우이 메서드는 비가상 메서드라고 합니다.

가상 메서드가 호출되면 호출이 발생하는 인스턴스의 런타임 형식에 따라 호출할 실제 메서드 구현이 결정됩니다. 비가상 메서드 호출에서는 인스턴스의 컴파일 타임 형식이 결정 요인입니다.

가상 메서드는 파생된 클래스에서 재정의될 수 있습니다. 인스턴스 메서드 선언에 한정자가 `override` 포함된 경우 메서드는 동일한 서명으로 상속 된 가상 메서드를 재정의 합니다. 가상 메서드 선언은 새 메서드를 도입하지만 재정의 메서드 선언은 해당 메서드의 새 구현을 제공하여 기존의 상속된 가상 메서드를 특수화합니다.

추상 메서드는 구현이 없는 가상 메서드입니다. 추상 메서드는 `abstract` 한정자를 사용 하여 선언 되고 선

언 `abstract` 된 클래스에서만 허용 됩니다. 추상 메서드는 모든 비추상 파생 클래스에서 재정의해야 합니다.

다음 예제에서는 식 트리 노드를 나타내는 추상 클래스 `Expression` 와 상수, 변수 참조 및 산술 연산에 대한 식 트리 노드를 구현하는 세 개의 파생 클래스 `Constant`, `VariableReference` 및 `Operation`을 선언합니다. 이는 유사하지만 [식 트리 형식](#)에서 도입 된 식 트리 형식과 혼동 하지 않아야 합니다.

```
using System;
using System.Collections;

public abstract class Expression
{
    public abstract double Evaluate(Hashtable vars);
}

public class Constant: Expression
{
    double value;

    public Constant(double value) {
        this.value = value;
    }

    public override double Evaluate(Hashtable vars) {
        return value;
    }
}

public class VariableReference: Expression
{
    string name;

    public VariableReference(string name) {
        this.name = name;
    }

    public override double Evaluate(Hashtable vars) {
        object value = vars[name];
        if (value == null) {
            throw new Exception("Unknown variable: " + name);
        }
        return Convert.ToDouble(value);
    }
}

public class Operation: Expression
{
    Expression left;
    char op;
    Expression right;

    public Operation(Expression left, char op, Expression right) {
        this.left = left;
        this.op = op;
        this.right = right;
    }

    public override double Evaluate(Hashtable vars) {
        double x = left.Evaluate(vars);
        double y = right.Evaluate(vars);
        switch (op) {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;
        }
        throw new Exception("Unknown operator");
    }
}
```

이전의 4개 클래스는 산술 연산자를 모델링하는데 사용할 수 있습니다. 예를 들어 이러한 클래스의 인스턴스를 사용할 경우 식 `x + 3` 을 다음과 같이 나타낼 수 있습니다.

```
Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));
```

`Expression` 인스턴스의 `Evaluate` 메서드는 지정된 식을 계산하고 `double` 값을 생성하기 위해 호출됩니다. 메서드는 변수 이름 (항목의 `Hashtable` 키)과 값 (항목의 값)을 포함하는를 인수로 사용 합니다. 메서드 `Evaluate`

는 가상 추상 메서드이므로 추상이 아닌 파생 클래스에서 실제 구현을 제공하기 위해 재정의 해야 하는 것을 의미 합니다.

`Evaluate` 의 `Constant` 구현은 단순히 저장된 상수를 반환합니다. `VariableReference` 의 구현은 hashtable에서 변수 이름을 조회하고 결과 값을 반환합니다. `Operation` 의 구현은 먼저 왼쪽 및 오른쪽 피연산자를 계산하고 (재귀적으로 해당 `Evaluate` 메서드 호출) 지정된 산술 연산을 수행합니다.

다음 프로그램에서는 `Expression` 클래스를 사용하여 `x` 및 `y`의 다른 값에 대해 식 `x * (y + 2)`를 계산합니다.

```
using System;
using System.Collections;

class Test
{
    static void Main() {
        Expression e = new Operation(
            new VariableReference("x"),
            '*',
            new Operation(
                new VariableReference("y"),
                '+',
                new Constant(2)
            )
        );
        Hashtable vars = new Hashtable();
        vars["x"] = 3;
        vars["y"] = 5;
        Console.WriteLine(e.Evaluate(vars));           // Outputs "21"
        vars["x"] = 1.5;
        vars["y"] = 9;
        Console.WriteLine(e.Evaluate(vars));           // Outputs "16.5"
    }
}
```

#### 메서드 오버로드

메서드 **오버로드**는 동일한 클래스가 고유한 시그니처를 갖는 한, 동일한 이름을 갖도록 허용합니다. 오버로드된 메서드의 호출을 컴파일할 때 컴파일러는 **오버로드 확인**을 사용하여 호출할 특정 메서드를 결정합니다. 오버로드 확인은 인수와 가장 적합하게 일치하는 단일 메서드를 찾으며, 최상의 일치 메서드를 찾을 수 있는 경우 오류를 보고합니다. 다음 예제에서는 실제로 진행되는 오버로드 확인을 보여 줍니다. `Main` 메서드의 각 호출에 대한 주석은 실제로 호출되는 메서드를 보여 줍니다.

```

class Test
{
    static void F() {
        Console.WriteLine("F()");
    }

    static void F(object x) {
        Console.WriteLine("F(object)");
    }

    static void F(int x) {
        Console.WriteLine("F(int)");
    }

    static void F(double x) {
        Console.WriteLine("F(double)");
    }

    static void F<T>(T x) {
        Console.WriteLine("F<T>(T)");
    }

    static void F(double x, double y) {
        Console.WriteLine("F(double, double)");
    }

    static void Main() {
        F();                      // Invokes F()
        F(1);                     // Invokes F(int)
        F(1.0);                   // Invokes F(double)
        F("abc");                 // Invokes F(object)
        F((double)1);             // Invokes F(double)
        F((object)1);             // Invokes F(object)
        F<int>(1);                // Invokes F<T>(T)
        F(1, 1);                  // Invokes F(double, double)
    }
}

```

예제와 같이, 인수를 정확한 매개 변수 형식으로 명시적으로 캐스팅하거나 형식 인수를 명시적으로 제공하여 항상 특정 메서드를 선택할 수 있습니다.

#### 기타 함수 멤버

실행 코드를 포함하는 멤버를 통칭하여 클래스의 **함수 멤버**라고 합니다. 이전 섹션에서는 함수 멤버의 기본 종류인 메서드에 대해 설명합니다. 이 섹션에서는 생성자, 속성, 인덱서, 이벤트, C#연산자 및 소멸자에서 지원하는 다른 종류의 함수 멤버에 대해 설명 합니다.

다음 코드에서는 개체의 growable 목록을 구현 `List<T>` 하는 라는 제네릭 클래스를 보여 줍니다. 이 클래스는 함수 멤버의 가장 일반적인 몇 가지 예제를 포함합니다.

```

public class List<T> {
    // Constant...
    const int defaultCapacity = 4;

    // Fields...
    T[] items;
    int count;

    // Constructors...
    public List(int capacity = defaultCapacity) {
        items = new T[capacity];
    }

    // Properties...
    public int Count {
        get { return count; }
    }
    public int Capacity {
        get {
            return items.Length;
        }
        set {
            if (value < count) value = count;
            if (value != items.Length) {
                T[] newItems = new T[value];
                Array.Copy(items, 0, newItems, 0, count);
                items = newItems;
            }
        }
    }

    // Indexer...
    public T this[int index] {
        get {
            return items[index];
        }
        set {
            items[index] = value;
            OnChanged();
        }
    }

    // Methods...
    public void Add(T item) {
        if (count == Capacity) Capacity = count * 2;
        items[count] = item;
        count++;
        OnChanged();
    }
    protected virtual void OnChanged() {
        if (Changed != null) Changed(this, EventArgs.Empty);
    }
    public override bool Equals(object other) {
        return Equals(this, other as List<T>);
    }
    static bool Equals(List<T> a, List<T> b) {
        if (a == null) return b == null;
        if (b == null || a.count != b.count) return false;
        for (int i = 0; i < a.count; i++) {
            if (!object.Equals(a.items[i], b.items[i])) {
                return false;
            }
        }
        return true;
    }

    // Event...
    public event EventHandler Changed;

    // Operators...
    public static bool operator ==(List<T> a, List<T> b) {
        return Equals(a, b);
    }
    public static bool operator !=(List<T> a, List<T> b) {
        return !Equals(a, b);
    }
}

```

생성자

C#은 인스턴스 및 정적 생성자를 모두 지원합니다. **인스턴스 생성자는** 클래스의 인스턴스를 초기화하는 데 필요한 작업을 구현하는 멤버입니다. **정적 생성자는** 처음 로드될 때 클래스 자체를 인스턴스화하는 데 필요한 작

업을 구현하는 멤버입니다.

생성자는 반환 형식이 없고 포함하는 클래스와 동일한 이름을 갖는 메서드처럼 선언됩니다. 생성자 선언에 한정자가 `static` 포함된 경우 정적 생성자를 선언 합니다. 그렇지 않으면 인스턴스 생성자를 선언합니다.

인스턴스 생성자를 오버 로드할 수 있습니다. 예를 들어 `List<T>` 클래스는 2개의 인스턴스 생성자, 즉, 매개 변수가 없는 생성자와 `int` 매개 변수를 취하는 생성자를 선언합니다. 인스턴스 생성자는 `new` 연산자를 사용하여 호출됩니다. 다음 문은 `List` 클래스의 각 `List<string>` 생성자를 사용하여 두 개의 인스턴스를 할당합니다.

```
List<string> list1 = new List<string>();
List<string> list2 = new List<string>(10);
```

다른 멤버와 달리 인스턴스 생성자는 상속되지 않으며 클래스에는 클래스에서 실제로 선언된 인스턴스 생성자만 포함됩니다. 클래스에 대해 인스턴스 생성자가 제공되지 않으면 매개 변수가 없는 빈 인스턴스 생성자가 자동으로 제공됩니다.

속성

속성은 필드의 기본 확장입니다. 둘 다 연결된 형식으로 명명되는 멤버이며, 필드 및 속성에 액세스하는 구문은 동일합니다. 그러나 필드와 달리 속성은 스토리지 위치를 명시하지 않습니다. 대신, 속성에는 해당 값을 읽거나 쓸 때 실행될 문을 지정하는 접근자가 있습니다.

속성은 필드처럼 선언 됩니다 `get`. 단, 선언은 접근자 및 / `set` 또는 구분 기호 `{` 사이에 `}` 쓰여진 접근자 및 / 또는 접근자를 사용하여 끝나는 대신 세미콜론으로 끝나지 않습니다. `get` `get` 접근자 `set`과 접근자가 모두 있는 속성은 읽기/쓰기 속성이이며 접근자만 있는 속성은 읽기 전용 속성이이며 접근자만 있는 속성은입니다. `set` 쓰기 전용 속성입니다.

접근자 `get` 는 속성 형식의 반환 값이 있는 매개 변수가 없는 메서드에 해당 합니다. 할당의 대상이 아닌 `get`에서 속성을 참조하는 경우 속성의 접근자가 호출되어 속성의 값을 계산 합니다.

접근자 `set` 는 라는 `value` 단일 매개 변수가 있고 반환 형식이 없는 메서드에 해당 합니다. 속성이 할당의 대상이 나 또는 `++` `--`의 피연산자로 참조 되는 경우 접근자는 `set` 새 값을 제공하는 인수를 사용하여 호출 됩니다.

`List<T>` 클래스는 각각 읽기 전용 및 읽기/쓰기 특성을 갖는 두 개의 속성 `Count` 및 `Capacity`를 선언합니다. 다음은 이러한 속성 사용의 예입니다.

```
List<string> names = new List<string>();
names.Capacity = 100;           // Invokes set accessor
int i = names.Count;           // Invokes get accessor
int j = names.Capacity;        // Invokes get accessor
```

필드 및 메서드와 마찬가지로, C#은 인스턴스 속성 및 정적 속성을 모두 지원합니다. 정적 속성은 `static` 한정자를 사용하여 선언되고, 인스턴스 속성은 포함하지 않고 선언됩니다.

속성의 접근자는 가상일 수 있습니다. 속성 선언에 `virtual`, `abstract`, 또는 `override` 한정자가 포함되면 속성의 접근자에 적용됩니다.

인덱서

인덱서는 개체가 배열과 같은 방식으로 인덱싱될 수 있도록 하는 멤버입니다. 인덱서는 `this` 과(와) 구분 기호 `[` 및 `]` 사이에 작성된 매개 변수 목록을 합쳐서 구성원 이름으로 사용한다는 점을 제외하고 속성처럼 선언됩니다. 매개 변수는 인덱서의 접근자에서 사용할 수 있습니다. 속성과 마찬가지로 인덱서는 읽기/쓰기, 읽기 전용 및 쓰기 전용일 수 있으며 인덱서의 접근자는 가상일 수 있습니다.

`List` 클래스는 `int` 매개 변수를 사용하는 단일 읽기/쓰기 인덱서를 선언합니다. 인덱서는 `List` 인스턴스를 `int` 값으로 인덱싱할 수 있도록 합니다. 예

```
List<string> names = new List<string>();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++) {
    string s = names[i];
    names[i] = s.ToUpper();
}
```

인덱서는 오버로드될 수 있습니다. 즉, 해당 매개 변수의 수와 형식이 다를 경우 한 클래스가 여러 인덱서를 선언할 수 있습니다.

이벤트

**이벤트**는 클래스 또는 개체가 알림을 제공할 수 있도록 하는 멤버입니다. 이벤트는 선언에 `event` 키워드가 포함되고 형식이 대리자 형식이어야 한다는 점만 제외하고 필드처럼 선언 됩니다.

이벤트 멤버를 선언하는 클래스 내에서 이벤트는 대리자 형식의 필드처럼 동작합니다(이벤트가 추상이 아니고 접근자를 선언하지 않을 경우). 필드는 이벤트에 추가된 이벤트 처리기를 나타내는 대리자에 대한 참조를 저장합니다. 이벤트 핸들이 없는 경우 필드 `null`입니다.

`List<T>` 클래스는 `Changed`라는 단일 이벤트 멤버를 선언합니다. 이것은 새 항목이 목록에 추가되었음을 나타냅니다. 이벤트 `Changed`는 이벤트가 발생하는지 여부 `OnChanged` `null`를 먼저 확인하는 가상 메서드에 의해 발생합니다. 즉, 처리기가 없음을 의미 합니다. 이벤트 발생 개념은 이벤트가 나타내는 대리자를 호출하는 것과 정확히 동일하므로 이벤트 발생을 위한 특수한 언어 구문은 없습니다.

클라이언트는 **이벤트 처리기**를 통해 이벤트에 반응합니다. 이벤트 처리기는 `+=` 연산자를 사용하여 추가되고, `-=` 연산자를 사용하여 제거됩니다. 다음 예제에서는 이벤트 처리기를 `List<string>`의 `Changed` 이벤트에 추가합니다.

```
using System;

class Test
{
    static int changeCount;

    static void ListChanged(object sender, EventArgs e) {
        changeCount++;
    }

    static void Main() {
        List<string> names = new List<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(changeCount);           // Outputs "3"
    }
}
```

이벤트의 기본 스토리지를 제어하려고 하는 고급 시나리오의 경우 이벤트 선언에서 속성의 `set` 접근자와 비슷한 `add` 및 `remove` 접근자를 명시적으로 제공할 수 있습니다.

연산자

**연산자는** 클래스 인스턴스에 특정 식 연산자를 적용하는 것의 의미를 정의하는 멤버입니다. 세 가지 종류의 연산자, 즉, 단항 연산자, 이항 연산자 및 변환 연산자를 정의할 수 있습니다. 모든 연산자는 `public` 및 `static`으로 선언해야 합니다.

`List<T>` 클래스는 두 가지 연산자인 `operator==` 및 `operator!=`를 선언하므로 해당 연산자를 `List` 인스턴스에 적용하는 새로운 의미를 식에 지정합니다. 특히 연산자는 해당 `List<T>` `Equals` 메서드를 사용하여 각 포함된 개체를 비교하는 것처럼 두 인스턴스의 같음을 정의 합니다. 다음 예제에서는 `==` 연산자를 사용하여 두 `List<int>` 인스턴스를 비교합니다.

```
using System;

class Test
{
    static void Main() {
        List<int> a = new List<int>();
        a.Add(1);
        a.Add(2);
        List<int> b = new List<int>();
        b.Add(1);
        b.Add(2);
        Console.WriteLine(a == b);           // Outputs "True"
        b.Add(3);
        Console.WriteLine(a == b);           // Outputs "False"
    }
}
```

두 목록은 같은 순서로 같은 값을 갖는 동일한 수의 개체를 포함하므로 첫 번째 `Console.WriteLine`은 `True`를 출력합니다. `List<T>`에서 `operator==`이 정의되지 않았으면 `a` 및 `b`은 다른 `List<int>` 인스턴스를 참조하므로 첫 번째 `Console.WriteLine`은 `False`를 출력합니다.

소멸자

**소멸자는** 클래스의 인스턴스를 소멸시키는데 필요한 작업을 구현하는 멤버입니다. 소멸자에는 매개 변수를 사용할 수 없으며, 액세스 가능성 한정자를 가질 수 없으며 명시적으로 호출할 수 없습니다. 인스턴스의 소멸자

는 가비지 수집 중에 자동으로 호출 됩니다.

가비지 수집기는 개체를 수집 하고 소멸자를 실행 하는 시기를 결정 하는 데 사용할 수 있는 광범위 한 위도입니다. 특히 소멸자 호출의 타이밍은 결정적이 아니며 소멸자가 모든 스레드에서 실행 될 수 있습니다. 이러한 다른 이유 때문에 클래스는 다른 솔루션이 불가능 한 경우에만 소멸자를 구현 해야 합니다.

`using` 문은 개체 소멸을 위한 더 나은 방법을 제공합니다.

## 구조체

**구조체**는 클래스처럼 데이터 멤버 및 함수 멤버를 포함할 수 있는 데이터 구조이지만 값 형식이며 힙 할당이 필요하지 않는 점이 클래스와 다릅니다. 구조체 형식의 변수는 구조체의 데이터를 직접 저장하지만 클래스 형식의 변수는 동적으로 할당된 개체에 대한 참조를 저장합니다. 구조체 형식은 사용자 지정 상속을 지원하지 않으며 모든 구조체 형식은 `object` 형식에서 암시적으로 상속됩니다.

구조체는 값 의미 체계를 갖는 작은 데이터 구조에 특히 유용합니다. 복소수, 좌표계의 점 또는 사전의 키-값 쌍이 모두 구조체의 좋은 예입니다. 작은 데이터 구조에 클래스 대신 구조체를 사용하는 것은 애플리케이션이 사용하는 메모리 할당 수에서 큰 차이를 보일 수 있습니다. 예를 들어 다음 프로그램은 100개의 점 배열을 만들고 초기화합니다. `Point` 가 클래스로 구현될 경우 배열에 대해 1개, 100개 요소에 대해 각각 1개씩 101개의 별도 개체가 인스턴스화됩니다.

```
class Point
{
    public int x, y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

class Test
{
    static void Main() {
        Point[] points = new Point[100];
        for (int i = 0; i < 100; i++) points[i] = new Point(i, i);
    }
}
```

또 다른 방법은 구조체를 `Point` 만드는 것입니다.

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

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

이제 1개의 개체만 인스턴스화되고(배열에 대해 1개) `Point` 인스턴스는 배열에 인라인으로 저장됩니다.

구조체 생성자는 `new` 연산자로 호출되지만 메모리가 할당된다는 것을 의미하지는 않습니다. 개체를 동적으로 할당하고 그에 대한 참조를 반환하는 대신, 구조체 생성자는 단순히 구조체 값 자체(일반적으로 스택의 임시 위치에 있음)를 반환하며 이 값은 필요할 때 복사됩니다.

클래스에서는 두 가지 변수가 같은 개체를 참조할 수 있으므로 한 변수에 대한 작업이 다른 변수에서 참조하는 개체에 영향을 미칠 수 있습니다. 구조체에서는 변수 각각에 데이터의 자체 사본이 들어 있으며 한 변수의 작업이 다른 변수에 영향을 미칠 수 없습니다. 예를 들어, 다음 코드 조각에 의해 생성 된 출력은가 `Point` 클래스 인지 구조체 인지에 따라 달라집니다.

```
Point a = new Point(10, 10);
Point b = a;
a.x = 20;
Console.WriteLine(b.x);
```

가 `Point` 클래스 이면 출력은와 `20` `b` 가 같은 개체 `a` 를 참조 하기 때문입니다. 가 `Point` struct 인 경우는 `a` 에 `10` `b` 대 한 할당이 값의 복사본을 만들지만이 복사본은에 대 `a.x` 한 후속 할당의 영향을 받지 않기 때문입니다.

이전 예제에는 구조체의 제한 사항 중 두 가지를 집중적으로 보여 줍니다. 첫째, 일반적으로 전체 구조체를 복사

하는 것이 개체 참조를 복사하는 것보다 덜 효율적이므로 대입 및 값 매개 변수 전달이 참조 형식보다 덜 경제적일 수 있습니다. 둘째, `ref` 및 `out` 매개 변수를 제외하고, 구조체에 대한 참조를 만들 수 없으므로 다양한 상황에서 사용하는 것이 불가능합니다.

## 배열

**배열**은 계산된 인덱스를 통해 액세스되는 여러 변수를 포함하는 데이터 구조입니다. 배열에 포함된 변수, 즉 배열의 요소라고도 하는 배열은 모두 같은 형식이며, 이 형식을 배열의 **요소 형식**이라고 합니다.

배열 형식은 참조 형식이고 배열 변수의 선언은 배열 인스턴스에 대한 참조를 위한 공간을 설정합니다. 실제 배열 인스턴스는 런타임에 연산자를 `new` 사용하여 동적으로 생성됩니다. 작업 `new`은 새 배열 인스턴스의 길이 `0`를 지정하며, 이는 인스턴스의 수명 동안 고정됩니다. 배열 요소의 인덱스 범위는 `0`에서 `Length - 1` 사이입니다. `new` 연산자는 배열의 요소를 모든 숫자 형식에 대해 0이고, 모든 참조 형식에 대해 `null`인 기본값으로 자동으로 초기화합니다.

다음 예제에서는 `int` 요소의 배열을 만들고, 배열을 초기화하고, 배열의 콘텐츠를 출력합니다.

```
using System;

class Test
{
    static void Main() {
        int[] a = new int[10];
        for (int i = 0; i < a.Length; i++) {
            a[i] = i * i;
        }
        for (int i = 0; i < a.Length; i++) {
            Console.WriteLine("a[{0}] = {1}", i, a[i]);
        }
    }
}
```

이 예제에서는 **1차원 배열**을 만들고 작업을 수행합니다. C#에서는 **다차원 배열s**을 지원하지 않습니다. 배열 형식의 **순위**라고도 하는 배열 형식의 차원 수는 배열 형식의 대괄호 사이에 사용된 쉼표 수에 1을 더한 값입니다. 다음 예제에서는 1 차원, 2 차원 및 3 차원 배열을 할당 합니다.

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

`a1` 배열에는 10개의 요소가 들어 있고 `a2` 배열에는 50( $10 \times 5$ )개의 요소가 들어 있고 `a3` 배열에는 100( $10 \times 5 \times 2$ )개 요소가 들어 있습니다.

배열의 요소 형식은 배열 형식을 비롯한 어떤 형식도 될 수 있습니다. 배열 형식의 요소가 있는 배열을 **가변 배열**이라고도 합니다. 요소 배열의 길이가 항상 동일할 필요는 없기 때문입니다. 다음 예제에서는 `int` 배열의 배열을 할당합니다.

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

첫 번째 줄은 형식이 `int[]`이고 초기 값이 `null`인 3개 요소가 있는 배열을 만듭니다. 다음 줄은 가변 길이의 개별 배열 인스턴스에 대한 참조로 3개 요소를 초기화합니다.

연산자 `new`는 구분 `{ 기호와 }` 사이에 쓰여진 식의 목록 인 **배열 이니셜라이저**를 사용하여 배열 요소의 초기 값을 지정할 수 있도록 허용 합니다. 다음 예제에서는 3개 요소로 `int[]`를 할당하고 초기화합니다.

```
int[] a = new int[] {1, 2, 3};
```

배열의 길이는와 `{ } 사이의 식` 수에서 유추 됩니다. 지역 변수 및 필드 선언은 배열 형식을 다시 시작할 필요가 없도록 좀 더 줄일 수 있습니다.

```
int[] a = {1, 2, 3};
```

앞의 두 예제는 다음 예제와 동일합니다.

```
int[] t = new int[3];
t[0] = 1;
t[1] = 2;
t[2] = 3;
int[] a = t;
```

## 인터페이스

인터페이스는 클래스 및 구조체에서 구현될 수 있는 계약을 정의합니다. 인터페이스는 메서드, 속성, 이벤트 및 인덱서를 포함할 수 있습니다. 인터페이스는 정의하는 멤버의 구현을 제공하지 않으며 단순히 인터페이스를 구현하는 클래스 또는 구조체에서 제공해야 하는 멤버를 지정합니다.

인터페이스는 다중 상속을 사용할 수 있습니다. 다음 예제에서 인터페이스 `IComboBox` 는 `ITextBox` 및 `IListBox` 를 둘 다 상속합니다.

```
interface IControl
{
    void Paint();
}

interface ITextBox: IControl
{
    void SetText(string text);
}

interface IListBox: IControl
{
    void SetItems(string[] items);
}

interface IComboBox: ITextBox, IListBox {}
```

클래스 및 구조체는 여러 인터페이스를 구현할 수 있습니다. 다음 예제에서 클래스 `EditBox` 는 `IControl` 및 `IDataBound` 를 둘 다 구현합니다.

```
interface IDataBound
{
    void Bind(Binder b);
}

public class EditBox: IControl, IDataBound
{
    public void Paint() {...}
    public void Bind(Binder b) {...}
}
```

클래스 또는 구조체가 특정 인터페이스를 구현하는 경우 해당 클래스 또는 구조체의 인스턴스를 해당 인터페이스 형식으로 암시적으로 변환할 수 있습니다. 예

```
EditBox editBox = new EditBox();
IControl control = editBox;
IDataBound dataBound = editBox;
```

인스턴스가 정적으로 특정 인터페이스를 구현하는 것으로 알려지지 않은 경우 동적 형식 캐스팅을 사용할 수 있습니다. 예를 들어 다음 문은 동적 형식 캐스팅을 사용하여 개체의 `IControl` 및 `IDataBound` 인터페이스 구현을 가져옵니다. 개체 `EditBox` 의 실제 형식이 이기 때문에 캐스팅이 성공 합니다.

```
object obj = new EditBox();
IControl control = (IControl)obj;
IDataBound dataBound = (IDataBound)obj;
```

이전 `EditBox`, `IDataBound`, `public` 클래스 `IControl`에서 인터페이스의 `Bind` 메서드와 인터페이스의 메서드는 멤버를 사용하여 구현됩니다. `Paint` C#은 클래스 또는 구조체에서 멤버 `public`를 만들 수 없도록 하는 **명시적 인터페이스 멤버 구현**도 지원합니다. 명시적 인터페이스 멤버 구현은 정규화된 인터페이스 멤버 이름을 사용하여 작성됩니다. 예를 들어 `EditBox` 클래스는 다음과 같이 명시적 인터페이스 멤버 구현을 사용하여 `IControl.Paint` 및 `IDataBound.Bind` 메서드를 구현할 수 있습니다.

```

public class EditBox: IControl, IDataBound
{
    void IControl.Paint() {...}
    void IDataBound.Bind(Binder b) {...}
}

```

명시적 인터페이스 멤버는 인터페이스 형식을 통해서만 액세스할 수 있습니다. `IControl.Paint` 예를 들어 이전 `EditBox` 클래스에서 제공하는 구현은 먼저 `IControl` 인터페이스 형식에 대한 `EditBox` 참조를 변환하는 경우에만 호출할 수 있습니다.

```

EditBox editBox = new EditBox();
editBox.Paint();                                // Error, no such method
IControl control = editBox;
control.Paint();                                 // Ok

```

## 열거형

열거형 형식은 명명된 상수 집합을 갖는 고유 값 형식입니다. 다음 예제 `Color`에서는 세 개의 상수 `Red` 값 `Green`, 및 `Blue`가 포함된 열거형 형식을 선언하고 사용합니다.

```

using System;

enum Color
{
    Red,
    Green,
    Blue
}

class Test
{
    static void PrintColor(Color color) {
        switch (color) {
            case Color.Red:
                Console.WriteLine("Red");
                break;
            case Color.Green:
                Console.WriteLine("Green");
                break;
            case Color.Blue:
                Console.WriteLine("Blue");
                break;
            default:
                Console.WriteLine("Unknown color");
                break;
        }
    }

    static void Main() {
        Color c = Color.Red;
        PrintColor(c);
        PrintColor(Color.Blue);
    }
}

```

각 열거형 형식에는 열거형 형식의 기본 형식이라고 하는 해당 정수 형식이 있습니다. 내부 형식을 명시적으로 선언하지 않는 열거형 형식에는의 `int` 기본 형식이 있습니다. 열거형 형식의 저장소 형식 및 가능한 값의 범위는 해당 기본 형식에 의해 결정됩니다. 열거형 형식이 사용할 수 있는 값 집합은 열거형 멤버로 제한되지 않습니다. 특히 열거형의 기본 형식 값은 열거형 형식으로 캐스팅 될 수 있으며 해당 열거형 형식의 유효한 고유 값입니다.

다음 예제에서는의 `Alignment` 기본 형식을 사용하여 라는 열거형 형식을 선언 합니다.

```

enum Alignment: sbyte
{
    Left = -1,
    Center = 0,
    Right = 1
}

```

앞의 예제와 같이 열거형 멤버 선언에는 멤버의 값을 지정하는 상수 식이 포함될 수 있습니다. 각 열거형 멤버에 대한 상수 값은 열거형의 기본 형식 범위에 속해야 합니다. 열거형 멤버 선언에서 값을 명시적으로 지정하

지 않으면 멤버에 값 0 (열거형 형식의 첫 번째 멤버인 경우) 또는 열거형 멤버 앞에 있는 형식 (열거형)의 값이 지정 됩니다.

형식 캐스팅을 사용 하여 열거형 값을 정수 값으로 변환하거나 그 반대로 변환할 수 있습니다. 예

```
int i = (int)Color.Blue;           // int i = 2;
Color c = (Color)2;                // Color c = Color.Blue;
```

열거형 형식의 기본값은 열거형 형식으로 변환된 정수 값 0입니다. 변수가 기본값으로 자동 초기화 되는 경우 열거형 형식의 변수에 지정 된 값입니다. 열거형 형식의 기본값을 쉽게 사용할 수 있도록 리터럴 `0` 은 암시적으로 열거형 형식으로 변환 합니다. 따라서 다음이 허용됩니다.

```
Color c = 0;
```

## 대리자

대리자는 특정 매개 변수 목록 및 반환 형식이 있는 메서드에 대한 참조를 나타내는 형식입니다. 대리자는 메서드를 변수에 할당되고 매개 변수로 전달될 수 있는 엔터티로 취급할 수 있도록 합니다. 또한 대리자는 다른 언어에 나오는 함수 포인터의 개념과 비슷하지만 함수 포인터와 달리 대리자는 개체 지향적이며 형식 안전 방식입니다.

다음 예제에서는 `Function` 라는 대리자 형식을 선언하고 사용합니다.

```
using System;

delegate double Function(double x);

class Multiplier
{
    double factor;

    public Multiplier(double factor) {
        this.factor = factor;
    }

    public double Multiply(double x) {
        return x * factor;
    }
}

class Test
{
    static double Square(double x) {
        return x * x;
    }

    static double[] Apply(double[] a, Function f) {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }

    static void Main() {
        double[] a = {0.0, 0.5, 1.0};
        double[] squares = Apply(a, Square);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new Multiplier(2.0);
        double[] doubles = Apply(a, m.Multiply);
    }
}
```

`Function` 대리자 형식의 인스턴스는 `double` 인수를 사용하고 `double` 값을 반환하는 메서드를 참조할 수 있습니다. 메서드 `Apply` 는 지정 `Function` 된 블록의 `double[]` 오른쪽에 적용 하여 결과와 `double[]` 함께를 반환 합니다. `Main` 메서드에서 `Apply` 는 세 가지 다른 함수를 `double[]`에 적용하는 데 사용됩니다.

대리자는 정적 메서드(예: 이전 예제의 `Square` 또는 `Math.Sin`) 또는 인스턴스 메서드(예: 이전 예제의 `m.Multiply`)를 참조할 수 있습니다. 인스턴스 메서드를 참조하는 대리자는 특정 개체도 참조하며, 인스턴스 메서드가 대리자를 통해 호출되는 경우 해당 개체도 이 호출에서 `this` 가 됩니다.

또한 즉석에서 만들어지는 "인라인 메서드"인 익명 함수를 사용하여 대리자를 만들 수도 있습니다. 익명 함수는 주변 메서드의 지역 변수를 볼 수 있습니다. 따라서 위의 승수 예제는 클래스를 `Multiplier` 사용 하지 않고 더 쉽게 작성할 수 있습니다.

```
double[] doubles = Apply(a, (double x) => x * 2.0);
```

대리자의 흥미롭고 유용한 속성은 참조하는 메서드의 클래스를 알지 못하거나 관심을 두지 않는다는 것입니다. 참조되는 메서드가 대리자와 동일한 매개 변수 및 반환 형식을 갖는다는 것만 중요합니다.

## 특성

C# 프로그램의 형식, 멤버 및 기타 엔터티는 동작의 특정 측면을 제어하는 한정자를 지원합니다. 예를 들어 메서드의 액세스 가능성은 `public`, `protected`, `internal` 및 `private` 한정자를 사용하여 제어됩니다. C#은 선언적 정보의 사용자 정의 형식을 프로그램 엔터티에 연결하고 런타임에 검색할 수 있도록 이러한 기능을 일반화합니다. 프로그램은 특성을 정의하고 사용하여 이러한 추가적인 선언적 정보를 지정합니다.

다음 예제에서는 관련 설명서에 대한 링크를 제공하기 위해 프로그램 엔터티에 배치될 수 있는 `HelpAttribute` 특성을 선언합니다.

```
using System;

public class HelpAttribute: Attribute
{
    string url;
    string topic;

    public HelpAttribute(string url) {
        this.url = url;
    }

    public string Url {
        get { return url; }
    }

    public string Topic {
        get { return topic; }
        set { topic = value; }
    }
}
```

모든 특성 클래스는 .NET Framework에서 `System.Attribute` 제공하는 기본 클래스에서 파생 됩니다. 연결된 선언 바로 앞에 대괄호로 묶은 특성 이름을 인수와 함께 적용할 수 있습니다. 특성 이름이에서 `Attribute` 종료되는 경우 특성을 참조 하는 경우 이름의 해당 부분을 생략할 수 있습니다. 예를 들어 `HelpAttribute` 특성을 다음과 같이 사용할 수 있습니다.

```
[Help("http://msdn.microsoft.com/.../MyClass.htm")]
public class Widget
{
    [Help("http://msdn.microsoft.com/.../MyClass.htm", Topic = "Display")]
    public void Display(string text) {}
}
```

이 예제에서는 `HelpAttribute` `Widget` 클래스에 `HelpAttribute`, 다른 클래스의 메서드에 연결합니다.

`Display` 특성 클래스의 공용 생성자는 프로그램 엔터티에 특성을 추가할 때 제공해야 하는 정보를 제어합니다. 특성 클래스의 공용 읽기/쓰기 속성을 참조하여 추가 정보를 제공할 수 있습니다(예: 앞에 나온 `Topic` 속성 참조).

다음 예제에서는 리플렉션을 사용 하 여 런타임에 지정 된 프로그램 엔터티에 대 한 특성 정보를 검색 하는 방법을 보여 줍니다.

```
using System;
using System.Reflection;

class Test
{
    static void ShowHelp(MemberInfo member) {
        HelpAttribute a = Attribute.GetCustomAttribute(member,
            typeof(HelpAttribute)) as HelpAttribute;
        if (a == null) {
            Console.WriteLine("No help for {0}", member);
        }
        else {
            Console.WriteLine("Help for {0}:", member);
            Console.WriteLine(" Url={0}, Topic={1}", a.Url, a.Topic);
        }
    }

    static void Main() {
        ShowHelp(typeof(Widget));
        ShowHelp(typeof(Widget).GetMethod("Display"));
    }
}
```

리플렉션을 통해 특정 속성이 요청되면 속성 클래스에 대한 생성자가 프로그램 소스에 제공된 정보와 함께 호출되고 결과 속성 인스턴스가 반환됩니다. 속성을 통해 추가 정보를 제공한 경우 해당 속성은 속성 인스턴스가 반환되기 전에 지정된 값으로 설정됩니다.

# C # 7의 패턴 일치

2020-11-02 • 40 minutes to read • [Edit Online](#)

C #에 대한 패턴 일치 확장은 기능 언어에서의 대수 데이터 형식 및 패턴 일치의 많은 이점을 제공 하지만, 기본 언어의 느낌을 원활하게 통합하는 방법입니다. 기본 기능은 데이터의 모양에 의미 체계를 설명하는 형식인 [레코드 형식](#)입니다. 및 패턴 일치는 이러한 데이터 형식에 대해 매우 간결한 단계 분해를 가능하게 해주는 새로운 형식입니다. 이 방법의 요소는 [F #](#) 및 [Scala](#) 프로그래밍 언어의 관련 기능을 통해 아이디어를 제공합니다.

## Is 식

`is` 연산자는 [패턴](#)에 대해 식을 테스트하기 위해 확장됩니다.

```
relational_expression
  : relational_expression 'is' pattern
  ;
```

이 형식의 `relational_expression`는 C# 사양의 기존 품에 추가됩니다. 토큰의 왼쪽에 있는 `relational_expression` `is` 값을 지정하지 않거나 형식이 없는 경우 컴파일 시간 오류가 발생합니다.

패턴의 모든 식별자에는 연산자가 *definitely assigned* `is` `true` (즉, `true` 일 때 명확하게 할당됨) 이후에는 명확하게 할당된 새 지역 변수가 도입되었습니다.

참고: `constant_pattern` 및의 형식 사이에는 기술적으로 모호성이 있으며, 이 중 `is-expression` 하나는 정규화된 식별자의 유효한 구문 분석 일 수 있습니다. `constant_pattern` 이전 버전의 언어와의 호환성을 위해 형식으로 바인딩하려고 합니다. 이 작업이 실패하는 경우에만 다른 컨텍스트에서 수행하는 것과 같은 방법으로 문제를 해결합니다 (상수 또는 형식 이어야 함). 이러한 모호성은 식의 오른쪽에만 표시 됩니다 `is`.

## 패턴

패턴은 `is` 연산자와 `switch_statement`에서 사용되어 들어오는 데이터를 비교할 데이터의 모양을 표현합니다. 패턴은 재귀적으로 수행되어 데이터의 일부가 하위 패턴과 일치하도록 할 수 있습니다.

```
pattern
  : declaration_pattern
  | constant_pattern
  | var_pattern
  ;

declaration_pattern
  : type simple_designation
  ;

constant_pattern
  : shift_expression
  ;

var_pattern
  : 'var' simple_designation
  ;
```

참고: `constant_pattern` 및의 형식 사이에는 기술적으로 모호성이 있으며, 이 중 `is-expression` 하나는 정규

화 된 식별자의 유효한 구문 분석 일 수 있습니다. *constant\_pattern* 이전 버전의 언어와의 호환성을 위해 형식으로 바인딩 하려고 합니다. 이 작업이 실패 하는 경우에만 다른 컨텍스트에서 수행 하는 것과 같은 방법으로 문제를 해결 합니다 (상수 또는 형식 이어야 함). 이러한 모호성은 식의 오른쪽에만 표시 됩니다 `is` .

## 선언 패턴

*Declaration\_pattern* 는 식이 지정 된 형식 인지 테스트 하고 테스트가 성공 하면 해당 형식으로 캐스팅 합니다. *Simple\_designation* 식별자 인 경우 지정 된 식별자에 의해 명명 된 지정 된 형식의 지역 변수를 도입 합니다. 패턴 일치 작업의 결과가 `true` 인 경우 해당 지역 변수는 명확하게 할당됩니다.

```
declaration_pattern
  : type simple_designation
  ;
```

이 식의 런타임 의미 체계는 패턴의 형식에 대해 왼쪽 *relational\_expression* 피연산자의 런타임 형식을 테스트 하는 것입니다. 해당 런타임 형식 (또는 일부 하위 형식)의 경우의 결과는 `is operator true` 입니다. 결과가 인 경우 왼쪽 피연산자의 값이 할당 된 식별자에 의해 이름이 지정 된 새 지역 변수를 선언 합니다 `true` .

왼쪽 및 지정 된 형식에 대 한 정적 형식의 특정 조합은 호환 되지 않는 것으로 간주 되어 컴파일 타임 오류가 발생 합니다. 정적 형식의 값 `E` 은 *pattern compatible* `T` id 변환, 암시적 참조 변환, boxing 변환, 명시적 참조 변환 또는에서로의 unboxing 변환이 있는 경우 형식과 호환 되는 패턴 이라고 합니다 `E T` . 형식의 식이 `E` 패턴과 일치 하는 형식 패턴의 형식과 호환 되지 않는 경우 컴파일 시간 오류가 발생 합니다.

참고: [c# 7.1에서는](#) 입력 형식 또는 형식이 개방형 형식이 면 패턴 일치 작업을 허용하도록 이를 확장 `T` 합니다. 이 단락은 다음으로 대체 됩니다.

왼쪽 및 지정 된 형식에 대 한 정적 형식의 특정 조합은 호환 되지 않는 것으로 간주 되어 컴파일 타임 오류가 발생 합니다. 정적 형식의 값 `E` 은 *pattern compatible* `T` id 변환, 암시적 참조 변환, boxing 변환, 명시적 참조 변환 또는에서로의 unboxing 변환이 있거나 `E T ** E` 또는 `T` 가 개방형 형식인 경우 형식과 호환 되는 패턴 이라고 합니다. 형식의 식이 `E` 패턴과 일치 하는 형식 패턴의 형식과 호환 되지 않는 경우 컴파일 시간 오류가 발생 합니다.

선언 패턴은 참조 형식의 런타임 형식 테스트를 수행 하 고이를 대체 하는 데 유용 합니다.

```
var v = expr as Type;
if (v != null) { // code using v }
```

약간 더 간결한

```
if (expr is Type v) { // code using v }
```

형식이 nullable 값 형식인 경우 오류가 발생 합니다.

선언 패턴은 nullable 형식의 값을 테스트 하는 데 사용할 수 있습니다. `Nullable<T> T` `T2 id` 값이 null이 아니고의 형식이 이거나 `T2 T` 의 기본 형식 또는 인터페이스인 경우 형식 (또는 boxed)의 값이 형식 패턴과 일치 합니다. 예를 들어 코드 조각에서

```
int? x = 3;
if (x is int v) { // code using v }
```

명령문의 조건은 `if true` 런타임에 이며 변수에는 `v 3` 블록 내의 형식 값이 포함 됩니다 `int` .

## 상수 패턴

```
constant_pattern
: shift_expression
;
```

상수 패턴은 상수 값에 대해 식의 값을 테스트 합니다. 상수는 리터럴, 선언 된 `const` 변수 이름, 열거형 상수 또는 식과 같은 임의의 상수 식일 수 있습니다 `typeof` .

`e`와 `c`가 모두 정수 계열 형식이면 식의 결과가 인 경우 패턴이 일치 하는 것으로 간주 `e == c` 됩니다 `true` .

그렇지 않으면 패턴이 반환할 때 일치 하는 것으로 간주 됩니다 `object.Equals(e, c)` `true` . 이 경우 `e`의 정적 형식이 상수 형식과 호환되지 않는 경우 컴파일 시간 오류가 발생 합니다.

### Var 패턴

```
var_pattern
: 'var' simple_designation
;
```

식 `e`는 항상 `var_pattern` 와 일치 합니다. 즉, `var` 패턴에 대한 일치는 항상 성공 합니다. *Simple designation* 식 별자인 경우 런타임에 `e`값은 새로 도입 된 지역 변수에 바인딩됩니다. 지역 변수의 형식은 `e`의 정적 형식입니다.

이름이 형식에 바인딩되는 경우 오류가 발생 `var` 합니다.

## Switch 문

`switch` 문을 확장 하여 스위치/ 식과 일치 하는 연결된 패턴을 가진 첫 번째 블록을 실행 하도록 선택 합니다.

```
switch_label
: 'case' complex_pattern case_guard? ':'
| 'case' constant_expression case_guard? ':'
| 'default' ':'
;

case_guard
: 'when' expression
;
```

패턴이 일치 하는 순서는 정의 되지 않습니다. 컴파일러는 순서가 잘못 된 패턴을 일치 시킬 수 있으며 이미 일치 한 패턴의 결과를 다시 사용 하여 다른 패턴과 일치 하는 결과를 계산할 수 있습니다.

대/소문자 보호가 있는 경우 해당 식은 형식입니다 `bool` . 이는 사례가 총족 되는 것으로 간주 되려면 총족 해야 하는 추가 조건으로 평가 됩니다.

`Switch_label` 가 이전 경우에는 해당 패턴이 스페이스가 때문에 런타임에 영향을 주지 않을 수 있는 경우 오류가 발생 합니다. [TODO: 이러한 판단에 도달 하려면 컴파일러가 사용 해야 하는 기술에 대해 보다 정확하게 알아야 합니다.]

Case 블록에 정확히 하나의 `switch_label` 포함 된 경우에만 `switch_label`에 선언 된 패턴 변수가 `case` 블록에 할당 됩니다.

[TODO: 스위치 블록에 연결할 수 있는 경우를 지정 해야 합니다.]

### 패턴 변수의 범위

패턴에 선언 된 변수의 범위는 다음과 같습니다.

- 패턴이 `case` 레이블이면 변수의 범위가 `case` 블록입니다.

그렇지 않으면 변수가 *is\_pattern* 식에서 선언 되고 해당 범위는 다음과 같이 *is\_pattern* 식이 포함 된 식을 바로 루는 구문을 기반으로 합니다.

- 식이 식 본문 람다에 있는 경우 해당 범위는 람다의 본문입니다.
- 식이 식 본문 메서드나 속성에 있으면 해당 범위는 메서드나 속성의 본문입니다.
- 식이 `when` 절의 절에 있으면 `catch` 해당 범위는 해당 `catch` 절입니다.
- 식이 *iteration\_statement*에 있는 경우 해당 문만 해당 됩니다.
- 그렇지 않고 식이 다른 문 형식인 경우 해당 범위는 문을 포함 하는 범위입니다.

범위를 결정 하기 위해 *embedded\_statement*는 자체 범위에 있는 것으로 간주 됩니다. 예를 들어 *if\_statement*에 대 한 문법은

```
if_statement
: 'if' '(' boolean_expression ')' embedded_statement
| 'if' '(' boolean_expression ')' embedded_statement 'else' embedded_statement
;
```

따라서 *if\_statement*의 제어 된 문이 패턴 변수를 선언 하는 경우 해당 범위는 해당 *embedded\_statement*로 제한 됩니다.

```
if (x) M(y is var z);
```

이 경우의 범위는 `z` 포함 문입니다 `M(y is var z)`.

다른 경우에는 다른 이유로 오류가 발생 합니다. 예를 들어 매개 변수의 기본값 또는 특성에서 이러한 컨텍스트는 상수 식이 필요 하기 때문입니다.

C # 7.3에서는 패턴 변수를 선언할 수 있는 다음 컨텍스트를 추가 했습니다.

- 식이 생성자 이니셜라이저에 있으면 해당 범위는 생성자 이니셜라이저와 생성자의 본문입니다.
- 식이 필드 이니셜라이저에 있으면 해당 범위는 표시 되는 *equals\_value\_clause*입니다.
- 식이 람다 본문으로 변환 되도록 지정 된 쿼리 절에 있는 경우 해당 식의 범위는 해당 식 뿐입니다.

## 구문 명확성의 변경 내용

C # 문법이 모호 하고 언어 사양에 이러한 모호성을 해결 하는 방법이 포함 된 제네릭을 포함 하는 경우가 있습니다.

### 7.6.5.2 문법 모호성

단순 *이름*(§ 7.6.3) 및 *멤버 액세스*(§ 7.6.5)에 대 한 생성은 식의 문법에서 모호성을 증가 시킬 수 있습니다. 예를 들어 문은 다음과 같습니다.

```
F(G<A,B>(7));
```

는 `F` 두 개의 인수 및를 사용 하 여에 대 한 호출로 해석 될 수 있습니다 `G < A B > (7)`. 또는 하나의 인수를 사용 하 여에 대 한 호출로 해석 될 수 있습니다 `F`. 이는 `G` 두 개의 형식 인수와 하나의 일반 인수가 있는 제네릭 메서드를 호출 하는 것입니다.

토큰 시퀀스를 단순 *이름*(§ 7.6.3), *멤버 액세스*(§ 7.6.5) 또는 형식-인수 목록(§ 4.4.1)으로 끝나는(§ 18.5.2) 형식으로 구문 분석 할 수 있는 경우 닫는 토큰 바로 다음에 오는 토큰이 `>` 검사 됩니다. 다음 중 하나인 경우

```
( ) ] } : ; , . ? == != | ^
```

그런 다음, 형식 인수 목록은 단순한 이름, 멤버 액세스 또는 포인터 멤버 액세스의 일부로 유지 되 고 토큰 시퀀스의 다른 모든 구문 분석은 무시 됩니다. 그렇지 않은 경우에는 토큰 시퀀스에 대 한 다른 가능한 구문 분석이 없더라도 형식 인수 목록이 단순한 이름, 멤버 액세스 또는 > 포인터 멤버 액세스의 일부로 간주 되 지 않습니다. 네임 스페이스 또는 형식 이름에서 형식 인수 목록을 구문 분석 할 때는 이러한 규칙이 적용 되 지 않습니다 (§ 3.8). 다음 문은

```
F(G<A,B>(7));
```

는이 규칙에 따라 하나의 인수를 사용 하 여에 대 한 호출로 해석 됩니다. 이 인수 F 는 G 두 개의 형식 인수와 하나의 일반 인수가 있는 제네릭 메서드를 호출 하는 것입니다. 문

```
F(G < A, B > 7);  
F(G < A, B >> 7);
```

각는 두 인수를 사용 하 여에 대 한 호출로 해석 됩니다 F . 다음 문은

```
x = F < A > +y;
```

는 보다 작은 연산자, 보다 큼 연산자, 단항 더하기 연산자로 해석 됩니다 x = (F < A) > (+y) . 즉, 형식 인수 목록을 포함 하는 단순한 이름 대신 이항 더하기 연산자가 오는 것 처럼 연산자가 작성 됩니다. 문에서

```
x = y is C<T> + z;
```

토큰은 C<T> 형식 인수 목록을 사용 하 여 네임 스페이스 또는 형식 이름으로 해석 됩니다.

C # 7에 도입 된 다양한 변경 내용으로 인해 이러한 명확성 규칙을 사용 하 여 언어의 복잡성을 더 이상 처리 할 수 없습니다.

#### 출력 변수 선언

이제 out 인수에서 변수를 선언 할 수 있습니다.

```
M(out Type name);
```

그러나 형식은 제네릭일 수 있습니다.

```
M(out A<B> name);
```

인수에 대 한 언어 문법에 식이 사용 되므로 컨텍스트는 명확성 규칙의 영향을 받습니다. 이 경우 닫는 뒤에는 > 형식 인수 목록으로 처리 될 수 있도록 하는 토큰 중 하나가 아닌 식별자가 옵니다. 따라서 \*\* 형식 인수 목록에 명확성을 트리거하는 토큰 집합에 식별자를 추가\*\* 하는 것을 제안 합니다.

#### 튜플 및 분해 선언

튜플 리터럴은 정확히 동일한 문제에 실행 됩니다. 튜플 식 고려

```
(A < B, C > D, E < F, G > H)
```

이전 c # 6 규칙에서 인수 목록을 구문 분석 하는 경우 첫 번째로 시작 하는 네 개의 요소가 포함 된 튜플로 구문

분석 됩니다 `A < B`. 그러나 분해의 왼쪽에 표시 되는 경우 위에 설명 된 대로 식별자 토큰에 의해 트리거되는 명확성을 원합니다.

```
(A<B,C> D, E<F,G> H) = e;
```

이는 두 변수를 선언 하는 분해 선언입니다. 첫 번째 변수는 형식이 `A<B,C>` 고 명명 된 변수입니다 `D`. 즉, 튜플 리터럴은 각각 선언 식인 두 개의 식을 포함 합니다.

사양과 컴파일러의 편의를 위해 이 튜플 리터럴은 표시 될 때마다 두 요소 튜플로 구문 분석 되는 것으로 제안 합니다 (할당의 왼쪽에 표시 되는지 여부에 관계 없음). 이는 이전 섹션에서 설명 하는 명확성의 자연 스러운 결과입니다.

#### 패턴 일치

패턴 일치는 식 형식 모호성이 발생 하는 새 컨텍스트를 도입 합니다. 이전에 연산자의 오른쪽은 `is` 형식 이었습니다. 이제 형식이 나 식일 수 있으며 형식이 면 식별자가 뒤에 올 수 있습니다. 이는 기술적으로 기존 코드의 의미를 변경 할 수 있습니다.

```
var x = e is T < A > B;
```

이는 c# 6 규칙에 따라 구문 분석 될 수 있습니다.

```
var x = ((e is T) < A) > B;
```

그러나 c# 7 규칙 (위에서 명확성을 제안 함) 아래에는 다음과 같이 구문 분석 됩니다.

```
var x = e is T<A> B;
```

형식의 변수를 선언 하는입니다 `B T<A>`. 다행히 네이티브 및 Roslyn 컴파일러에는 c# 6 코드에서 구문 오류를 제공 하는 버그가 있습니다. 따라서 이 특정 주요 변경 사항은 중요 하지 않습니다.

패턴 일치에는 형식을 선택 하기 위해 모호성 해결을 유도 하는 추가 토큰이 도입 되었습니다. 다음의 기존 유효한 c# 6 코드 예제는 추가 명확성 규칙 없이 중단 됩니다.

```
var x = e is A<B> && f;           // &&
var x = e is A<B> || f;            // ||
var x = e is A<B> & f;            // &
var x = e is A<B>[];             // [
```

#### 명확성 규칙의 제안 된 변경 내용

명확히 구분 토큰 목록을 변경 하는 사양을 수정 하는 것을 제안 합니다.

```
( ) ] } : ; , . ? == != | ^
```

을

```
( ) ] } : ; , . ? == != | ^ && || & [
```

그리고 특정 컨텍스트에서는 식별자를 명확히 구분 토큰으로 처리 합니다. 이러한 컨텍스트는 명확 하 게 하는 토큰 시퀀스가 바로 앞에 와야 합니다. 즉 `is`, `case` `out` 튜플 리터럴의 첫 번째 요소를 구문 분석 하는 동안

(토큰 앞에 `(` 또는 `:` 이 오고 식별자 뒤에 오는 경우 `,`) 또는 튜플 리터럴의 후속 요소가 됩니다.

수정 된 명확성 규칙

수정 된 명확성 규칙은 다음과 같습니다.

토큰 시퀀스를 단순 이름(§ 7.6.3), 멤버 액세스(§ 7.6.5) 또는 (§ 18.5.2) 형식으로 끝 (\$)으로 구문 분석할 수 있는 경우, 닫는 토큰 바로 다음에 오는 토큰을 검사 하여 다음을 확인 합니다. >

- 중 하나 `( ) ] } : ; , . ? == != | ^ && || & [`, 또는
- 관계형 연산자 중 하나 `< > <= >= is as` 입니다.
- 쿼리 식 내에 나타나는 상황별 쿼리 키워드입니다. 디스크나
- 특정 컨텍스트에서는 식별자를 명확히 구분 토큰으로 처리 합니다. 이러한 컨텍스트는 명확하게 구분 되는 토큰 시퀀스가 바로 앞에와야 합니다 `is`. 즉 `case` `out`, 튜플 리터럴의 첫 번째 요소를 구문 분석 하는 동안, 또는이 발생 합니다. 이 경우 토큰 앞에는 `(` 또는 `:` 이 오고 식별자 뒤에는 `,` 튜플 리터럴의 후속 요소가 있습니다.

다음 토큰이이 목록 또는 해당 컨텍스트의 식별자 중 하나인 경우 형식 인수 목록은 단순한 이름, 멤버 액세스 또는 포인터 멤버 액세스의 일부로 유지 되고 토큰 시퀀스의 다른 모든 구문 분석은 무시 됩니다. 그렇지 않으면 토큰 시퀀스에 대한 다른 가능한 구문 분석이 없더라도 형식 인수 목록이 단순한 이름, 멤버 액세스 또는 포인터 멤버 액세스의 일부로 간주 되지 않습니다. 네임 스페이스 또는 형식 이름에서 형식 인수 목록을 구문 분석 할 때는 이러한 규칙이 적용 되지 않습니다 (§ 3.8).

이 제안으로 인한 주요 변경 내용

이 제안 된 명확성 규칙으로 인해 주요 변경 내용이 알려지지 않습니다.

흥미로운 예

이러한 명확성 규칙의 몇 가지 흥미로운 결과는 다음과 같습니다.

식은 `(A < B, C > D)` 각각 비교 하는 두 개의 요소가 있는 튜플입니다.

식은 `(A<B,C> D, E)` 두 개의 요소가 포함 된 튜플입니다. 첫 번째 요소는 선언 식입니다.

호출에는 `M(A < B, C > D, E)` 세 개의 인수가 있습니다.

호출에는 `M(out A<B,C> D, E)` 두 개의 인수가 있으며, 이 중 첫 번째는 `out` 선언입니다.

식에서 `e is A<B> C` 선언 식을 사용 합니다.

Case 레이블은 `case A<B> c:` 선언 식을 사용 합니다.

## 패턴 일치의 몇 가지 예

다음으로

이 방법을 바꿀 수 있습니다.

```
var v = expr as Type;
if (v != null) {
    // code using v
}
```

약간 더 간결 하고 직접적인

```
if (expr is Type v) {
    // code using v
}
```

## Nullable 테스트

이 방법을 바꿀 수 있습니다.

```
Type? v = x?.y?.z;
if (v.HasValue) {
    var value = v.GetValueOrDefault();
    // code using value
}
```

약간 더 간결 하고 직접적인

```
if (x?.y?.z is Type value) {
    // code using value
}
```

산술 간소화

별도의 제안에 따라 식을 나타내는 재귀 형식 집합을 정의 있다고 가정 합니다.

```
abstract class Expr;
class X() : Expr;
class Const(double Value) : Expr;
class Add(Expr Left, Expr Right) : Expr;
class Mult(Expr Left, Expr Right) : Expr;
class Neg(Expr Value) : Expr;
```

이제 식의 (축소 되지 않은)를 계산 하는 함수를 정의할 수 있습니다.

```
Expr Deriv(Expr e)
{
    switch (e) {
        case X(): return Const(1);
        case Const(*): return Const(0);
        case Add(var Left, var Right):
            return Add(Deriv(Left), Deriv(Right));
        case Mult(var Left, var Right):
            return Add(Mult(Deriv(Left), Right), Mult(Left, Deriv(Right)));
        case Neg(var Value):
            return Neg(Deriv(Value));
    }
}
```

식 simplifier는 위치 패턴을 보여 줍니다.

```
Expr Simplify(Expr e)
{
    switch (e) {
        case Mult(Const(0), *): return Const(0);
        case Mult(*, Const(0)): return Const(0);
        case Mult(Const(1), var x): return Simplify(x);
        case Mult(var x, Const(1)): return Simplify(x);
        case Mult(Const(var l), Const(var r)): return Const(l*r);
        case Add(Const(0), var x): return Simplify(x);
        case Add(var x, Const(0)): return Simplify(x);
        case Add(Const(var l), Const(var r)): return Const(l+r);
        case Neg(Const(var k)): return Const(-k);
        default: return e;
    }
}
```

# C# 연습

2020-11-02 • 6 minutes to read • [Edit Online](#)

연습에서는 일반 시나리오에 대한 단계별 지침을 제공하므로 제품 또는 특정 기능 영역 파악을 효율적으로 시작할 수 있습니다.

이 섹션에는 C# 프로그래밍 연습에 대한 링크가 포함되어 있습니다.

## 단원 내용

- [완료되면 비동기 작업 처리](#)

`async` 및 `await`를 사용하여 비동기 솔루션을 만드는 방법을 보여 줍니다.

- [C# 또는 Visual Basic에서 Windows 런타임 구성 요소를 만든 다음 JavaScript에서 호출](#)

Windows 런타임 형식을 만들고 Windows 런타임 구성 요소에 패키지한 다음, JavaScript를 사용하여 Windows용으로 빌드된 Windows 8.x 스토어 앱에서 해당 구성 요소를 호출하는 방법을 보여 줍니다.

- [Office 프로그래밍\(C# 및 Visual Basic\)](#)

C# 및 Visual Basic을 사용하여 Excel 통합 문서와 Word 문서를 만드는 방법을 보여 줍니다.

- [동적 개체 만들기 및 사용\(C# 및 Visual Basic\)](#)

텍스트 파일의 내용을 동적으로 노출하는 사용자 지정 개체를 만들고 `IronPython` 라이브러리를 사용하는 프로젝트를 만드는 방법을 보여 줍니다.

- [Visual C#을 사용하여 복합 컨트롤 작성](#)

간단한 복합 컨트롤을 만들고 상속을 통해 컨트롤 기능을 확장하는 방법을 보여 줍니다.

- [Visual Studio의 디자인 타임 기능을 활용하는 Windows Forms 컨트롤 만들기](#)

사용자 지정 컨트롤을 사용자 지정 디자이너를 만드는 방법을 보여 줍니다.

- [Visual C#을 사용하여 Windows Forms 컨트롤에서 상속](#)

상속 가능한 간단한 단추 컨트롤을 만드는 방법을 보여 줍니다. 이 단추는 표준 Windows Forms 단추에서 기능을 상속하며 사용자 지정 멤버를 노출합니다.

- [디자인 타임에 사용자 지정 Windows Forms 컨트롤 디버깅](#)

사용자 지정 컨트롤의 디자인 타임 동작을 디버그하는 방법을 설명합니다.

- [연습: 디자이너 작업을 사용하여 일반 작업 수행](#)

`TabControl`에서 탭 추가/제거, 부모 항목에 컨트롤 도킹, `SplitContainer` 컨트롤 방향 변경 등 일반적으로 수행하는 몇 가지 작업을 보여 줍니다.

- [C#에서 쿼리 작성\(LINQ\)](#)

LINQ 쿼리 식을 작성하는 데 사용되는 C# 언어 기능을 보여 줍니다.

- [데이터 조작\(C#\)\(LINQ to SQL\)](#)

데이터베이스에서 데이터를 추가/수정/삭제하는 LINQ to SQL 시나리오에 대해 설명합니다.

- [간단한 개체 모델 및 쿼리\(C#\)\(LINQ to SQL\)](#)

엔티티 클래스 및 해당 엔티티 클래스를 필터링하는 간단한 쿼리를 만드는 방법을 보여 줍니다.

- [저장 프로시저만 사용\(C#\)\(LINQ to SQL\)](#)

LINQ to SQL을 사용해 저장 프로시저만 실행하여 데이터에 액세스하는 방법을 보여 줍니다.

- [관계 간 쿼리\(C#\)\(LINQ to SQL\)](#)

LINQ to SQL 연결을 사용하여 데이터베이스에서 외래 키 관계를 표시하는 방법을 보여 줍니다.

- [C#에서 시각화 도우미 작성](#)

C#을 사용하여 간단한 시각화 도우미를 작성하는 방법을 보여 줍니다.

## 관련 단원

- [배포 샘플 및 연습](#)

일반적인 배포 시나리오의 단계별 예제를 제공합니다.

## 참고 항목

- [C# 프로그래밍 가이드](#)

- [Visual Studio 샘플](#)