# 일반적인 C# 코드 규칙

기사 • 08/01/2023

코드 표준은 개발 팀 내에서 코드 가독성, 일관성 및 협업을 유지하는 데 필수적입니다. 업계 관행과 확립된 지침을 따르는 코드는 더 쉽게 이해하고, 유지 관리하고, 확장할 수 있습니다. 대부분의 프로젝트는 코드 규칙을 통해 일관된 스타일을 적용합니다. dotnet/docs 및 dotnet/samples 프로젝트도 예외는 아닙니다. 이 문서 시리즈에서는 코딩 규칙과 이를 적용하는 데 사용하는 도구에 대해 알아봅니다. 규칙을 있는 그대로 사용하거나 팀의 필요에 맞게 수정할 수 있습니다.

우리는 다음과 같은 목표에 따라 대회를 선택했습니다.

1. *정확성*: 샘플이 복사되어 응용 프로그램에 붙여넣어집니다. 우리는 그것을 기대하므로 여러 번 편집한 후에도 복원력 있고 올바른 코드를 만들어야 합니다.
2. *교육*: 샘플의 목적은 모든 .NET 및 C#을 교육하는 것입니다. 이러한 이유로 언어 기능이나 API에 제한을 두지 않습니다. 대신, 이러한 샘플은 기능이 좋은 선택일 때를 알려줍니다.
3. *일관성*: 독자는 콘텐츠 전반에 걸쳐 일관된 경험을 기대합니다. 모든 샘플은 동일한 스타일을 준수해야 합니다.
4. *채택*: 새로운 언어 기능을 사용하기 위해 샘플을 적극적으로 업데이트합니다. 이러한 방법을 통해 새로운 기능에 대한 인식이 높아지고 모든 C# 개발자가 더 친숙하게 사용할 수 있습니다.

> **ⓘ 중요하다**
>
> 이러한 지침은 Microsoft에서 샘플 및 설명서를 개발하는 데 사용됩니다. **.NET 런타임, C# 코딩 스타일** 및 **C# 컴파일러(roslyn)** 지침에서 채택되었습니다. 이러한 가이드라인을 선택한 이유는 수년간의 오픈소스 개발 과정을 거쳤기 때문입니다. 그들은 커뮤니티 구성원이 런타임 및 컴파일러 프로젝트에 참여할 수 있도록 도왔습니다. 이는 일반적인 C# 규칙의 예이며 신뢰할 수 있는 목록이 아닙니다(**자세한 내용은 프레임워크 디자인 지침** 참조).
>
> *교육* 및 *채택* 목표는 docs 코딩 규칙이 런타임 및 컴파일러 규칙과 다른 이유입니다. 런타임과 컴파일러 모두 실행 부하 과다 경로에 대한 엄격한 성능 메트릭을 가지고 있습니다. 다른 많은 응용 프로그램은 그렇지 않습니다. 우리의 *교육* 목표는 어떤 구성도 금지하지 않도록 명령합니다. 대신 샘플은 구문을 사용해야 하는 경우를 보여줍니다. 우리는 대부분의 생산 응용 프로그램보다 더 적극적으로 샘플을 업데이트합니다. *채택* 목표는 작년에 작성된 코드를 변경할 필요가 없는 경우에도 오늘 작성해야 하는 코드를 표시해야 합니다.

이 문서에서는 가이드라인에 대해 설명합니다. 가이드라인은 시간이 지남에 따라 발전해 왔으며 가이드라인을 따르지 않는 샘플을 찾을 수 있습니다. 이러한 샘플을 규정 준수하도록 하는 PR 또는 업데이트해야 하는 샘플에 주의를 기울이는 문제를 환영합니다. 우리의 지침은 오픈 소스이며 PR과 문제를 환영합니다. 그러나 제출로 인해 이러한 권장 사항이 변경되는 경우 먼저 토론을 위해 문제를 엽니다. 가이드라인을 사용하거나 필요에 맞게 조정할 수 있습니다.

# 도구 및 분석기

도구는 팀이 표준을 시행하는 데 도움이 될 수 있습니다. 코드 분석을 사용하도록 설정하여 원하는 규칙을 적용할 수 있습니다. Visual Studio에서 스타일 지침을 자동으로 적용하도록 editorconfig를 만들 수도 있습니다. 시작점으로 dotnet/docs 리포지토리의 파일을 복사하여 스타일을 사용할 수 있습니다.

이러한 도구를 사용하면 팀이 선호하는 지침을 더 쉽게 채택할 수 있습니다. Visual Studio는 범위의 모든 파일에 규칙을 적용하여 코드의 서식을 지정합니다. 여러 구성을 사용하여 회사 전체 표준, 팀 표준 및 세분화된 프로젝트 표준을 적용할 수 있습니다. `.editorconfig`

코드 분석은 사용하도록 설정된 규칙을 위반할 때 경고 및 진단을 생성합니다. 프로젝트에 적용할 규칙을 구성합니다. 그런 다음 각 CI 빌드는 규칙을 위반할 때 개발자에게 알립니다.

## 진단 ID

- 자체 분석기를 빌드할 때 적절한 진단 ID 선택

# 언어 지침

다음 섹션에서는 .NET 문서 팀이 코드 예제 및 샘플을 준비하기 위해 따르는 방법을 설명합니다. 일반적으로 다음 방법을 따르십시오.

- 가능하면 최신 언어 기능 및 C# 버전을 활용합니다.
- 더 이상 사용되지 않거나 오래된 언어 구문을 사용하지 마십시오.
- 제대로 처리할 수 있는 예외만 catch합니다. 제네릭 예외를 catch하지 마십시오.
- 특정 예외 유형을 사용하여 의미 있는 오류 메시지를 제공합니다.
- 컬렉션 조작에 LINQ 쿼리 및 메서드를 사용하여 코드 가독성을 높입니다.
- 비동기 프로그래밍을 사용하여 비동기 프로그래밍을 사용하고 I/O 바인딩된 작업을 기다립니다.
- 교착 상태에 주의하고 적절한 경우 Task.ConfigureAwait를 사용합니다.

- 런타임 형식 대신 데이터 형식에 language 키워드를 사용합니다. 예를 들어 System.String 대신 또는 System.Int32 대신 사용합니다. `string int`
- 부호 없는 형식 대신 사용합니다. 의 사용은 C# 전체에서 일반적이며 를 사용할 때 다른 라이브러리와 더 쉽게 상호 작용할 수 있습니다. 서명되지 않은 데이터 형식과 관련된 설명서는 예외입니다. `int int int`
- 판독기가 식에서 형식을 유추할 수 있는 경우에만 사용합니다. 독자는 문서 플랫폼에서 샘플을 볼 수 있습니다. 변수 유형을 표시하는 가리키기 또는 도구 설명이 없습니다. `var`
- 명확성과 단순성을 염두에 두고 코드를 작성합니다.
- 지나치게 복잡하고 난해한 코드 논리를 피하십시오.

보다 구체적인 지침은 다음과 같습니다.

## 문자열 데이터

- 다음 코드와 같이 문자열 보간을 사용하여 짧은 문자열을 연결합니다.

```
C#
```

```
string displayName = $"{nameList[n].LastName},
{nameList[n].FirstName}";
```

- 루프에 문자열을 추가하려면, 특히 많은 양의 텍스트로 작업하는 경우 System.Text.StringBuilder 개체를 사용합니다.

```
C#
```

```
var phrase =
"lalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

## 배열

- 선언 줄에서 배열을 초기화할 때 간결한 구문을 사용합니다. 다음 예제에서는 대신 사용할 수 없습니다. `var string[]`

```
C#
```

```
string[] vowels1 = { "a", "e", "i", "o", "u" };
```

- 명시적 인스턴스화를 사용하는 경우 . `var`

C#

```
var vowels2 = new string[] { "a", "e", "i", "o", "u" };
```

## 대리자

- 대리자 형식을 정의하는 대신 Func<> 및 Action<>을 사용합니다. 클래스에서 대리자 메서드를 정의합니다.

C#

```
Action<string> actionExample1 = x => Console.WriteLine($"x is: {x}");

Action<string, string> actionExample2 = (x, y) =>
    Console.WriteLine($"x is: {x}, y is {y}");

Func<string, int> funcExample1 = x => Convert.ToInt32(x);

Func<int, int, int> funcExample2 = (x, y) => x + y;
```

- 또는 대리자에 의해 정의된 서명을 사용하여 메서드를 호출합니다. `Func<>` `Action<>`

C#

```
actionExample1("string for x");

actionExample2("string for x", "string for y");

Console.WriteLine($"The value is {funcExample1("1")}");

Console.WriteLine($"The sum is {funcExample2(1, 2)}");
```

- 대리자 형식의 인스턴스를 만드는 경우 간결한 구문을 사용합니다. 클래스에서 대리자 형식과 일치하는 시그니처가 있는 메서드를 정의합니다.

  C#

  ```
  public delegate void Del(string message);

  public static void DelMethod(string str)
  {
  ```

```
    Console.WriteLine("DelMethod argument: {0}", str);
}
```

- 대리자 형식의 인스턴스를 만들고 호출합니다. 다음 선언에서는 압축된 구문을 보여 줍니다.

```C#
Del exampleDel2 = DelMethod;
exampleDel2("Hey");
```

- 다음 선언에서는 전체 구문을 사용합니다.

```C#
Del exampleDel1 = new Del(DelMethod);
exampleDel1("Hey");
```

## try-catch 예외 처리의 AND 문 using

- 대부분의 예외 처리에는 try-catch 문을 사용합니다.

```C#
static double ComputeDistance(double x1, double y1, double x2, double
y2)
{
    try
    {
        return Math.Sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 -
y2));
    }
    catch (System.ArithmeticException ex)
    {
        Console.WriteLine($"Arithmetic overflow or underflow: {ex}");
        throw;
    }
}
```

- C# using 문을 사용하여 코드를 단순화합니다. 블록의 유일한 코드가 Dispose 메서드에 대한 호출인 try-finally 문이 있는 경우 문을 대신 사용합니다. `finally using`

  다음 예제에서 문은 블록에서만 호출합니다. `try-finally Dispose finally`

```C#
```

```
Font bodyStyle = new Font("Arial", 10.0f);
try
{
    byte charset = bodyStyle.GdiCharSet;
}
finally
{
    if (bodyStyle != null)
    {
        ((IDisposable)bodyStyle).Dispose();
    }
}
```

명령문으로 동일한 작업을 수행할 수 있습니다. `using`

C#

```
using (Font arial = new Font("Arial", 10.0f))
{
    byte charset2 = arial.GdiCharSet;
}
```

중괄호가 필요하지 않은 새로운 using 구문을 사용합니다.

C#

```
using Font normalStyle = new Font("Arial", 10.0f);
byte charset3 = normalStyle.GdiCharSet;
```

## && 및 연산자 ||

- 다음 예제와 같이 비교를 수행할 때 & 대신 &&를 사용하고 | 대신 ||를 사용합니다.

C#

```
Console.Write("Enter a dividend: ");
int dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");
int divisor = Convert.ToInt32(Console.ReadLine());

if ((divisor != 0) && (dividend / divisor) is var result)
{
    Console.WriteLine("Quotient: {0}", result);
}
else
{
```

```
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

제수가 0이면 문의 두 번째 절로 인해 런타임 오류가 발생합니다. 그러나 && 연산자는 첫 번째 표현식이 false일 때 단락됩니다. 즉, 두 번째 식을 계산하지 않습니다. & 연산자는 둘 다 평가하므로 가 0일 때 런타임 오류가 발생합니다. `if divisor`

## `new` 연산자

- 다음 선언과 같이 간결한 형식의 개체 인스턴스화 중 하나를 사용합니다.

C#

```
var firstExample = new ExampleClass();
```

C#

```
ExampleClass instance2 = new();
```

앞의 선언은 다음 선언과 동일합니다.

C#

```
ExampleClass secondExample = new ExampleClass();
```

- 다음 예제와 같이 개체 이니셜라이저를 사용하여 개체 만들기를 단순화합니다.

C#

```
var thirdExample = new ExampleClass { Name = "Desktop", ID = 37414,
    Location = "Redmond", Age = 2.3 };
```

다음 예제에서는 앞의 예제와 동일한 속성을 설정하지만 이니셜라이저를 사용하지 않습니다.

C#

```
var fourthExample = new ExampleClass();
fourthExample.Name = "Desktop";
fourthExample.ID = 37414;
fourthExample.Location = "Redmond";
fourthExample.Age = 2.3;
```

## 이벤트 처리

- 람다 식을 사용하여 나중에 제거할 필요가 없는 이벤트 처리기를 정의합니다.

C#

```
public Form2()
{
    this.Click += (s, e) =>
        {
            MessageBox.Show(
                ((MouseEventArgs)e).Location.ToString());
        };
}
```

람다 식은 다음과 같은 전통적인 정의를 단축합니다.

C#

```
public Form1()
{
    this.Click += new EventHandler(Form1_Click);
}

void Form1_Click(object? sender, EventArgs e)
{
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());
}
```

## 정적 멤버

클래스 이름(*ClassName.StaticMember*)을 사용하여 정적 멤버를 호출합니다. 이 방법은 정적 액세스를 명확하게 하여 코드를 더 읽기 쉽게 만듭니다. 기본 클래스에 정의된 정적 멤버를 파생 클래스의 이름으로 한정하지 마세요. 해당 코드가 컴파일되는 동안 코드 가독성이 잘못될 수 있으며 나중에 파생 클래스에 동일한 이름의 정적 멤버를 추가하면 코드가 중단될 수 있습니다.

## LINQ 쿼리

- 쿼리 변수에 의미 있는 이름을 사용합니다. 다음 예에서는 시애틀에 있는 고객을 대상으로 합니다. `seattleCustomers`

  C#

  ```
  var seattleCustomers = from customer in customers
                         where customer.City == "Seattle"
  ```

```
                select customer.Name;
```

- 별칭을 사용하여 익명 형식의 속성 이름이 파스칼식 대/소문자를 사용하여 올바르게 대문자로 표시되는지 확인합니다.

```
C#
```

```
var localDistributors =
    from customer in customers
    join distributor in distributors on customer.City equals
distributor.City
    select new { Customer = customer, Distributor = distributor };
```

- 결과의 속성 이름이 모호한 경우 속성 이름을 바꿉니다. 예를 들어 쿼리에서 고객 이름과 배포자 ID를 반환하는 경우 결과에 그대로 두는 대신 이름을 바꿔 고객 이름과 배포자 ID임을 명확히 합니다. `Name ID Name ID`

```
C#
```

```
var localDistributors2 =
    from customer in customers
    join distributor in distributors on customer.City equals
distributor.City
    select new { CustomerName = customer.Name, DistributorID =
distributor.ID };
```

- 쿼리 변수와 범위 변수의 선언에 암시적 입력을 사용합니다. LINQ 쿼리의 암시적 형식 지정에 대한 이 지침은 암시적으로 형식화된 지역 변수에 대한 일반 규칙을 재정의합니다. LINQ 쿼리는 익명 형식을 만드는 프로젝션을 사용하는 경우가 많습니다. 다른 쿼리 식은 중첩된 제네릭 형식으로 결과를 만듭니다. 암시적 형식화된 변수는 종종 더 읽기 쉽습니다.

```
C#
```

```
var seattleCustomers = from customer in customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

- 이전 예제에 표시된 대로 from 절 아래에 쿼리 절을 맞춥니다.

- 다른 쿼리 절 앞에 where 절을 사용하여 이후 쿼리 절이 축소되고 필터링된 데이터 집합에서 작동하도록 합니다.

```
C#
```

```
var seattleCustomers2 = from customer in customers
                        where customer.City == "Seattle"
                        orderby customer.Name
                        select customer;
```

- join 절 대신 여러 절을 사용하여 내부 컬렉션에 액세스합니다. 예를 들어 개체 컬렉션에는 각각 테스트 점수 컬렉션이 포함될 수 있습니다. 다음 쿼리를 실행하면 90점을 초과하는 각 점수와 점수를 받은 학생의 성이 반환됩니다. `from Student`

C#

```
var scoreQuery = from student in students
                 from score in student.Scores!
                 where score > 90
                 select new { Last = student.LastName, score };
```

## 암시적으로 형식화된 지역 변수

- 지역 변수에 대한 암시적 형식 지정은 변수의 형식이 할당의 오른쪽에서 분명한 경우에 사용합니다.

C#

```
var message = "This is clearly a string.";
var currentTemperature = 27;
```

- 형식이 할당의 오른쪽에서 명확하지 않은 경우 var를 사용하지 마세요. 메서드 이름에서 형식이 명확하다고 가정하지 마세요. 변수 형식은 연산자, 명시적 캐스트 또는 리터럴 값에 대한 할당인 경우 명확한 것으로 간주됩니다. `new`

C#

```
int numberOfIterations = Convert.ToInt32(Console.ReadLine());
int currentMaximum = ExampleClass.ResultSoFar();
```

- 변수 이름을 사용하여 변수의 형식을 지정하지 마세요. 정확하지 않을 수 있습니다. 대신 type을 사용하여 유형을 지정하고 변수 이름을 사용하여 변수의 의미 체계 정보를 나타냅니다. 다음 예제는 콘솔에서 읽은 정보의 의미를 나타내기 위해 유형 및 이와 유사한 것을 사용해야 합니다. `string iterations`

C#

```
var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- 동적 대신 사용하지 마십시오. 런타임 형식 유추를 원할 때 사용합니다. 자세한 내용은 dynamic 형식 사용(C# 프로그래밍 가이드)을 참조하십시오. `var` `dynamic`

- for 루프에서 loop 변수에 대해 암시적 형식 지정을 사용합니다.

  다음 예제에서는 문에서 암시적 형식 지정을 사용합니다. `for`

  C#

  ```
  var phrase =
  "lalalalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
  var manyPhrases = new StringBuilder();
  for (var i = 0; i < 10000; i++)
  {
      manyPhrases.Append(phrase);
  }
  //Console.WriteLine("tra" + manyPhrases);
  ```

- 암시적 형식 지정을 사용하여 foreach 루프에서 루프 변수의 형식을 확인하지 마세요. 대부분의 경우 컬렉션의 요소 형식은 즉시 명확하지 않습니다. 컬렉션의 이름은 해당 요소의 형식을 유추하는 데만 의존해서는 안 됩니다.

  다음 예제에서는 문에 명시적 형식 지정을 사용합니다. `foreach`

  C#

  ```
  foreach (char ch in laugh)
  {
      if (ch == 'h')
          Console.Write("H");
      else
          Console.Write(ch);
  }
  Console.WriteLine();
  ```

- LINQ 쿼리의 결과 시퀀스에 암시적 형식을 사용합니다. LINQ에 대한 단원에서는 많은 LINQ 쿼리로 인해 암시적 형식을 사용해야 하는 익명 형식이 생성된다고 설명합니다. 다른 쿼리는 더 읽기 쉬운 중첩된 제네릭 형식을 생성합니다. `var`

  **ⓘ 메모**

이터러블 컬렉션의 요소 형식을 실수로 변경하지 않도록 주의하십시오. 예를 들어 문에서 System.Linq.IQueryable에서 System.Collections.IEnumerable로 쉽게 전환하여 쿼리 실행을 변경할 수 있습니다. `foreach`

일부 샘플은 표현식의 *자연 유형을* 설명합니다. 이러한 샘플은 컴파일러가 자연 형식을 선택하도록 사용해야 합니다. 이러한 예는 명확하지 않지만 샘플에는 의 사용이 필요합니다. 텍스트는 동작을 설명해야 합니다. `var var`

## using 지시문을 네임스페이스 선언 외부에 배치합니다

지시문이 네임스페이스 선언 외부에 있는 경우 가져온 네임스페이스는 정규화된 이름입니다. 정규화된 이름이 더 명확합니다. 지시문이 네임스페이스 내에 있는 경우 해당 네임스페이스에 상대적이거나 정규화된 이름일 수 있습니다. `using using`

C#

```
using Azure;

namespace CoolStuff.AwesomeFeature
{
    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

WaitUntil 클래스에 대한 참조(직접 또는 간접)가 있다고 가정합니다.

이제 약간 변경해 보겠습니다.

C#

```
namespace CoolStuff.AwesomeFeature
{
    using Azure;

    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
```

```
    }
}
```

And it compiles today. And tomorrow. But then sometime next week the preceding (untouched) code fails with two errors:

Console

```
- error CS0246: The type or namespace name 'WaitUntil' could not be found
(are you missing a using directive or an assembly reference?)
- error CS0103: The name 'WaitUntil' does not exist in the current context
```

One of the dependencies has introduced this class in a namespace then ends with `: .Azure`

C#

```
namespace CoolStuff.Azure
{
    public class SecretsManagement
    {
        public string FetchFromKeyVault(string vaultId, string secretId) {
return null; }
    }
}
```

A directive placed inside a namespace is context-sensitive and complicates name resolution. In this example, it's the first namespace that it finds. `using`

- `CoolStuff.AwesomeFeature.Azure`
- `CoolStuff.Azure`
- `Azure`

Adding a new namespace that matches either or would match before the global namespace. You could resolve it by adding the modifier to the declaration. However, it's easier to place declarations outside the namespace instead. `CoolStuff.Azure CoolStuff.AwesomeFeature.Azure Azure global:: using using`

C#

```
namespace CoolStuff.AwesomeFeature
{
    using global::Azure;

    public class Awesome
    {
```

```
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

# Style guidelines

In general, use the following format for code samples:

- Use four spaces for indentation. Don't use tabs.
- Align code consistently to improve readability.
- Limit lines to 65 characters to enhance code readability on docs, especially on mobile screens.
- Break long statements into multiple lines to improve clarity.
- Use the "Allman" style for braces: open and closing brace its own new line. Braces line up with current indentation level.
- Line breaks should occur before binary operators, if necessary.

## Comment style

- Use single-line comments () for brief explanations. `//`
- Avoid multi-line comments () for longer explanations. Comments aren't localized. Instead, longer explanations are in the companion article. `/* */`
- For describing methods, classes, fields, and all public members use XML comments.
- Place the comment on a separate line, not at the end of a line of code.
- Begin comment text with an uppercase letter.
- End comment text with a period.
- Insert one space between the comment delimiter () and the comment text, as shown in the following example. `//`

C#

```
// The following declaration creates a query. It does not run
// the query.
```

## Layout conventions

Good layout uses formatting to emphasize the structure of your code and to make the code easier to read. Microsoft examples and samples conform to the following conventions:

- Use the default Code Editor settings (smart indenting, four-character indents, tabs saved as spaces). For more information, see Options, Text Editor, C#, Formatting.
- Write only one statement per line.
- Write only one declaration per line.
- If continuation lines aren't indented automatically, indent them one tab stop (four spaces).
- Add at least one blank line between method definitions and property definitions.
- Use parentheses to make clauses in an expression apparent, as shown in the following code.

```
C#
```

```
if ((startX > endX) && (startX > previousX))
{
    // Take appropriate action.
}
```

Exceptions are when the sample explains operator or expression precedence.

# Security

Follow the guidelines in Secure Coding Guidelines.

### Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

## .NET feedback

.NET is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback