using directive

Article • 03/14/2023

The `using` directive allows you to use types defined in a namespace without specifying the fully qualified namespace of that type. In its basic form, the `using` directive imports all the types from a single namespace, as shown in the following example:

```C#
using System.Text;
```

You can apply two modifiers to a `using` directive:

- The `global` modifier has the same effect as adding the same `using` directive to every source file in your project. This modifier was introduced in C# 10.
- The `static` modifier imports the `static` members and nested types from a single type rather than importing all the types in a namespace.

You can combine both modifiers to import the static members from a type in all source files in your project.

You can also create an alias for a namespace or a type with a *using alias directive*.

```C#
using Project = PC.MyCompany.Project;
```

You can use the `global` modifier on a *using alias directive*.

> **Note**
>
> The `using` keyword is also used to create *using statements*, which help ensure that **IDisposable** objects such as files and fonts are handled correctly. For more information about the *using statement*, see **using statement**.

The scope of a `using` directive without the `global` modifier is the file in which it appears.

The `using` directive can appear:

- At the beginning of a source code file, before any namespace or type declarations.

- In any namespace, but before any namespaces or types declared in that namespace, unless the `global` modifier is used, in which case the directive must appear before all namespace and type declarations.

Otherwise, compiler error CS1529 is generated.

Create a `using` directive to use the types in a namespace without having to specify the namespace. A `using` directive doesn't give you access to any namespaces that are nested in the namespace you specify. Namespaces come in two categories: user-defined and system-defined. User-defined namespaces are namespaces defined in your code. For a list of the system-defined namespaces, see .NET API Browser.

global modifier

Adding the `global` modifier to a `using` directive means that using is applied to all files in the compilation (typically a project). The `global using` directive was added in C# 10. Its syntax is:

C#

```
global using <fully-qualified-namespace>;
```

where *fully-qualified-namespace* is the fully qualified name of the namespace whose types can be referenced without specifying the namespace.

A *global using* directive can appear at the beginning of any source code file. All `global using` directives in a single file must appear before:

- All `using` directives without the `global` modifier.
- All namespace and type declarations in the file.

You may add `global using` directives to any source file. Typically, you'll want to keep them in a single location. The order of `global using` directives doesn't matter, either in a single file, or between files.

The `global` modifier may be combined with the `static` modifier. The `global` modifier may be applied to a *using alias directive*. In both cases, the directive's scope is all files in the current compilation. The following example enables using all the methods declared in the System.Math in all files in your project:

C#

```
global using static System.Math;
```

You can also globally include a namespace by adding a `<Using>` item to your project file, for example, `<Using Include="My.Awesome.Namespace" />`. For more information, see <Using> item.

ⓘ **Important**

The C# templates for .NET 6 use *top level statements*. Your application may not match the code in this article, if you've already upgraded to the .NET 6. For more information see the article on **New C# templates generate top level statements**

The .NET 6 SDK also adds a set of *implicit* `global using` directives for projects that use the following SDKs:

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

These implicit `global using` directives include the most common namespaces for the project type.

For more information, see the article on **Implicit using directives**

static modifier

The `using static` directive names a type whose static members and nested types you can access without specifying a type name. Its syntax is:

```C#
using static <fully-qualified-type-name>;
```

The `<fully-qualified-type-name>` is the name of the type whose static members and nested types can be referenced without specifying a type name. If you don't provide a fully qualified type name (the full namespace name along with the type name), C# generates compiler error CS0246: "The type or namespace name 'type/namespace' couldn't be found (are you missing a using directive or an assembly reference?)".

The `using static` directive applies to any type that has static members (or nested types), even if it also has instance members. However, instance members can only be invoked through the type instance.

You can access static members of a type without having to qualify the access with the type name:

C#

```
using static System.Console;
using static System.Math;
class Program
{
    static void Main()
    {
        WriteLine(Sqrt(3*3 + 4*4));
    }
}
```

Ordinarily, when you call a static member, you provide the type name along with the member name. Repeatedly entering the same type name to invoke members of the type can result in verbose, obscure code. For example, the following definition of a `Circle` class references many members of the Math class.

C#

```
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); }
```

```
    }
}
```

By eliminating the need to explicitly reference the Math class each time a member is referenced, the `using static` directive produces cleaner code:

```C#
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` imports only accessible static members and nested types declared in the specified type. Inherited members aren't imported. You can import from any named type with a `using static` directive, including Visual Basic modules. If F# top-level functions appear in metadata as static members of a named type whose name is a valid C# identifier, then the F# functions can be imported.

`using static` makes extension methods declared in the specified type available for extension method lookup. However, the names of the extension methods aren't imported into scope for unqualified reference in code.

Methods with the same name imported from different types by different `using static` directives in the same compilation unit or namespace form a method group. Overload resolution within these method groups follows normal C# rules.

The following example uses the `using static` directive to make the static members of the Console, Math, and String classes available without having to specify their type name.

```C#
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 ra-
dius)) {
         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
```

In the example, the `using static` directive could also have been applied to the Double type. Adding that directive would make it possible to call the TryParse(String, Double) method without specifying a type name. However, using `TryParse` without a type name creates less readable code, since it becomes necessary to check the `using static` directives to determine which numeric type's `TryParse` method is called.

`using static` also applies to `enum` types. By adding `using static` with the enum, the type is no longer required to use the enum members.

C#

```
using static Color;

enum Color
{
    Red,
    Green,
    Blue
}

class Program
{
    public static void Main()
    {
        Color color = Green;
    }
}
```

using alias

Create a `using` alias directive to make it easier to qualify an identifier to a namespace or type. In any `using` directive, the fully qualified namespace or type must be used regardless of the `using` directives that come before it. No `using` alias can be used in the declaration of a `using` directive. For example, the following example generates a compiler error:

C#

```
using s = System.Text;
using s.RegularExpressions; // Generates a compiler error.
```

The following example shows how to define and use a `using` alias for a namespace:

C#

```
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 { }
        }
    }
}
```

A using alias directive can't have an open generic type on the right-hand side. For example, you can't create a using alias for a `List<T>`, but you can create one for a `List<int>`.

The following example shows how to define a `using` directive and a `using` alias for a class:

C#

```
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.
```

Beginning with C# 12, you can create aliases for types that were previously restricted, including tuple types, pointer types, and other unsafe types. For more information on the updated rules, see the feature spec.

How to use the Visual Basic `My` namespace

The Microsoft.VisualBasic.MyServices namespace (`My` in Visual Basic) provides easy and intuitive access to a number of .NET classes, enabling you to write code that interacts with the computer, application, settings, resources, and so on. Although originally designed for use with Visual Basic, the `MyServices` namespace can be used in C# applications.

For more information about using the `MyServices` namespace from Visual Basic, see Development with My.

You need to add a reference to the *Microsoft.VisualBasic.dll* assembly in your project. Not all the classes in the `MyServices` namespace can be called from a C# application: for example, the FileSystemProxy class is not compatible. In this particular case, the static methods that are part of FileSystem, which are also contained in VisualBasic.dll, can be used instead. For example, here is how to use one such method to duplicate a directory:

C#

```csharp
// Duplicate a directory
Microsoft.VisualBasic.FileIO.FileSystem.CopyDirectory(
    @"C:\original_directory",
    @"C:\copy_of_original_directory");
```

C# language specification

For more information, see Using directives in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

For more information on the *global using* modifier, see the global usings feature specification - C# 10.

See also

- C# reference
- C# keywords
- Namespaces
- Style rule IDE0005 - Remove unnecessary 'using' directives
- Style rule IDE0065 - 'using' directive placement
- using statement