

B E G I N N E R S ' S
C # T U T O R I A L
6 . N A M E S P A C E S

WRITTEN BY JOE MAYO
JMAYO@CSHARP-STATION.COM
UPDATED 15/10/00, 11/11/01, 12/03/03

CONVERTED TO PDF BY ASH WEAVER 02/09/03

WWW.CSHARP-STATION.COM

This lesson introduces you to C# Namespaces. Our objectives are as follows:

- Understand what Namespace is.
- Learn how to implement the *using* directive.
- Learn to use *alias* directive.
- Understand what are namespace members.

In Lesson 1, you saw the *using System;* directive in the *SimpleHello* program. This directive allowed you to use members of the *System namespace*. Because of the narrow focus of that lesson, we needed to delay explanation until now. When you've completed this lesson you will understand the *using* directive and more.

Namespaces are C# program elements designed to help you organize your programs. They also provide assistance in avoiding name clashes between two sets of code. Implementing Namespaces in your own code is a good habit because it is likely to save you from problems later when you want to reuse some of your code.

Namespaces don't correspond to file or directory names. If naming directories and files to correspond to *namespaces* helps you organize your code, then you may do so, but it is not required.

Listing 6-1. The C# Station Namespace: NamespaceCSS.cs

```
// Namespace Declaration
using System;

// The C# Station Namespace
namespace csharp_station
{
    // Program start class
    class NamespaceCSS
    {
        // Main begins program execution.
        public static void Main()
        {
            // Write to console
            Console.WriteLine("This is the new C# Station Namespace.");
        }
    }
}
```

Listing 6-1 shows how to create a *namespace*. We declare the new *namespace* by putting the word *namespace* in front of *csharp_station*. Curly braces surround the members inside the *csharp_station namespace*.

Listing 6-2. Nested Namespace 1: *NestedNamespace1.cs*

```
// Namespace Declaration
using System;

// The C# Station Tutorial Namespace
namespace cssharp_station
{
    namespace tutorial
    {
        // Program start class
        class NamespaceCSS
        {
            // Main begins program execution.
            public static void Main()
            {
                // Write to console
                Console.WriteLine("This is the new C# Station Tutorial Namespace.");
            }
        }
    }
}
```

Namespaces allow you to create a system to organize your code. A good way to organize your *namespaces* is via a hierarchical system. You put the more general names at the top of the hierarchy and get more specific as you go down. This hierarchical system can be represented by nested *namespaces*. Listing 6-2 shows how to create a nested *namespace*. By placing code in different sub-namespaces, you can keep your code organized.

Listing 6-3. Nested Namespace 2: *NestedNamespace2.cs*

```
// Namespace Declaration
using System;

// The C# Station Tutorial Namespace
namespace cssharp_station.tutorial
{
    // Program start class
    class NamespaceCSS
    {
        // Main begins program execution.
        public static void Main()
        {
            // Write to console
            Console.WriteLine("This is the new C# Station Tutorial Namespace.");
        }
    }
}
```

Listing 6-3 shows another way of writing nested *namespaces*. It specifies the nested *namespace* with the dot operator between *cssharp_station* and *tutorial*. The result is exactly the same as Listing 6-2. However, Listing 6-3 is easier to write.

Listing 6-4. Calling Namespace Members: NamespaceCall.cs

```
// Namespace Declaration
using System;

namespace csharp_station
{
    // nested namespace
    namespace tutorial
    {
        class myExample1
        {
            public static void myPrint1()
            {
                Console.WriteLine("First Example of calling another namespace member.");
            }
        }
    }

    // Program start class
    class NamespaceCalling
    {
        // Main begins program execution.
        public static void Main()
        {
            // Write to console
            tutorial.myExample1.myPrint1();
            tutorial.myExample2.myPrint2();
        }
    }
}

// same namespace as nested namespace above
namespace csharp_station.tutorial
{
    class myExample2
    {
        public static void myPrint2()
        {
            Console.WriteLine("Second Example of calling another namespace member.");
        }
    }
}
```

Listing 6-4 provides an example of how to call *namespace* members with fully qualified names. A fully qualified name contains every language element from the *namespace* name down to the method call. At the top of the listing there is a nested *namespace tutorial* within the *csharp-station namespace* with *class myExample1* and method *myPrint1*. *Main()* calls this method with the fully qualified name of *tutorial.myExample1.myPrint()*. Since *Main()* and the *tutorial namespace* are located in the same *namespace*, using *csharp_station* in the fully qualified name is unnecessary.

At the bottom of Listing 6-4 is an addition to the *csharp_station.tutorial namespace*. The classes *myExample1* and *myExample2* both belong to the

same *namespace*. Additionally, they could be written in separate files and still belong to the same *namespace*. In *Main()*, the *myPrint2()* method is called with the fully qualified name *tutorial.myExample2.myPrint2()*. Although the class *myExample2* is outside the bounding braces of where the method *myPrint2* is called, the *namespace csharp_station* does not need to be a part of the fully qualified name. This is because both classes belong to the same *namespace*, *csharp_station*.

Notice that I used different names for the two classes *myExample1* and *myExample2*. This was necessary because every *namespace* member of the same type must have a unique name. Remember, they are both in the same *namespace* and you wouldn't want any ambiguity about which class to use. The methods *myPrint1()* and *myPrint2()* have different names only because it would make the lesson a little easier to follow. They could have had the same name with no effect, because their classes are different, thus avoiding any ambiguity.

Listing 6-5. The using Directive: UsingDirective.cs

```
// Namespace Declaration
using System;
using csharp_station.tutorial;

// Program start class
class UsingDirective
{
    // Main begins program execution.
    public static void Main()
    {
        // Call namespace member
        myExample.myPrint();
    }
}

// C# Station Tutorial Namespace
namespace csharp_station.tutorial
{
    class myExample
    {
        public static void myPrint()
        {
            Console.WriteLine("Example of using a using directive.");
        }
    }
}
```

If you would like to call methods without typing their fully qualified name, you can implement the *using* directive. In Listing 6-5, we show two *using* directives. The first, *using System*, is the same *using* directive you have seen in every program in this tutorial. It allows you to type the method names of members of the *System namespace* without typing the word *System* every time. In *myPrint()*, *Console* is a class member of the *System*

namespace with the method *WriteLine()*. Its fully qualified name is *System.Console.WriteLine(...)*.

Similarly, the *using* directive *using csharp_station.tutorial* allows us to implement members of the *csharp_station.tutorial namespace* without typing the fully qualified name. This is why we can type *myExample.myPrint()*. Without the *using* directive, we would have to type *csharp_station.tutorial.myExample.myPrint()* every time we wanted to implement that method.

Listing 6-6. The Alias Directive: *AliasDirective.cs*

```
// Namespace Declaration
using System;
using csTut = csharp_station.tutorial.myExample; // alias

// Program start class
class AliasDirective
{
    // Main begins program execution.
    public static void Main()
    {
        // Call namespace member
        csTut.myPrint();
        myPrint();
    }

    // Potentially ambiguous method.
    static void myPrint()
    {
        Console.WriteLine("Not a member of csharp_station.tutorial.myExample.");
    }
}

// C# Station Tutorial Namespace
namespace csharp_station.tutorial
{
    class myExample
    {
        public static void myPrint()
        {
            Console.WriteLine("This is a member of csharp_station.tutorial.myExample.");
        }
    }
}
```

Sometimes you may encounter a long namespace and wish to have it shorter. This could improve readability and still avoid name clashes with similarly named methods. Listing 6-6 shows how to create an alias with the alias directive *using csTut = csharp_station.tutorial.myExample*. Now the expression *csTut* can be used anywhere, in this file, in place of *csharp_station.tutorial.myExample*. We use it in *Main()*.

Also in *Main()* is a call to the *myPrint()* method of the *AliasDirective class*. This is the same name as the *myPrint()* method in the *myExample*

class . The reason both of these methods can be called in the same method call is because the *myPrint()* method in the *myExample class* is qualified with the *csTut* alias. This lets the compiler know exactly which method is to be executed. Had we mistakenly omitted *csTut* from the method call, the compiler would have set up the *myPrint()* method of the *AliasDirective class* to run twice.

So far, all we've shown in our namespaces are classes. However, namespaces can hold other types as follows:

- Classes
- Structures
- Interfaces
- Enumerations
- Delegates

Future chapters we will cover what these types are in more detail.

In summary, you know what a *namespace* is and you can declare your own namespaces. If you don't want to type a fully qualified name, you know how to implement the *using* directive. When you want to shorten a long namespace declaration, you can use the *alias* directive. Also, you have been introduced to some of the other *namespace* members in addition to the *class* type.