The Hello World Project

In this lesson, you're going to learn about the different parts of a very simple program that displays the message, "Hello World!" You'll also learn what happens when you build and run the application, and you'll learn about some common errors you may encounter and how to correct them.

A C# program begins with a *Main* method, usually found in a file called *Program.cs*, like this one:

```
using System;

namespace ConsoleApplication
{
    public class Program
    {
        public static void Main()
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```


```
using System;

namespace ConsoleApplication
{
    public class Program
    {
        public static void Main()
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```


Tip {.tip .newLanguage }
A *method* is a named group of individual commands the program will run. You'll learn more about creating methods in a bit.

Program.cs

Program.cs is a (usually small) text file. Its file extension is "cs" because it contains C# source code. When you build the program from a command prompt, the `dotnet build` tool will build all of the files that end in ".cs" using the C# compiler. Although it's a small program, there are a number of important bits of syntax in it that you should understand. First, remember that C# is case-sensitive, so keywords won't work unless they're lowercase, and other named elements within the source code must exactly match the case of the element being referenced.

The first line of the program is

```
using System;
```

The `using` statement is a programmer convenience. It allows us to refer to elements that exist within the listed *namespace* (in this case, `System`) without prefixing them with the namespace name. What's a namespace? A namespace is a way of organizing programming constructs. They're similar to folders or directories in your file system. You don't have to use them, but they make it much easier to find and organize things. The reason this program includes the `System` namespace is that the `Console` type (used to print "Hello World!") is in that namespace. If the `using` statement were removed, the `Console.WriteLine` statement would need to include the namespace, becoming `System.Console.WriteLine`. `using` statements must end with a semicolon (`;`). In C#, most statements that aren't defining a scope end with a semicolon.

After the `using` statements, the code declares its namespace:

```
namespace ConsoleApplication
```

Again, it's a good idea to use namespaces to keep larger codebases organized. `namespace` is a language keyword; *ConsoleApplication* is an identifier. In this case, the `ConsoleApplication` namespace has only one element in it (the `Program` class), but this would grow as the program grew in complexity. Namespaces use curly braces (`{` and `}`) to denote which types belong within the namespace. Namespaces are optional; you'll frequently see they're omitted from the small samples shown in this tutorial.

Inside the namespace's scope (defined by its curly braces), a `class` called "Program" is created:

```
public class Program
```

This line includes two keywords and one identifier. The `public` keyword describes the class's accessibility level. This defines how the class may be accessed by other parts of the program, and `public` means there are no restrictions to its access. The `class` keyword is used to define classes in C#, one of the primary constructs used to define *types* you will work with. C# is a *strongly typed* language, meaning that most of the time you'll need to explicitly define a type in your source code before it can be referenced from a program.

Inside the class's scope, a *method* called "Main" is defined:

```
public static void Main()
```

The "Main" method is this program's entry point - the first code that runs when the application is run. Like classes, methods can have accessibility modifiers, too. In this case, `public` means there are no limitations on access to this method.

Next, the `static` keyword marks this method as global and associated with the type it's defined on, not a particular *instance* of that type. You'll learn more about this distinction in later lessons.

The `void` keyword indicates that this method doesn't return a value. The method is named *Main*.

Finally, inside of parentheses (`(` and `)`), the method defines any *parameters* it requires. In this case, the method has no parameters, but a command line program might accept arguments by specifying a parameter of type *string array*. This parameter is typically defined as `string[] args`, where *args* in this case is short for *arguments*. Arguments correspond to parameters. A method defines the parameters it requires; when calling a method, the values passed to its parameters are referred to as arguments. Like namespaces and classes, methods have scope defined by curly braces.

A class can contain many methods, which are one kind of *member* of that class.

Within the method's scope, there is one line:

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

You've already learned that `Console` is a type inside of the `System` namespace. It's worth noting that this code does not create an *instance* of the `Console` type - it is simply calling the `WriteLine` method on the type directly. This tells you that `WriteLine`, like the `Main` method in this program, is declared as a `static` method. This means that any part of the application that calls this method will be calling the same method, doing the same thing. The program won't, for instance, open several different console windows and write to them separately. Every call to `Console.WriteLine` is going to write to the same console window.

Inside of the parentheses, the program is passing in "Hello World!" to the method. This is an *argument*, and will be used by the `WriteLine` method internally. C# defines a number of built-in types, one of which is a *string*. A string is a series of text characters. In this case, the program is passing the string `"Hello World!"` as an argument to the `WriteLine` method, which has defined a string parameter type. At the end of the line, the statement ends with a semicolon.

After the `Console.WriteLine` statement, there are three closing curly braces (`}`). These close the scopes for the `Main` method, the `Program` class, and the `ConsoleApplication` namespace, respectively. Note that the program uses indentation to make it easy to see which elements of the code belong to which scope. This is a good practice to follow, and will make it much easier for you (or others) to quickly read and understand the code you write.