# **Module 1: Write your first C#**

**Write your first code**

Console.WriteLine("Hello World!");

**Common mistakes new programmers make:**

* Entering lower-case letters instead of capitalizing C in Console, or the letters W or L in WriteLine.
* Entering a comma instead of a period between Console and WriteLine.
* Forgetting to use double-quotation marks, or using single-quotation marks to surround the phrase Hello World!.
* Forgetting a semi-colon at the end of the command.

**The difference between Console.Write and Console.WriteLine**

* The three new lines of code you added demonstrated the difference between the [Console.WriteLine()](https://learn.microsoft.com/en-us/dotnet/api/system.console.writeline#system-console-writeline) and [Console.Write](https://learn.microsoft.com/en-us/dotnet/api/system.console.write) methods.
* To print an entire message to the output console, you used the first technique, Console.WriteLine(). At the end of the line, it added a line feed similar to how to create a new line of text by pressing Enter or Return.
* To print to the output console, but without adding a line feed at the end, you used the second technique, Console.Write(). So, the next call to Console.Write() prints another message to the same line.
* Congratulations on writing your first lines of code!

# Learn how it works

## What is compilation?

A special program called a **compiler** converts your source code into a different format that the computer's central processing unit (CPU) can execute. When you used the green **Run** button in the previous unit, the code you wrote was first compiled, then executed.

Why does code need to be compiled? Although most programming languages seem cryptic at first, they can be more easily understood by humans than the computer's *preferred* language. The CPU understands instructions that are expressed by turning thousands or millions of tiny switches either on or off. Compilers bridge these two worlds by translating your human-readable instructions into a computer-understandable set of instructions.

The Console part is called a **class**. Classes "own" methods; There's also a dot (or period) that separates the class name Console and the method name WriteLine(). The period is the *member access operator*. In other words, the dot is how you "navigate" from the class to one of its methods.

The WriteLine() part is called a **method**. The parentheses are known as the *method invocation operator*.

Finally, the semicolon is the *end of statement operator*. A **statement** is a complete instruction in C#. The semicolon tells the compiler that you've finished entering the command.

Module 1 Achievement: https://learn.microsoft.com/en-us/training/achievements/learn.languages.csharp-write-first.badge?username=NguyenAnhTuanVPI-3001&sharingId=2ECA20722B251E9F

**What is a literal value?**

A literal value is a constant value that never changes.

### Use character literals

If you only wanted a single alphanumeric character printed to screen, you could create a **char literal** by surrounding one alphanumeric character in single quotes. The term char is short for character. In C#, this data type is officially named "char", but frequently referred to as a "character".

C# Datatypes:

A screenshot of a computer

Description automatically generated

### Use floating-point literals

A floating-point number is a number that contains a decimal, for example 3.14159. C# supports three data types to represent decimal numbers: float, double, and decimal. Each type supports varying degrees of precision.

Float Type Precision

----------------------------

float ~6-9 digits

double ~15-17 digits

decimal 28-29 digits

Console.WriteLine(0.25F);

To create a float literal, append the letter F after the number. In this context, the F is called a *literal suffix*. The literal suffix tells the compiler you wish to work with a value of float type. You can use either a lower-case f or upper-case F as the literal suffix for a float.

Console.WriteLine(12.39816m);

To create a decimal literal, append the letter m after the number. In this context, the m is called a *literal suffix*. The literal suffix tells the compiler you wish to work with a value of decimal type. You can use either a lower-case m or upper-case M as the literal suffix for a decimal.

Console.WriteLine(true);

Console.WriteLine(false);

### Why emphasize data types?

Data types play a central role in C#. In fact, the emphasis on data types is one of the key distinguishing features of C# compared to other languages like JavaScript. The designers of C# believed they can help developers avoid common software bugs by enforcing data types. You'll see this concept unfold as you learn more about C#.

## Recap

The main takeaway is that there are many data types, but you'll focus on just a few for now:

* string for words, phrases, or any alphanumeric data for presentation, not calculation
* char for a single alphanumeric character
* int for a whole number
* decimal for a number with a fractional component
* bool for a true/false value

Here's a few important considerations about variable names:

* Variable names can contain alphanumeric characters and the underscore character. Special characters like the hash symbol # (also known as the number symbol or pound symbol) or dollar symbol $ are not allowed.
* Variable names must begin with an alphabetical letter or an underscore, not a number.
* Variable names are case-sensitive, meaning that string Value; and string value; are two different variables.
* Variable names must **not** be a C# keyword. For example, you cannot use the following variable declarations: decimal decimal; or string string;.

Here are some coding conventions for variables:

* Variable names should use **camel case**, which is a style of writing that uses a lower-case letter at the beginning of the first word and an upper-case letter at the beginning of each subsequent word. For example, string thisIsCamelCase;.
* Variable names should begin with an alphabetical letter. Developers use the underscore for a special purpose, so try to not use that for now.
* Variable names should be descriptive and meaningful in your app. Choose a name for your variable that represents the kind of data it will hold.
* Variable names should be one or more entire words appended together. Don't use contractions or abbreviations because the name of the variable (and therefore, its purpose) may be unclear to others who are reading your code.
* Variable names shouldn't include the data type of the variable. You might see some advice to use a style like string strValue;. That advice is no longer current.

### Variable name examples

Here's a few examples of variable declarations using the data types you learned about thus far:

char userOption;

int gameScore;

decimal particlesPerMillion;

bool processedCustomer;

# Declare implicitly typed local variables

An implicitly typed local variable is created by using the var keyword followed by a variable initialization. For example:

A close-up of a logo

Description automatically generated

### Variables using the var keyword must be initialized

It's important to understand that the var keyword is dependent on the value you use to initialize the variable. If you try to use the var keyword without initializing the variable, you'll receive an error when you attempt to compile your code.

A screenshot of a chat

Description automatically generated

Get started with C# Part 1: <https://learn.microsoft.com/en-us/training/achievements/learn.languages.csharp-literals-variables.badge?username=NguyenAnhTuanVPI-3001&sharingId=2ECA20722B251E9F>

### Character escape sequences

An **escape character sequence** is an instruction to the runtime to insert a special character that will affect the output of your string. In C#, the escape character sequence begins with a backslash \ followed by the character you're escaping. For example, the \n sequence will add a new line, and a \t sequence will add a tab.

What if you need to insert a double-quotation mark in a literal string? If you don't use the character escape sequence, you'll confuse the compiler because it will think you want to terminate the string prematurely. The compiler won't understand the purpose of the characters after the second double-quotation mark.

C#

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

The .NET Editor will put a red squiggly line under World. But if you attempt to run the code anyway, you would see the following output:

(1,27): error CS1003: Syntax error, ',' expected

(1,27): error CS0103: The name 'World' does not exist in the current context

(1,32): error CS1003: Syntax error, ',' expected

To handle that situation, use the \" escape sequence:

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

What if you need to use the backslash for other purposes, like to display a file path?

Console.WriteLine("c:\source\repos");

Unfortunately, C# reserves the backslash for escape sequences, so if you run the code, the compiler will display the following error:

Output

(1,22): error CS1009: Unrecognized escape sequence

The problem is the sequence \s. The \r doesn't produce an error because it's a valid escape sequence for a carriage return. However, you don't want to use a carriage return in this context.

To solve this problem, you use the \\ to display a single backslash.

C#

Console.WriteLine("c:\\source\\repos");

Escaping the back slash character produces the output you intended:

Output

c:\source\repos

## Verbatim string literal

A verbatim string literal will keep all whitespace and characters without the need to escape the backslash. To create a verbatim string, use the @ directive before the literal string.

C#

Console.WriteLine(@" c:\source\repos

(this is where your code goes)");

Notice that the string spans two lines and the whitespace generated by this C# instruction is kept in the following output.

Output

c:\source\repos

(this is where your code goes)

## Recap

Here's what you've learned about formatting literal strings so far:

* Use character escape sequences when you need to insert a special character into a literal string, like a tab \t, new line \n, or a double quotation mark \".
* Use an escape character for the backslash \\ when you need to use a backslash in all other scenarios.
* Use the @ directive to create a verbatim string literal that keeps all whitespace formatting and backslash characters in a string.
* Use the \u plus a four-character code to represent Unicode characters (UTF-16) in a string.
* Unicode characters may not print correctly depending on the application.

## What is string interpolation?

String interpolation combines multiple values into a single literal string by using a "template" and one or more interpolation expressions. An **interpolation expression** is a variable surrounded by an opening and closing curly brace symbol { }. The literal string becomes a template when it's prefixed by the $ character.

In other words, instead of writing the following line of code:

C#

string message = greeting + " " + firstName + "!";

You can write this more concise line of code instead:

C#

string message = $"{greeting} {firstName}!";

ACHIEVEMENT  
<https://learn.microsoft.com/en-us/training/achievements/learn.languages.csharp-basic-formatting.badge?username=NguyenAnhTuanVPI-3001&sharingId=2ECA20722B251E9F>