Write your first C# code

The reason for taking this course is that knowledge of C# is a prerequisite for the "Get started with Azure Cosmos DB for NoSQL" module.

In this module we will:

- Write our first lines of C# code
- Use two different techniques to print a message as output
- Diagnose errors when code is incorrect
- Identify different C# syntax elements like operators, classes, and methods

The fact that diagnosing errors when code is incorrect makes me hopeful for what I'll learn in this module.

Exercise - Write your first code

The first exercise was simple. We printed `Hello World!` which then we commented out and then printed `Congratulations! You wrote your first lines of code.` with a new line after `Congratulations!`

To print those we used two commands:

- `// Console.WriteLine("Hello World!");`
- `Console.WriteLine("Congratulations!");`
- `Console.Write("You wrote your first lines of code.");`

Doing this helped us understand understand:

- The difference between `WriteLine` and `Write`
    - The difference is that `WriteLine` at the end of the line, it adds a line feed similar to pressing Enter (essentially it creates a new line at the end). Using `Write` the output will be printed on the same line.
- C# is case-sensitive
- We can comment a line of code out with two forward slashes `//`
- we declare the end of a line using `;`

Store and retrieve data using literal and variable values in C#

As a data engineer and scientist, many of the applications that you build in C# will require you to work with data.

Constant or Literal:

Sometimes that data will be hard-coded in your application. Hard-coded values are values that are constant and unchanged throughout the execution of the program.

For example, you may need to print a message to the user when some operation succeeds. A "success" message would likely be the same every time the application is executed. This hard-coded value can also be called a `constant`, or a `literal` value.

Hard-coded + Define variables = Formatted Messages:

Suppose you want to display a formatted message to the end user containing different types of data. the message would include hard-coded strings combined with information your app collects from the user. To display a formatted message you'll need to create both hard-coded values and define variables that can store data of a certain type, whether numeric, alphanumeric, and so on.

To define the character `b` we need to use single codes:

```
Console.WriteLine('b');
```

The single quote creates a character or `char` while earlier on we learned that double codes creates a `string` data type.

Integer literals

To display a numeric whole number (no fractions) value in the output console, you can use an `int literal`.

This is simple: `Console.WriteLine(123);`

Floating-point literals

A floating-point number is a number that contains a decimal, for example 3.14159. There are three data types that C# supports to represent decimal numbers:

- float
- double
- decimal

Each type supports varying degrees of precisions. Precision reflects the number of digits past the decimal that are accurate.

Float

To create a float literal, append the letter `F` (literal suffix) after the number:

- `Console.WriteLine(0.25F);` will output `0.25`
- `Console.WriteLine(0.25f);` will output `0.25`
    - Lower-case `f` will work the same way.

Double

To create a double literal, just enter a decimal number as the compiler defaults to a double literal when a decimal number is entered without a literal suffix.

- `Console.WriteLine(2.625);` will output 2.625

Decimal

To create a decimal literal, append the letter `m` after the number. Just likethe `f` the `m` is a literal suffix which tells the compiler that you want to work with a value of `decimal` type. As with float, for decimal literal suffix you can use lower-case `m` or upper-case `M`:

- `Console.WriteLine(12.39816m);`

Boolean literals

Booleans are simply `true` or `false` values. In C# you use the `bool` literal for that.

The below lines of code return `True` and `False`: `Console.WriteLine(true);`
`Console.WriteLine(false);`

Why emphasize data types?

`Data types play a central role in C#. This 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.`

Declare variables

When you need to work with data that isn't hard-coded, you'll declare a variable.

What is a variable?

A `variable` is a container for storing a type of value. Variables are important because their values can change, or vary, throughout the execution of a program. Variables can be assigned, read, and changed. You use variables to store values that you intend to use in your code.

A variable name is a human-friendly label that the compiler assigns to a memory address. when you want to store or change a value in that memory address, or whenever you want to retrieve the stored value, you just use the variable name you created.

Declare a variable

To create a new variable, you must first declare the data type of the variable, and then give it a name. For example:

- `string firstName;`
    - Above we are creating a new variable of type `string` called `firstName`. From now on, this variable can only hold string values.

You can choose any name for your variables, as long as it adheres to a few C# syntax rules for naming variables.

Variable name rules and conventions

A few important considerations about variable names:

- Variable names can contain alphanumeric characters and the underscore character. Special characters like the 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 thisIsCaleCase;`.
- 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 style like `string strValue;`. That advice is no longer current.

Some examples that follow the conventions for variable naming:

```
char userOption;

int gameScore;

decimal particlesPerMillion;

bool processedCustomer;
```

We can also assign value to a declared variable on one line: `string firstName = "Bob";`

Declare implicitly typed local variables

The C# compiler works behind the scenes to assist you as your write your code. It can infer your variable's data type by its initialised value. This feature is called implicitly typed local variables.

What are implicitly typed local variables?

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

```
var message = "Hello World!";
```

In this example, a string variable was created using the `var` keyword instead of the `string` keyword.

The `var` keyword tells the C# compiler that the data type is implied by the assigned value. After the type is implied, the variable acts the same as if the actual data type had been used to declare it. The `var` keyword is used to save on keystrokes when types are lengthy or when the type is obvious from the context.

In the above example, because the variable `message` is immediately set to the `string` value `Hello World!`, the C# compiler understands the intent and treats every instance of `message` as an instance of type `string`.

In fact, the `message` variable is typed to be a `string` and can never be changed. For example, consider the following code:

```
var message = "Hello World!";
message = 10.703m;
```

If you run this code, you'll see the following error message:

```
(2,11): error CS0029: Cannot implicitly convert type 'decimal' to 'string'
```

In C# the type of a variable is locked in at the time of declaration and therefore, will never be able to hold values of a different data type.

Variables using the `var` keyword must be initialised

If you used the `var` keyword without initialising the variable, you'll receive an error when you attempt to compile your code.

```
var message;
```

If you attempt to run the above you will receive the following error.

```
(1,5): error CS0818: Implicitly-typed variables must be initialized
```

Exercise

For the exercise that asks me to declare `Bob`, `3`, and `34.4` and print the below message in the console

```
Bob!You have3messages in your inbox.The temperature is 34.4 celsius.
```

My solution is the below:

```
var firstName = "Bob";
int numberOfMessages = 3;
float temperatureInCelsius = 34.4F;

Console.Write(firstName+"!");
Console.Write("You have" + numberOfMessages);
Console.Write("messages in your inbox.");
Console.Write("The temperature is " + temperatureInCelsius + " celsius.");
```

Perform basic string formatting in C#

To escape string literals we use the backslash `\`:

- Escape `"` `Console.WriteLine("Hello \"World\"!");`
- New line `Console.WriteLine("Hello\nWorld!");`
- Add tab `Console.WriteLine("Hello\tWorld!");`

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

```
Console.WriteLine(@"    c:\source\repos
        (this is where you code goes)");
```

The output of the above

```
    c:\source\repos
            (this is where your code goes)
```

Unicode escape characters

You can also add encoded characters in literal strings using the `\u` escape sequence, then a four-character code representing some character in Unicode (UTF-16).

```
// Kon'nichiwa World
Console.WriteLine("\u3053\u3093\u306B\u3061\u306F World!");
```

Some caveats:

- *Some consoles like the Windows command Prompt won't display all Unicode characters.*
    - *It will replace some characters with question mark characters instead.*

- *The examples used here are UTF-16.*
    - *Some characters require UTF-32 and therefore require a different escape sequence.*
        - *This is a complicated subject and we won't go there in this learning path.*

Format output using unicode escape characters

To complete the mock-up of the command-line tool, you'll add a phrase in Japanese that translates: "To generate Japanese invoices". Then you'll display a verbatim literal string that represents a command the user can enter. You'll also add some escape sequences for formatting.

We are adding the following to our application:

```
// To generate Japanese invoices:
// Nihon no seikyu-sho o seisei suru ni wa:
Console.Write("\n\n\u65e5\u672c\u306e\u8acb\u6c42\u66f8\u3092\u751f\u6210\
u3059\u308b\u306b\u306f\uff1a\n\t");
// User command to run an application
Console.WriteLine(@"c:\invoices\app.exe -j");
```

The final code:

```
Console.WriteLine("Generating invoices for customer \"Contoso Corp\" ...
\n");
Console.WriteLine("Invoice: 1021\t\tComplete!");
Console.WriteLine("Invoice: 1022\t\tComplete!");
Console.WriteLine("\nOutput Directory:\t");
Console.WriteLine(@"c:invoices");

// To generate Japanese invoices:
// Nihon no seikyu-sho o seisei suru ni wa:
Console.Write("\n\n\u65e5\u672c\u306e\u8acb\u6c42\u66f8\u3092\u751f\u6210\
u3059\u308b\u306b\u306f\uff1a\n\t");
// User command to run an application
Console.WriteLine(@"c:\invoices\app.exe -j");
```

The final output:

```
Generating invoices for customer "Contoso Corp" ...

Invoice: 1021           Complete!
Invoice: 1022           Complete!

Output Directory:
c:\invoices

日本の請求書を生成するには：
    c:\invoices\app.exe -j
```

Exercise - Combine Strings using string concatenation

In a nutshell, how do you combine literal strings and variables containing both text and numeric data? We use string concatenation.

We kinda did it by accident earlier one with the `+`. I was jumping ahead.

A simple example:

```
string firstName = "Bob";
string message = "Hello " + firstName;
Console.WriteLine(message);
```

Using a variable for `Hello`:

```
string firstName = "Bob";
string greeting = "Hello";
string message = greeting + " " + firstName + "!";
Console.WriteLine(message);
```

No need for the `message` variable:

```
string firstName = "Bob";
string greeting = "Hello";
Console.WriteLine(greeting + " " + firstName + "!");
```

Exercise - Combine strings using string interpolation

While string concatenation is simple and convenient, string interpolation is growing in popularity in situations where you need to combine many literal strings and variables into a single formatted message.

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 indicated by an opening and closing curly brace symbol `{ }`.

You can put any C# expression that returns a value inside the braces. The literal string becomes a template when it's prefixed by the `$` character.

So instead of writing the following:

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

You can write:

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

It reminds me of f-strings from Python

In the above example we save a few keystrokes, but we can imagine that it can be more useful when we work with a more complex operation. Also, many find the string interpolation syntax cleaner and easier to read.

Use string interpolation to combine a literal string and a variable value

```
string firstName = "Bob";
string message = $"Hello {firstName}!";
Console.WriteLine(message);
```

Use string interpolation with multiple values and literal strings

```
int version = 11;
string updateText = "Update to Windows";
string message = $"{updateText} {version}";
Console.WriteLine(message);
```

Avoid intermediate variables

```
int version = 11;
string updateText = "Update to Windows";
Console.WriteLine($"{updateText} {version}");
```

Combine verbatim literals and string interpolation

This can be very useful:

```
string projectName = "Frist-Project";
Console.WriteLine($@"C:\Output\{projectName}\Data");
```

Final exercise

My solution to the final exercise:

```
string projectName = "ACME";
string russianMessage =
"\u041f\u043e\u0441\u043c\u043e\u0442\u0440\u0435\u0442\u044c
\u0440\u0443\u0441\u0441\u043a\u0438\u0439
\u0432\u044b\u0432\u043e\u0434";
string englishMessage = "View English Output";

Console.WriteLine($@"{englishMessage}:
        c:\Exercise\{projectName}\data.txt
        ");

Console.WriteLine($@"{russianMessage}:
        c:\Exercise\{projectName}\ru-RU\data.txt");
```

The code produces outcome:

```
View English Output:
        c:\Exercise\ACME\data.txt

Посмотреть русский вывод:
        c:\Exercise\ACME\ru-RU\data.txt
```

The suggested solution:

```
string projectName = "ACME";
string englishLocation = $@"c:\Exercise\{projectName}\data.txt";
Console.WriteLine($"View English output:\n\t\t{englishLocation}\n");

string russianMessage =
"\u041f\u043e\u0441\u043c\u043e\u0442\u0440\u0435\u0442\u044c
\u0440\u0443\u0441\u0441\u043a\u0438\u0439
\u0432\u044b\u0432\u043e\u0434";
string russianLocation = $@"c:\Exercise\{projectName}\ru-RU\data.txt";
Console.WriteLine($"{russianMessage}:\n\t\t{russianLocation}\n");
```