

Contents

| | |
|---|----------|
| C# Fundamentals | 1 |
| Introduction to the C# Language | 1 |
| The Object-Oriented Paradigm | 2 |
| First Program | 3 |
| Hello World | 3 |
| Rules of C# Syntax | 5 |
| Conventions of C# Programs | 6 |
| Reserved Words and Identifiers | 6 |
| Write and WriteLine | 7 |
| Escape Sequences | 9 |

C# Fundamentals

Introduction to the C# Language

- C# is a managed language (as discussed previously)
 - Write in a high-level language, compile to intermediate language, run intermediate language in interpreter
 - Intermediate language is called CIL (Common Intermediate Language)
 - Interpreter is called .NET run-time
 - Standard library is called .NET Framework, comes with the compiler and run-time
- It is widespread and popular
 - It is "programming language of the year 2023"¹ in the very well-respected TIOBE Index².
 - It was the first in the list of "3 Future Programming Languages You Should Learn Between 2022 and 2030"³, because of the growing popularity of Unity⁴.
 - 7th most "desired / admired" language on StackOverflow⁵
 - .NET is the first most used "other" library/framework⁶
 - More insights on its evolution can be found in this blog post⁷.

¹<https://www.tiobe.com/tiobe-index/>

²https://en.wikipedia.org/wiki/TIOBE_index

³<https://betterprogramming.pub/3-future-programming-languages-you-should-learn-between-2022-and-2030-8a618a15eca6>

⁴<https://unity.com/>

⁵<https://survey.stackoverflow.co/2023/#programming-scripting-and-markup-languages>

⁶<https://survey.stackoverflow.co/2023/#most-popular-technologies-misc-tech>

⁷<https://dottutorials.net/stats-surveys-about-net-core-future-2020/#stackoverflow-surveys>

The Object-Oriented Paradigm

- C# is called an "object-oriented" language
 - Programming languages have different *paradigms*: philosophies for organizing code, expressing ideas
 - Object-oriented is one such paradigm, C# uses it
 - Meaning of object-oriented: Program mostly consists of *objects*, which are reusable modules of code
 - Each object contains some data (*attributes*) and some functions related to that data (*methods*)
- Object-oriented terms
 - **Class**: A blueprint or template for an object. Code that defines what kind of data the object will contain and what operations (functions) you will be able to do with that data
 - **Object**: A single instance of a class, containing running code with specific values for the data. Each object is a separate "copy" based on the template given by the class.
Analogy: A *class* is like a floorplan while an *object* is the house build from the floorplan. Plus, you can make as many houses as you would like from a single floorplan.
 - **Attribute**: A piece of data stored in an object.
Example: A *House* class has a spot for a color property while an house object has a color (e.g. "Green").
 - **Method**: A function that modifies an object. This code is part of the class, but when it is executed, it modifies only a specific object and not the class.
Example: A *House* class with a method to change the house color. Using this method changes the color a single house object but does not change the *House* class or the color on any other house objects.
- Examples:
 - A *Car Class*
 - * Attributes: Color, engine status (on/off), gear position
 - * Methods: Press gas or brake pedal, turn key on/off, shift transmission
 - A *Car Object*
Example: A *Porsche911* object that is Red, Engine On, and in 1st gear
 - An "Audio File" *Class* represents a song being played in a music player
 - * Attributes: Sound wave data, current playback position, target speaker device
 - * Methods: Play, pause, stop, fast-forward, rewind
 - An Audio File *Object*
Example: A *NeverGonnaGiveYouUp* object that is "rolled wave data", 0:00, speaker01

First Program

It is customary to start the study of a programming language with a “Hello World” program⁸, that simply displays “Hello World”. It is a simple way of seeing a first, simple example of the basic structure of a program. Here’s a simple “hello world” program in the C# language:

Hello World

```
!include code/snippets/helloWorld.cs
```

Features of this program:

- A multi-line comment: everything between the `/*` and `*/` is considered a *comment*, i.e. text for humans to read. It will be ignored by the C# compiler and has no effect on the program.
- A `using` statement: This imports code definitions from the *System namespace*, which is part of the .NET Framework (the standard library).
 - In C#, code is organized into **namespaces**, which group related classes together
 - If you want to use code from a different namespace, you need a `using` statement to “import” that namespace
 - All the standard library code is in different namespaces from the code you will be writing, so you’ll need `using` statements to access it
- A class declaration⁹
 - Syntax:

```
class [name of class]
{
    [body of the class]
}
```
 - All code between opening `{` and closing `}` is the *body* of the class named by the `class [name of class]` statement
- A method declaration
 - A collection of instructions with a name
 - Can be used by typing its name

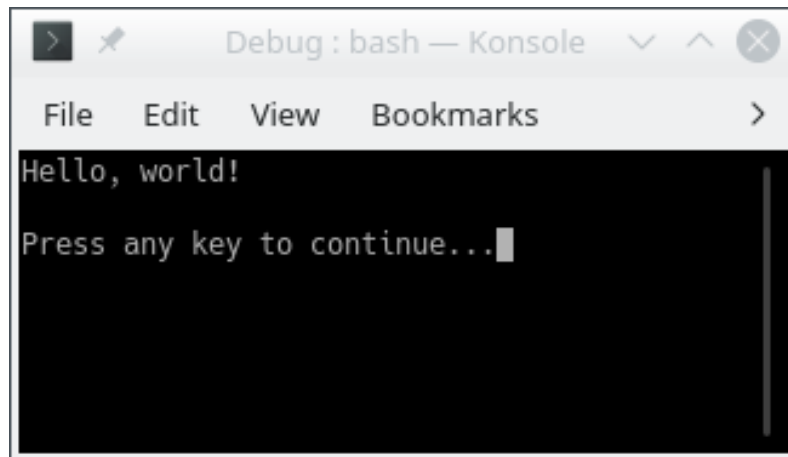
⁸https://www.wikiwand.com/en/%22Hello,_World!%22_program

⁹We use the notation [...] to denote what “should” be there, but this is just a place holder: you are not supposed to *actually* have the braces in the code.

- A method is similar to a paragraph, in that it can contain multiple statements, and a class is similar to a chapter, in that it can have multiple methods within its body.
 - A C# program requires a method called `Main`, and, in our example, is followed by empty parentheses (we will get to those later, but they are required)
 - Just like the class declaration, the body of the method begins with `{` and ends with `}`
- A statement inside the body of the method:

```
Console.WriteLine("Hello, world!"); // I'm an in-line comment.
```






- This is the part of the program that actually “does something”: It displays a line of text to the console:



- This statement contains a class name (`Console`), followed by a method name (`WriteLine`). It calls the `WriteLine` method in the `Console` class.
- The **argument** to the `WriteLine` method is the text “Hello, world!”, which is in parentheses after the name of the method. This is the text that gets printed in the console: The `WriteLine` method (which is in the standard library) takes an argument and prints it to the console.
- Note that the argument to `WriteLine` is inside double-quotes. This means it is a **string**, i.e. textual data, not a piece of C# code. The quotes are required in order to distinguish between text and code.
- A statement *must* end in a semicolon (the class header and method header are not statements)

- An in-line comment: All the text from the `//` to the end of the line is considered a comment, and is ignored by the C# compiler.

Rules of C# Syntax

- Each statement must end in a semicolon (`;`), except for some statements that we will study in the future that contains opening `{` and closing `}`, that do not end in a `;`.
 - Note that class and method declarations, as well as comments, are not statements¹⁰ and hence do not need to ends with a `;`. Typically, a method *contains* some statements, but it is not a statement.
- All words are case-sensitive
 - A class named `Program` is not the same as one named `program`
 - A method named `writeline` is not the same as one named `WriteLine`
- Braces and parentheses must always be matched
 - Once you start a class or method definition with `{`, you must end it with `}`
- Whitespace has *almost* no meaning
 - “Whitespaces” refer to spaces (sometimes denoted “ ”, “” or “”), tabs¹¹ (which consists in 4 spaces), and newlines (sometimes denoted “”, “” or “”)
 - There must be at least 1 space between words
 - Other than that, spaces and new lines are just to help humans read the code
 - Spaces are counted exactly if they are inside string data, e.g. `"Hello world!"` is different from `"Hello world!"`
 - Otherwise, entire program could be written on one line¹²; it would have the same meaning
- All C# applications must have a `Main` method
 - Name must match exactly, otherwise .NET run-time will get confused
 - This is the first code to execute when the application starts – any other code (in methods) will only execute when its method is called

¹⁰<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/statements>

¹¹https://www.wikiwand.com/en/Tab_key#Tab_characters

¹²Well, if there are no in-line comments in it. Can you figure out why?

Conventions of C# Programs

- Conventions: Not enforced by the compiler/language, but expected by humans
 - Program will still work if you break them, but other programmers will be confused
- Indentation
 - After a class or method declaration (header), put the opening { on a new line underneath it
 - Then indent the next line by 4 spaces, and all other lines “inside” the class or method body
 - De-indent by 4 spaces at end of method body, so ending } aligns vertically with opening {
 - Method definition inside class definition: Indent body of method by another 4 spaces
 - In general, any code between { and } should be indented by 4 spaces relative to the { and }
- Code files
 - C# code is stored in files that end with the extension “.cs”
 - Each “.cs” file contains exactly one class
 - The name of the file is the same as the name of the class (Program.cs contains class Program)

Note that some of those conventions are actually rules in different programming languages (typically, the last two regarding code files are mandatory rules in java).

Reserved Words and Identifiers

- Reserved words: Keywords in the C# language
 - Note they have a distinct color in the code sample and in your IDE
 - Built-in commands/features of the language
 - Can only be used for one specific purpose; meaning cannot be changed
 - Examples:
 - * using
 - * class
 - * public
 - * private
 - * namespace
 - * this
 - * if
 - * else
 - * for
 - * while

- * `do`
- * `return`
- There is no need to memorize the whole list of keywords¹³, as we will only introduce the ones we need on a “per need” basis.
- Identifiers: Human-chosen names
 - Names for classes (`Rectangle`, `ClassRoom`, etc.), variables (age, name, etc.), methods (`ComputeArea`, `GetLength`, etc), namespaces, etc.
 - Some have already been chosen for the standard library (e.g. `system`, `Console`, `WriteLine`, `Main`), but they are still identifiers, not keywords
 - Rules for identifiers:
 - * Must not be a reserved word
 - * Must contain only letters (lower case, from a to z, or upper case, from A to Z), numbers (made of digits from 0 to 9), and underscore (`_`). But they cannot contain spaces.
 - * Must not begin with a number
 - * Are case sensitive
 - * Must be unique (you cannot re-use the same identifier twice in the same scope – a concept we will discuss later)
 - Conventions for identifiers
 - * Should be descriptive, e.g. “`AudioFile`” or “`userInput`” not “`a`” or “`x`”
 - * Should be easy for humans to read and type
 - * If name is multiple words, use CamelCase¹⁴ (or its variation Pascal case¹⁵) to distinguish words, e.g. `myHeightInMeters` or `distanceFromEarthToMoon`.
 - * Class and method names should start with capitals, e.g. “`class AudioFile`”
 - * Variable names should start with lowercase letters, then capitalize subsequent words, e.g. “`myFavoriteNumber`”

Write and WriteLine

- The `WriteLine` method
 - We saw this in the “Hello World” program: `Console.WriteLine("Hello World!");` results in “Hello World!” being displayed in the terminal
 - In general, `Console.WriteLine("text");` will display the text but not the “\n” in the terminal, then *start a new line*

¹³<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/>

¹⁴https://www.wikiwand.com/en/Camel_case

¹⁵<https://www.c-sharpcorner.com/UploadFile/8a67c0/C-Sharp-coding-standards-and-naming-conventions/>

- This means a second `Console.WriteLine` will display its text on the next line of the terminal. For example, this program:

```
!include code/snippets/writelineTwoLines.cs
```

will display the following output in the terminal:

```
Hello
World!
```

- Methods with multiple statements
 - Note that our two-line example has a `Main` method with multiple statements
 - In C#, each statement must end in a semicolon
 - Class and method declarations are not statements
 - Each line of code in your .cs file is not necessarily a statement
 - A single invocation/call of the `WriteLine` method is a statement
- The `Write` method
 - `Console.WriteLine("text")` prints the text, then starts a new line in the terminal – it effectively “hits enter” after printing the text
 - `Console.Write("text")` just prints the text, without starting a new line. It’s like typing the text without hitting “enter” afterwards.
 - Even though two `Console.Write` calls are two statements, and appear on two lines, they will result in the text being printed on just one line. For example, this program:


```
!include code/snippets/writeTwoLines.cs
```

will display the following output in the terminal:

```
HelloWorld!
```
 - Note that there is no space between “Hello” and “World!” because we did not type one in the argument to `Console.Write`
- Combining `Write` and `WriteLine`
 - We can use both `WriteLine` and `Write` in the same program
 - After a call to `Write`, the “cursor” is on the same line after the printed text; after a call to `WriteLine` the “cursor” is at the beginning of the next line
 - This program:


```
!include code/snippets/writeAndWriteline.cs
```


will display the following output in the terminal:

```
Hello world!  
Welcome to CSCI 1301!
```

Escape Sequences

- Explicitly writing a new line
 - So far we've used `WriteLine` when we want to create a new line in the output
 - The **escape sequence** `\n` can also be used to create a new line – it represents the “newline character,” which is what gets printed when you type “enter”
 - This program will produce the same output as our two-line “Hello World” example, with each word on its own line:

```
!include code/snippets/writeWithNewline.cs
```

- Escape sequences in detail
 - An **escape sequence** uses “normal” letters to represent “special”, hard-to-type characters
 - `\n` represents the newline character, i.e. the result of pressing “enter”
 - `\t` represents the tab character, which is a single extra-wide space (you usually get it by pressing the “tab” key)
 - `\"` represents a double-quote character that will get printed on the screen, rather than ending the text string in the C# code.
 - * Without this, you couldn't write a sentence with quotation marks in a `Console.WriteLine`, because the C# compiler would assume the quotation marks meant the string was ending
 - * This program will not compile because `in quotes` is not valid C# code, and the compiler thinks it is not part of the string:

```
// Incorrect Code  
class Welcome  
{  
    static void Main()  
    {  
        Console.WriteLine("This is "in quotes");  
        // This is parsed as if the string was "This is "  
        // followed by in quotes, which is not valid C#,  
    }  
}
```

```

        // followed by the empty string "".
    }
}

```

- * This program will display the sentence including the quotation marks:

```
!include code/snippets/escapeQuotes.cs
```

- Note that all escape sequences begin with a backslash character (\), called the "escape character"
- General format is \[key letter] – the letter after the backslash is like a "keyword" indicating which special character to display. You can refer to the full list on microsoft documentation¹⁶.
- If you want to put an actual backslash in your string, you need the escape sequence \\, which prints a single backslash
 - * This will result in a compile error because \U is not a valid escape sequence:

```
Console.WriteLine("Go to C:\Users\Edward");
```

- * This will display the path correctly:

```
Console.WriteLine("Go to C:\\Users\\Edward");
```

¹⁶<https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/strings/#string-escape-sequences>