CHAPTER 18

INTRODUCING OUR LANGUAGE: C#

This chapter introduces you to the key features of C# and describes some important reasons why I chose it as the language for this book. It also examines the basic syntax of C#, explaining what is meant by the structure of some simple C# statements.

By the end of this chapter, you will better understand C# and be ready to tackle the more in-depth chapters that follow.
Understanding the Features of C#

As covered in Chapter 16, "Thinking in Digital Systems," programming consists of giving the computer a series of simple commands, and C# is the language through which we will do so. However, many different programming languages exist out there, each of which has benefits and drawbacks. Some of the features of C# are that it is:

- A compiled language
- Managed code
- Strongly typed
- Function based
- Object oriented

Each of these features is described further in the following sections, and each will help you in various ways.

C# Is a Compiled Language

When most people write computer programs, they are not actually writing in a language that the computer itself understands. In fact, each computer chip on the market has a slightly different set of very simple commands that it understands, known as machine language. This language is very, very fast for the chip to execute, but it is incredibly difficult for a person to read. For example, the machine language line

```
000000 00001 00010 00110 00000 100000
```

would certainly mean something to the right computer chip, but it means next to nothing to human readers. You might have noticed, however, that every character of that machine code is either a 0 or 1. That’s because all the more complex types of data—numbers, letters, and so on—have been converted down to individual bits of data (i.e., ones or zeros). If you’ve ever heard of people programming computers using punch cards, this is exactly what they were doing: For most formats of binary punch cards, physically punching a hole in card stock represented a one, and an unpunched hole represented a zero.

For people to be able to write code more easily, human-readable programming languages—sometimes called authoring languages—were created. You can think of an authoring language as an intermediate language meant to act as a go-between from you to the computer. Authoring languages like C# are logical and simple enough for a computer to interpret while also being close enough to written human languages to allow programmers to easily read and understand them.
There is also a major division in authoring languages between compiled languages such as BASIC, C++, C#, and Java, and interpreted languages such as JavaScript, Perl, PHP, and Python (see Figure 18.1).

In an interpreted language, authoring and executing code is a two-step process:

- The programmer writes the code.
- Then, each time any player plays the game, the code is converted from the authoring language to machine language in real time on the player’s machine.

The good thing about this is that it enables code portability, because the authoring code can be interpreted specifically for the type of computer on which it is running. For example, the JavaScript of a given web page will run on almost any modern computer regardless of whether the computer is running macOS, Windows, Linux, or one of many mobile operating systems like iOS, Android, Windows Phone, and so on. However, this flexibility also causes the code to execute more slowly due to the time required to interpret the code on the player’s computer, the authoring language not being well optimized for the device on which it will run, and a host of other reasons. Because the same interpreted code is run on all devices, optimizing for the specific device on which it happens to be running is impossible. This is the reason why 3D games created in an interpreted language like JavaScript generally run much more slowly than those created in a compiled language, even when running on the same computer.

In a compiled language, such as C#, the programming process comprises three separate steps:

- The programmer writes the code in an authoring language like C#.
- A compiler converts the code from the authoring language to a compiled application in machine language for a specific kind of machine.
- The computer executes the compiled application.
This added middle process of compilation converts the code from the authoring language into an executable (that is, an application or app) that a computer can run directly without needing an interpreter. Because the compiler has both a complete understanding of the program and a complete understanding of the execution platform on which the program will run, it is able to incorporate many optimizations into the process. In games, these optimizations translate directly into higher frame rates, more detailed graphics, and more responsive interactions. Most high-budget games are authored in a compiled language because of this optimization and speed advantage, but this means that a different executable must be compiled for each execution platform.

In many cases, compiled authoring languages are only suited for specific execution platforms. For instance, Objective-C is Apple Computer’s proprietary authoring language for making applications for both macOS and iOS. This language is based on C (a predecessor of C++), but it includes a number of features that are unique to macOS or iOS development. Similarly, XNA was a flavor of C# developed by Microsoft specifically to enable students to author games for both Windows-based personal computers and the Xbox 360.

As mentioned in Chapter 17, “Introducing the Unity Development Environment,” Unity uses either C# or a JavaScript flavor named UnityScript to create games. Either of these languages are compiled into a Common Intermediate Language (CIL) in an additional compilation step, and that CIL is then compiled to target any number of platforms, from iOS to Android to macOS, Windows PC, game consoles such as the PlayStation and Xbox, and even interpreted languages such as WebGL (a specific form of JavaScript used in web pages). This additional CIL step ensures that Unity programs can be compiled across many platforms regardless of whether they are written in C# or UnityScript.

The ability to write once and compile anywhere is not unique to Unity, but it is one of Unity Technologies’ core goals for the Unity game engine, and it is better integrated into Unity than any other game development software I have seen. However, as a game designer, you will still need to think seriously about the design differences between a game meant for a handheld phone controlled by touch, a game meant to run on a personal computer controlled by mouse and keyboard, or a game built for virtual or augmented reality, so you will usually have slightly different code for the different platforms.

C# Is Managed Code

More traditional compiled languages such as BASIC, C++, and Objective-C require programmers to directly manage computer memory, obliging a programmer to manually allocate and de-allocate memory any time she creates or destroys a variable.1 If a programmer doesn't

---

1. Memory allocation is the process of setting aside a certain amount of Random-Access Memory (RAM) in the computer to enable it to hold a chunk of data. While computers now often have hundreds of gigabytes (GB) of hard drive space, they still usually have less than 20GB of RAM. RAM is much faster than hard drive memory, so all applications pull assets like images and sounds from the hard drive, allocate some space for them in RAM, and then store them in RAM for fast access.
manually de-allocate RAM in these languages, her programs will have a “memory leak” and eventually allocate more than the maximum amount of the computer’s RAM, causing it to crash.

Luckily for us, C# is managed code, which means that the allocation and de-allocation of memory is handled automatically. You can still cause memory leaks in managed code, but it is more difficult to do so accidentally.

C# Is Strongly Typed

Later chapters cover variables in more detail, but there are a couple of things that you should know about them now. First, a variable is just a named container for a value. For instance, in algebra, you might have seen an expression like this:

\[ x = 5 \]

In this one line, you have created a variable, named it \( x \), and assigned it the value 5. Later, if I asked you the value of \( x+2 \), I’m sure you could tell me that the answer is 7 because you remember that \( x \) was holding the value 5 and know to add 2 to that value. That is exactly what variables do for you in programming.

In most interpreted languages, like JavaScript, a single variable can hold any kind of data. The variable \( x \) could hold the number 5 one minute, an image the next, and a sound file thereafter. This capability for a single variable to hold any type of value is what is meant when we say that a programming language is weakly typed.

C#, in contrast, is strongly typed. This means that when you initially create a variable, you must tell it at that moment what kind of value it can hold:

```csharp
int x = 5;
```

In the preceding statement, you have created a variable named \( x \) that it is exclusively allowed to hold \( int \) values (that is, positive or negative numbers without a decimal point), and assigned it the integer value 5. Although it might seem like strong typing would make programming more difficult, the use of strong typing enables the compiler to make several optimizations and makes it possible for the authoring environment, MonoDevelop, to perform real-time syntax checking on the code you write (much like the grammar checking that is performed by Microsoft Word). This also enables and enhances code-completion, a technology in MonoDevelop that enables it to predict the words you’re typing and provide you with valid completion options based on the other code that you’ve written. With code-completion, if you’re typing and see MonoDevelop suggest the correct completion of the word, you simply press Tab to accept the suggestion. When you’ve become used to this, it can save you hundreds of keystrokes every minute.

---

2. One disadvantage of managed code is that it makes controlling exactly when memory is deallocated and reclaimed very difficult. Instead, memory is automatically reclaimed in a process called garbage collection. This can sometimes lead to a hitch in the frame rate of a game on less powerful devices such as cell phones, but it’s usually not noticeable.
C# Is Function-Based

In the early days of programming, a program was composed of a single series of commands. These programs were run directly from beginning to end much like the directions you would give to a friend who was trying to drive to your house:

1. From school, head north on Vermont.
2. Head west on I-10 for about 7.5 miles.
3. At the intersection with I-405, take the 405 south for 2 miles.
4. Take the exit for Venice Blvd.
5. Turn right onto Sawtelle Blvd.
6. My place is just north of Venice on Sawtelle.

As authoring languages improved, repeatable sections were added to programming in the form of things like loops (a section of code that repeats itself) and subroutines (an otherwise inaccessible section of code that is jumped to, executed, and then returned from).

The development of procedural languages (i.e., those that make use of functions) allowed programmers to name chunks of code and thereby encapsulate functionality (that is, group a series of actions under a single function name). For example, if in addition to giving someone detailed directions to your house as described in the preceding list, you also asked him to pick up some milk for you on the way, he would know that if he saw a grocery store on the way, he should stop the car, get out, walk to find milk, pay for it, return to his car, and continue on his way to your house. Because your friend already knows how to buy milk, you just need to request that he do so rather than giving him explicit instructions for every tiny step. This could look something like this:

"If you see a store on the way, could you please BuySomeMilk()?"

This statement encapsulates all the instructions to buy milk into the single function named BuySomeMilk(). You can do the same thing in any procedural language. When the computer is processing C# and encounters a function name followed by parentheses, it will call that function (that is, it will execute all the actions encapsulated in the function). You will learn much more about functions in Chapter 24, "Functions and Parameters."

The other fantastic thing about functions is that after you have written the code for the function BuySomeMilk() one time, you should never have to write it again. Even if you're working on a completely different program, you can often copy and paste functions like BuySomeMilk() and reuse them without having to write the whole thing again from scratch. The C# script named Utils.cs that you will see in several of the tutorials in this book includes several reusable functions.

---

3. There are also functional languages like Lisp, Scheme, Mathematica (Wolfram Language), and Haskell, but for these functional languages, "functional" means something different than the capabilities we have to write functions in C#.
C# Is Object-Oriented

Many years after functions were invented, the idea of *Object-Oriented Programming* (OOP) was created. In OOP, not only functionality but also data are encapsulated together into something called an *object*, or more correctly a *class*. This is covered extensively in Chapter 26, “Classes,” but here’s a metaphor for now.

Consider a group of various animals. Each animal has specific information that it knows about itself. Some examples of this data could be its species, age, size, emotional state, level of hunger, current location, and so on. Each animal also has certain things that it can do: eat, move, breath, etc. The data about each animal are analogous to variables in code, and the actions that the animal can perform are analogous to functions.

Before OOP, an animal represented in code could hold information (i.e., variables) but could not perform any actions. Those actions were performed by functions that were not directly connected to the animal. A programmer could write a function named `Move()` that could move any kind of animal, but she would have to write several lines of code in that function that determined what kind of animal it was and what type of movement was appropriate for it. For example, dogs walk, fish swim, and birds fly. Any time the programmer added a new animal, she was required to change `Move()` to accommodate the new type of locomotion, and `Move()` would thereby grow larger and more complex with the addition of each new animal.

Object orientation changed all of this by introducing the ideas of *classes* and *class inheritance*. A *class* combines both variables and functions into one whole object. In OOP, instead of having a huge `Move()` function that can handle any animal, a much smaller and more specific `Move()` function is attached to each animal. This eliminates the need for you to expand `Move()` every time you add a new type of animal, and it eliminates the need for all the type-checking of animal types in the non-OOP version of `Move()`. Instead, each new animal class is given its own small `Move()` function when it is created.

Object orientation also includes the concept of *class inheritance*. This enables classes to have *subclasses* that are more specific, and it allows the subclasses to either inherit or override functions in their *superclasses*. Through inheritance, a single *Animal* class could be created that included declarations of all the data types that are shared by all animals. This class would also have a `Move()` function, but it would be nonspecific. In subclasses of *Animal*, like *Dog* or *Fish*, the function `Move()` could be overridden to cause specific behavior like walking or swimming. This is a key element of modern game programming, and it will serve you well when you want to create something like a basic *Enemy* class that is then further specified into various subclasses for each individual enemy type that you want to create.
Reading and Understanding C# Syntax

Just like any other language, C# has a specific syntax that you must follow. Take a look at these example statements in English:

- The dog barked at the squirrel.
- At the squirrel the dog barked.
- The dog at the squirrel. barked
- barked The dog at the squirrel.

Each of these English statements has the same words and punctuation, but they are in a different order, and the punctuation and capitalization is changed. Because you are familiar with the English language, you can easily tell that the first is correct and the others are just wrong. Another way of examining this is to look at it more abstractly as just the parts of speech:

- [Subject] [verb] [object].
- [Object] [subject] [verb].
- [Subject] [object]. [verb]
- [verb] [Subject] [object].

When parts of speech are rearranged like this, doing so alters the syntax of the sentence, and the latter three sentences are incorrect because they have syntax errors.

Just like any language, C# has specific syntax rules for how statements must be written. Let’s examine this simple statement in detail:

```csharp
int x = 5;
```

As explained earlier, this statement does several things:

- Declares a variable named `x` of the type `int`
  
  Any time a statement starts with a variable type, the second word of the statement becomes the name of a new variable of that type (see Chapter 20, "Variables and Components"). This is called declaring a variable.

- Assigns `x` the value `5`
  
  The `=` symbol is used to assign values to variables (which is also called initializing a variable if it is the first time that any value has been assigned to the variable). When you do this, the variable name is on the left, and the value assigned is on the right.

- Ends with a semicolon (`;`)
  
  Every simple statement in C# must end with a semicolon (`;`). This is similar in use to the period at the end of sentences in the English language.
Why not end C# statements with a period? Computer programming languages are meant to be very clear. The period is not used at the end of statements in C# because it is already in use in numbers as a decimal point (for example, the period in 3.14159). For clarity, the only use of the semicolon in C# is to end statements.

Now, let’s add a second simple statement:

```csharp
    int x = 5;
    int y = x * (3 + x);
```

The second line does the following:

- Declares a variable named `y` of the type `int`
- Adds `3 + x` (which is 3 + 5, for a result of 8)
  Just like in algebra, order of operations follows parentheses first, meaning that `3 + x` is evaluated first because it is surrounded by parentheses. The sum is 8 because the value of `x` was set to 5 in the previous statement. In Appendix B, "Useful Concepts," you can read the section Operator Precedence and Order of Operations, to learn more about order of operations in C#, but the main thing to remember for your programs is that if you have any doubt about the order in which things will occur, you should use parentheses to remove that doubt (and increase the readability of your code).
- Multiplies `x * 8` (x is 5, so the result is 40)
  If there had been no parentheses, the order of operations would have handled multiplication and division before addition and subtraction. This would have resulted in `x * 3 + 5`, which would become `5 * 3 + 5`, then `15 + 5`, and finally `20`.
- Assigns the value `40` to `y`
- Ends with a semicolon (`;`)

This chapter finishes with a breakdown of one final couplet of C# statements. In this example, the statements are now numbered. Line numbers can make referencing a specific line in code much simpler, and my hope is that they will make it easier for you to read and understand the code in this book when you’re typing it into your computer. The important thing to remember is that **you do not need to type the line numbers** into MonoDevelop. MonoDevelop automatically numbers (and rennumbers) your lines as you work:

```csharp
1    string greeting = "Hello World!";
2    print( greeting );
```
These two statements deal with strings (a series of characters like a word or sentence) rather than integers. The first statement (numbered 1):

- Declares a variable named `greeting` of the type `string`.
  
  A string is another type of variable just like `int`.  

- Assigns the string value "Hello World!" to `greeting`.
  
  The double quotes around "Hello World!" tell C# that the characters in between them are to be treated as a string literal and not interpreted by the compiler to have any additional meaning. Putting the string literal "x = 10" in your code will not assign the value 10 to `x` because the compiler knows to ignore all string literals between quotes and does not try to interpret them as C# code.  

- Ends with a semicolon (;)

The second statement (numbered 2):

- Calls the function `print()`.
  
  As discussed earlier, functions are named collections of actions. When a function is called, the function executes the actions it contains. As you might expect, `print()` contains actions that will output a string to the Console pane. Any time you see a word in code followed by parentheses, it is either calling or defining a function. Writing the name of a function followed by parentheses calls the function, causing that functionality to execute. You'll see an example of defining a function in the next chapter.

- Passes `greeting` to `print()`.
  
  Some functions just do things and don’t require parameters, but many require that you pass something in. Any variable placed between the parentheses of a function call is passed into that function as an argument. In this case, the string `greeting` is passed into the function `print()`, and the characters Hello World! are output to the Console pane.

- Ends with a semicolon (;)
  
  Every simple statement ends with a semicolon.

**Summary**

Now that you understand a little about C# and about Unity, it’s time to put the two together into your first program. The next chapter takes you through the process of creating a new Unity project, making a few C# scripts, adding some simple code to those scripts, and manipulating 3D GameObjects.