

3

First Steps

This section will cover many basic terms and some of the basic ideas behind writing proper code. C# is among the many different programming languages which use the same words and concepts to convey instructions to the computer. Once we get through the first few chapters we'll be introduced to some more of the basics which are directly related to reading and writing code.

Before we can get to that we'll need to cover some of the terms and concepts that hold true for many different programming languages, not just C#. The methods and systems that allow converting words into executable code require fairly strict rules.

When writing English or chatting online we tend to ignore formatting and punctuation. The brevity allows for faster communication, though only because we as humans have learned many new words and can use context to better interpret the contractions and acronyms in the written form of English used online or in text messages. Computers aren't so smart as to be able to interpret our words so easily.

Unity 3D is a 3D game engine. By 3D we mean three dimensions, which means we have an x , y , and z coordinate space. 2D or two-dimensional game engines like Game Maker only use x and y . This is the difference between a cube and a square. One has depth to each shape and the other does not.

Later on we will be working with three-dimensional vectors. Unity 3D uses Euclidian vectors to describe where objects are located in a scene. When a position is described in terms of $x = 1$, $y = 2$, and $z = 3$, you should understand that this describes a position in 3D space.

3.1 What Will Be Covered in This Chapter

We'll go over the specifics of the most basic parts of the C# programming language. The C# language has a great deal in common with other programming languages, so much of the knowledge here is transferable to many other programming languages:

- Tokens, the smallest elements of C#.
- How statements, giving tokens meaning.
- Important words that exist in C# with specific functions.
- How to use white space and proper formatting.
- How statements are grouped together into a code block.
- How classes are organized and what they're used for.
- What a variable is and how it's used.
- Types and how they behave when converted between one another.
- Commenting on code and leaving messages to yourself and others within the code.

3.2 Review

In Chapter 2 we looked at some short sample code examples. Before going too much further we'll want to make sure that we are able to open each of the different projects in Unity 3D to follow along.

Near the end of Chapter 2 we had to venture onto GitHub to grab the projects. There are some alternative locations on the Internet to find the projects for this book. These "cloud" services, as they are called,

are available to everyone with an Internet connection. They're all public and openly available and they can be updated. If you discover a typo or any errors in the process that weren't intended then you can feel free to find the author's web site and post a note to the forum.

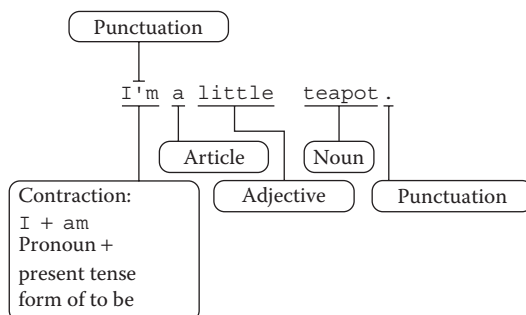
Before continuing you should be able to run Unity 3D, create and open C# files, and attach the C# files to an object in a scene. Once all of this is done, you should be able to press the Play in Editor button without any errors. You should also be able to manipulate objects in the scene using the various navigation tools in the editor.

If any of this seems unfamiliar go back and review Chapter 2; we'll be doing quite a lot here which depends on your being able to move objects and create and edit C# files. It's time to get into what code is, so keep calm and carry on!

3.3 Tokens

In written English the smallest elements of the language are letters and numbers. Individually, most letters and numbers lack specific meaning. The next larger element after the letter is the word. Each word has more meaning, but complex thoughts are difficult to convey in a single word.

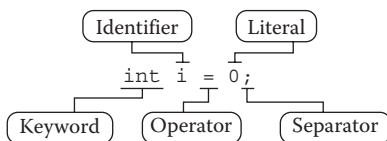
To communicate a thought, we use sentences. The words in a sentence each have a specific use, as seen in the diagram below. To convey a concept we use a collection of sentences grouped into a paragraph. And to convey emotion we use a collection of many paragraphs organized into chapters. To tell a story we use a book, a collection of chapters.



Programming has similar organizational mechanisms. The smallest meaningful element is a token, followed by statements, code blocks, functions, followed by classes and namespaces and eventually a program, or in our case a game. We will begin with the smallest element and work our way up to writing our own classes. However, it's important to know the very smallest element to understand how all the parts fit together before we start writing any complex code.

3.3.1 Writing C#

C# is an English-based programming language; like any other C-like programming language it can be broken into tokens, the smallest meaningful fragment of text that the computer can understand. Tokens are made of a single character or a series of characters. For instance, in a simple statement like the following there are five tokens.



Tokens can be categorized as a *keyword*, *identifier*, *literal*, *operator*, *separator*, and *white space*. The above statement contains five tokens with spacing provided with white space. The keyword `int` is

followed by the identifier `i`. This is followed by an operator `=` which assigns the identifier on the left of the operator and the value of the literal `0` on the right. The last token `;` is a separator which ends the statement. White spaces and comments, which we have yet to cover, are not considered tokens, although they can act as separators.

It's important that proper formatting or style be used. Let's examine a sample of code:

```
int i = 0; int j = 1;
```

The code shown above is two statements, the separator keeps the computer from misreading the code as a single statement. However, there are humans involved with writing code, and thus code needs to be readable by humans, or any other life-form which might read your code. Proper code means following a specific formatting style. We'll dive into more about proper code style later in this chapter.

Each group of characters, or text, is converted by a *lexical analyzer*, sometimes called a *lexer*, in the computer and turned a *symbol*. This process is called *tokenization*, or as a computer scientist would say, a lexer tokenizes your code into symbols. Once tokenized, the symbols are *parsed*; this process organizes the tokens into instructions for the computer to follow. This unique vocabulary does frame programmers as a strange group of alien beings with a language all to themselves; I promise, however, conversations about lexical analyzers don't come up very often when talking with most programmers.

This may seem like a great deal of work and we are jumping ahead into more complex computer science topics, but all of this happens behind the scenes so you don't need to see any of this taking place. It's important to understand what a compiler does, and how it works so you can write code in a way that the computer can understand. This is why a simple typo will stop the lexer from building code. We'll get to writing code to be analyzed, tokenized, and parsed soon enough.

A computer can't interpret or guess at what it is you're trying to do. Therefore, if you mistype `int i = 0:` and not `int i = 0;` the last token cannot be converted by the lexer into a proper symbol, and thus this statement will not be parsed at all. This code results in an error before the code is even compiled.

3.3.2 Separator Tokens

The most common separator is the `;`, semicolon. This mark of punctuation is used to end a code statement. Other separators come in pairs. Curly braces are used in pairs to separate groups of code statements. The opening curly brace is the `{` with the pointy end pointed to the outside of the group; this is paired with the `}` which closes the curly brace pair.

Parentheses start with the `(` and end with the `)` while square brackets start with `[` and end with `]`. The different opening and closing braces, brackets, and parentheses have specific purposes and are used to help identify what they are being used for. There are angle brackets `<` and `>` as well; these are used to surround data types. And don't forget both single quotes `' '` and double quotes `" "`, which are used in pairs as well to surround symbols called strings.

NOTE: In many word processors a beginning and an ending quote (“ and ”) are often used. The lexical analyzer doesn't recognize these smart quotes, so the text editor for programming uses straight quote marks (") instead. These subtle changes aren't so subtle once you realize that letters, numbers, and every other character used in programming are all parsed as American Standard Code for Information Interchange (ASCII) or unicode transformation format (UTF)-8, 16, or 32. Both ASCII and UTF are names given to the database of characters that the computer uses to read text. Computers don't use eyes to read. Rather, they use a database of numbers to look up each character being used. Therefore, to a computer “ looks more like `0x201C` and " like `0x0022`, if we were using UTF-16.

Curly braces are used in code to separate statements. Parentheses are usually used to accept data. Square brackets are often used to help identify arrays. We'll get into arrays later, but imagine something like a spreadsheet or a graph paper till we get to their explanation.

```
int[] arrayNumbers = {1, (int) 3.0, 9000};
```

The above statement assigns three numbers to an array. The curly braces contain a pair of statements separated by a comma. The second statement in the curly braces is converting a number with a dot (.) into a number without a dot in it. This will take a bit of explanation, but it's good to get used to seeing this sort of thing. We'll get to what an array is soon enough, as well as a dot operator.

Often, when learning to program you'll see many strange tokens which might not seem to have any meaning. It's up to you to observe every token you come across and try to understand what it does. If none of the statements makes sense that's fine. At this point you're not expected to know what all this means yet, but you soon will.

3.3.3 Operator Tokens

Operators are like keywords but use only a few non-word characters. The colon or semi-colon is an operator; unlike keywords, operators change what they do based on where they appear, or how they are used. This behavior shifting is called operator overloading, and it's a complex subject that we'll get into later. It's also important to know that you can make up your own operators when you need to.

Commas in C# have a different meaning. They're used to separate different data values like `Vector3(x, y, z)` or `for(int i = 0, j = 1; i++, j++)`.

```
public class Example : MonoBehaviour {
```

Here, the `:` was used following the name of the class to inform the compiler we're going to be adding to a preexisting class called `MonoBehaviour`. In this case, `:` will tell our class to extend `MonoBehaviour`. This means to build upon a class which already exists. We're creating a new child object related to `MonoBehaviour`; this is a class that was created by the developers at Unity 3D.

3.3.4 Other Operator Tokens

Operators in general are special characters that take care of specific operations between variables. Notation or the order in which operators and variables appear is normally taught with the following notation or mathematic grammar: $a + b = c$. However, in programming the following is preferred: `c = a + b`; the change is made because the `=` operator in C# works differently from its function commonly taught in math class.

The left side of the `=` operator is the assignment side, and the right side is the operation side. This puts the emphasis on the value assigned rather than how it's assigned. The result is more important than how the problem is solved since the computer is doing the math. In programming the above example should be read as "c is assigned a plus b."

3.3.5 Literals

Literals come in many different forms; similar to operators which appear as things like `+`, `-`, or `=` literals are the tokens that often go between them. Numbers are considered literals, or tokens that are used in a literal manner. Numbers can be called numeric literals.

When something is put into quotes, as in "I'm a literal," you are writing a string literal. Literals are common throughout nearly all programming languages and form the base of C#'s data types. Right now, we're dealing with only a few different types of literals, but there are many more we will be aware of soon.

3.3.6 Transitive and Non-Transitive Operations

In math class it's sometimes taught how some operations are transitive and others not. For instance $2 + 3 + 4$ and $4 + 3 + 2$ result in the same values. Where each number appears between the operators doesn't matter, so the `+` operator is considered transitive. This concept holds true in C#. Results change when we start to mix operators.

```
int a = 1 + 2 - 4 + 7; //a = 6
int b = 7 + 4 - 2 + 1; //b = 10
```

This code fragment shows two different results using the same operators but with different number placement. We'll see how to overcome these problems later on when we start using them in our game. Operator order is something to be aware of, and it's important to test each step as we build up our calculations to make sure our results end up with values we expect.

To ensure that the operations happen as you expect, either you can use parentheses to contain operands or you can do each calculation. We'll dive further into more detail with a second look at operators later on, but this will have to wait till we have a better understanding of more of the basics of C#.

3.3.7 Putting It All Together

Coming up with all of the systems that make up a fully functional class for use in a game takes a great deal of work, and that's what the folks over at Unity 3D have taken care of for you. Classes often work together to share and manipulate data. There are exceptions to this rule, but C# allows for many different writing styles and paradigms.

When you create data you've produced components of an object. In general you've created something that can be reused many times. Once you have created a zombie by adding in brain-seeking logic and how many bullets are required to stop him, it's only a matter of duplicating that zombie to create a mob to terrorize your player.

As we proceed we'll become more familiar with what it means to use classes as objects; the tutorials will be aimed at making sure that this is all clear.

```
void MyFunction()
{
    int i = 0;
    while(i < 10)
    {
        print(i);
        i ++;
    }
}
```

Let's break this code sample down. The first word is `void`, which is a keyword. `MyFunction` is an identifier we created. It's called `MyFunction` because we need to identify this function by some name we can remember.

The word used to identify the function could be practically anything; in this case, we're calling it `MyFunction`. Now that we know the parts of this line we should observe that this particular arrangement of tokens `void MyFunction()` is called a function declaration statement.

The parentheses after the identifier are operators used when we need parameters for our function. The parameters are discussed in Section 3.3.2. For now, take in the fact that both opening (and closing) parentheses are necessary after identifiers which are used to declare functions. They're really useful; trust me.

To begin the contents of the function we start with curly braces, the opening { and closing } curly braces. Everything that the function will do must be contained between the two braces. Any text appearing outside of them will not be part of the function. The code contained in the curly braces becomes the function block of code or code block.

The first statement in the function's code block is `int i = 0;` this is an assignment statement, where we declare `int i` and then use the `=` to assign `i` a value of 0. This is then separated from the next statement with the `;` operator.

The assignment statement is followed by `while(i < 10)`, which is called a looping condition. We'll read more on looping conditions later in Section 4.12. This looping statement is followed by an opening { (curly brace). This indicates a code block specifically written for the while loop.

The contained block contains the statement `print(i);`, which is a function call using `i` as a parameter or argument. After the `print` function call is `i++;`, which applies the post-increment operator `++` to the variable `i`. This code block is ended with the closing `}` (curly brace).

The `while` code block and the `int i = 0;` are both within the `MyFunction()` code block. To make the code more readable, tabs are added to indicate a separation between each section of code. Now bask in the knowledge that we've got words to describe everything presented in the above code sample. We may not understand everything that has been written, but we've got some vocabulary where we can get started.

3.3.8 What We've Learned

Programmers are meticulous about syntax. They have to be. The following two statements are very different: `int myint0 = -1;` and `Int my 0int = - 1;`. The first one will compile; the second one has at least five errors. You may not be able to spot the problems in the second statement, but by the end of the next section you will.

When trying to ask a programmer questions it's best to limit your subject and consider your words carefully. Asking "how would I write a video game where you shoot zombies with a flame thrower" involves so many different concepts it's impossible for anyone to tell you where to start. Most likely, you might be told to go to school, or at best you might get a link to a book on Amazon. Either way, you're asking for too much in a single forum post.

If you formulate a smaller question like "How do I attach flames to a zombie after it's been hit by a projectile?" you're more likely to get an answer. When talking to a programmer, you'll have to use similar wording. The more specific your question, the more likely you'll get the answer you were looking for.

3.4 Statements and Expressions

When reading a book or story you extract meaning from an ordered chain of words. In a similar way, computers extract commands from a chain of ordered instructions. In English we call this a sentence; programmers call this a *statement*. A statement is considered to be any chunk of code which accomplishes some sort of task separated by a semicolon.

At the center of any given task is the *algorithm*, not to be confused with a logarithm. An algorithm is a systematic process that accomplishes something. In many ways you can think of it as a recipe, or rather a recipe is an algorithm for making food.

It can take just one or two statements to accomplish a task, or it could take many hundreds of statements. This all depends on the difficulty of a given task. Each individual statement is a step toward a goal. This is the difference between spending a few minutes frying an egg, or spending an hour baking a soufflé. Each step is usually fairly simple; it's the final result that matters.

Like sentences, statements have different forms. Statements can declare and assign values to variables. These are called *declaration* and *assignment* statements. These statements are used to set up various types of data and give them names.

3.4.1 Expressions

The subjects of your statements are called *identifiers*. An *assignment statement* is used to give an identifier a value. When you read "Jack is a boy and Jill is a girl," you've mentally assigned two subjects their genders. In C# this might look more like:

```
gender Jack = male;
gender Jill = female;
```

Assignment statements often incorporate some sort of operation. These are called *expressive statements*. Different from expressing an emotion, expressions in code look more like "`x + y`." Expressions process data. After processing, the result of an *expression* can be assigned to a variable. We'll learn more about variables and assignments in Section 3.10.2.

A collection of statements is called a *code block*, not like a roadblock which might stop your code, but more like a building block, anything that's used to build. When writing a story, we call a collection of sentences a paragraph. The statements in a block of code work with each other to accomplish a task.

3.4.2 How Code Is Executed

When a class is created, all of its instructions are not carried out at the same time. As it's been explained, each collection of parts in a class is made up of lines of text called *statements*. Each statement is carried out in order. Where the computer starts is usually dependent on which collection of statements it's told to start with. You're in charge of how the code is started.

Unity 3D provides us with a function called `Start ()`, which we'll go into later. However, this is the preferred place to begin our code using Unity 3D. Once `Start ()` is called, the computer goes to the first line in `Start ()` and begins to carry out the instructions written there. As the computer gets to each statement the computer starts on the left of the statement and works its way to the right. Basically, a computer reads code like you're reading this sentence.

Other development environments often use `Main()` as their first place to begin running the code. When you start writing software in other environments you'll most likely start with `Main()`. We don't have to get into that right now, but it's good to know once you're done with this book.

3.4.3 Thinking in Algorithms

An algorithm is a step-by-step process. This lends itself to being written in an imperative programming language like C#, which executes operations one at a time and in order. Algorithms are written with statements starting at a beginning statement and finishing that statement before moving on to the next.

Just because we know what an algorithm is doesn't mean it's any easier to create one. Therefore, how do we go about writing the all-important algorithm? The first thing should be how you might accomplish any given task by hand.

As an example, say we're a computer surrounded by a mob of zombies. Thankfully, computers are very fast so we can do a lot before the zombies can get to us. With this in mind we're going to want to focus on the zombie closest to us. Unfortunately, computers are also very dumb, so there's nothing which allows us to just guess at which one is the closest.

First, we're going to need to know how far away every zombie is from our point of view. Therefore, the computer would probably start with measuring the distance from itself to every zombie available to make a measurement to. After that we're going to have to compare zombies and distances, so we'll start with the first zombie we found and compare its distance to the next zombie we found.

If the next zombie is closer we'll have to remember that one as being the closest one we've come across. Otherwise, if the next one is farther away, then we can forget about him and move on to the next zombie and compare numbers again. Once we've gone through all of the zombies, we should have identified the zombie who is the closest one to us.

The process which we just thought through is an algorithm for finding the zombie closest to us. It's not complex, but we had to do a few steps to get to it. First, we needed to get a list of all the zombies we needed to sort through. After that it was an iterative process, by which we needed to remember one zombie as the closest one, then compare others against him to decide whether or not he would retain the status of being the zombie nearest to us. We'll be going over many of these problem-solving thought processes as we learn how to program.

This algorithm would look something like the following code block:

```
void Update () {
    float closestDistance = Mathf.Infinity;
    GameObject closestGameObject = null;
    GameObject[] allObjects =
        (GameObject[]) GameObject.FindSceneObjectsOfType(
            typeof(GameObject));
    foreach(GameObject g in allObjects) {
```



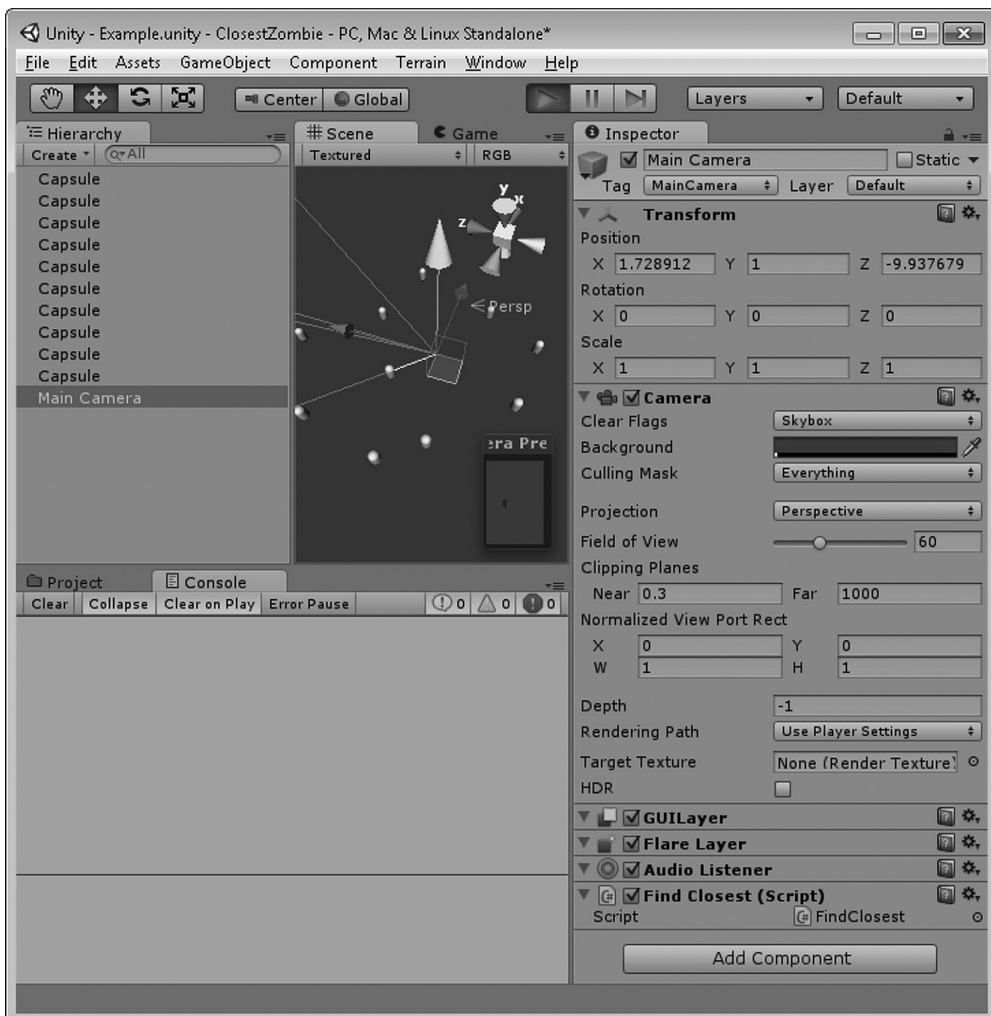
```

        if (g.name != this.name) {
            float distance = (g.transform.position -
                this.transform.position).sqrMagnitude;
            if (distance < closestDistance) {
                closestDistance = distance;
                closestGameObject = g;
            }
        }
    }
    Debug.DrawLine(this.transform.position,
        closestGameObject.transform.position);
}

```

We won't cover the entirety of how this code works, as we are yet to discuss the bulk of what's going on here, but in short we are indeed creating a list of objects called `allObjects`. Then we enter a loop where we compare the distance of each object against the one we've decided was the closest. If we find a closer object then we change the `closestGameObject` to the compared object.

Just for a proof of concept we draw a line from the object. Because the script is attached to the object, we draw a line from the object the script is attached to toward the object selected as the closest. The scene will look something like this:



Programming is a lot more about a thought process than it is about processing. In other words, programming is less about how to do multiplication as it is about what to do with the result of a multiplication process. In your everyday life we take for granted the everyday things we do on a regular basis. However, take a moment and really consider the complexity involved with something as simple as breathing.

Your diaphragm is receiving signals from your autonomic nervous system, causing several chemical reactions in the diaphragm's muscle tissue, which then pull proteins toward one another causing it to contract. This in turn creates a lowered air pressure in your lungs, which pulls in air. There's a whole number of chemical processes which then allow the oxygen in the atmosphere to enter your blood stream. Thankfully, all of this happens without our thinking of it because of chemistry and physics. In programming, nothing is so automatic.

3.4.3.1 Wash, Rinse, Repeat

In our every day life we see instructions written everywhere. However, if we follow instructions too closely we'd run into problems with our everyday lives. If you interpreted, for example, the instructions on most shampoo labels like a computer we'd run out of shampoo the first time we used it.

To interpret the “wash, rinse, repeat” instructions like a computer we'd wash our hair, rinse it; then wash our hair then rinse it; then wash our hair then rinse it; ... ; you get the picture. You'd continue the process until you've run out of shampoo. Because there are no further instructions, once the shampoo has run out, a computer would send out an error of some kind. There wasn't a number given for how many times the instructions should be followed; there was no way to *terminate* the process, as a programmer would say. A computer would need to have the “shampooing” process killed to stop washing its hair. Simply put, computers lack common sense.

Here's a simple scenario: A spouse tells the programmer to go to the grocery store and says “Get some bacon, if there's milk get three.” The programmer comes home with three packs of bacon, and no milk. If this makes sense, then you're thinking like a programmer. There are conditions to tell the programmer to get three bottles of milk if it was available; just the fact that there was milk meant bringing home 3× bacon.

3.4.4 What We've Learned

Programmers are often a group of literal thinkers. The thought process a programmer uses is different from what the artist uses. In terms of order of operation, programmers take things one step at a time. This process in programming terms is an imperative procedure, a step-by-step process in which operations are carried out in a linear fashion.

As an artist or newcomer to programming, this imperative process can be a bit out of the ordinary. With some time and practice, however, this paradigm of thinking will become second nature. The approaches a programmer and an artist would take to solve a problem often diverge.

To learn how to write code is to learn how to think like a programmer. This may sound a bit awkward, but perhaps after some work, you'll understand why some programmers seem to think differently than everyone else.

3.5 Keywords

Keywords are special words, or symbols, that tell the compiler to do specific things. For instance, the keyword `class` at the beginning of a line tells the compiler you are creating a class. A *class declaration* is the line of code that tells the compiler you're about to create a new class.

The order in which words appear is important. English requires sentences and grammar to properly convey our thoughts to the reader. In code, C# or otherwise, programming requires statements and syntax to properly convey instructions to the computer.

```
class className
{
}
```

Every class needs a name; in the above example we named our class `className`, although we could have easily named the class `Charles`. When a new class is named the name becomes a new *identifier*. This also holds true for every variable's name, though *scope* limits how long and where the variable's identifier exists. We'll learn more about variables and scope soon.

You can't use keywords for anything other than what C# expects them to be used for. There are exceptions, but in general, keywords provide you with specific commands. Altogether in C# there are roughly 80 keywords you should be aware of, we don't need to go over all of them now, so instead we'll learn them as we come across them.

3.5.1 Class

To create a class named `Charles` we used the keyword `class`. The keyword commands C# to expect a word to identify the new class. The following word becomes a new identifier for the class. In this case `Charles` is used to name the class. Keywords often precede a word, and the computer uses the word following as its identifier.

The contents of `Charles` are defined between curly braces. After the class definition the opening curly brace `{` is followed by any statements that `Charles` needs to be a proper `Charles`. When you're finished defining the contents of `Charles`, you use the closing curly brace `}` to indicate you're done with defining the `Charles` class.

```
class Charles { }
```

It's important to note that keywords start with lowercase letters. C# is a case-sensitive programming language, and as such, `Class` and `class` are two different things. In C# there is no keyword `Class`; only `class` starting with a lowercase `c` can be used. More of the same goes for every keyword in C#.

Writing a new class or any function starts off as a nonlinear process. By this I mean you write out `class Charles { }` with both curly braces. When writing a sentence in English we don't write the period at the end of a sentence first, for example, when we would write something like "It was a dark and stormy night." We don't start with "It." and then go back and add the rest of the sentence later, between "It" and the period.

However, programming works this way: "`class Charles { }`" begins with the start of the class indicated with the open curly brace `{` and the end of the class indicated by the closing curly brace `}`, after which we go back and add the body later. Not everything works in this way, and it takes a bit of getting used to before you understand the flow. Certainly, however, this will come naturally after some practice.

Now we have a new class called `Charles`, which has no data and does nothing. This is to be expected since we haven't added any data or functions yet. You can also use the terminology this class is identified as `Charles`.

This class can't be used directly by Unity 3D. It lacks some of the necessary additions needed by the Unity 3D game engine to directly interact with this class. To be specific, this class doesn't have enough information to allow Unity 3D to know what to do with the contents of this class. However, there are methods which make a minimal class, such as `Charles`, quite useful inside of Unity 3D, but we need to cover a great deal of ground before we can do that.

In section 2.4 we looked at a complete source file, `Example.cs` which is ready for use in Unity 3D. However, it's important to know what creates a minimal class before adding all of the rest of the keywords and declarations which make it actually do anything in Unity 3D, our game engine.

Keep in mind no two classes can share the same name. We'll go further into detail about class names and what's called a namespace later on. The contents of the class are contained in the following pair of curly braces. All of the variables and functions that live in between the opening curly brace `{` and the closing brace `}` become the contents of the class object.

Other keywords which we'll come across in this chapter include keywords that indicate a variable type. Some variable keywords which will be covered in the next chapter or two are as follows.

```
int
float
string
```

These keywords indicate a type of variable. There are many more, but we're just going to look at these as an example. An `int` variable looks something like the following:

```
int i = 0;
```

A `float` looks like this:

```
float f = 3.14159f;
```

A `string` is used to store words like this:

```
string s = "some words in quotes";
```

If `int` and `float` seem similar, it's because they both deal with numbers, but as we will see, numbers behave quite differently when computers use them. Keywords are used quite often, and they are case sensitive. Therefore, `int` and `Int` are different things, and thus using `Int i = 0;` will not work. Only `int` is recognized as a keyword, and there has been no definition written to tell C# what `Int` is. Keywords are also used to indicate special changes in code behavior. The keyword `if` is what is used to control code execution based on a conditional statement.

```

      Keyword      Condition
      |            |
      |            |
  if ( i < 10 )
  {
      // Code goes here.
  }
```

The diagram shows an `if` statement. A box labeled 'Keyword' has a line pointing to the word `if`. A box labeled 'Condition' has a line pointing to the expression `(i < 10)`. The rest of the code block is indented.

As you are introduced to more and more keywords, we'll be expanding our ability to create new code as well. Keywords make up the complex vocabulary that is required to use C#.

3.5.2 What We've Learned

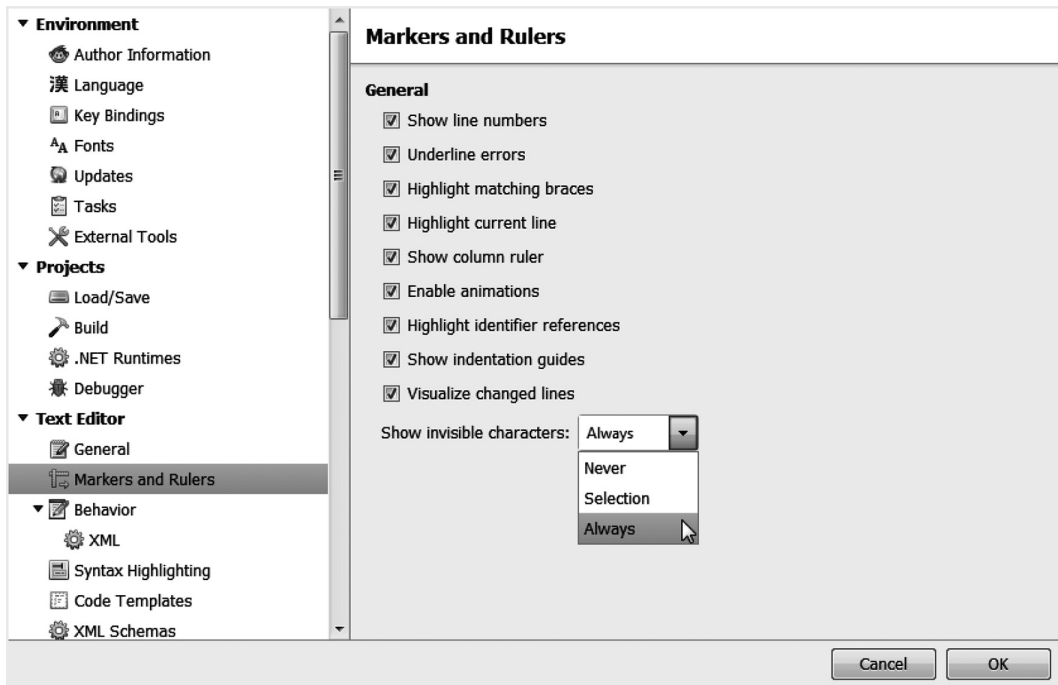
This was a short section, but it covered a fundamental component of a C# construct. Classes are both construction instructions and rules for objects to obey. Writing a complex class takes a step-by-step approach.

Keywords are an important component of any programming language. Some languages have very few keywords. Lisp has a mere 18 keywords, C# has about 76, and COBOL has over 400 words reserved for special purposes.

Most programming languages use the same keywords for similar purposes. The `var` keyword is often used to indicate that a variable is being declared. Because of the similarities between programming languages, once you've learned one language, it's often easy to pick up and learn new languages.

3.6 White Space

White space refers to the spaces between words and lines. When typewriters were still in fashion, new lines were entered by pushing on a lever which pushed the roller that held onto the paper to reposition the hammers with letters on them back to the beginning of the next line. Now, these are called line feeds, new lines, or returns.



Like letters and numbers, white spaces are also characters entered into your code. To see them along with the rest of your text in MonoDevelop, select Tools → Options, and in the Text Editor section select Markers and Rulers. Select Always from the Show invisible characters Dialog Box. Going back to your script, you should see faint dots, arrows, and \n marks spaced throughout your code.

```

1 using·UnityEngine; \n
2 using·System.Collections; \n
3 \n
4 public·class·WhiteSpace·:·MonoBehaviour \n
5 { \n
6     //·Use·this·for·initialization \n
7     void·Start·()·{ \n
8         \n
9     } \n
10 \n
11     //·Update·is·called·once·per·frame \n
12     void·Update·()·{ \n
13         \n
14     } \n
15 } \n
16

```

In many cases the new lines and spaces and tabs are there only to help make the code more readable. And depending on where or what a programmer was taught, the white spaces will be used differently. For instance, adding white space after a function declaration is unnecessary and the location of the curly

braces can pretty much be moved anywhere before or after the contents of the function itself. Leaving the white spaces showing isn't necessary, but it's good to know what white space means when talking to a programmer. If none of this makes sense right now, it will later.

```

1  using·UnityEngine;\n
2  using·System.Collections;\n
3  \n
4  public·class·WhiteSpace·:·MonoBehaviour\n
5  {\n
6      //·Use·this·for·initialization\n
7      void·Start·()\n
8      {\n
9      \n
10     }\n
11     \n
12     //·Update·is·called·once·per·frame\n
13     void·Update·()\n
14     {\n
15     \n
16     }\n
17 }\n
18 EOF

```

The placement of curly braces is easy to change; these decisions often lead to debates on standards among programmers. In reality there's little difference either way. However, keeping your own code consistent helps you read your code and find errors faster.

It's important to remember that if you're going to be working with more classically trained programmers they might have a headache when reading poorly formatted code. It's best to copy your programmer friends' style before coming up with your own way to format your classes and functions. No doubt, before the end of this book you will have formed opinions of your own when it comes to the use of white space and formatting your own code.

Normally, code might look a bit like the following:

```

void MyFunction()
{
    int i = 0;
    while(i < 10)
    {
        print(i);
        i ++;
    }
}

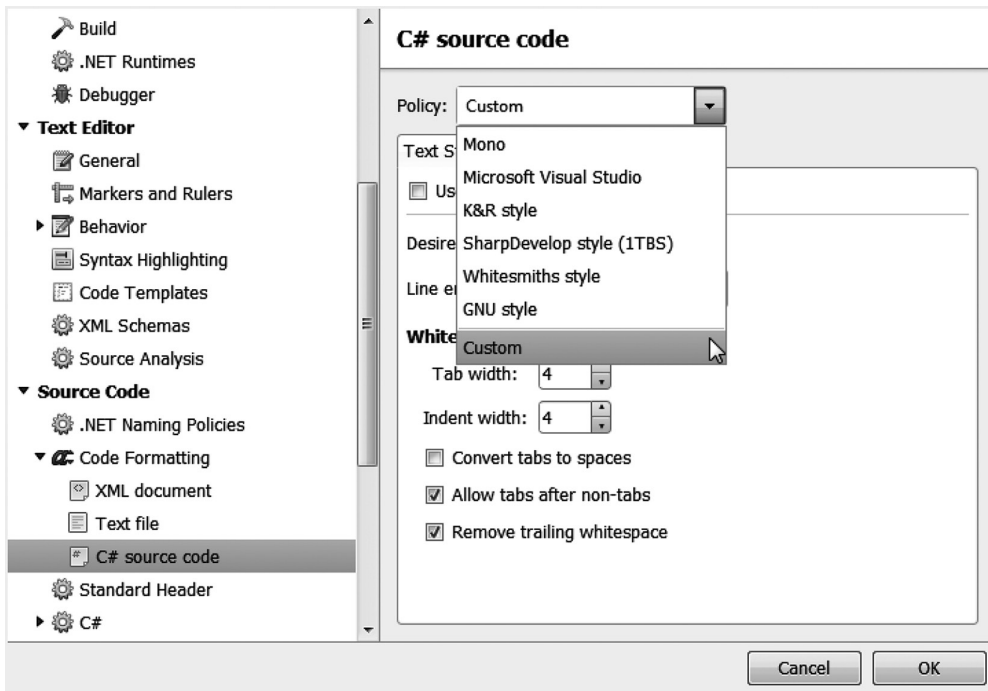
```

This looks like a readable collection of statements. However, the compiler could care less about how you use white space. Therefore, here is what would this code sample look like without the unnecessary white space:

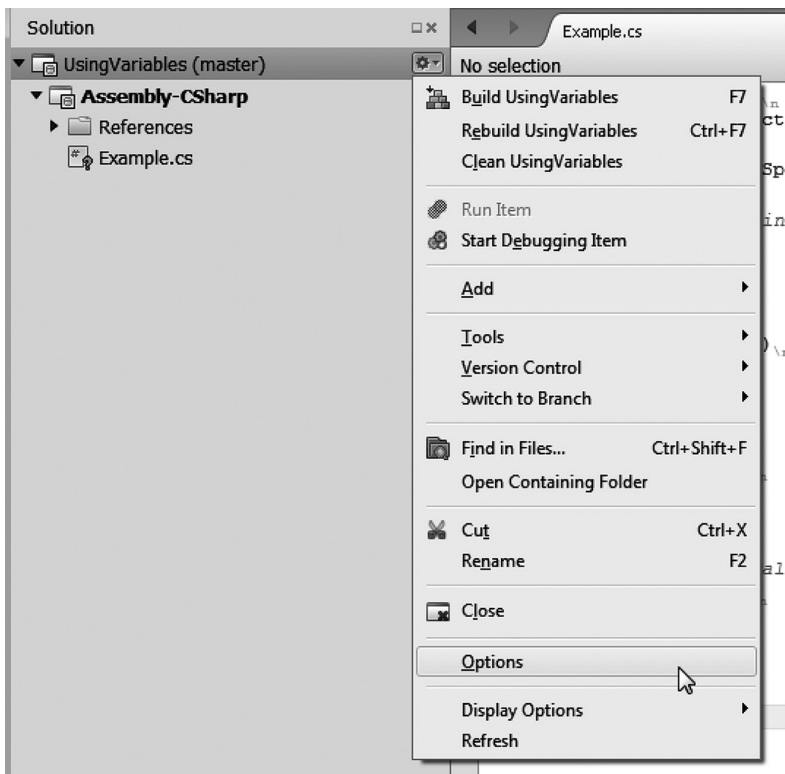
```
void MyFunction(){int i = 0;while(i<10){print(i);i++;}}
```

There are still some necessary white space characters used to separate the different tokens in the function. There is the space between `void` and `MyFunction`, and the space between `int` and `i`. In the latest version of MonoDevelop saving a file automatically reformats your code into a more proper format. Rather than fight this it's better to let it format your code, though this isn't always an option with other editors.

To turn this feature off you can select Tools → Options.



And change some of the behaviors MonoDevelop will follow when saving a file. In an already existing project, for example, you have to change these properties in the project settings.



These options all follow different coding styles; you'll want to pick one that suits your taste. Likewise, if you're working in a team you'll want to come to some decision as a group to avoid starting a formatting war.

In any case the white space is required after the use of a keyword. Keywords are unique words reserved for C#. If we used `voidMyFunction()` then there would be an error stating something about not knowing what a `voidMyFunction()` is.

White space is also used to make code more readable. Without the white space the code becomes more difficult to interpret.

```
imagineforamomentwhatenglishwouldlooklikewithoutanyspacesorpunctuation
```

White space should also be used as needed. I once worked with a programmer who would put a few pages of white space between different functions. He said it made it easier to focus on what he was looking at, when there was only one line of code on the screen at a time. Of course, this drove other programmers nuts. It's hardly a common practice.

Although he had a point, being able to focus on one code block at a time is important, but there's a thing called code folding that we'll learn about later on to help make your code more easy to organize.

3.6.1 Pick a Flavor

Between programmers there's often a discussion of code style. Many heated debates might flare up based on the use of white space. The differences are entirely based on style and in most cases have no effect on the efficacy of the code. In the following fragments the `{` is placed in different places.

```
void MyFunction () {  
    //some code here...  
}  
void aDifferentFunction ()  
{  
    //some more code here...  
}
```

The execution of the two functions will perform equally, but the placement of the first curly brace `{` can make the code more or less readable depending on what you're comfortable with. Unity's developers offer us the first version, but for clarity we'll stick to the second version.

The finer points on where variables appear in a statement or how many statements can appear on a single line are usually dictated by how a programmer has learned to write code. Old school programmers prefer no line of code extending beyond 80 characters. This limit formed in 1928, when IBM punch cards had only 80 columns to poke holes in the paper. Though it's unlikely you'll have to work with an old curmudgeon, but just in case, at least now you know why they might be grumpy.

Today, there's no physical limit other than what would make a statement difficult to read. If a single line of code spans several widths of your screen it's probably unreadable. In these cases it's better to break apart your statement into a few smaller statements. This will help keep the code more readable. Within the confines of this book we're somewhat more limited by space, so we too will be limiting the length of our statements based on the width of the margins of the book.

Likewise, if you've got a lovely 4k monitor you might be comfortable reading mile-long lines, but if you're stuck on a low res laptop then you might hate having to scroll left and right just to read a single statement. Consider the fact that not everyone has a giant monitor.

Interpreting your own code several days or weeks after you've written it is just as important as handing your code off to someone else, with that person being able to read it. There are many times when I've come back to a project written several weeks ago and needed to spend a few minutes trying to figure out what I was thinking when I wrote the code. Sometimes, it takes only a few days for me to forget about the code I've written. Worse yet, if I write the code half asleep then I might forget what I was doing before I've even reached the end of a statement.

In the end it's up to you to pick what you'll stick to. However, it's considered rude to pick through someone else's code and reformat it. Code can sometimes become quite personal, and reformatting code might end up feeling like someone sneaking into your bedroom and reorganizing your sock drawer.

3.6.2 What We've Learned

White space and formatting are important to creating a clean easy-to-read class. Every large company will have a set of rules for their programmers to follow. These coding standards, as they are called, are usually maintained by a lead programmer and can vary from company to company.

Smaller teams of engineers can have a less restrictive set of rules, but this often leads to some confusion and large bodies of unreadable code. Everyone has his or her own coding style and preferences. If there is an established set of coding standards, then it's best to follow it.

So far as this book is concerned, it's best that you learn how to make your code work before concerning yourself with strict coding rules. Code formatting and style are important. How you express yourself through code is up to you, so long as it works. Getting your code to work is a big enough challenge; style and proper formatting will come naturally with time as you see the benefits of different code styles, and how they contribute to your code's readability.

3.7 Code Blocks

Blocks are collections of statements grouped together when they're evaluated. This evaluation of code is usually controlled by an opening and closing curly brace. C# programmers often use tabs to indent different blocks of code. This makes the different blocks more readable. Some languages like python rely on the tabs to indicate separate blocks of code.

Getting used to how tabs are used is essential for writing readable code. Next, we'll use a logic control statement followed by a block of code. The block for this `if` statement is indented to the right to indicate it's confined within the logic control statement.

```
if (true)
{
    //code goes here
    //im also a part of the code
}
```

If we take a look at a more complex section of code we can see why it's important to differentiate code by using tabs. When looking at code it might not be apparent, but programmers are very fastidious about text layout. Each programmer has a preferred style, but it's safe to say that most, if not all, programmers hate seeing badly formatted code.

```
int i = 7;
int j = 13;
if (i < 13){
int j = 7;}
```

In the above example we can clearly see `int j` being set twice. What isn't obvious is why it's declared twice. All of the lines start at the same position and look crowded. Programming tools offer many text layout options, so white space is used to separate different blocks of code.

```
int i = 7;
int j = 13;
if (i < 13)
{
    int j = 7;
}
```

By adding in some white space we can make the `if` statement more visible. Adding white space around our blocks of code is important to maintain readability. There are also blocks of code that can live within another block of code. In general it's all related to the same block. Blocks inside of a block are called nested code blocks. White space is used to separate the nested code from the block it's found in.

```
if(true)
{
    //all of the code
    //im also a part of the code
    if(true)
    {
        //im another block of code
        //living inside of a bigger
        //block of code
    }
    //yep more code here too
}
```

The block of code following the second `if` statement is considered to be nested inside of the first block of code. This is because of the placement of the first opening curly brace `{` and the related closing curly brace `}`. You form hierarchies by placing the different curly braces within one another.

```
if(true)
{
    //all of the code
    //im also a part of the code
    if(true)
    {
        //im another block of code
        //living inside of another
        //block of code
    }
    //yep more code here too
}
```

Poorly indented curly braces make for confusing code; the above code is still valid, but it's less readable compared to the previous version. Most code-editing tools will automatically add in the proper number of indents for you. It's something you'll have to get used to when writing your own code.

We'll go into further detail as to what this all means later. Just gather from this discussion that a bunch of statements appearing between a pair of curly braces is considered a block of code.

3.7.1 What We've Learned

Much of programming involves proper use of style and a great deal of white space to accomplish that style. So far we've just been covering vocabulary, but without that vocabulary we can't have a proper conversation about programming. This too includes talking to other programmer friends, who will have a hard time understanding you if you can't speak in their language.

3.8 Classes

The most important part of any C# program is the class. Classes here are not the sort that students go to; rather, they are classes in terms of classification. Classes in C# appear in several different forms. Some are called partial classes, and some classes can even appear inside of another class; these classes are called nested classes.

A class is a collection of instructions and data. The instructions can be referred to as functions or methods. The data stored in a class is called a field or property. All of the naming aside, what's important is what classes do and how they're used.

Classes are flexible and easily changed, but this flexibility is not without some complexity. As we've seen before, the class declaration can contain a lot more than the following example. However, much of the complexity comes later in this book, when needed.

When you approach a game or any sort of programming task it's important to first collect data, and then figure out how to use the data. In most programming paradigms it's best to start off with variables at the beginning of your new class. For instance, in this pseudo code we might want to know what day it is before going out:

```
Time day = today;
if (day == Friday)
{
    PartyTonight = true;
}
```

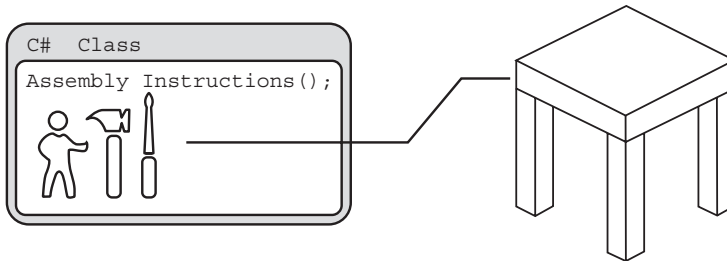
We start off with getting a day, and then deciding what to do tonight based on the day. We should refrain from partying and then checking what the day is unless we want to show up for work late. We'll go into more detail on this topic later on, after we've learned about how to use a variable in a class properly.

3.8.1 Objects

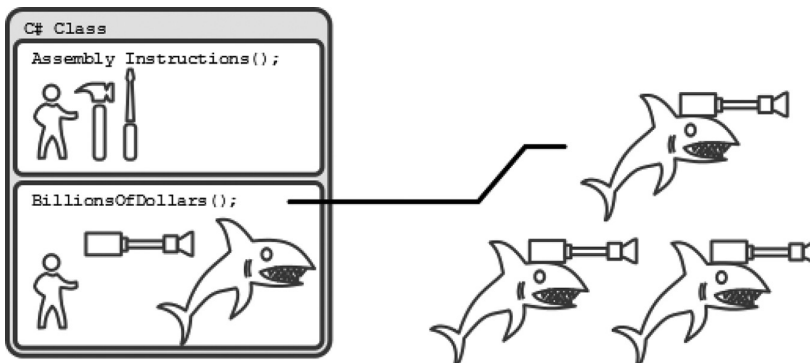
Object oriented programming (OOP) was invented to associate data and logic into nicely packaged bundles of code. When you write a class in C# you're creating blueprints and instructions for a new object. Any number of objects can then be constructed based on your blueprints. Programmers use the term *instantiate* when talking about creating an object from the class blueprint.

Each instantiated object is an *instance* of the class from which it was constructed. The class is not itself one of the instances created by it. The newly created object contains all of the functions and data written into the class.

To make this clear, a class in of itself is not an object, just a plan for making objects. We start with a C# class which contains instructions to build an object.

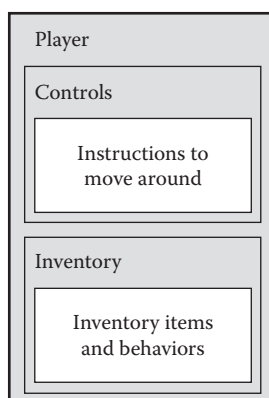


Classes can be simple, or complex. A class might be nothing more than a container for some data. On the other hand a class can also contain behaviors for a complex boss who plans to crash a comet into the earth, or any other instruction for world domination.



Just as important, a class can create more than one instance of an object. This process is called *instantiating*. Each instance is created from the same blueprint and thus behaves the same as all the other instances of the same class. Each instance can have unique values. In Unity 3D, each instance of an object can have unique positions in 3D space.

OOP is particularly useful for video games. Individual classes can address different aspects of your game. A zombie class can be instantiated many times to create a mob of mindless zombies. Likewise, a solitary class can be written to manage how to move your player's character around when keys or buttons are pressed. A separate class can manage inventory and how to deal with items.



Monsters, and bullets, cameras, and lights all have classes from which they are created. Almost every object you see in any level in Unity 3D can be instantiated from a class. An instance of an object communicates to another object through *messages* and *events*.

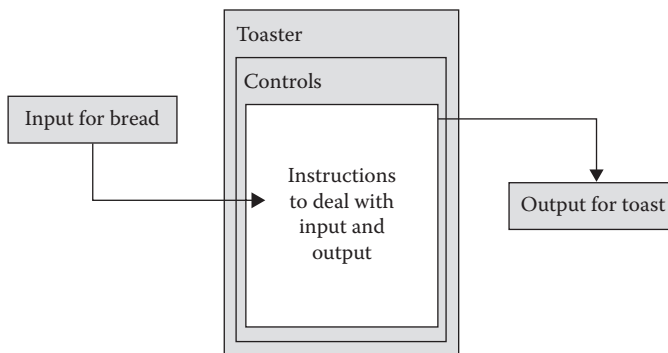
When people in the real world deal with household items they are rarely concerned with how they function. Knowing how a toaster works has little effect on the toaster's performance. We put bread in a toaster, push down on a button, and wait for a delicious toast.

A toaster, however, is more than a button and a receptacle for bread. It's a collection of wires, heating elements, transformers, and other electrical components which are hidden under a shiny cover. We interact with an *abstraction* of a toaster, never dealing with the insides ourselves.

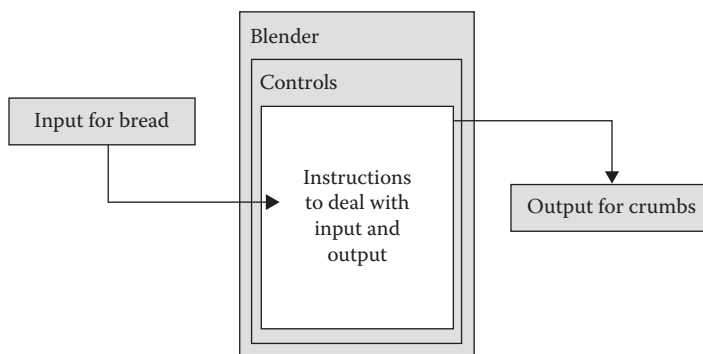
Abstraction presents only the parts we need to accomplish a given task and hides the rest. To make toast we need a place to put bread and a button to start the machine. The toaster does its work and toast happens, or at least it should.

Messing with the insides of the toaster class isn't recommended; that's why there are labels telling us to keep forks out of the toaster. To protect a human class from fumbling around with the insides of a toaster class the toaster is protected by *encapsulation*.

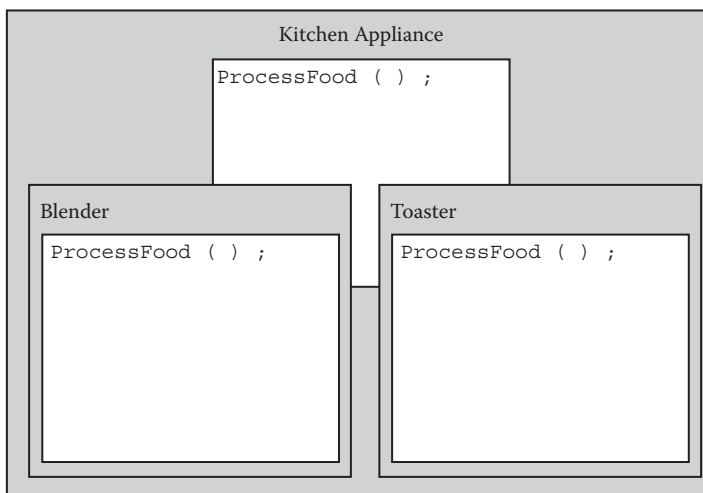
Encapsulation is the term used to define to how classes hide data and functions from the outside world. To interact with the toaster we must use the toaster's *interface*, but there are other objects in the kitchen which have buttons, and the human class is very good at pushing buttons.



When you put bread into a blender and press a button you will get a very different result from the toaster, but they are both kitchen appliances and use the same power outlet. They're both made of metal and run on the same electronics, but they do different things.



When and where it's possible, a programmer will reuse as much code as possible. A blender and a toaster could share similar components, only the operation carried out on a slice of bread is different when they are used. The capability to change the contents of an operation is described as *polymorphism*.



A generic kitchen appliance would be a place to put food and a button to press to process food. Each version of a kitchen appliance does different things to food. A wise programmer would make sure that all of the appliances inherit from the generic kitchen appliance.

Inheritance allows many classes to share common attributes and behaviors. If you wanted a toaster with more than one slot and a knob to adjust the darkness of your toast you would want to inherit the original toaster's functions first. This means that the base of both Toaster and Blender is a Kitchen Appliance. A dual slot toaster inherits from Toaster, which inherits from Kitchen Appliance. This in turn means that if any changes are made to Kitchen Appliance that change or addition is inherited by all of the sub-classes inheriting from Kitchen Appliance.

This would mean inheriting the original toaster's electronic systems, casing, and original button. All of these components may also be made from other classes. This would mean only changing one electronic part to adapt a toaster for sale, say, in Europe.

Aggregating many different classes into a single class allows another layer of flexibility. By aggregating many separate components or classes into a single useful toaster or blender means you can more easily swap individual parts for new behaviors. A toaster is the aggregate total of many smaller parts.

You may find yourself writing code for a zombie or a toaster and see some similarities between those things and other objects in your game. When your player attacks a zombie or a toaster, they may both break apart. Both the zombie and toaster can share functions related to getting attacked.

As far as your code may be concerned, one object chases after you trying to eat your brains, and the other object turns bread into delicious toast. Keep this in mind when you start writing your own code. Even though some things may seem unrelated, maybe even humans and zombies can both enjoy toast.

3.8.2 What We've Learned

I'll let this sink in a little bit. Being new to programming is usually a very intimidating experience. Having to learn any new language is a daunting task. Because much of programming is taught in English we've got a slight advantage. At least the mentioned 80 keywords aren't written in a foreign language.

The main take away from this chapter is the fact that OOP is the process of building objects, instantiating them, and then using them. Using objects involves reading data, using logic on the data, and then carrying out tasks based on the logic.

As we move on, feel free to jump back between chapters to refresh your memory if there are concepts which you might have forgotten. It's important to proceed at your own pace; jumping ahead before you're clear on a topic leads to confusion, then frustration, and leading to quitting altogether.

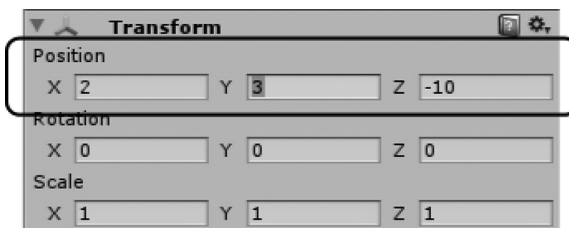
We're here to learn and have fun while learning. Building a game for modern gaming platforms is easy once you've got your head wrapped around this whole C# thing, so stick to it and you'll reap the rewards. Not only will you be able to write your own games from scratch, you'll also be able to write plenty of other apps using other development tools which use C#.

We've also come across a great deal of vocabulary related to classes. If you're not completely clear as to what these words mean we'll be covering chapters dedicated to these more complex concepts. When you're ready it's time to continue learning and getting ready to make some games.

3.9 Variables

Programming is a combination of behavior and data. Variables are the little bits of data or information which make up the world. Everything you see in a game scene, and even things that are implicitly assumed, like gravity, can be interpreted as data that can be stored into a variable. Logic behavior can be controlled by comparing values stored in variables.

A variable can be either something that's constantly changing or something you've set once and shouldn't change unless you act on the variable to change it. For instance, an object's location should be updating if it's falling or getting pushed on by other forces in the world. This object's location is a variable, which the programmers at Unity have named `transform.position`. This brings us to how to find a variable.



A variable's name is called an *identifier*. For the most part an identifier is a unique word that a programmer, or in this case you, picks to name a variable. An identifier is always something that a programmer invented to describe a variable; it's like naming a new pet or a baby.

3.9.1 Identifiers

Identifiers, which are considered symbols or tokens, are words that you invent which you then assign a meaning. Identifiers can be as simple as the letter `i` or as complex as `@OhHAICanIHasIdentifier01`. We'll get into properly creating names later on, but *identifier* is the word that's used to name any function, variable, or type of data you create.

When the keyword `class` is used it's followed by another word called an identifier. After properly identifying a function, or other chunk of data, you can now refer to that data by its identifier. In other words, when you name some data `Charles` you access that data by the name `Charles`.

```
class MyNewClassImWriting
{
}
```

3.9.2 Data

Data, in a general sense, is sort of like a noun. Like nouns, data can be a person, place, or thing. Programmers refer to these nouns as objects, and these objects are stored in memory as variables. The word *variable* infers something that might change, but this isn't always the case. It's better to think of a variable as a space in your computer's memory to put information.

When you play word games like `MadLibs` you might ask for someone's name, an object, an adverb, and a place. Your result could turn out like "*Garth* ate a *jacket*, and *studiously* played at the *laundry-mat*." In this case the name, object, adverb, and place are variable types. The data is the word you use to assign the variable with.

Programmers use the word `type` to denote what kind of data is going to be stored. Computers aren't fluent in English and don't usually know the difference between the English types noun and adjective, but they do know the difference between letters and a whole variety of numbers. There are many predefined types that Unity 3D is aware of. If you add that to the ability to create new types of data, the kinds of data we can store is practically unlimited.

The C# built-in types are sometimes called POD, or plain old data. The term *POD* came from the original C++ standard which finds its origin dating back to 1979. POD types have not fundamentally changed from their original implementation.

So far we've used the word `type` several times. Programmers define the word `type` to describe the variety of data to be stored. Variables are created using declarations. Declaration is defined as a formal statement or announcement.

Each set of words a programmer writes is called a statement. In English we'd use the word *sentence*, but programmers like to use their own vocabulary. Declaration statements for variables define both the type and the identifier for a variable.

```
public class Example : MonoBehaviour
{
    int i;
}
```

Programmers use a semicolon (`;`) rather than a period to end the statement. Therefore, if you want to sound like a programmer you can say you can "write a statement to declare a variable of type `int` with the identifier `i`." Or if you want to be overly dramatic you can proclaim "I declare a new variable of type `int` to be known as `i`!" and so it shall be.

3.9.3 Declaring a Variable

Programmers use their own grammar to make complete statements. To ask the computer we'd like some space in memory for a number, we use the following statement. Declaring a variable starts with a type of data followed by the name or identifier we're assigning to the variable.


```

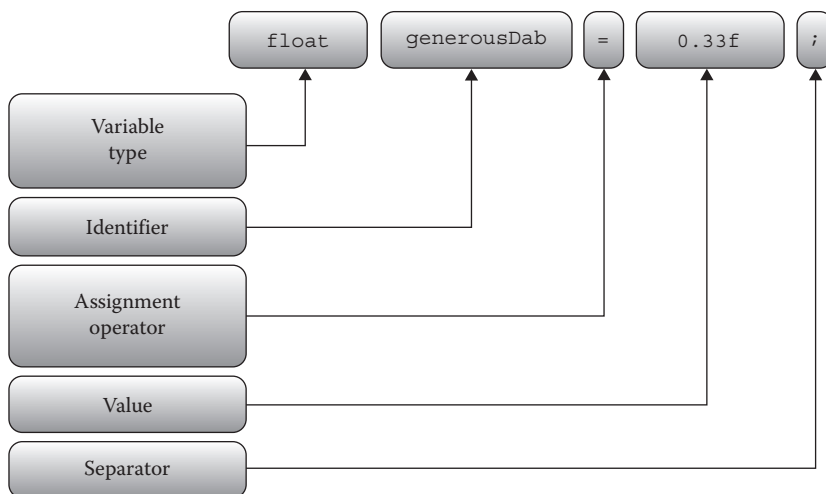
1  using UnityEngine;
2  using System.Collections;
3
4  public class Example : MonoBehaviour
5  {
6      int SomeNumber;
7      // Use this for initialization
8      void Start ()
9      {
10
11      }
12
13      // Update is called once per frame
14      void Update ()
15      {
16
17      }
18  }
19

```

There are three important parts to the statement you see above. First observe the placement of the statement, after the lines. We've added the `int SomeNumber` after the first line of code `public class Example : MonoBehaviour {`; this is significant because of the first curly brace. Any line of code between the first and last curly brace operator is a part of the `Example` class. Anything outside of the curly braces either supports the `Example` class by external means or is created independently and requires special means to interact with the `Example` class we're writing.

Let's go back to the shampoo example. I've seen some rather vague descriptions for how much shampoo to use at any time. The clause *apply a generous dab* of shampoo means nothing to a computer. Computers like actual numbers; they hate descriptions of a number without a number. Computers would be happier if you said the following by *defining* a generous dab as:

```
float generousDab = 0.33f;
```



Now we can say we have defined `generousDab` as a type `float`, and then assigned the value `0.33f` to `generousDab`. We will explain what the `f` is doing after the numbers in Section 4.7.1. For now, remember it's just a way to tell C# the value to match the `float` type it's being assigned to.

The above statement is still rather vague: Does it mean 0.33 fluid ounce or 0.33 gram? This really depends on the hardware a computer would be using to measure an amount of shampoo. It's fair to say that computers would have a hard time washing hair without the help of robotic parts.

Now that we know what parts are required to make a variable declaration, what can we do with them? Once a variable has been declared and assigned it can be reassigned at any time.

```
int i = 0;
i = 1;
```

On the second line the variable `i` is no longer 0; it's 1 now. If we check what value is being stored at `i` we'd find a 1. So long as `i` is not assigned again, `i` will retain its value.

```
i = 2;
```

And, of course, `i` is no longer 1; it's 2 now. The variable will retain whatever the last assignment was until it's changed again. Once a variable has been declared with `int i;`, it cannot be declared again. Consider this bit of code:

```
int i = 0;
int i = 1;//invalid, i has already been declared!
int i = 2;//still won't work, stop trying.
```

This code leads to conflicts in memory, as we're trying to create a declaration for `i` twice.

3.9.3.1 A Basic Example

In a very simple class you need to declare variables inside of the curly braces. In the following example, declaring the variable outside of the curly braces will create invalid code.

```
class SomeClass
{
    int SomeNumber;
}
```

Once the code has an invalid character or line of code the rest of the class will be broken and won't compile.

```
int SomeNumber;
class SomeClass
{
}
```

We are using the keyword `int` to tell the computer the type of data we want. This is followed by the name of the variable. In this case we're naming the `int` to `SomeNumber`. If we were to use programmer speak, we are assigning the identifier `SomeNumber` to data type `int`. The last part of our statement is the `;` operator. The semicolon is like the period at the end of a sentence. The `;` ends the statement, and tells the computer we're done making a statement.

```
class SomeClass
{
    int SomeNumber = 0;
}
```

One classic mistake many, new and old, programmers make is forgetting to add the semicolon to the end of a statement. MonoDevelop will sometimes tell you when there might be an error, but Unity 3D will also stop everything when it finds an error like this.

We've now completed two important tasks. We created a variable identifier giving the variable a way to access the contents of the variable. C# under the hood gave that variable some space in your computer's memory to put an integer. The `int` variable is now defined as `SomeNumber`, an object we can fill with a number with practically any `int` value.

When variables are declared for the first time, or initialized, in C# they are usually immediately given a value. This is not true with all types of variables, but number values are assigned 0 when they are created. For more complex data types their default value is null when they are initialized. This initialization behavior will become useful later when we start inspecting the values of variables.

3.9.4 Dynamic Initialization

When we both define and assign a variable we use the term *dynamic initialization*. Some data types automatically give themselves a default value when they are initialized without an assignment. For instance, a Boolean automatically assumes it's false when it's created.

```
class SomeClass
{
    bool SomeBool;
    bool AnotherBool = false;
}
```

In this case, if we were to use `SomeBool` before assigning it then the result would act as though it were initialized false. In the above example both `SomeBool` and `AnotherBool` have the same value. Likewise, most numbers are initialized as 0.

```
class SomeClass
{
    int SomeInt;
    int AnotherInt = 0;
}
```

So in this case both `SomeInt` and `AnotherInt` have the same value. However, this changes when we dynamically assign a value to a variable when initializing.

```
class SomeClass
{
    int SomeInt = 7;
    int AnotherInt = 11;
}
```

When we assign a value to an `int` when it's initialized we skip the default initialization, and the value assumes its assignment when it's created.

3.9.5 What We've Learned

Variables are stored in your computer memory. By defining a variable by its type you instruct the computer to set aside a block of memory in the computer's RAM (Random Access Memory) to store data. To access the data you give it a name, or an identifier.

Once the identifier has been defined you can assign or read the data stored in memory through its identifier. In many cases the identifier needs to be unique. Depending on when it was created an identifier might have the same name as one created before it.

When this happens you get an error telling you that a variable has already been defined with that identifier. Changing one or the other's name solves the problem. When creating identifiers we need to consider what they're used for and how to name them.

Some code standards place rules on how an identifier should be named. For instance, a bool might always start with `is` to indicate how it's used. Therefore, a zombie might have `isWalking` or `isRunning`, indicating what it's doing. Other standards might use a prefix `b` for bools, so `bWalking` or `bRunning` could be used to indicate the same thing. How you decide to name variables is up to you, but it's good to know what we're about to get into so far as naming variables, which we'll be covering next.

3.10 Variable Names

It's important to know that variable identifiers and class identifiers can be pretty much anything. There are some rules to naming anything when programming. Here are some guidelines to help. Long names are more prone to typos, so keep identifiers short. A naming convention for variables should consider the following points.

The variable name should indicate what it's used for, or at least what you're going to do with it. This should be obvious, but a variable name shouldn't be misleading. Or rather, if you're using a variable named `radius`, you shouldn't be using it as a character's `velocity`. It's also helpful if you can pronounce the variable's name; otherwise you're likely to have issues when trying to explain your code to another programmer.

```
int someLong_complex_hardToRememberVariableName;
```

There is an advantage to keeping names as short as possible but still quite clear. Your computer screen, and most computers for that matter, can only fit so many characters on a single line. You could make the font smaller, but then you run into readability issues when letters get too small. Consider the following function, which requires more than one variable to work.

```
SomeCleverFunction(TopLeftCorner - SomeThickness + OffsetFromSomePosition,
BottomRightCorner - SomeThickness + OffsetFromSomePosition);
```

The code above uses many long variable names. Because of the length of each variable, the statement takes up multiple lines making a single statement harder to read. We could shorten the variable names, but it's easy to shorten them too much.

```
CleverFunc(TL-Thk+Ofst, LR-Thk+Ofst);
```

Variable names should be descriptive, so you know what you're going to be using them for: too short and they might lose their meaning.

```
int a;
```

While short and easy to remember, it's hard for anyone else coming in to read your code and know what you're using the variable `a` for. This becomes especially difficult when working with other programmers. Variable naming isn't completely without rules.

```
int 8;
```

A variable name can't be a number. This is bad; numbers have a special place in programming as much of it has other uses for them. MonoDevelop will try to help you spot problems. A squiggly red line will appear under any problems it spots. And speaking of variable names with numbers, you can use a number as part of a variable name.

```
int varNumber2;
```

The above name is perfectly valid, and can be useful, but conversely consider the following.

```
int 13thInt;
```

Variable names can't start with any numbers. To be perfectly honest, I'm not sure why this case breaks the compiler, but it does seem to be related to why numbers alone can't be used as variable names.

```
int $;  
int this-that;  
int (^_^);
```

Most special characters also have meanings, and are reserved for other uses. For instance, in C#, a `-` is used for subtracting; in this case C# may think you're trying to subtract that from this. Keywords, you should remember, are also invalid variable names as they already have a special meaning for C#. In MonoDevelop, you might notice that the word `this` is highlighted, indicating that it's a keyword. Spaces in the middle of a variable are also invalid.

```
int Spaces are bad;
```

Most likely, adding characters that aren't letters will break the compiler. Only the underscore and letters can be used for identifier names. As fun as it might be to use emoticons for a variable, it would be quite difficult to read when in use with the rest of the code.

```
int ADifferenceInCase;  
int adifferenceincase;
```

The two variables here are actually different. Case-sensitive languages like C# do pay attention to the case of a character; this goes for everything else when calling things by name. Considering this: `A` is different from `a`.

NOTE: Trained programmers are often taught a variation of naming conventions, which yields easier-to-read code. Much of this is dependent on scope, which we will discuss in Section 4.4.4. There are also conventions which always prefix variable names with an indication of what sort of data is stored by that variable.

As a programmer, you need to consider what a variable should be named. It must be clear to you and anyone else with whom you'll be sharing your work with. You'll also be typing your variable name many times, so they should be short and easy to remember and type.

The last character we discuss here is the little strange `@` or `at`. The `@` can be used only if it's the first character in a variable's name.

```
int @home;  
int noone@home;
```

In the second variable declared here we'll get an error. Some of these less regular characters are easy to spot in your code. When you have a long list of variables it's sometimes best to make them stand out visually. Some classically trained programmers like to use an underscore to indicate a class scope variable. The underscore is omitted in variables which exist only within a function.

You would find the reason for the odd rule regarding `@` when you use `int`, which is reserved as a keyword. You're allowed to use `int @int`, after which you can assign `@int` any integer value. However, many programmers tend to use `MyInt`, `mInt`, or `_int` instead of `@int` based on their programming upbringing.

Good programmers will spend a great deal of time coming up with useful names for their variables and functions. Coming up with short descriptive names takes some getting used to, but here are some useful tips. Variables are often named using nouns or adjectives as they describe an attribute related to what they're used for.

A human character can often have a health attribute. Naming this `HealthPoints` or `NumberOfHealthPoints` is sometimes considered too wordy, or even redundant. However, if you and your friends are accustomed to paper role-playing games then perhaps `HitPoints` would be preferred.

In the end once you start using the name of the variable throughout the rest of your code, it becomes harder to change it as it will need to be changed everywhere it's used. Doing a global change is called *refactoring*, and this happens so often that there is software available to help you "refactor" class, variable, and function names.

NOTE: You may also notice the pattern in which uppercase and lowercase letters are used. This is referred to as either BumpyCase or CamelCase. Sometimes, the leading letter is lowercase, in which case it will look like headlessCamelCase rather than NormalCamelCase.

Many long debates arise between programmers as to which is correct, but in the end either one will do. Because C# is case sensitive, you and anyone helping you should agree whether or not to use a leading uppercase letter.

These differences usually come from where a person learned how to write software or who taught that person. The use of intermixed uppercase and lowercase is a part of programming style. Style also includes how white space is used.

When you name variables in Unity 3D the use of CamelCase or BumpyCase will automatically separate the different words indicated by an uppercase letter in the Inspector panel. This doesn't actually affect how the declaration was written. This will only change how your variable's name appears in the Unity 3D editor.

3.10.1 UsingVariables in Unity 3D

Open the UsingVariables project in Unity 3D to follow along.

Unity 3D is a great way to demonstrate how these variables work. When you add variables of different types and Unity 3D parses the information, the variables will show up in Unity 3D editor. The Inspector panel we looked at before is useful for a great number of reasons.

```

1  using UnityEngine;
2  using System.Collections;
3
4  public class Example : MonoBehaviour
5  {
6      int SomeInt;
7      int OtherInt = 7;
8      // Use this for initialization
9      void Start ()
10     {
11
12     }
13
14     // Update is called once per frame
15     void Update ()
16     {
17
18     }
19 }
20

```

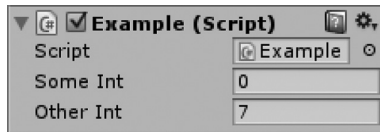
I've added `int SomeInt;` and then an `int OtherInt = 7;` on the line after. To make these visible to the editor we need to add the keyword "Public" before the variable in MonoDevelop.

```

public int SomeInt;
public int OtherInt = 7;

```

The keyword `public` is used to share data outside of the class's encapsulation. With the `public` keyword added before the `int` keyword, the `SomeInt` and `OtherInt` will now appear in the Inspector panel in Unity 3D editor. Don't forget to save the file before jumping back to Unity 3D to observe the changes.



When we added `= 7` to the `OtherInt` we told C# that we're creating a default value to the Object's Script. The first `int` we declared is initialized to 0 when it's added to the Inspector. Many variable types are initialized with a default value if nothing is provided when they are declared.

Initialization is the process by which variables and other chunks of data are created and stored for the first time in a class object. Initialization can be either automatically declared, like `SomeInt`, or declared as variables are initialized, such as `OtherInt`. Later on, we can control how to be more explicit how variables are initialized.

If you make changes to the default values and save the scene, the changes will be saved to the script. To bring them back right click on the script's component title bar and select [Reset] to revert the values to those declared in the script.

In the Inspector panel in Unity 3D we have some new things to look at under the Example script's parameters. Our variables have been added as number fields that we can make changes to. With the new parameters added you can make changes as long as they are valid `int` values. You might notice that you can't add in a . as this would change the number into something other than an integer.

Integers or `int` are whole numbers. Numbers which have a decimal in them are not integer values. Such numbers, having values with fractions, have several different names, for example, float, or double. We'll go into some more depth as to what types mean in Chapter 4.

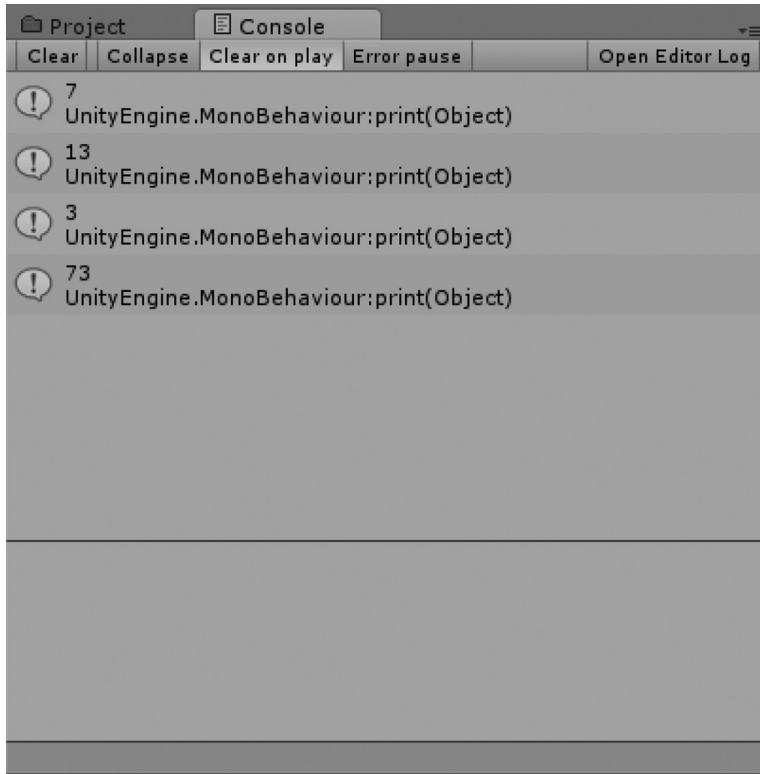
3.10.2 Variable Assignment

We stepped through some basic guidelines regarding assignment of variables, but what happens when we change the assignment? The variables' assignment happens one line at a time. The computer starts at the top of the code and works its way down one line at a time. What this means is that the value of a variable shows its changes only after the assignment is made.

For instance, in the following code sample we'll start off with one variable and change its assignment several times and print the results.

```
using UnityEngine;
using System.Collections;
public class Variables : MonoBehaviour {
    //Use this for initialization
    void Start () {
        int MyVariable = 7;
        print (MyVariable);
        MyVariable = 13;
        print (MyVariable);
        MyVariable = 3;
        print (MyVariable);
        MyVariable = 73;
        print (MyVariable);
    }
    //Update is called once per frame
    void Update () {
    }
}
```

With the code included in a new C# script named `Variables` assigned to the Main Camera in a new scene, we'll get the following output in the Console panel.



Using the same variable named `MyVariable` and after assigning it different values, we get different numbers printed to the Console panel. This is because the identifier `MyVariable` is nothing more than its value; it's just a number. In other words we could use the following code fragment and get the same result from both print outputs.

```
int MyVariable = 7;
print (MyVariable + MyVariable); //prints 14
print (7 + 7); //prints 14
```

If we want to mix variables together we can do that too. Because `MyVariable` is an integer value, we can treat it like any other integer, or whole number. This can be demonstrated best with another `int` variable.

```
int MyVariable = 7;
int MyOtherVariable = 3;
print(MyVariable * MyOtherVariable); //prints 21
print(MyVariable * 3); //prints 21
```

Here we'll get an expected 21 printed out to the Console panel from both of the print functions. To be clear, `MyVariable * MyOtherVariable` and `MyVariable * 3` are equivalent; next, let's consider what happens when we assign a variable to another variable:

```
int MyVariable = 7;
int MyOtherVariable = 3;
print(MyVariable * MyOtherVariable); //prints 21
MyVariable = MyOtherVariable; //what's happening here?
print(MyVariable * MyOtherVariable);
```

Now we have `MyVariable` being assigned the value that is `MyOtherVariable`. Therefore, let's think through this logic. `MyOtherVariable` is assigned a value of 3, so if we assign `MyVariable` a

value of 3 the second print function will return `MyVariable * MyOtherVariable`. This means it's actually just $3 * 3$, which means 9 will be printed out to the Console panel.

3.10.3 Putting It Together

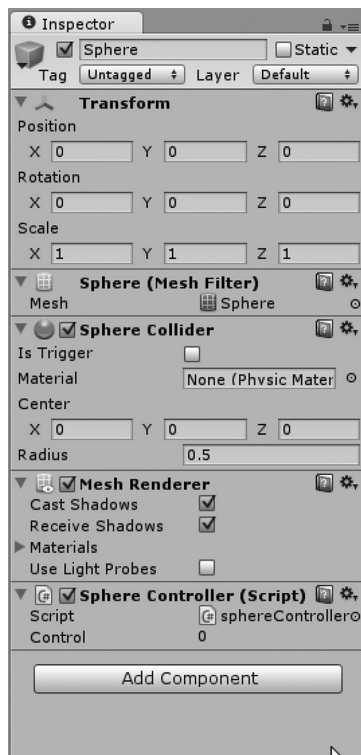
We've named variables and given them assignments. It's about time we start to consider how to apply these new concepts to something useful in Unity 3D. Because variables can be used as the values that they store we'll need to see how that works in Unity 3D.

We'll start off with a basic scene with a little sphere in it; we'll be doing this often. Select the `GameObject` → `Create Other...` → `Sphere` and a sphere will appear in your scene. Create a new C# file named `sphereController` and assign it to the sphere we just created.

Next, open the `sphereController` script in `MonoDevelop`. To start with, we're going to add in a public variable at the class level. We will be getting to what this all means in Section 4.4.3, but for now what we're doing is making this variable available to the rest of the functions as well as the editor.

```
using UnityEngine;
using System.Collections;
public class sphereController : MonoBehaviour {
    public float Control;
    //Use this for initialization
    void Start () {
    }
    //Update is called once per frame
    void Update () {
    }
}
```

Your code should look like the one above. Now hopping back into Unity 3D you should see `Control` in the Inspector panel.

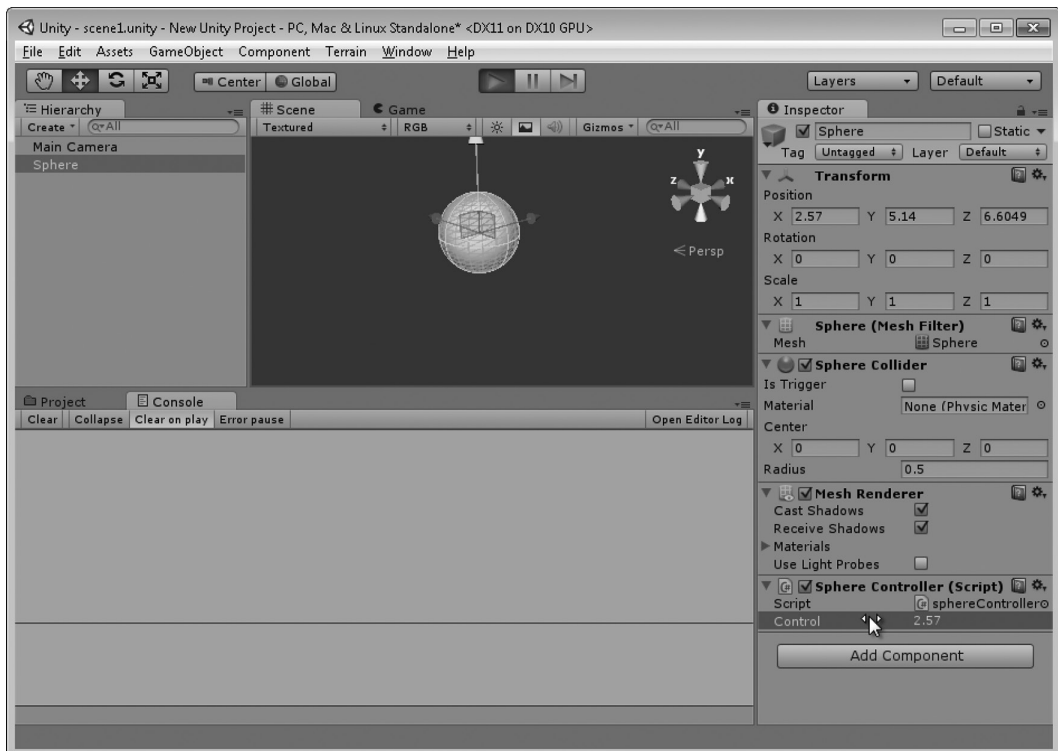


This little thing has a 0 in it and so far that's what we want. We'll be making use of this variable in the `Update ()` function found in the class. Let's add the following code to the `Update ()` function and see how it works.

```
void Update () {
    transform.position = new Vector3(Control, Control + Control,
Control * Control);
}
```

This will move the sphere such that its x position is simply the number in `Control`. The y value is `Control + Control` and the z value is `Control * Control`. To play with this simply select the Sphere in the Hierarchy panel, and then left click + drag on the `Control` label in the Inspector panel. Press the Play in Editor button up at the top and watch the update happen as we change the value stored at `Control`.

When the Play button is pressed the class we attached to the sphere is immediately instantiated into an object. This process is easily overlooked and taken for granted, but remember, a class is a blueprint for making objects. Therefore, every class that is attached to a `GameObject` tells Unity 3D to instance that class and attach it to the `GameObject`. So the `GameObject` behaves as expected.



The value that is being modified in the `Control` script will be directly reflected in the Position under the Transform information at the top of the Inspector. By adding another variable we can make this even more interesting.

```
public float OtherControl;
```

Add in another `public float OtherControl;` on the line following `public float Control;` this will add in another value which we can slide around.

```
void Update ()
{
    transform.position = new Vector3(
        Mathf.Sin(Control) * OtherControl,
        Mathf.Cos(Control) * OtherControl,
        Control * OtherControl);
}
```

Play with this for a little bit; just make sure `OtherControl` is a value other than 0. You should see a spiraling ball moving around, thanks to `Mathf.Sin()` and `Mathf.Cos()`. Basically, these two `Mathf` functions return the sine and cosine of control. We've now seen some basic examples of how to use variables in Unity 3D.

An alternative to the above code which is more readable may look like the following:

```
void Update ()
{
    Vector3 vec = new Vector3();
    vec.x = Mathf.Sin(Control) * OtherControl;
    vec.y = Mathf.Cos(Control) * OtherControl;
    vec.z = Control * OtherControl;
    transform.position = vec;
}
```

Another possible option is to use the following:

```
void Update ()
{
    float x = Mathf.Sin(Control) * OtherControl;
    float y = Mathf.Cos(Control) * OtherControl;
    float z = Control * OtherControl;
    transform.position = new Vector3(x, y, z);
}
```

The nuances between the different versions require some understanding of the different data types involved. The variables that you were just introduced to are being put to use in unfamiliar ways. The different uses will become more clear as we are exposed to a more varied set of situations. As we explore the code throughout the rest of the book, we'll become more acclimated to unique uses of different data types.

Of course, there are many things we can try out here, and it's best you try them out on your own. There are many other `Mathf` functions to play with, but we'll leave that for Section 6.5.1.2. For now it's best to explore how variables behave and how we can use them.

3.10.4 What We've Learned

We're taking very small steps to start off with. There's a great deal of new concepts for you to absorb and many questions to be asked. If anything up to this point is unclear then take a moment to go back and review.

So far we've become familiar with some basic terms related to writing C#. Learning about programming is somewhat about learning new vocabulary, and we've covered keywords and a few operators and variables. We just added some variables to a C# class object and added it to an object in our Unity 3D scene.

Variables, numbers in particular, demand different amounts of memory based on what type of number is being stored.

C# is an OOP paradigm. Every class is an object. Once we learn how to write our own objects, we'll learn how to read other objects and possibly even begin to understand the complexity laid out in Unity 3D's library.

3.11 Types: A First Look

When we assign a variable a value such as `int i = 1;` we're making an interesting assumption. We assume using `1` automatically infers a value type that matches the variable type. Computers use a wide variety of different types of data for storing numbers.

So far we've been seeing the word `int` being thrown around as though you know what an `int` is. In our everyday lives we hardly think there's any difference between numbers `1` and `1.0` or even the word *one*. As humans we can easily conceptualize numbers as units of measure and converting between them. The conversion between the word *one* and the number `1` isn't so easy for a computer.

The keyword `int` is short for integer. Integers are whole number values. The integer is a C# built-in type. Basically, this means that integers are a fundamental part of C#. Other built-in types include `float`, `double`, `string`, `bool`, and `object`. Less commonly used built-in types are `char`, `short`, and `byte`. Altogether, there are 15 different built-in types. All of these, except `object` and `string`, are value types.

Every type of data you're able to build must be based on all of these built-in types. The system which creates all of these `floats`, `doubles`, and `bools` is rooted in the origin of computing. Remember the punch tapes and pieces of paper with holes in them? Those were records of 1s and 0s which were fed into computers for storing in the form of mechanical switches, either on or off. The patterns represented numbers or instructions and logic.

Today, we still use the same system of 1s and 0s, or binary, only we don't need to punch holes in paper to tell the computer what to do. The methods of creating and storing the instructions have become many times more complex, but nonetheless the principle of the 1 and 0 are the same. These are called *bits*; one possible origin for that name is the little bits of paper that were left over from punching all of the holes in the paper cards.

3.11.1 Value and Reference Types

The `int`, `float`, `double`, and `bool` are commonly used value types. The basics of this usage pattern relate to the system that stores and organizes the computer's memory. When you talk about computer storage you use the word *megabyte* or *gigabyte*. Value types are stored with a very specific purpose in mind: to keep track of a numeric value.

NOTE: The word *mega* refers to how many millions of bytes a component in your computer can store. *Giga* indicates how every many billions of bytes are being stored. Have you ever thought about what a byte actually is?

A byte is what is called an 8-bit unsigned integer, or a system of using 8 bits to form a number. What this means is that it's a whole number like 0, 7, or 32,767. The word *unsigned* indicates that the number cannot be negative, like -512.

A 1 or 0 in computer terms is called a bit. We won't go into detail on how computers use bits to count, and it's a rather fun thing to learn on your own. However, it's important to know that a byte has a limited range, from 0 to 255. I'll just leave you with the idea that you could count up to 1023 on your 10 fingers if you used binary rather than decimal. Your fingers can be used to represent a 10-bit unsigned integer.

The computer's calculation capabilities used to be far more limited than they are today. An 8-bit game console made in the 1980s had a limited number of colors, 256 to be exact. This limitation was based on the number of bits that the processor could handle. The processor had a limited number of transistors which could be used at any one time. Shortly after, floating point coprocessors were introduced, which had a much larger numeric range allowed, by having closer to 32 bits to work with.

In all of these cases, when you use a value type, that number is actually stored as 1s and 0s in your computer's RAM. This should be considered fairly remarkable. In the past you'd have to go through flaming hoops to store a value. Now you just type in `float f = 3.1415926535;` and you can measure the circumference of the universe accurately to within the width of a single atom.

What all of these types have in common is that they are stored as a single element. Large values like `double gigawatts = 1210000000.0;` and `double mint = 2.0;` use the same amount of memory.

Your computer doesn't assign one double or the other more space in memory. These types are referred to as *primitive types*. The only difference between an `int` and a `float` is the number of bits they use at a time, 8 and 32 respectively; doubles use 64 bits.

The `string` and `object` types differ a bit in how they are stored. These types are a composite of any number smaller elements. The bigger a string or object the bigger chunk of memory the computer opens up to place that object or string. These are called *reference types* or sometimes called *nullable types* because C# doesn't look to a single element in memory to get data from it.

We will explain nullable types in Section 4.4.1.2, but in short a nullable type is a space in memory which is reserved for data that has yet to be fulfilled. Primitive or value types are commonly reserved and assigned at the same time. There are systems which allow this to be changed, but we're getting ahead of ourselves for now.

In Unity 3D, a commonly used type is the `Vector3`. Vectors, in the math world, are directions in *x*, *y*, and *z* with a length. Unity 3D uses vectors to keep track of a position in 3D space. The `Vector3` type is a composite of three `float` variables. This means that a `Vector3` is made up of different components. From this, we may infer that a `Vector3` is a *nullable* type. Each float is labeled *x*, *y*, and *z*. However, unlike the primitive types the `Vector3` needs to be instantiated. This means that you can't simply use the following syntax.

```
int i = 0; // a valid initialization of an int.
Vector3 v1 = (x = 0, y = 0, z = 0); // not valid initialization.
Vector3 v2 = new Vector3(); // this is how it's done.
```

Nullable types need to be initialized; that is, they need some form of initial value. We'll go into further detail on this later. It's important to remember there are differences between types when they are being initialized. Later on, as you begin to create and use objects and variables it'll be important to remember how to initialize the different variable types.

3.11.2 What We've Learned

There are many different forms which data can take. Each form is called a type. The different forms depend on how they are organized and how they are initialized. Reference or nullable types require some form of initial parameters before they can be used. Value or primitive types usually have an initial value when they are created.

When variables are declared, they are assigned a type. The type changes how the variable behaves and what values it's allowed to store. As we move forward, remembering how types behave is important. C#, and many other C-like programming languages, has specific rules with how to deal with data of different types.

3.12 Strong Typing

This has nothing to do with how hard you type on the keyboard! C# sees a difference between numbers in how they are written. A whole number like 1 is different from 1.0, even though we can think of them as having the same value. Some of these differences can be added by adding a letter. Therefore, 1.0 and 1.0f are actually different numbers as well.

C# is a *strongly* typed programming language. When a programming language uses strong types you're expected to keep the different types separated. This means that a `1 + 1.0f` can create a problem. Because the two numbers are actually different types of numbers we have to convert one of them to match the other before the operation is allowed to take place.

To convert from one type to another we have to tell C# that we intend to make that conversion from a 1 to a 1.0 by using what's called a *cast operator*. Aside from changing how memory is used, types also limit any problems which might come about when working in a team of programmers.

If one programmer is new to your game and has made assumptions about hit points being in floating point values, he'd quickly find out as soon as he looked that you're using integer values and change how he intends to use those numbers. Value types not only store a value assigned to them but also provide a great deal of information to be inferred by the programmer reading the code.

In Unity 3D a `Vector3 vec = new Vector3(1.0, 1.0, 1.0);` will throw an error. The number 1.0 is called a `double`, which uses 64 bits, whereas `1.0f`, which is a `float`, uses 32 bits. The difference means `float 1.0f` is half the size of the `double 1.0` in memory. A `Vector3` is made up of three `float` values, not three `doubles`.

Therefore, in terms of declaring a variable and assigning it `float f = 1.0f` and `double d = 1.0;` the same type problems exist as with assigning a `Vector3()`, which is made of three different `float` values. To properly tell C# you mean a 32-bit version of 1.0 would be to add an `f` after the number: `1.0f` means you want to use the 32-bit version of 1.0 and not the 64-bit version of 1.0. Therefore, to declare `Vector3 vec = new Vector3(1.0f, 1.0f, 1.0f);` is the correct way to assign a proper `Vector3()`.

Confused? I'd imagine so; we'll clear things up in a bit. Just hang in there.

For our purposes of learning how to code, knowing anything about bits may not mean a lot. However, if you ever talk to a programmer it'll be very important that you know the difference. Before getting too far ahead we should know that Unity 3D uses `floats`, `ints`, and `doubles` quite often. There is also a difference between the different data types and that they cannot be simply used interchangeably without consequence.

3.12.1 Dynamic Typing

Not all programming languages are so strict with types, but it's a good to get used to working with strict types before learning a more lazily typed programming languages, like Lua (<http://www.lua.org/>). Type conversion is important for learning more complex languages like C, or C++. These languages offer a wider number of platforms and a more detailed level of control.

With dynamically typed languages like Lua or UnrealScript, you can run into strange hard-to-track-down bugs. These are sometimes called "Duck"-typed languages; this refers to the proverb "if it looks like a duck and it sounds like a duck, it's probably a duck." However, when it comes to a type we might see something like this:

```
var i = 1;
if (i)
{
    Console.WriteLine("is it true or a 1?");
    i = i * 0.1;
}
```

In the above it's not clear what the variable `i` is supposed to be, is it a `bool` or an `int` or a `double`? When `var` above starts off as an integer but we use it as a `bool` and then multiply it by a floating point value, what is `i`? What have we turned it into if we want to use it as something else, or how do we turn it back?

As we will see in Section 7.14.4, we are allowed to make some exceptions to the type of a variable. The keywords `var` and `dynamic` do mean you're expecting an unexpected type to be coming, but once it's there, you had better know what to do with it.

3.12.2 What We've Learned

We have glossed over the specifics of how numbers work in the computer's memory, but we'll get into that in Section 6.20.2. Superficially, we should know that not all numbers are the same. A byte is 8 bits, a nibble is 4. An `int` is 32 bits, and an `int64` is 64 bits, but that's not all. A `float` is also 32 bits and a `double` is 64 bits, but these can't hold the same values as an `int` or `int64`.

The `float` and `double` use some of their bits called a mantissa to hold the numbers after the decimal. Therefore, 1.01 requires some of the bits to represent the 1, and these are called the significant digits.

The mantissa uses some bits to represent the other 1 in the number. Then there are more bits set aside to tell C# where the dot (.) goes in the number, and these bits are called the exponent. Therefore, with 1.02, the exponent tells the mantissa 2 to move an extra zero after the dot, so 1.02 is the final result.

We take for granted the work going on under the hood to store a simple number. All of this happens thanks to many computer scientists who have worked in the last several decades. Konrad Zuse and John von Neumann introduced a system of floating point representation to computing in the 1940s, with the Z1 and later the Z3. Their system is roughly the same system in use today.

3.13 Type Casting, Numbers

This is a simple exercise, but an important one nonetheless. When you start to deal with keeping score, or recording injuries to a monster, it's important that behaviors are predictable. Many problems begin to show up if the math you're using involves mixing numbers with decimal values like floats or doubles, or with integers which have no decimal value.

```
int i = 100;
```

Decimals are not allowed on an `int` value.

```
float f = 100.0f;
double d = 100.0;
```

Decimals are allowed for both `float` and `double`. The only caveat is that a float requires an `f` post fixed to the number when it's assigned.

Converting types between one another is pretty simple when you need to deal with numbers. When we start creating our own classes the task of casting becomes a bit more detailed, but not necessarily more difficult; there's just a bit more work involved. We'll get to that soon enough.

When we go between number types or the built-in value types (POD), we work with things which seem to come with C#. So far we've seen things like `int` and `float`. Integer values like 1, 7, and 19 are useful for counting. We use these for counting numbers of items in an array, or how many zombies are in a scene, for instance. Integers are whole numbers, even though we might be counting zombies whole or not.

Once we start needing numbers with fractions we need to use float values. To get an object's speed or *x*, *y*, and *z* coordinates in space we need to use values like 12.612f, or `x = 13.33f`, to accurately place an object in space. If not, objects would move around in a scene as though they were chess pieces locked to a grid.

When a float value exists as 0.9f we lose some values after the decimal when converting the floating point value to an integer.

```
void Start ()
{
    float a = 0.9f;
    int b = (int)a;
    Debug.Log(b);
}
```

If we use the above code in a new script called `Casting.cs` attached to the Main Camera in a new Unity 3D project we'll get the following output to the Console panel.

```
0
UnityEngine.Debug:Log(Object)
Casting:Start () (at Assets/Casting.cs:10)
```

Without the cast operator `int b = a;` we get an error telling us we need to cast the value before assigning it.

```
Assets/Casting.cs(9,21): error CS0266: Cannot implicitly convert type 'float'
to 'int'. An explicit conversion exists (are you missing a cast?)
```

The first part of the error message says “Cannot implicitly convert type,” which has a very specific meaning that we’ll get to before the end of this chapter, but first let’s cover explicit type casting.

3.13.1 Explicit versus Implicit Casting

As we have seen, in the `Start ()` function where we declared `float a = 0.9f`; `a` was converted to 0. Even though we were only 0.1f away from 1 and 0.9f away from 0 we’re left with a 0 instead assigned to `int b`. To do these conversions we use the explicit casting method, as shown with `(int)a`, which explicitly tells C# to convert the value stored at `a` to a float value. When we are required to use an explicit cast, we’re also warned that there may be some data lost in the conversion.

Casting is a process by which C# changes the type of a variable to another type. This may or may not involve a loss of information. In some cases, the conversion doesn’t require the explicit cast operator. The following code casts an `int` to a `float`.

```
int c = 3;
float d = c;
Debug.Log(d);
```

With the above code we get a 3 printed out to the Console panel. We didn’t need to use an explicit casting operator. This is because when we cast from an `int`, in this case 3, there would be no data loss when it’s cast to a float. This is possible through an implicit cast. Implicit casts are allowed only when there is no data lost in the process of converting from one type to another.

There are no integer values which will result in a loss of any value when it’s converted to a float. Technically, there are more float numbers between 0 and 1 than there are integer numbers between 0 and 1, or any other sequence of two numbers for that matter.

3.13.1.1 A Basic Example

Create a new project called `NumberTypes` and assign a C# script to the Main Camera called `NumberTypes.cs`. In the `Start ()` function include the following code.

```
void Start ()
{
    int a = 1;
    double b = 0.9;
    int c = a * b;
    Debug.Log (c);
}
```

If we look at what is going on here we’ll want to think for a moment about what it means. The integer `a` is set to 1 and `double b` has 0.9 assigned. We should assume that 0.9 will be assigned to `c` after the multiplication, but this assignment is stopped by a type conversion error. C# usually gives us pretty clear reasons for its errors. In this case we know there’s a cast missing between an `int` and a `double`.

```
Assets/NumberTypes.cs(10,21): error CS0266: Cannot implicitly convert type
'double' to 'int'. An explicit conversion exists (are you missing a cast?)
```

The error states we can’t implicitly convert a `double` to an `int`; why not? 1 certainly looks like 1.0, or so it seems. However, 0.9 can’t be turned into an integer. There’s no integer between 0 and 1. Even though 0.9 is very close to being 1, it’s not.

```
int c = a * (int)b;
```

We use type conversion to tell C# to explicitly change `b` from a `double` to an `int` by preceding the `b` with `(int)`. What value is going to be assigned to `c` if it can't be 0.9? The `(int)` operator is an explicit cast.

```
0
UnityEngine.Debug:Log(Object)
NumberTypes:Start () (at Assets/NumberTypes.cs:11)
```

0 is less than 0.9; we've lost data going from the double value to an integer value. Even though 0.9 is almost a 1, the first `int` value is 0, followed by values that are cut off by the type conversion. This is why type conversion matters.

We will find other casts which look like this in Sections 6.14 and 6.20, but we're introducing the concept early on as it's a very common problem to come across. The syntax `(int)` turns a number following this operator into an `int`. Likewise `int i = 0; double d = (double) i ;` is a way to cast an `int` into a `double`. However, this isn't always necessary.

Some conversions take place automatically as an implicit cast. In the above example we can use the following code without any problems.

```
void Start ()
{
    int a = 1;
    double b = 0.9;
    int c = a * (int)b;
    Debug.Log (c);
    double d = a;
    Debug.Log(d);
}
```

Here we assign `double d = a;` where `a` is `int 1`, which we know isn't a `double`. This produces no errors. The same goes if we add another line `float f = a;`. In this case `f` is a `float`. These are called implicit casts. An integer value doesn't have a mantissa or an exponent. Both the `double` and `float` do have these two possibly large and important values. These two terms were explained at the end of Chapter 2.

When mashing a `double` into an `int` we lose the mantissa and exponent. By using an explicit cast, we tell C# that we don't mind losing the data. This doesn't mean that an implicit cast will not lose any data either. An `int` can hold more significant values than a `float`. We can observe this with the following lines of code added to the `Start ()` function.

```
Debug.Log(largeInt);
float largeFloat = largeInt;
Debug.Log(largeFloat);
int backAgain = (int)largeFloat;
Debug.Log(backAgain);
```

This code produces the following console output, after a bit of cleaning up:

```
2147483647
2.147484E+09
-2147483648
```

When we start with the value 2147483647 assigned to `int` we're at one extreme of the integer value. This is the biggest number the `int` can hold, for reasons discussed in Section 3.11.1, but in short, it's because it's using only 32 bits to store this number.

If we cast this into `largeFloat` we can use an implicit cast, which converts the value from `int` to `float`. In this case we see only 2.147484 followed by the exponent E+09, which tells us that the dot (.) is actually nine places over to the right.

When we convert the float back into an `int` with an explicit cast we get a -2147483648, which is certainly not what we started with. This tells that there's some significant information lost in the conversion from float to `int`, but there were also numbers lost in the implicit cast from `int` to float.

This still happens if we remove a digit from the end.

```
int largeInt = 214748361;//cutting off a digit and ending in 1
Debug.Log(largeInt);
float largeFloat = largeInt;
Debug.Log(largeFloat);
int backAgain = (int)largeFloat;
Debug.Log(backAgain);
```

This sends the following numbers to the console:

```
214748361
2.147484E+08
214748368
```

The last digit is changed from 1 to 8. The 8 is coming from some strange conversion when we go from `int` into `float`. Even if we look at the numbers being stored in the float value we can tell that there's already a change in value. This tells us that in general, we need to be very careful when converting from one type into another.

Logically, another difficult problem is converting from a string to a number value. When you use `string s = "1";` you're not able to use `(int)s` to convert from a string to an `int`. The types of data are very different since strings can also contain letters and symbols. The conversion of `(int) "five"` has no meaning to the computer. There's no dictionary which is built into C# to make this conversion. This doesn't mean that there are no options.

```
string s = "1";
int fromString = int.Parse(s);
Debug.Log(fromString);
```

The code added to the `Start ()` function will produce the following number:

```
1
```

There are plenty of options when faced with dealing with significantly different types. The `int.Parse()` function is an option that we can use to convert one value into another. Again, we might be getting a bit ahead of ourselves, but it's important to see that there are many different ways to convert between different types.

3.13.2 What We've Learned

In this chapter we used some syntax which has not been formerly introduced. For the sake of seeing what happens to numbers between casting we needed to do this. In Section 6.20, we will cover the details of what happened in this chapter, so don't worry. Everything in this chapter will be explained in detail soon enough.

We've still got a lot to cover about types. It's a bit early to see how they relate to one another and why there is a difference. Once we move into evaluating numbers we'll start to see how the different types interact. Accuracy is one place where we begin to see the effects of type conversion. A float with the value of 0.9 turns into 0 when it's converted to an `int`.

This chapter is just a cursory look at types. There's a lot more to conversion than just losing some numbers. Later on we'll see what happens when we need to convert between different game play characters like zombies and humans. Many attributes need to have specific conversions but then they're both bipedal creatures.

3.14 Comments

Just before the `Start ()` and the `Update ()` functions you'll notice some green text in MonoDevelop. These two lines, which state `//Use this for initialization` and `//Update is called once per frame`, are comments. The `//` operator means that the `//` and anything following it is now invisible to the compiler.

Comments are left as breadcrumbs to help keep notes in your code. Once a new line is started the `//` no longer applies and the next line is now visible.

```
//these are not the lines you are looking for...
```

Programmers use comments for many reasons. Mostly, comments are used to describe what a function is used for. Quite often, comments are left as notes for both the person who wrote the code and for others. Sometimes, when a function is confusing, or isn't always working, a comment can be left as a "to do" list.

Once we start writing our own functions we'll want to leave comments to help ourselves remember what we were thinking when we wrote them. Once in a while, if we need help, we can indicate to our friends our intentions. You can leave comments like `//I'm still working on this, I'll get back to it tomorrow...` if you're still working on some code.

Or you can ask for help: `//help, I can't figure out how to rotate the player, can someone else do this?!` is a simple short comment. Even professional programmers leave comments for their coworkers, letting everyone who may be looking at their work a clue as to what was going on in their code.

In more than one case, the comments in a large code base needed to be cleaned up before releasing to the public. The engineers responsible needed to remove foul language from the comments to avoid any public ridicule. In some instances the comments in the code were also defaming competing game engines.

When a comment requires more than a single line, use the following notation:

```
/* this is a multiline comment
which can carry on to other lines
you can keep on writing anything
until you get to the following */
```

Anything between the `/*` and the `*/` will be ignored by the C# compiler. It's often a good habit for a programmer to introduce a class by leaving a comment at the top of the file. For instance, it's common to see the following in a class written for other people to edit.

```
/* Player class written by Alex Okita
This class manages the player's data and logic
*/
```

The use of comments isn't limited to simple statements. They are often decorated by extra elements to help them stand out.

```
/******
* This comment was written by Alex Okita *
******/
```

Comments have several other uses which will come in handy; for instance, when testing code you can comment a section of code that you know works and test out new code next to the code commented out. If your test doesn't work you can always clear it out and un-comment the code that does work. We'll play with this idea later in this chapter.

Using comments on the same line of code is also a regular practice.

```
void MyFunction () {
    int someInt = 0;//declaring some regular integer as 0
}
```

We can inform anyone reading our code what a specific statement is doing. Although we want to be smart with how our comments are written, if we add too many comments in our code the actual statements that matter get lost between a bunch of words that don't matter.

Comments are also useful for testing different collections of code. When you're learning about how different behaviors affect the results of your functions it's often useful to keep different versions of your code around. This is often the case when trying to debug a problem and eliminate lines of code without having to completely get rid of them.

```
void MyFunction () {
    int someInt = 0;//I print 0 through 10
    //int someInt = 3;//starts at 3
    //int someInt = 11;//this won't print
    while(someInt < 10) {
        print(someInt);
        someInt++;
    }
}
```

It's possible to test out various test cases using comments to allow you to pick and choose between different statements. In the above code fragment, we've got `someInt` declared three different ways. Depending on which line is left un-commented we can choose between different values.

We can leave another comment on the line after the statement, reminding us what the result was. For covering an entire section of code it's easier to use the `/* */` form of commenting.

```
void MyFunction () {
    int someInt = 0;//I print 0 through 10
    //int someInt = 3;//starts at 3
    //int someInt = 11;//this won't work, so leave me out.
    //trying out new code here
    for(i = 0; i < someInt; i++) {
        print(someInt);
    }
    //the code below will do the same thing.
    /*
    while(someInt < 10) {
        print(someInt);
        someInt++;
    }
    */
}
```

In above code, we've commented out the while loop following the for loop which accomplishes the same task. In this way we can easily switch between the two different loops by switching which loop is commented out. Programmers often leave sections of code in their functions commented out so they verify the validity of the new code they've added in.

The `/* */` notation for comments can also be used in smaller places. Anytime the `//` notation appears everything on the same line is commented out. Using the `/* */` notation we can hide a segment of a line of code.

```
while(someInt < /*10*/100) {
    print(someInt);
    someInt++;
}
```

For instance, we can hide the 10 in the while loop and replace it with another number altogether. The `/*10*/` is hidden from the computer so the 100 is the only part of the condition that's seen by the computer. This can easily get a bit more ugly as a single statement can begin to get stretched out and become unreadable.

```
Vector3(/* this is the x variable */1.0f, 2.0f/* <- that was the Y  
variable*/,/* the z */3.0f/*variable is here*/);
```

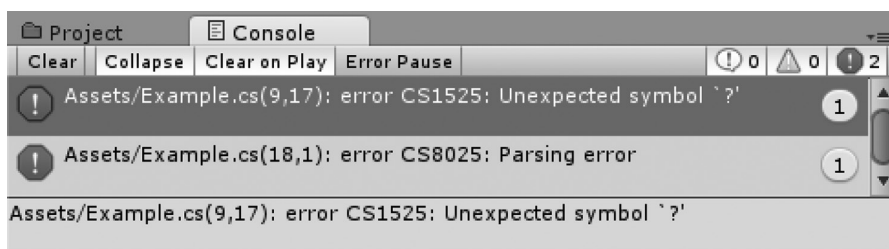
Although the above code is perfectly valid, it's not recommended that something like this appear in your code. When mixing in too many comments a simple declaration can get pretty ugly.

3.14.1 Line Numbers

Each line of code in MonoDevelop is indicated in the margin of the editor to the left. When Unity 3D comes across a typo or some unexpected character it's indicated in the Console panel.

```
// Use this for initialization  
void Start ()  
{  
    ?  
}
```

In this example I just added in `?` in the middle of the `Start ()` function. Unity 3D tells us about this addition with the following output to the Console panel.

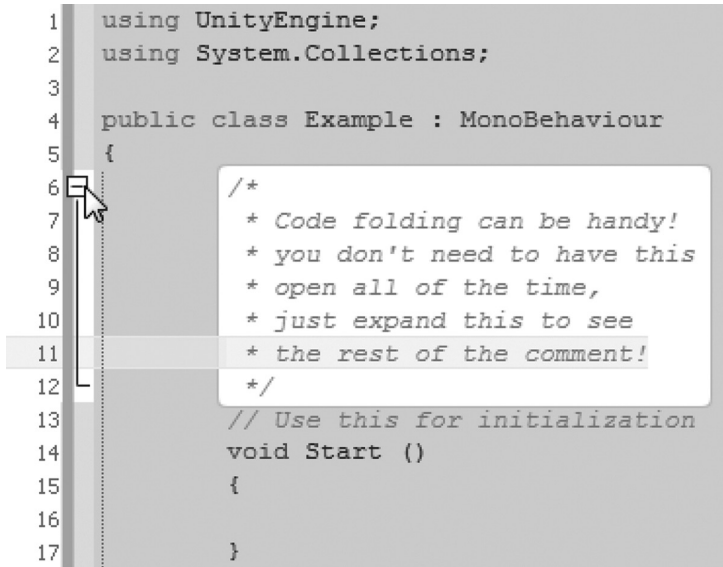


The error tells us: `Example.cs(9,17): error CS1525: Unexpected symbol `?'`. The (9,17) tells us the position where the error is occurring. The first number is the line number. In the source code mentioned, we added in the `?` on the 9th line. The 17 tells us that the character is on column 17, or the 17th character position from the left. If you were to count the number of spaces from the left you'd reach 17 and find the `?`, which Unity 3D is erroring on.

This is a more precise example of an error, but we may not always be so fortunate. In general, the line number indicated is more or less a starting place where to start looking for an error. It's difficult to say when and how various errors take place in code, but in general, syntax errors tend to be fairly easy to fix. Difficulties arise when the code itself is using proper syntax but the result is erroneous or deviates from expected behavior.

When this happens we'll have to use other techniques to track down the bugs. We'll save those techniques for when we're ready for them. For now we'll just stick to learning proper syntax and comments.

3.14.2 Code Folding

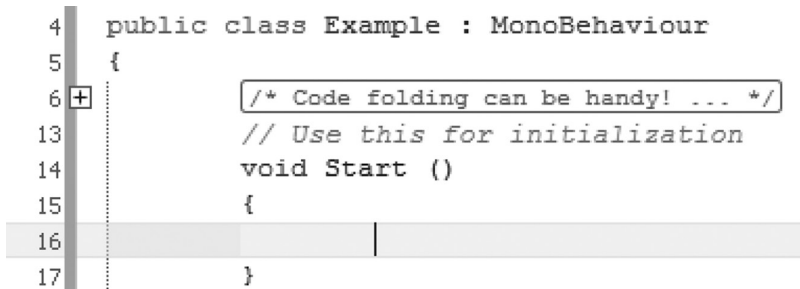


```

1  using UnityEngine;
2  using System.Collections;
3
4  public class Example : MonoBehaviour
5  {
6      /*
7       * Code folding can be handy!
8       * you don't need to have this
9       * open all of the time,
10      * just expand this to see
11      * the rest of the comment!
12      */
13      // Use this for initialization
14      void Start ()
15      {
16
17

```

When you add in a long comment using the `/* */` notation MonoDevelop recognizes your comment.

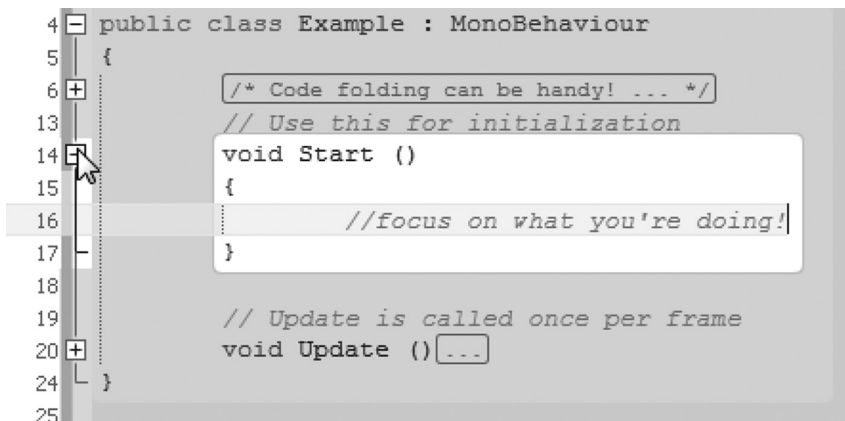


```

4  public class Example : MonoBehaviour
5  {
6      /* Code folding can be handy! ... */
13     // Use this for initialization
14     void Start ()
15     {
16
17

```

Clicking on the little box in the number column will collapse the comment into a single line. This feature comes in most of the modern IDEs, or integrated development environments, meant for writing in C#. This feature is also handy when you're rewriting large segments of code, in that it reduces any distractions.



```

4  public class Example : MonoBehaviour
5  {
6      /* Code folding can be handy! ... */
13     // Use this for initialization
14     void Start ()
15     {
16         //focus on what you're doing!
17     }
18
19     // Update is called once per frame
20     void Update () { ... }
24 }
25

```


We can collapse many blocks of code into a single line. If you look at the line numbering following the `Update ()` function you'll see the numbers jump from 20 to 24. This process of collapsing has hidden three lines of code; moreover, this feature will be particularly handy when you need to see two different functions which might have a great deal of code between them.

```

/* Code folding can be handy! ... */
// Use this for initialization
void Start ()
{
    SomeDistantFunction ();
}

// Update is called once per frame
void Update ()[...]

void SomeDistantFunction ()
{
    //more code here
}

```

Here, we can see the `SomeDistantFunction()` and where it's being used at the same time. If the `Update ()` function wasn't collapsed we might have to scroll up and down to see both the use of the function and the function's contents.

```

void Start ()
{
    for (int i = 0; i < 100; i++) [...]
    {
        // code!
    }
}

```

This also works for any instance where the opening curly brace `{` is followed by the closing curly brace `}`. This folds the `for` statement into a more compact form, so we don't have to look at what it's doing. This allows us to focus on the code block we're most interested in focusing on.

3.14.3 Summary Comments

```

/// <summary>
/// Clevers the comments.
/// </summary>
void CleverComments ()
{
    /*
     * comments inside
     * of a function
     */
}

```

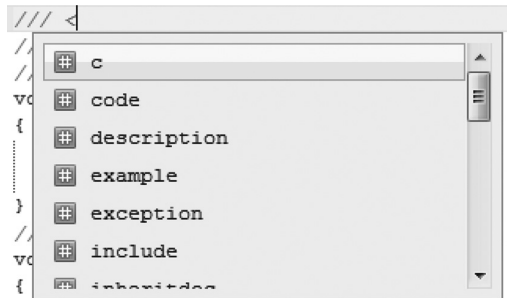
If you type in three `/s` over a function, MonoDevelop will automatically add in a summary comment. This will be filled in with an automatically generated comment summary, which is probably going to be wrong, but it's fun to see what it comes up with.

```

///<summary>
///Blends the fruit.
///</summary>
void BlendFruit() {
    int bananas = 2;
    int strawberries = 6;
}

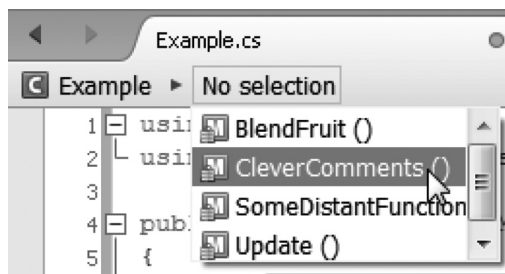
```

Like the other comments the summary comments are also collapsible. MonoDevelop also recognizes many different forms of comment tags.

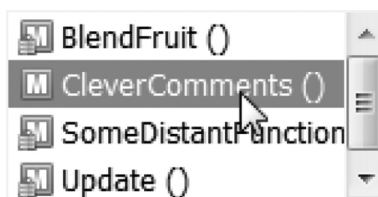


When you add in `///<` the above automatic code completion pop-up dialog appears. You can add in description examples and other tags. This feature of MonoDevelop demonstrates how important comments are in a well-written source file. Comments and comment tags have many different options. To avoid writing the rest of this book about comments I'll leave experimenting with the comment tags to you. They are important, but not essential, to learning how C# works.

3.14.4 Navigating in Code



There are also clever ways to get around in a source file. Up at the top of the Editor panel in MonoDevelop is a handy drop-down menu which has a list item for every function in the class. You'll also notice an icon next to each listing.



Add in a `public` keyword before the `void CleverComments()` function and the lock icon goes away. This indicates that the `CleverComments()` function is public. Selecting the item jumps the cursor

to the beginning of the function. This comes in particularly handy when dealing with large source files which might have many different functions in them.

3.14.5 What We've Learned

Comments are a necessary part of any good source file. When you've written several dozen different C# files you'll find that code written months ago might look alien. I often find myself thinking to myself "What was I thinking when I wrote this?" when I open a file from many months before.

In these cases I hope I wrote in detailed comments, leaving myself at least some clue as to what was going on to lead to the code that has been written; usually, however, I haven't and I need to read through the rest of my code to come up with an understanding of what was going on in my code.

It's time to get back to Unity 3D and start writing something more interesting than comments. We spent a great deal of time in this chapter writing things that are ignored by the computer. It's time to get to writing code that matters.

3.15 Leveling Up: Moving On to Basics

At this point we've looked at the components that make up C#. We should be able to look at code and identify the different tokens and roughly what they do. The `;` is a separator token which ends a code statement. Likewise, we should understand that a `:` and a `;` have different roles in code and cannot be used interchangeably.

Variables have a type, and this type determines how the variable can be used. We should also know that different types cannot interact without explicit conversions between them. Converting between different types can result in loss of data, and thus we should be mindful when doing so.

When we look at a statement, we should be able to identify the different tokens which the statement is composed of. In a statement like `int a = 6;` we should be able to see that we're using five different tokens.

What the different tokens are used for may not be clear, but that information will be presented as we continue. Learning a programming language is just as involved as learning any other language. Just like any language has grammar and vocabulary, C# has syntax, keywords, and structure.

