Lesson One

Snowmen and Variables

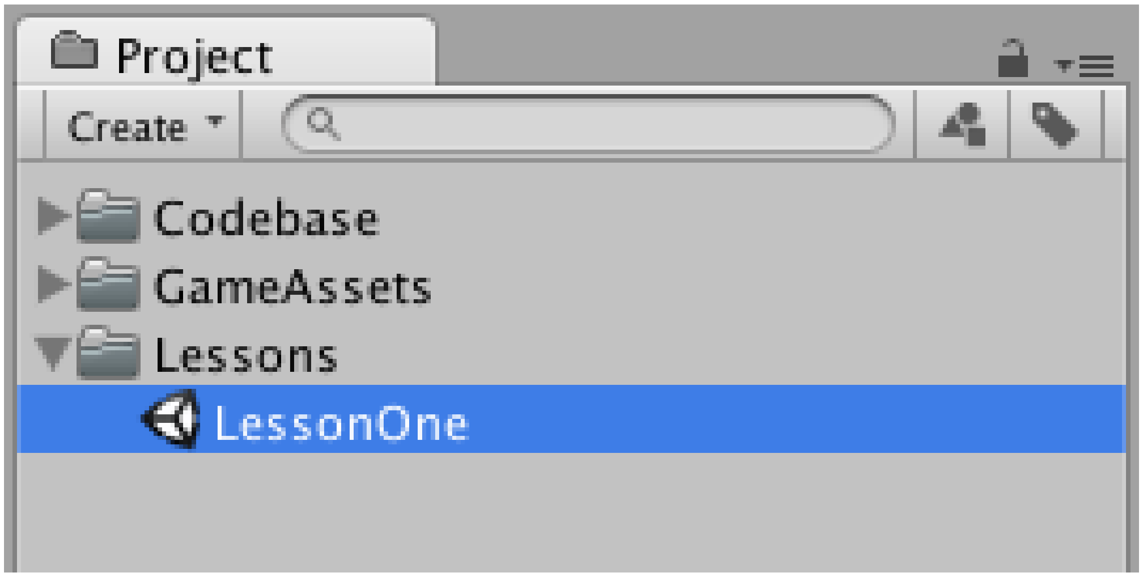


Figure 1: The lesson one "scene" file.

1 Introduction to Lesson One

Video games run on code. They share this trait with all applications on laptops, smartphones, and tablets. They all had to be written by a computer programmer (or more often several working together on a team). Learning to read and write code means that you can not only understand how these applications work, but can even make your own!

The following lessons will walk you through reading, understanding, and writing code. In the process, you'll also fix a series of broken video games and get the chance to make your own.

2 Lesson One: Snowmen and Variables

Definition!

VARIABLE:
The part of code that stores information for use.

Variables are the primary building blocks of all code. You may have run into variables in math class expressed as "x" or "y" and had to solve for them to determine their value. Variables take a far different form in code. They store information and due to the fact that they are "variable" (they can change), they can be used to make changes to the player, enemies, or world of a game.

To begin with open up the project in Unity. If you haven't yet downloaded Unity or the project onto your computer, take a moment to find the "instructions.txt" file that'll walk you through this process. Once Unity is opened, you'll want to click the "Open Other" button, and open the top-most folder of the downloaded project.

Once the project is open, In the "Project" window of Unity find the folder labelled "Lessons". Open the folder and then double click on the "LessonOne" file.

In this lesson we'll focus on fixing a game in which the player needs to stop angry snowmen from reaching a tower. The snowmen cannot be hurt, only stopped by building a wall to protect the tower.

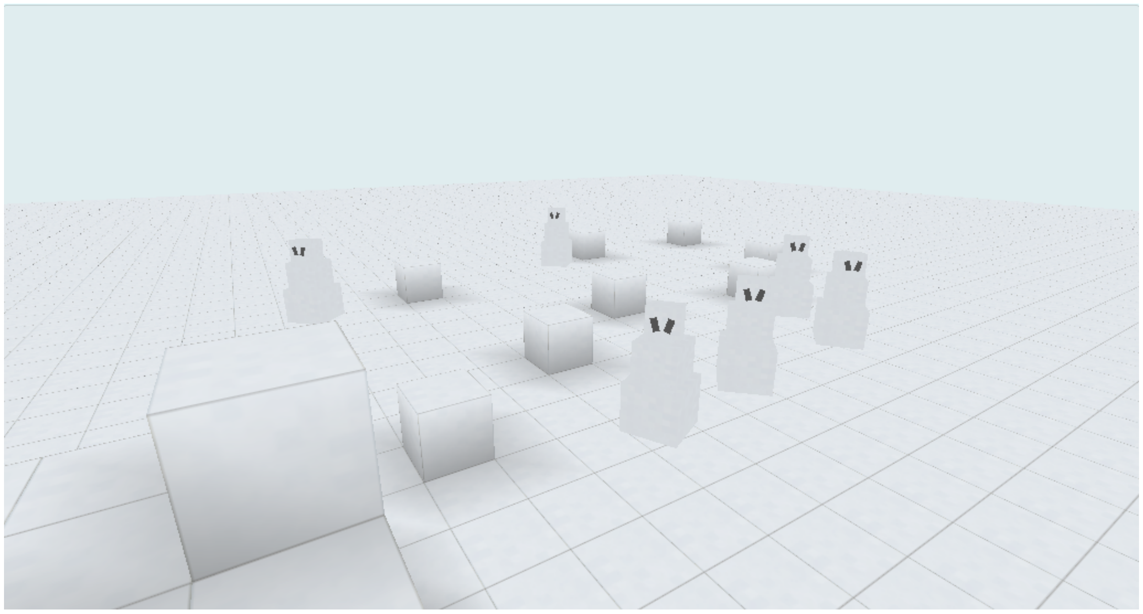


Figure 2: Attack of the Killer Snowmen.

First we'll need to determine how exactly the game is broken. You can start the game by pressing the "play" button near the top of the screen. The controls for the game are to use wasd to move, with mouse to look around. Left click with the mouse to remove blocks and right clock with the mouse to add a new "snow" block. Press space to jump. When the game is done, press escape or backspace to make your mouse cursor visible and click the "play" button again to stop it. (Note: If you can still see your mouse after pressing play you may have to click on the screen once after the game is loaded.)

Play the game, and try to build up a wall to stop the snowmen from reaching the tower as seen in Figure 3.

The problem with the game should be clear from just a single playthrough. The player cannot move fast enough to make a wall in time. The game is not winnable!

In order to resolve this problem, we'll need to make changes to some variables. For that we'll need to start looking at scripts. Scripts are files that store code. When the game engine runs it reads through the scripts in order to determine what to do. In this way they can be understood to be instructions for the game and everything that occurs within it. This means that in order to change something in the game (say the speed of the player) we'll need to change something in a script.

The script that we'll be looking at is called LessonOneGame.cs. You can open it from Unity in the Project View. Open up the "Codebase" folder, then the "Lesson1" folder. Double click on "LessonOneGame" to open it up. At this point, a new application called "MonoDevelop" will open that will actually let you modify the script.

Once MonoDevelop opens (this may take some time), you should see something that looks like Figure 4. (Note: you may get a message asking you to "convert line endings", go ahead and accept

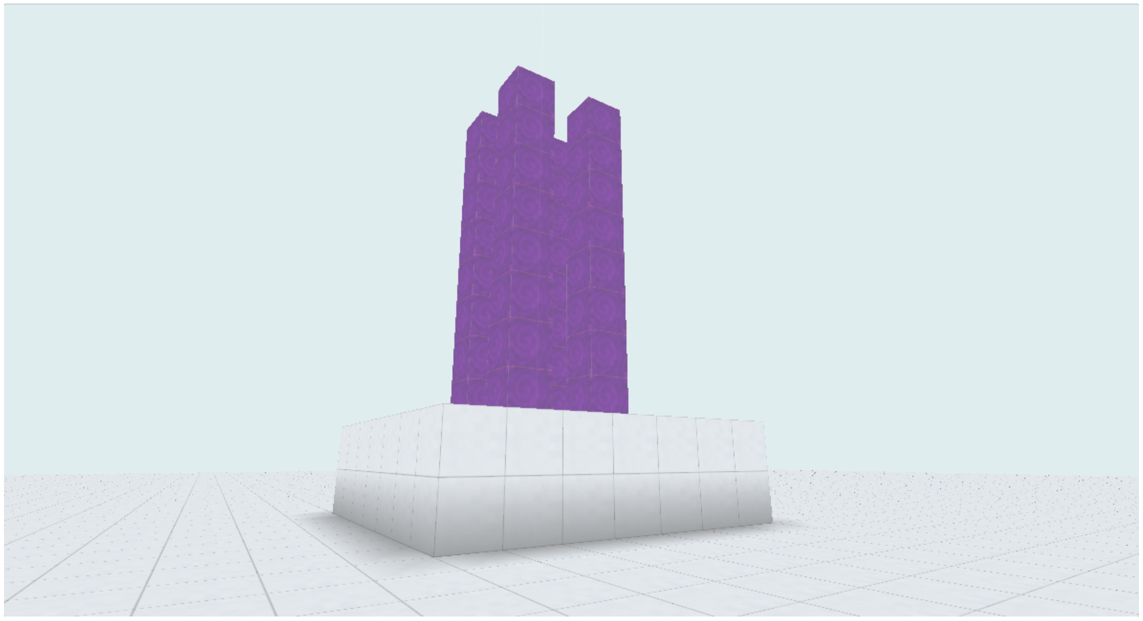


Figure 3: Protect the tower!

this. This is merely due to different system requirements.)

While even this simple game relies on many scripts, we'll be focusing on this script only for this section. Before we get much further let's go over a brief overview of the major sections of this script, and what exactly they're there for.

Definition!

COMMENT:
A part of code left by the code's author to explain what some code does in plain English. Starts with two forward slashes (//).

Definition!

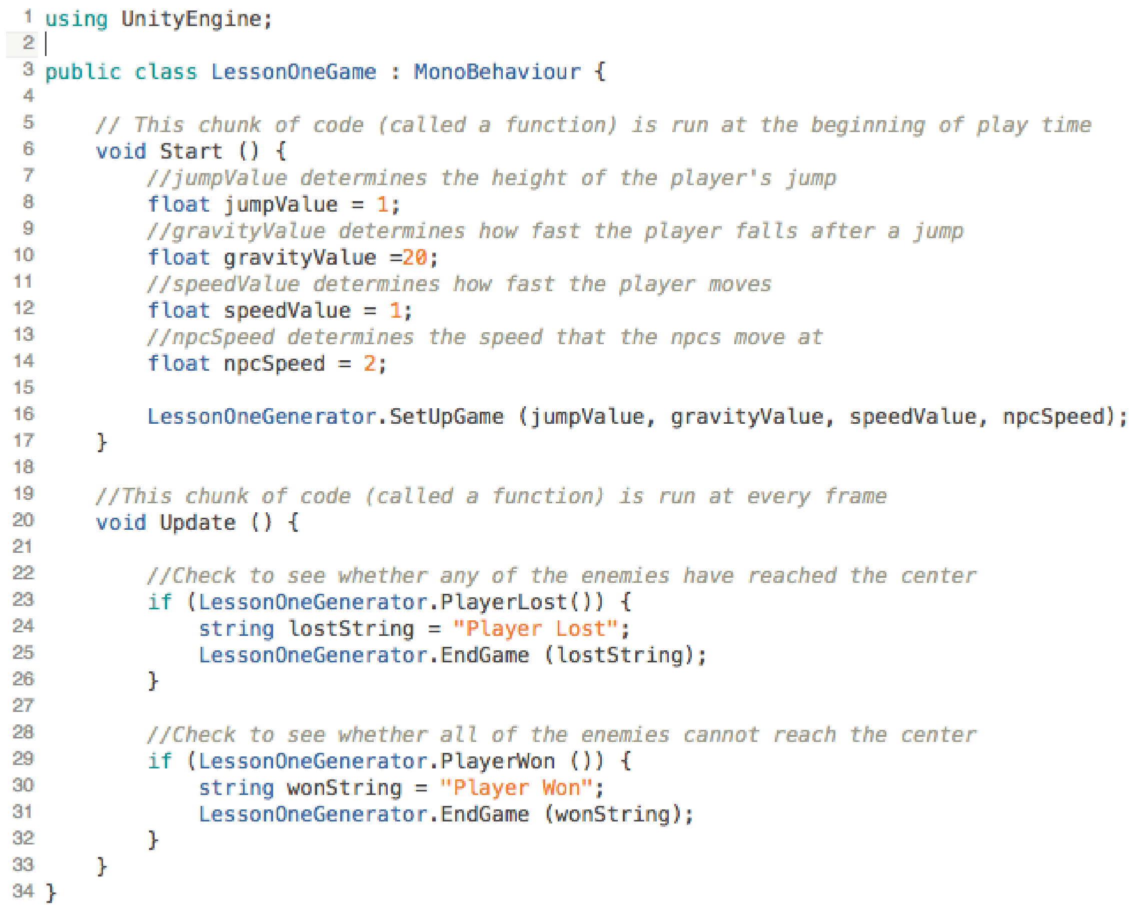
KEYWORD:
An English word with a specialized purpose in code beyond its normal definition.

2.1 Overview of LessonOneGame

Throughout the script you'll see sections of plain English text that start with two forward-slashes (//). This is what's referred to as a **comment** Comments are sections of code written by a programmer in order to explain what's going on in a specific section of code. When the game runs, it completely ignores these comments, meaning they have no impact on the game.

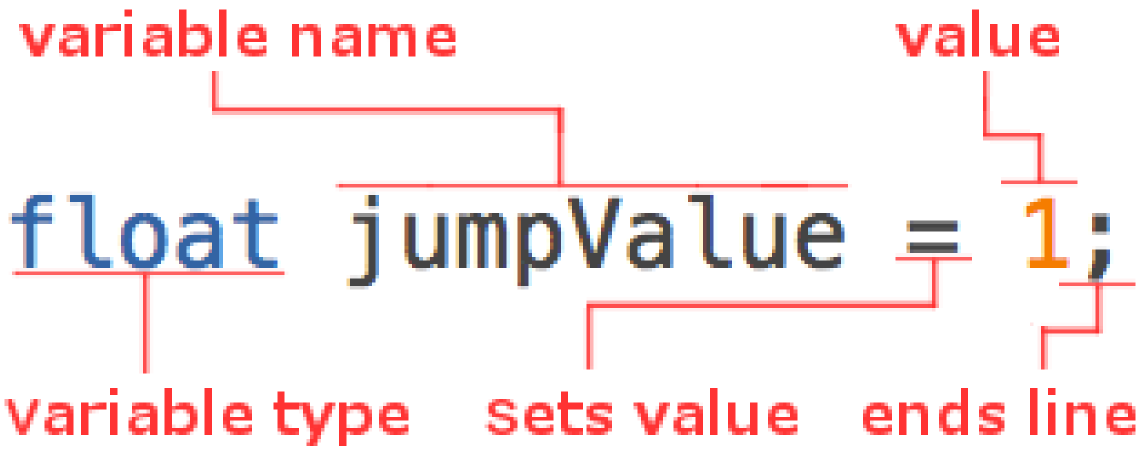
At the top of the file you have a line that begins with the **keyword** "using". Keywords are generally English words that have a specific meaning in a programming language, in that they do or represent something beyond their general English meaning. For example, the keyword "using" allows us to import and reference code from other codebases. In this case, we need to import the "UnityEngine" codebase as that's the game engine that this script is used in.

Just after line 3 (the line starting with using) we get to the main section of this script: the definition of the class "LessonOneGame". The line itself may seem fairly confusing at first, but let's break it down. The first word (another keyword, you can tell since it's in teal as well) "public" means that this class can be referenced from other scripts in the game (as opposed to if we used the word "private" here). The second word **class** specifies that the thing we're doing on this line is defining a class. We'll get more into this in a later lesson, but for now it's enough to know that a class defines a chunk of code that share a similar purpose. The third word is "LessonOneGame", which is the name of this script and the class. The last section ": MonoBehaviour" means that



```
using UnityEngine;

public class LessonOneGame : MonoBehaviour {

    // This chunk of code (called a function) is run at the beginning of play time
    void Start () {
        //jumpValue determines the height of the player's jump
        float jumpValue = 1;
        //gravityValue determines how fast the player falls after a jump
        float gravityValue =20;
        //speedValue determines how fast the player moves
        float speedValue = 1;
        //npcSpeed determines the speed that the npcs move at
        float npcSpeed = 2;

        LessonOneGenerator.SetUpGame (jumpValue, gravityValue, speedValue, npcSpeed);
    }

    //This chunk of code (called a function) is run at every frame
    void Update () {

        //Check to see whether any of the enemies have reached the center
        if (LessonOneGenerator.PlayerLost()) {
            string lostString = "Player Lost";
            LessonOneGenerator.EndGame (lostString);
        }

        //Check to see whether all of the enemies cannot reach the center
        if (LessonOneGenerator.PlayerWon ()) {
            string wonString = "Player Won";
            LessonOneGenerator.EndGame (wonString);
        }
    }
}
```

Figure 4: The LessonOneGame.cs file opened in MonoDevelop.



```
float jumpValue = 1;
```

Figure 5: A breakdown of a variable definition line.

Definition!

CLASS:
A collection of code including both variables and functions with a similar purpose.

this class named “LessonOneGame” takes some of its behaviors from the class “MonoBehavior”. This process is called inheritance. You’ll also notice that the line ends with a left curly bracket. If you click to the right of this curly bracket, you’ll see a right curly bracket highlight on the last line of this script. Curly brackets opening and closing like this specify a chunk of related code, in this case a class. This is similar to how parenthesis surround a distinct thought in writing (Like this!).

Definition!

FUNCTION:
A single section of code with a particular purpose, the equivalent of a paragraph in code.

Within the curly brackets of the class LessonOneGame we have two further chunks of code (held within curly brackets) referred to as **functions**. A function stores code that handles a particular activity. You can think of them as being like paragraphs for code with each line being a sentence. Like with classes, we’ll talk more about these later, but for now all that’s important to note is that the function “Start” runs at the beginning of the game just once, while “Update” runs every frame. Start handles setting up information about the game. Update handles checking whether or not the game should end and determines what text to display for the end screen. These two functions come from MonoBehavior, which specifies when the functions will be “called”. We use the term “call” to refer to running each line of a function one by one from top to bottom. It’s these two functions that you’ll be editing through the entirety of this section.

2.2 Making the Game Winnable

In order to fix the broken game, we’ll need to adjust variables. As we mentioned previously, variables are the building blocks of code. If we look at the function “Start” in “LessonOneGame” we can see four different *variable definitions*. A variable definition is a single line of code that creates a new variable, once created the variable can be used in later parts of the code. You can see a breakdown of one variable definition example in Figure 5. The word “float” indicates what kind of variable is being created and also specifies that this line is defining a new variable. We’ll get into variable types in a few sections, but for now just note that we have variables named "jumpValue", "gravityValue", "speedValue", and "npcSpeed".[1]

Since we know the major problem with the game seems to be how slow the player moves, it might make sense to try to raise the player's speed. All four of the definitions have an equals sign "=" and then a value after them. Unlike in math classes, this symbol doesn't mean two things have the same value, it actually stores the value on the right side into the variable on the left side. So the variable jumpValue currently stores the number 1, but we could change that to any other number we wanted to by changing the right side of the equals sign. We can use this fact to make the game winnable.

Based on the comments above each variable it looks like "speedValue" is our best bet. Try changing the number next to it from 1 to 5 and then playing the game again. WARNING: Don't forget to have a semi-colon (;) at the end of the line, so that it now reads:

```
float speedValue = 5;
```

Semi-colons are required to end individual lines of code as otherwise the game cannot determine where a line of code ends when reading a script. This will lead to an error and the game won't run till that error is resolved. Save the file after making this change and then open the Unity window.

Play the game by pressing the top play button on the Unity Window. As a quick note, while MonoDevelop does have a play button, it will not cause the game to play. Pressing that play button will lead to MonoDevelop trying to run the script as a standalone piece of code, which won't work well.

Now it's possible you *could* win the game. However, it's very unlikely that you can move fast enough to outrace the speedy snowmen. You might think that the issue is just that the player still isn't fast enough. That's possible! Why don't we try doubling the speed value from five to ten? The line should now read:

```
float speedValue = 10;
```

Try playing the game again before continuing.

Unfortunately, this increase of speed only makes the game much more difficult to control. Feel free to adjust speedValue more, trying an extreme value like 100 will make for an interesting but unplayable experience. We recommend setting the speedValue back to 5 for the remainder of this lesson, so that the line reads:

```
float speedValue = 5;
```

If adjusting the player's speed alone isn't enough to make the game winnable, then it makes sense that we should try adjusting another variable. We can determine which variable to adjust by thinking about what makes the game non-winnable: the snowmen can reach the tower before the player can build up a wall. To fix this, it makes sense that we should try to slow down the snowmen. Looking at the remaining variables it seems that "npcSpeed" is the variable we need to change. We'll start with an extreme change, let's make npcSpeed store 0.5.

However, if we just change the line to read:

```
float npcSpeed = 0.5;
```

[1]Note: the fact that all of these variable names have the first word uncapitalized, with the word afterward capitalized is referred to as "camel case". While its not required to name variables this way, it is a typical practice done by programmers to make it easier for readers to read the variable names.

Then we'll actually be unable to run the game! Make the change, save the script, and return to the Unity window. Look at the console (located on the bottom left of the Unity window; double click to open it) we'll see an error in red text that reads:

"Assets/Codebase/Lesson1/LessonOneGame.cs(20,23): error CS0664: Literal of type double cannot be implicitly converted to type 'float'. Add suffix 'f' to create a literal of this type". What does this mean?

Definition!

VARIABLE TYPE: An attribute of a variable, specified when it is defined, which determines what values can be stored into it.

Unfortunately we've just gotten our first type error (errors are problems with code that mean the game cannot run). All variables have a **type**, which dictates the kinds of values that can be stored in them. This can get confusing as there are actually multiple variable types for storing numbers.[2] For example all the four variables in this chunk of code are of the type "float", but including a decimal means that Unity thinks we're using another type called a "double". The major difference between a float and a double is that doubles are much more precise. For example, a float can hold values with up to 7 digits (0.3333333), while a double can hold values with up to 16 digits (0.3333333333333333). However, we generally don't need this level of precision.

In order to not confuse Unity into thinking we're using a double instead of a float we need to add a lower-case "f" to the end of the value we're storing in npcSpeed. This tells Unity that we want to create a float value instead of a double value, and then there's no problem storing this value into a float variable. The line should now read:

```
float npcSpeed = 0.5f;
```

Save the file and then return to the Unity window to play the game again.

This time it shouldn't be too difficult at this point to build up a wall to keep the snowmen away. Congratulations, you've made this game winnable! You might find that this speed of the snowmen is too slow for you, or maybe still a bit too fast. Feel free to adjust the value of the npcSpeed variable to adjust this. Just by changing this one variable you can directly impact the difficulty of the game.

Once you have a speed you're happy with feel free to move on to the next section of this lesson where we'll go beyond just making a game winnable and get into changing how the game feels to play.

2.3 Changing the "Game Feel"

The way it feels to control a virtual character in a video game is referred to as "game feel". In this section we'll cover making some additional changes that'll impact the way the player plays the game.

There are two float-type variables left in "Start" that we haven't yet considered. These being "jumpValue" and "gravityValue". As you might expect these impact the player's jumping and falling. One of the biggest complaints from video game players is that a a game's jump feels "floaty". In order to avoid that in the future let's see if we can replicate it here.

In order to turn a specific feeling ("floaty") into a gameplay mechanic it's useful to think about examples you might have seen from life. For example, you might have seen a gently floating feather, or bit of dust drifting through the air. We can then identify things that these "floaty" examples

[2]It may seem odd to have multiple types of variables to represent numbers, but its actually more efficient. Storing more information requires more memory, so it makes sense to use less "precise" variable types when we can. For example, if we know a variable will only hold a number between 1 and 100, there's no need to use a variable type that can store a wider range than that.

have in common and attempt to make the virtual character act more like them. The easiest of these shared attributes to implement is that both tend to fall very slowly. Let's see if we can make that happen in the game.

Of the two remaining variables, "gravityValue" makes the most sense in order to make the player fall less fast. Try altering the "gravityValue" to make the fall slower. Does it make sense to lower or raise the value? (Hint: You might have to make a large change before the effect is noticeable). Test out whether raising or lowering the value makes sense by making changes then testing how they impact the game.

Play the game and alter the gravityValue until it feels "floaty".

Depending on your ideas of floaty-ness, you might find that just changing the gravityValue doesn't seem to be enough. Especially given that the player character still moves at a relatively fast speed in the air when moving with wasd. To further floaty-ify the player, it might be necessary then to change speedValue as well. By adjusting "speedValue" as well you should be able to get a more satisfying floaty feeling. However, this will make the game impossible to win again, which is another reason to dislike floaty controls.

As mentioned above, most people who play video games don't like a floaty feeling with their controls. However, as with most rules, there is an exception. What if you wanted to set a video game on the moon, or some other place with low gravity? Making a player believe that a game takes place in a particular location takes more than just changing the way the game *looks*. The player character also needs to move as the player would expect for that location. You have to change how the game *feels*!

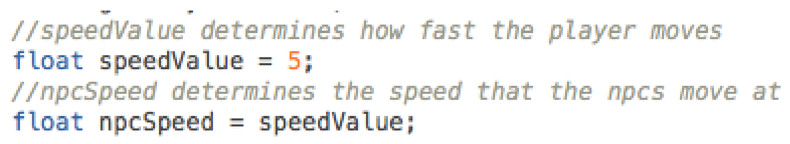
To demonstrate the importance of player controls to suggest a particular location, let's try transforming this Snowmen Blocking game into a Snowmen Blocking Game on the Moon. In order to make the player feel like they're on the moon we'll need to change both the gravity and the jump values. Make sure to reset the speedValue back to 5 (or even a bit higher!) as being on the moon wouldn't make one slower in terms of walking around speed. We won't give you exact values for gravityValue and jumpValue, play around with it! It is worth noting that jumpValue should be higher while gravityValue should be lower than its original value of 20.

Play the game to test your changes, and continue to make changes until you're happy with the feel.

Before too long you should find that you can bound around across the snowy landscape just like an astronaut on the moon. You'll also likely find that even though it doesn't impact the "gameplay" to any degree the game *feels* much more fun to play now. This altering of variables to get a specific feeling or behavior in a game is referred to as "tuning" and it's one of the most common tasks in game design.

2.4 Tuning and Relationships

In the process of tuning a game its sometimes useful to only change one variable, but keep a certain relationship between it and another variable. Let's say for example we wanted to change the speedValue variable from 5, but wanted to keep the current relationship between npcSpeed and speedValue (Where npcSpeed is 1/10 of speedValue at 0.5). Now you could make changes to speedValue and calculate what npcSpeed should be yourself, but that would get time consuming. Especially if you're making lots of little changes! (Say you wanted speedValue to be 4 then npcSpeed would be 0.4, or speedValue could be 4.5 then npcSpeed should be 0.45, and so on...). That doesn't sound like a lot of fun.



```
//speedValue determines how fast the player moves
float speedValue = 5;
//npcSpeed determines the speed that the npcs move at
float npcSpeed = speedValue;
```

Figure 6: Setting npcSpeed to speedValue.

Instead we can get the code to handle this for us! Remember that variables store specific values. However, they can also be used as if they *were* those values. For example, if we simply set npcSpeed to speedValue then npcSpeed will have the same value as speedValue, as seen in Figure 6. Trying to play the game now will be very difficult, as the snowmen will move at the same speed as the player.

Remember that our goal here is to have the same relationship between speedValue and npcSpeed; we want npcSpeed to be 1/10 of speedValue. We can do this very easily! Just change npcSpeed to be equal to "speedValue/10" so the line now should read:

```
float npcSpeed = speedValue/10;
```

You should now find that the game is now totally winnable! This is due to the fact that "/" in code is translated as a command to do division. So when Unity reads in the script it reads speedValue divided by 10 (or in this case 5 divided by 10 as 5 is stored in speedValue) which equals 0.5!

Try changing the value stored in speedValue, and you'll see that the relationship holds automatically. So setting speedValue to 4 will lead npcSpeed to equal 0.4 automatically and so forth. This is a powerful tool, as it not only allows tuning of two different variables to occur at once, but can even be used to change the relationship between two variables after the fact. So we can change "/10" to "/9" or whatever we like! Try changing this as well and see what kinds of different experiences you can make.

Division (using the "/" symbol) is not the only kind of mathematic operation that code can do for you automatically. We can use all of the "simple" mathematical operations with a single symbol. "+" tells the code to do addition, "-" tells the code to do subtraction, and "*" tells the code to do multiplication. See them all laid out with examples in the table below. We could have (for example) set npcSpeed with multiplication by changing the line to: "float npcSpeed = speedValue*0.1f;". Try using these different operations with the variables you have defined so far.

Name	Symbol	Example
Addition	+	float example = 4 + 5; (example equals 9)
Subtraction	-	float example = 4 - 5; (example equals -1)
Multiplication	*	float example = 4 * 5 ; (example equals 20)
Division	/	float example = 4 / 5 ; (example equals 0.8f)

2.5 Variable Types

So far we've dealt exclusively with variables that already existed, altering or "tuning" them towards different effects. In this section we'll go over creating and changing variable values with code instead of by hand.

Before we get too much further, we should take a moment to go over some of the most common variable types and what they store.

Type	English Name	Values Stored
uint	Unsigned Integer	0 to 4294967295
int	Signed Integer	-2,147,483,648 to 2,147,483,647
float	Float	-3.402823e38 to 3.402823e38
double	Double	-1.79769313486232e308 to 1.79769313486232e308
bool	Boolean	True or False
char	Character	Single text characters
string	String	Sequences of multiple text characters

The above table summarizes some of the most common primitive variable types. We classify these variable types as primitives via using a lower-case letter to define them in code. We've already mentioned "complex" variable types, though not by that name. Classes like LessonOneGame are complex variable types, and we'll explain those in detail in a later lesson.

Of these primitive variable types, you've already seen floats and interacted with doubles to some degree. There are two other types of variables that store number information here, signed and unsigned integers. You can think of an integer as being equivalent to a whole number in mathematics. So numbers that don't have decimal points that can be stored into integers. They're useful when it comes to counting things, where you can't just have "part" of something. Whether or not its signed merely indicates whether there can be positive or negative number values for them.

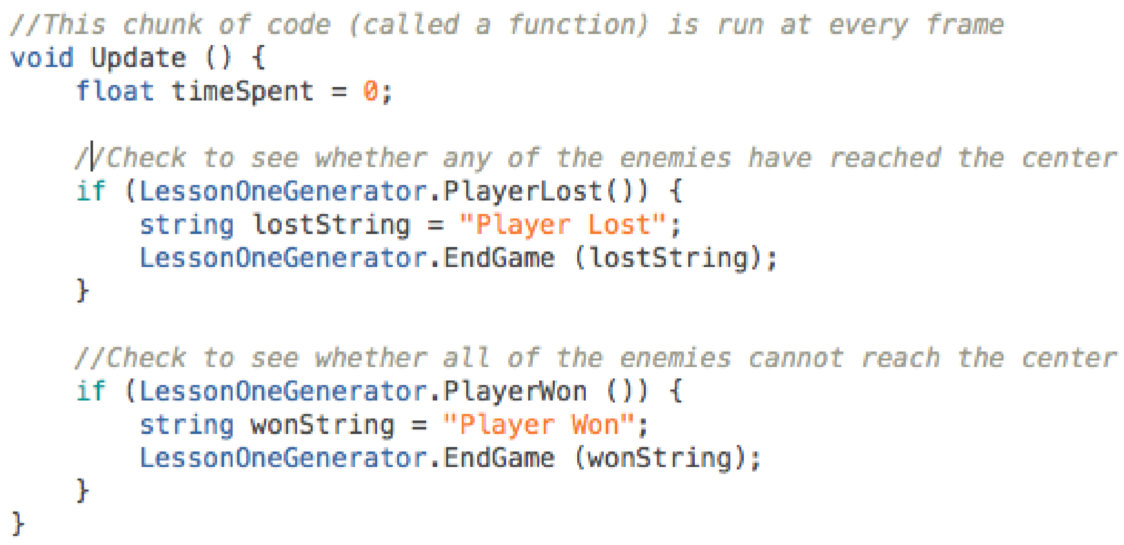
In the table we're also introducing three new types of variables entirely. First of all we have boolean values, which store true and false information. We'll come back to those in the next lesson. The last two types have to do with text information. We have characters, which can store a single text value ('a', 'b', 'c') and strings which can store sequences of text characters ("abc"). We have two examples of strings already in the script we've been working in, which can give you a sense of what these variable definitions look like. You'll note that just like how we had to add an f to the end of a float variable declaration in order to specify that it was a float, we specify that text is a string by surrounding it with quotation marks "like so". We have to do this as we technically use text characters throughout the code, including in variable names, so we need a way to differentiate when the text is a string.

2.6 Creating a Variable

Now that we've covered the most common types of primitive variables let's set about creating a new variable. To start with, it's possible individuals may want to know how long it took them to win or lose the game. To do that, we'll need to track how long the game is played for. Let's start by defining a new float variable to track the time. Instead of Start, try defining the new variable in Update. Let's use the name "timeSpent" and set it to 0 to begin with. You'll want to put the variable definition of the beginning of the Update function, something like Figure 7. As a reminder, variable definitions look like Figure 5. The first word ("float") determines the type of variable, and tells Unity that the line is creating a new variable. The second word ("timeSpent") is the name of the variable, and the way it can referred to later. The equals sign ("=") tells Unity to set the variable's value to it's right side ("0"). Then a semi-colon ends the line.

First ensure that the game still runs (you don't have to play it all the way through).

This will ensure that Unity can still read the script and that there have been no errors introduced. Its generally a good idea to ensure that the game still runs after every major change, as a



```
//This chunk of code (called a function) is run at every frame
void Update () {
    float timeSpent = 0;

    //Check to see whether any of the enemies have reached the center
    if (LessonOneGenerator.PlayerLost()) {
        string lostString = "Player Lost";
        LessonOneGenerator.EndGame (lostString);
    }

    //Check to see whether all of the enemies cannot reach the center
    if (LessonOneGenerator.PlayerWon ()) {
        string wonString = "Player Won";
        LessonOneGenerator.EndGame (wonString);
    }
}
```

Figure 7: Update after adding timeSpent.

means of ensuring that the change didn't break anything. It's worth noting at this stage that you may see a warning pop up in the Unity console (bottom left corner of the Unity window) with the text:

"Assets/Codebase/Lesson1/LessonOneGame.cs(21,23): warning CS0219: The variable 'timeSpent' is assigned but its value is never used".

This is not an error, and won't stop you from playing the game. Warnings appear in the console for issues in the code that still allow it to run, such as defining a variable that is never used.

Now that you've confirmed that the game still runs we have two problems.

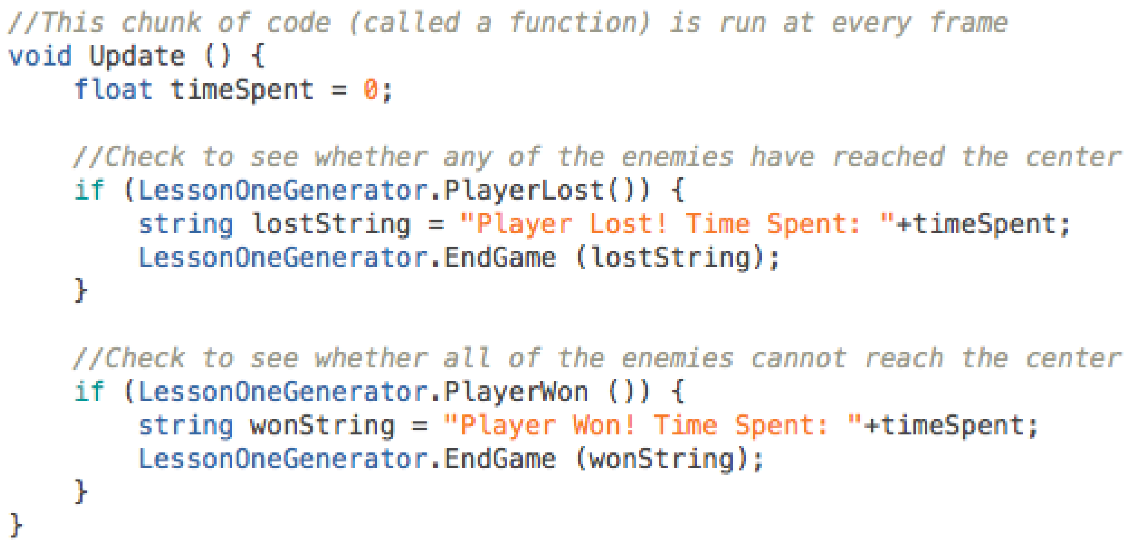
1. While we now have a variable to track the time spent playing the game, we don't have any means of displaying this information to the player.

2. The value of timeSpent will never change, meaning it won't actually hold the amount of time the player plays the game.

2.7 Displaying a Variable

We'll tackle the first problem first. We mentioned in a previous section that the "+" symbol tells the code to use addition, but it can actually be used for more than that. In certain situations the "+" symbol can be used to combine values of two different types. In this way we can turn a float (like our timeSpent variable) into a string (like our wonString variable). This can be accomplished in a number of ways, and we'll demonstrate one of them here. For example if we change the wonString line to read:

```
string wonString = "Player Won! Time Spent: "+timeSpent;
```



```
//This chunk of code (called a function) is run at every frame
void Update () {
    float timeSpent = 0;

    //Check to see whether any of the enemies have reached the center
    if (LessonOneGenerator.PlayerLost()) {
        string lostString = "Player Lost! Time Spent: "+timeSpent;
        LessonOneGenerator.EndGame (lostString);
    }

    //Check to see whether all of the enemies cannot reach the center
    if (LessonOneGenerator.PlayerWon ()) {
        string wonString = "Player Won! Time Spent: "+timeSpent;
        LessonOneGenerator.EndGame (wonString);
    }
}
```

Figure 8: Update after changing the end of game messages.

Then timeSpent will be converted to a string and combined with “Player Won. Time Spent: ” automatically. Now this is a really powerful and handy thing that programming languages can do, but it only works in specific cases and between certain types.

For example, you can convert numbers to strings but not strings to numbers in this way. Go ahead and change both the win and lose strings to include timeSpent. In the end your code should look like Figure 8. The text within the double quotation marks can be whatever you like, try writing your own win and lose messages with timeSpent included. Just try not to be too mean to the player, or they won’t play again.

Play the game until you either win or lose and you’ll see your new message appear! So that solves one problem, but we still have a fairly large one in that the value of timeSpent doesn’t change from the initial value of 0 we gave it.

The solution to this problem is a more complex one, but we’ll break it down into a number of smaller steps.

2.8 Updating Variables with Code

In order to have the timeSpent variable change from its initial value we’ll need to talk about updating variables. So far, we’ve made use of the values in variables after defining them (in the Game Feel section), but haven’t changed those values after defining them. As we mentioned at the very beginning of this unit, variables can vary (change). Let’s start with a very simple example, look back at ”Start” for a minute. On the line below npcSpeed’s definition insert a *new* line that reads:

```
npcSpeed = 5;
```

This is the first time that we’ve updated a variable’s value after defining it. We don’t include the “float” variable type here, as that would create a *new* variable named npcSpeed, and we want to use the same variable as defined above, but just change its value. We'll want to remove this line (and the following lines from this section) later as we're only using them as an example. We don't actually want the snowmen to be that fast!

If you run the game, you'll see that the snowmen all move at a much higher speed! This is how we update variables. We again use the "=" symbol to store a value on the right side into a variable on the left. The only thing missing here from the variable definition is the "float" type declaration. Since we've declared its type already the variable will stay at that type, meaning we don't need to include it when we update its value.

Updating a variable's value is important if a variable should store different values throughout the game. For example, if we're trying to track the time the player plays for! However, we've only scratched the surface when it comes to updating values. We can actually use the value stored in a variable and update that value *in the same line*. Remember the way the "=" symbol works. It takes the value on the right side of it and stores it in the variable on the left. That ordering is very important. As an example, change the line that updates the npcSpeed variable and change it to:

```
npcSpeed = npcSpeed+10;
```

When Unity reads through the script, what it does is take the value on the right hand side of the "=" symbol and runs any needed calculations first. So npcSpeed (which stores 0.5) is added to 10 (so it would be 10.5). Then that new value is stored back into the variable on the left hand side of the "=" symbol.

Go ahead and run the game, and you'll notice that the snowmen now move *much* faster. That's because their speed has a much higher value!

You can actually use addition (+), subtraction (-), division (/) and multiplication (*) in this same way. To prove it let's insert a new line just below our addition line that subtracts 10 from the npcSpeed then stores the new value back into npcSpeed. Before you play the game again, think about what this change should do. How should the value of npcSpeed that the game begins with relate to the last time you played? And the time before you added the addition line?

Once you've done that try using multiplication (*) and division (/) in the same way. Get rid of the two addition and subtraction lines and replace them with first just one multiplication line (so * 10 instead of + 10). Play through the game. Then add a division line by the same amount (/ 10). What would you expect to see with this change?

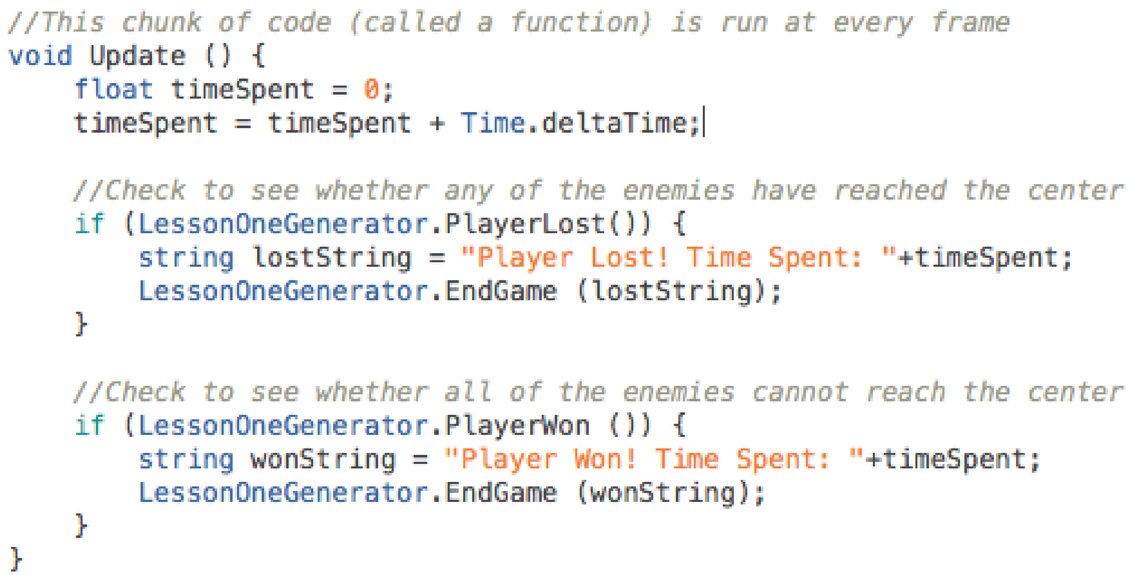
2.9 Using Mathematical Operations

Now that you're a coding math master we can get back to the original problem at hand. We want to make sure that "timeSpent" reflects the amount of time that the player has spent playing the game. For that we'll want to use a variable referred to as "Time.deltaTime". The variable "Time.deltaTime" stores the amount of time since the last time "Update" was called and is accessible due to the "using UnityEngine" line at the top of the script.

Let's go ahead and add "Time.deltaTime" to timeSpent. Add a new line below the definition of timeSpent where you set timeSpent equal to timeSpent plus "Time.deltaTime". Your Update function should now look like Figure 9.

Run the code now and either win or lose and you should see a display with a non-zero value.

However, this is only a fraction of a second, and definitely isn't representative of the player's entire play time. You may have already realized why that would be the case. Since "timeSpent" is defined as zero at the top of Update every frame, and "Time.deltaTime" added to it every frame, it



```
//This chunk of code (called a function) is run at every frame
void Update () {
    float timeSpent = 0;
    timeSpent = timeSpent + Time.deltaTime;

    //Check to see whether any of the enemies have reached the center
    if (LessonOneGenerator.PlayerLost()) {
        string lostString = "Player Lost! Time Spent: "+timeSpent;
        LessonOneGenerator.EndGame (lostString);
    }

    //Check to see whether all of the enemies cannot reach the center
    if (LessonOneGenerator.PlayerWon ()) {
        string wonString = "Player Won! Time Spent: "+timeSpent;
        LessonOneGenerator.EndGame (wonString);
    }
}
```

Figure 9: Update with Time.deltaTime added.

will never be larger than one value of Time.deltaTime. Essentially we set timeSpent back to zero every Update before adding Time.deltaTime, therefore it only has the value of one Time.deltaTime stored in it. So we need some way to define timeSpent elsewhere so we don't set it back to zero every frame.

Definition!

VARIABLE SCOPE: An attribute of a variable, specified by *where* it is defined, which determines where it can be used and updated.

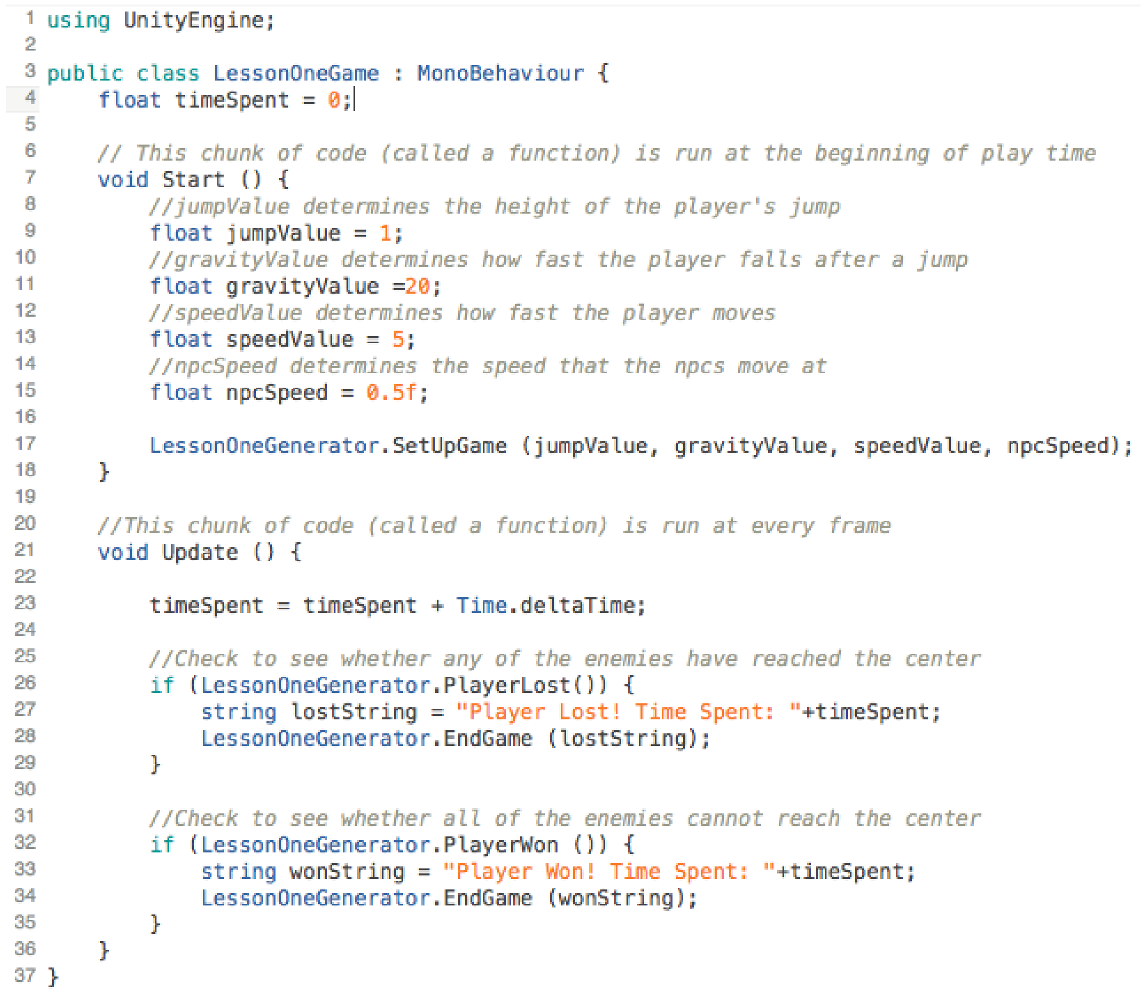
2.10 Variable Scope

At this stage we'll actually introduce something entirely new: **variable scope**. A variable's scope essentially determines where it can and cannot be used. At present you've only been defining variables within a function. We refer to this as a "local" definition. Variables with local scope cannot be used outside of the functions they are defined in. This means that when we define a definition in a specific function (as we have done so far) we cannot use it outside that function. For example, try adding a new line to Update instead of Start that reads:

```
npcSpeed=npcSpeed+5;
```

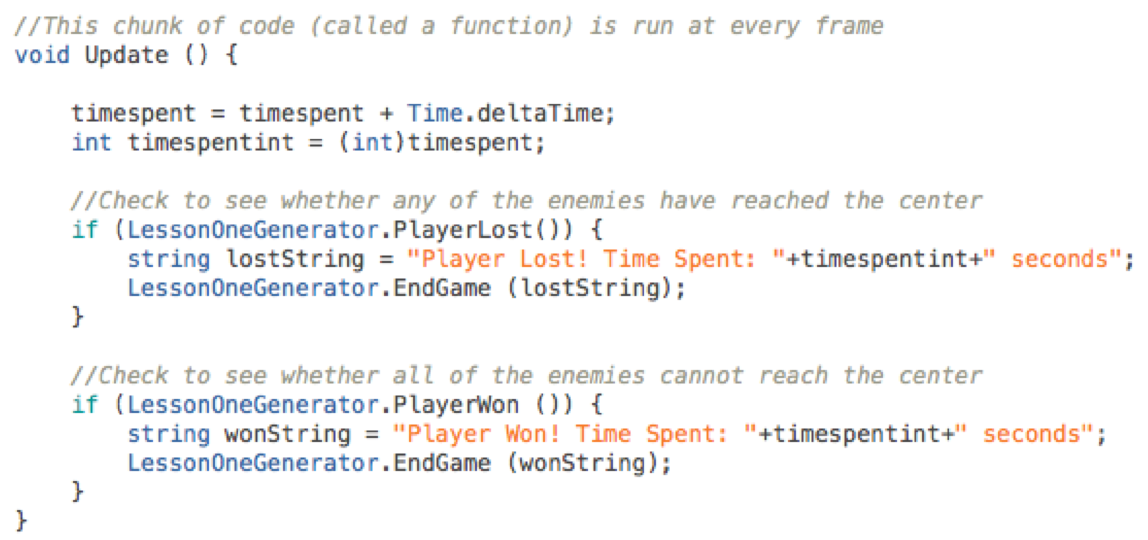
If you do that, the game will not run, as it will think that you're using a variable without defining it. You'll have to delete that line to get the game to play again. This might seem weird, because the variable "npcSpeed" clearly exists in Start, but this inability to access variables from other functions actually saves memory as it decreases how many variables the game has to keep track of at any one time.

The alternative to variables with local scope is variables with *global scope*. To set up a variable to have global scope we simply define it outside a function. Remove the variable definition of "timeSpent" from Update and move the entire line to be above Start. Your script should now look something like Figure 10.



```
 1 using UnityEngine;
 2
 3 public class LessonOneGame : MonoBehaviour {
 4     float timeSpent = 0;
 5
 6     // This chunk of code (called a function) is run at the beginning of play time
 7     void Start () {
 8         //jumpValue determines the height of the player's jump
 9         float jumpValue = 1;
10         //gravityValue determines how fast the player falls after a jump
11         float gravityValue =20;
12         //speedValue determines how fast the player moves
13         float speedValue = 5;
14         //npcSpeed determines the speed that the npcs move at
15         float npcSpeed = 0.5f;
16
17         LessonOneGenerator.SetUpGame (jumpValue, gravityValue, speedValue, npcSpeed);
18     }
19
20     //This chunk of code (called a function) is run at every frame
21     void Update () {
22
23         timeSpent = timeSpent + Time.deltaTime;
24
25         //Check to see whether any of the enemies have reached the center
26         if (LessonOneGenerator.PlayerLost()) {
27             string lostString = "Player Lost! Time Spent: "+timeSpent;
28             LessonOneGenerator.EndGame (lostString);
29         }
30
31         //Check to see whether all of the enemies cannot reach the center
32         if (LessonOneGenerator.PlayerWon ()) {
33             string wonString = "Player Won! Time Spent: "+timeSpent;
34             LessonOneGenerator.EndGame (wonString);
35         }
36     }
37 }
```

Figure 10: Moving timeSpent to have global scope (line 4).



```
//This chunk of code (called a function) is run at every frame
void Update () {

    timespent = timespent + Time.deltaTime;
    int timespentint = (int)timespent;

    //Check to see whether any of the enemies have reached the center
    if (LessonOneGenerator.PlayerLost()) {
        string lostString = "Player Lost! Time Spent: "+timespentint+" seconds";
        LessonOneGenerator.EndGame (lostString);
    }

    //Check to see whether all of the enemies cannot reach the center
    if (LessonOneGenerator.PlayerWon ()) {
        string wonString = "Player Won! Time Spent: "+timespentint+" seconds";
        LessonOneGenerator.EndGame (wonString);
    }
}
```

Figure 11: Using an explicit cast to get rid of decimals for timeSpent.

Play the game again, and you'll find that the value at the end of the game much better reflects your actual play time!

2.11 Cleaning up the Display

Definition!

CASTING:
A means of transferring a value from one variable type to another.

There's quite a lot of decimals in this output though, don't you think? This is due to the fact that we were using floats, which track decimal values. We could instead use an integer value, which can only store whole numbers. However, we can't just make timeSpent an integer as Time.deltaTime is a float value! Instead we can convert timeSpent from a float to an integer right before we print it out via a process called **casting**.

Definition!

IMPLICIT CAST:
A type of cast in which the type to cast to isn't specified, but assumed by the system (Unity).

We actually already used casting when we added a string value to timeSpent to make it into a string. That type of casting is referred to as an **implicit cast**. With implicit casts you as the programmer do not need to specify the variable type you want the variable converted to, as it happens automatically. However, implicit casts only work in specific cases (like casting from a number to a string, but not a string to a number).

Definition!

EXPLICIT CAST:
A type of cast in which the type to cast to is specified in parenthesis by the programmer.

To cast a float to an int we'll use something called an **explicit cast**. Instead of an implicit cast, an explicit cast requires you to spell out the type you want to convert to in parenthesis. So to cast to an integer you'd put (int) in front of the variable you wanted to cast from. We'll also need a new int variable to store the int "version" of timeSpent. So let's add the below line to Update:

```
int timeSpentInt = (int) timeSpent;
```

And then use timeSpentInt instead of timeSpent in the conversion to the string for end of game text. Your Update functions should now look like Figure 11.

Play the game, you should now see that your output is much easier to read!

However, while its much easier to read, the output isn't exactly clear. While we know that the number expressed is the seconds of game time, that's not something known to someone else playing the game. We can make this more clear by adding the word “seconds” to the output. This is actually very simple! We’ve already used “+” earlier to combine a string value and both timeSpentInt (an int) and timeSpent (a float). We can actually use the “+” symbol to combine two strings as well! So we can add the word “seconds” to the end of the output simply by adding + “ seconds” to the end of the line. The space before sounds is important as otherwise there won’t be one between the number from timeSpentInt and the word seconds.

2.12 Adding Complexity

In the last section we created a single variable to add new information to the game. In this section we’ll add new information that requires two different variables. Players of the game may want to know the rate at which they placed blocks through the game. If you are unfamiliar, the rate of block placement would be how many blocks were placed for a given amount of time. In this case we’ll say blocks placed per seconds. We already have one of the variables required: the play time in seconds thanks to timeSpent. The other half of the information we need is the number of blocks placed throughout the game.

In order to track the number of blocks placed we’ll use a very similar number of steps as with timeSpent, except that you’ll need to use an int value instead of a float and we’ll use Selector.blocksLastTick instead of Time.deltaTime. Selector.blocksLastTick holds the number of blocks placed since the last time “Update” was called, just like how Time.deltaTime held the time since “Update” was called. Try to walk through the entire process without glancing down at the image of the code further down, replacing timeSpentInt with your new variable. The name of your variables doesn’t actually matter as it doesn’t impact the game at all, its purely for yourself and other programmers looking at your code. If you’d like to remain consistent with us, we used the name “blocksPlaced”.

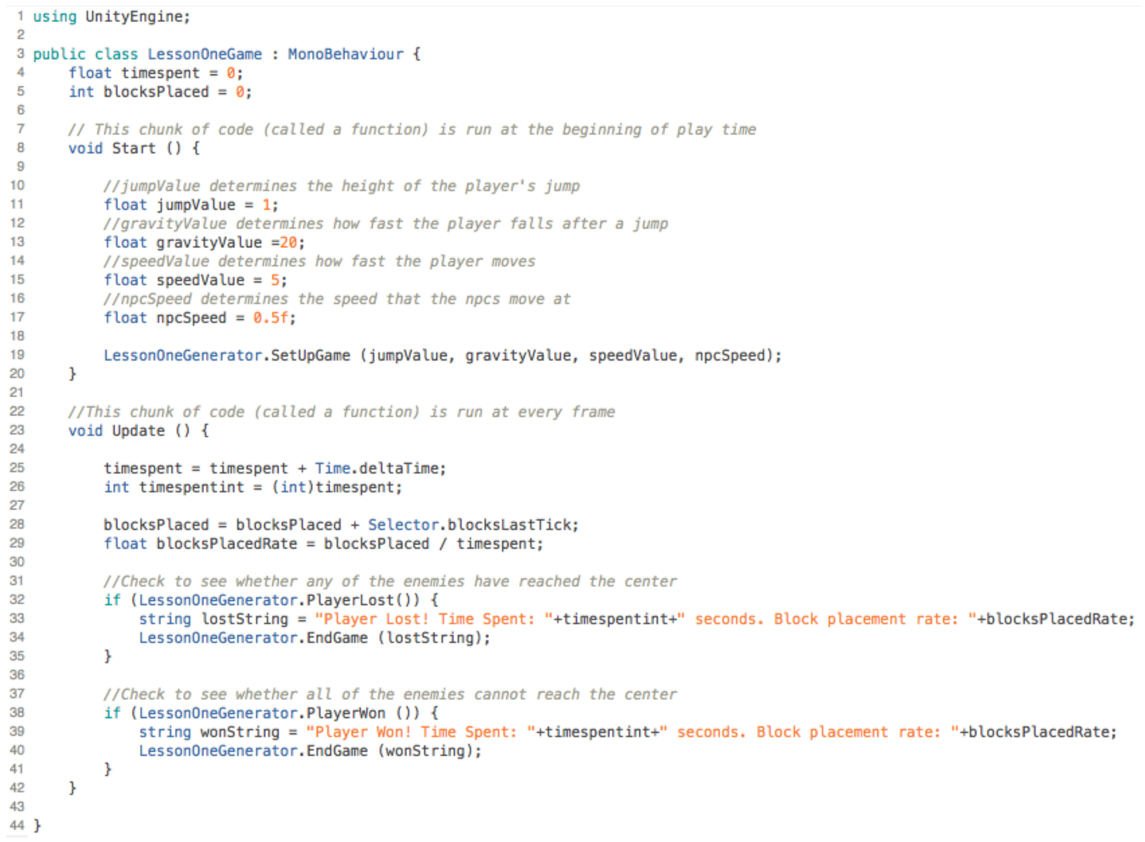
Once you have the variable working and displayed to the player at the end of the game we can go ahead and calculate the rate of blocks placed per second. All it takes is dividing blocksPlaced by timeSpent (or formally Selector.blocksPlaced/Time.deltaTime) to get the average amount of blocks placed each second of the game. You previously used division with code with a variable and a number value. Division with two variables is exactly the same. Can you figure out how to calculate it without looking at the Figure? It might help to break this process into each of the steps that you’ll need to go through to get this to work.

1. Define a new variable “blocksPlaced” that’s an int with global scope.
2. Add a line in update to add Selector.blocksLastTick to blocksPlaced every frame.
3. Calculate the rate by defining a new variable with the value of blocksPlaced /timeSpent. (We used blocksPlacedRate)
4. Add blocksPlacedRate (or whatever you named the variable) to the output.

3 End of Lesson One

That’s all for Lesson One! In this lesson you learned how to:

1. Alter scripts towards a specific purpose.



```
using UnityEngine;

public class LessonOneGame : MonoBehaviour {
    float timespent = 0;
    int blocksPlaced = 0;

    // This chunk of code (called a function) is run at the beginning of play time
    void Start () {

        //jumpValue determines the height of the player's jump
        float jumpValue = 1;
        //gravityValue determines how fast the player falls after a jump
        float gravityValue =20;
        //speedValue determines how fast the player moves
        float speedValue = 5;
        //npcSpeed determines the speed that the npcs move at
        float npcSpeed = 0.5f;

        LessonOneGenerator.SetUpGame (jumpValue, gravityValue, speedValue, npcSpeed);
    }

    //This chunk of code (called a function) is run at every frame
    void Update () {

        timespent = timespent + Time.deltaTime;
        int timespentint = (int)timespent;

        blocksPlaced = blocksPlaced + Selector.blocksLastTick;
        float blocksPlacedRate = blocksPlaced / timespent;

        //Check to see whether any of the enemies have reached the center
        if (LessonOneGenerator.PlayerLost()) {
            string lostString = "Player Lost! Time Spent: "+timespentint+" seconds. Block placement rate: "+blocksPlacedRate;
            LessonOneGenerator.EndGame (lostString);
        }

        //Check to see whether all of the enemies cannot reach the center
        if (LessonOneGenerator.PlayerWon ()) {
            string wonString = "Player Won! Time Spent: "+timespentint+" seconds. Block placement rate: "+blocksPlacedRate;
            LessonOneGenerator.EndGame (wonString);
        }
    }

}
```

Figure 12: The entire script including block rate

2. Use mathematic operations in code.
3. Differentiate different variable types
4. Convert between different variable types
5. Create new variables and integrate them into code

In the next lessons we'll cover using if statements in order to allow the game to make choices about when to run certain pieces of code, use loops to build up scenery, and using functions to create new possibilities with code.

4 Lesson One Glossary

Vocabulary Word	Definition	Examples
variable	The part of code that stores information for use.	jumpValue, gravityValue, speedValue, etc
comment	A part of code left by the code's author to explain what some code does in plain English. Starts with two forward slashes (//).	//jumpValue determines the height of the player's jump
keyword	An English word with a specialized purpose in code beyond its normal definition.	using, public, private
class	A collection of code including both variables and functions with a similar purpose.	LessonOneGame, Time, Selector, MonoBehavior, etc
function	A single section of code with a particular purpose, the equivalent of a paragraph in code.	Start and Update
variable type	An attribute of a variable, specified when it is defined, which determines what values can be stored into it.	float, double, int, bool, string, etc
variable scope	An attribute of a variable, specified by *where* it is defined, which determines where it can be used and updated.	[Not Appropriate]
casting	A means of transferring a value from one variable type to another.	[See Below]
implicit cast	A type of cast in which the type to cast to isn't specified, but assumed by the system (Unity). Only works between certain types.	string wonString = "Player Won! Time Spent: "+timespentint;
explicit cast	A type of cast in which the type to cast to is specified in parenthesis by the programmer.	int timespentint = (int)timespent;