UnityScript Basics for Noobs

By iGraphic Visuals LLC 2010

Especially for Noobs

1: Introduction

A word of warning before starting - this is not the "Be-all, end-all" manual on UnityScript. This manual is intended for the person that wants to use Unity, AND has no knowledge of any sort of programming. Use the info here to get your feet wet, to shed some of the fears of adding some scripts to your GameObjects.

Scripting, or programming in general, is learned in very basic steps, especially for total noobs. Chances are you need to overcome some unjustified fears about how amazingly difficult scripting appears to be. The basics of UnityScript really aren't difficult. We do things everyday that are just like the procedures followed in scripting.

There are books, manuals, and tutorials galore on programming that can show you how to really get into the details. There are also many video tutorials to learn Unity, and they just about all teach scripting basics. After going through the steps in this manual, you should feel a lot more comfortable learning from those tutorials. At least you'll have a basic understanding of what other authors are saying.

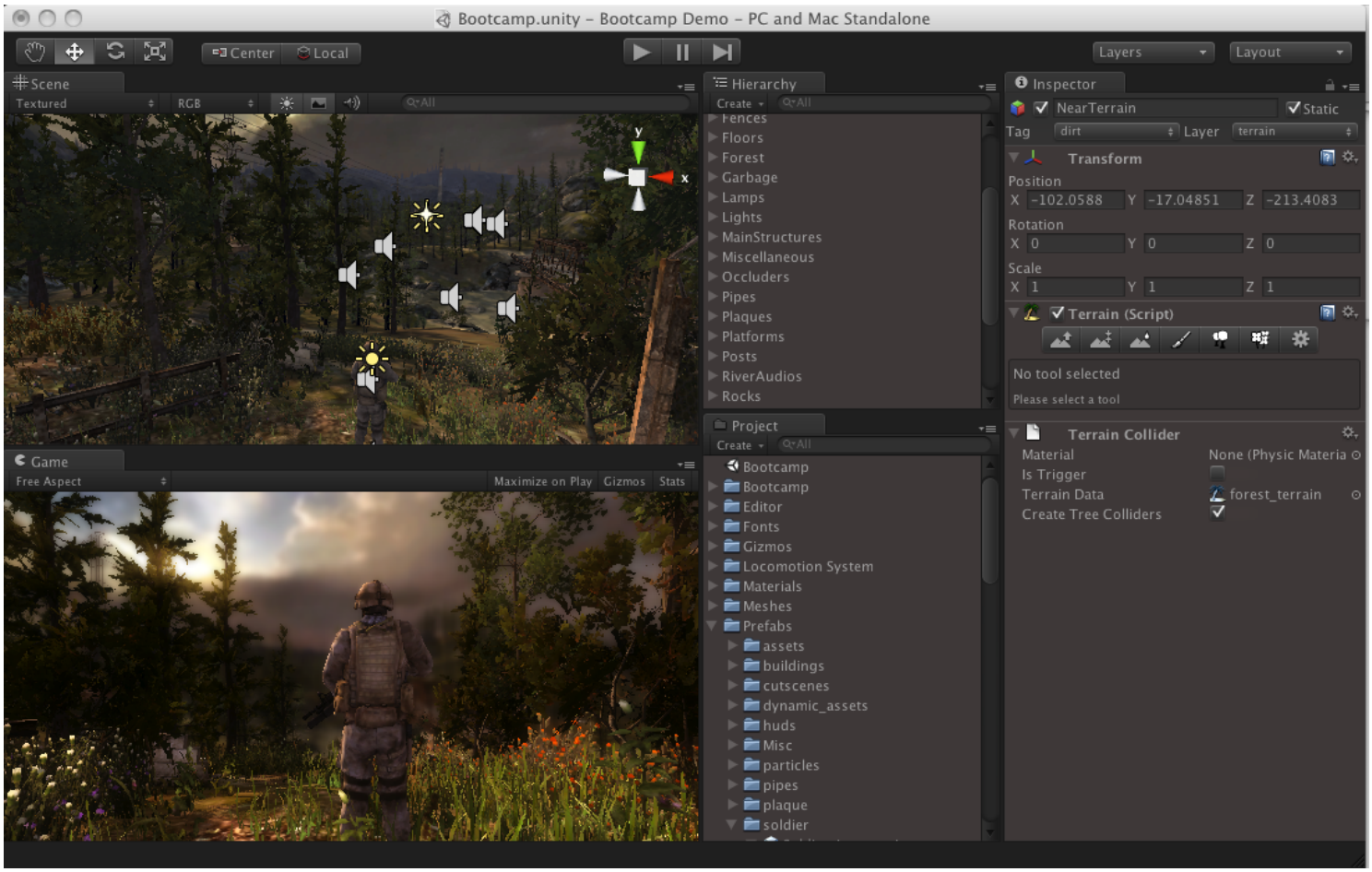
I'm a beginner of Unity myself. I also know what's it's like to learn other programming languages. I've had to go though the bewilderment just like you. I hope this manual will make clear the concepts of scripting that I've learned. I hope teaching you from a "new-user" perspective will turn on the lights for you, too.

Lessons 1 to 7 are for total noobs, those that have absolutely no clue about programming at all. I'm talking really low level and basic. I just attempt to explain the basic concepts in everyday life examples. I don't want you to always have that fuzzy spot in your mind that causes you to give up.

Lessons 8 and higher actually begin looking at applying the concepts to UnityScript.

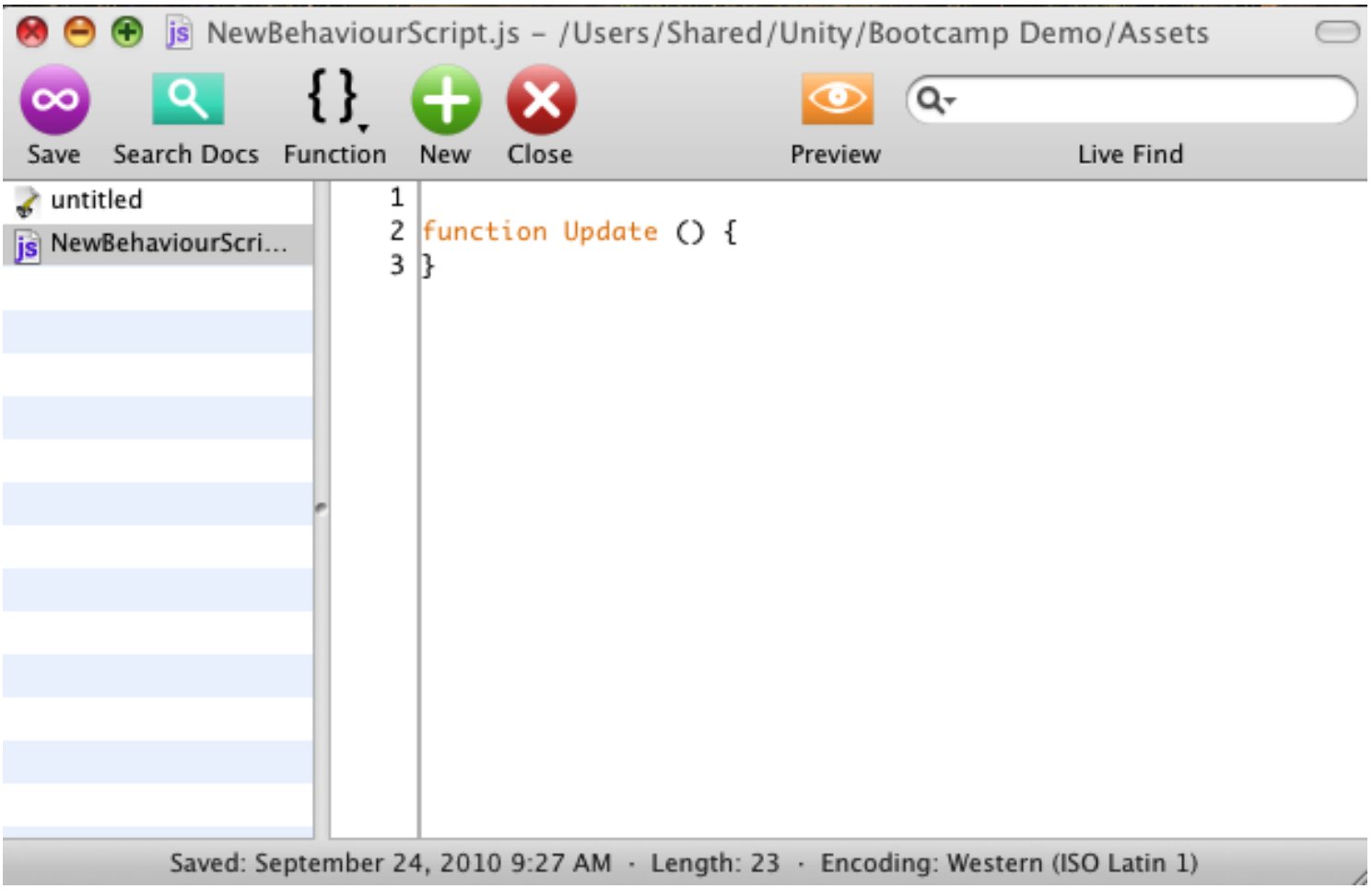
It seems that most beginners can learn fairly easily how to operate Unity, it's primarily the scripting they fear. This Noobs manual assumes you've learned, from other sources, your way around the Unity Editor, even working with asset files. This is strictly about learning the basics of scripting.

The Unity Editor



This is a view of the Unity 3 Bootcamp Demo in the Unity Editor. You should be familiar with the Unity Editor's layout by viewing other tutorials available. These lessons provided here are NOT about the layout or navigation in the Editor. These lessons are only about scripting in Unity using UnityScript (Unity's version of JavaScript).

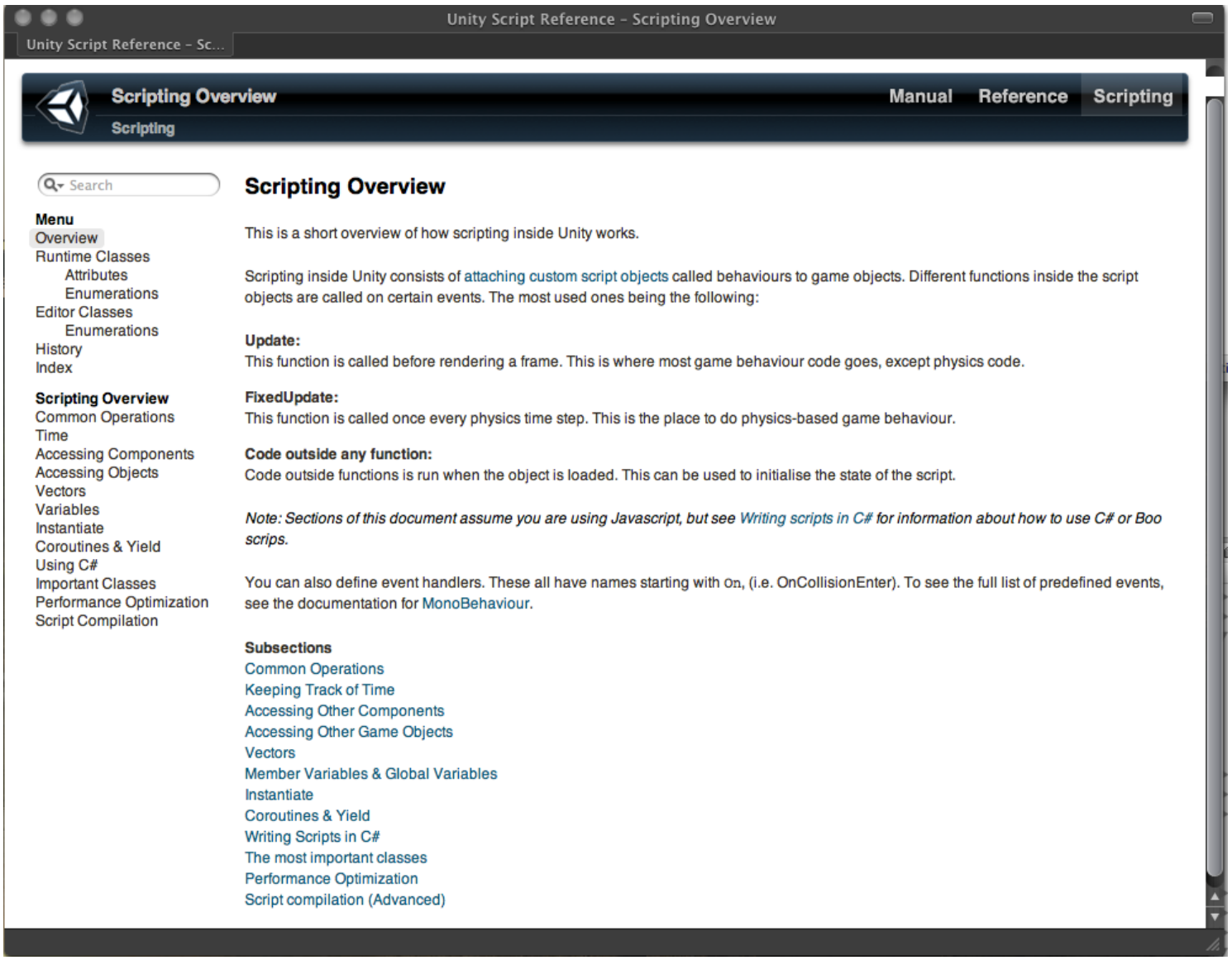
The Script Editor



This is Unitron, Unity's script editor on the Mac, which I use. It will be different on Windows, but they both serve the same purpose.

If you are an absolute noob to UnityScript, and have no idea what the word "script" even means, then these lessons are for you. If just looking at this picture of the editor scares you, then these lessons are for you. If you are an artist, wanting to use Unity to develop a game, and barely know how to think logically, then these lessons are for you.

What you will learn



Scripting Overview

You will learn that you are already familiar with scripts, even though you don't think you do. So I am going to get really basic in the first few lessons so you can see that scripting in Unity is just another way of doing things you already do, you just didn't realize it.

2: Your Game is Full of Stupid Objects

You're all set with your game assets. You have your scene view showing, which contains your GameObjects. You want these GameObjects to move around, listen, speak, pick up other objects, shoot the bad guys, or do whatever you can of dream for them to do. However, all these GameObjects are inherently stupid, they don't know how to do any of this interacting stuff you need them to do.

That's where scripting comes in. "But, but, but I don't know how to program" you say. Well, actually you do. You've been doing it everyday, you just didn't call it programming, or scripting. Do you live each day aimlessly bumping into things without any direction? Do you interact at all with other people, or even pets? Ever talk? Use a restroom?

Step 1

Let's pretend:

Imagine sitting on a couch talking to a friend.

Step 2

Then, the phone rings. You say excuse me, let me take this call.

Step 3

You finish the call and then continue your conversation with your friend.

Now think

Has this ever happened to you in real life, or any thing remotely similar? Was it hard to do? I'll bet you didn't even have to think about the whole process of taking a phone call in the middle of a conversation. You simply did it. Well, guess what? That's exactly the same thing that scripts do in GameObjects.

Let's break it down a bit:

Step 1

You happen to be an object, a person, a GameObject in this game called life. You were running your "**conversation"** script when the phone **interrupted** your "**conversation"** script.

Step 2

You were **called** to perform a different **function** - talk on the phone. therefore you are now running the "**talk on the phone**" script.

Step 3

When you were done on the phone, you **returned** to your "**conversation"** script with your friend.

NOTE:

Notice the words in red. They are very close to the exact words you'll use to write scripts. The same words you already know, understand, and use yourself already.

Scriptphobia

Now I ask you, just how hard was that process? Did you fret about it because you didn't believe you could do all that? Did you get on a forum to ask how to have a conversation, and then talk on a phone in the middle of it all?

> **Scriptphobia**: Fear of not being able to write instructions (I made that up)

Is that what you have? The fear that you cannot write down some instructions in a coherent manner?

The point is, you have been scripting your whole life, you just never had to write what you do on a piece of paper before doing anything. You could if you really wanted to, but it takes too much time and there's no need. But you do, in fact, know how to. Well, guess what? You have to start writing, not for yourself but for the world you're creating.

3: The "God Complex"

Creator of worlds

You have Unity because you wanted to create a game, or something interactive. Well, you're filling that game full of stupid GameObjects and you now have to play God. You created them, now you have to teach them everything they need to know to live in this make-believe world. The is the part where you have to write down the instructions so that your GameObjects can be smart, just like you.

Become the teacher

What you will have to do now is play teacher. You have to teach each GameObject, or Prefab, individually because they each have different things they need to learn. Luckily, every thing they need to know is in the Scripting Reference that Unity provides. All you'll have to write down the instructions in order for your GameObjects to perform.

Proper behavior

Scripting Overview

This is a short overview of how scripting inside Unity works.

Scripting inside Unity consists of attaching custom script objects called behaviours to game objects. Different functions inside the script objects are called on certain events. The most used ones being the following:

The Unity creators are pretty smart. They knew you, as a creator of worlds, would have to teach your GameObjects how to behave in any world you created for them. So why do I say Unity is smart? See the word in the image. (Script Reference)

Now isn't that a clever way to describe what you are going to do to your GameObjects. You will be giving them behaviors. Just like a parent teaches their child how to behave. Not only was Unity nice enough to provide a nice big list of all the things that GameObjects can do, but also how to use them in the Script Reference.

What this means is you can pick and chose, from this list of behaviors, anything you want any GameObject should do. Unity has done all the hard work of programming all these behaviors so that all you have to do is decide what you want to use, then include them in the instructions you write.

The GameObject: Scripts & Components

4: The Class, the Component, and the Script

Before anything else, you have to know what certain words mean. Then you need to know how to use those words. Think back to the time you were in grade school learning English. Boring stuff, but necessary to be able to write well. You'll probably already know some of these words from everyday use which have nothing to due with scripting, but similar enough to easily apply to scripting.

In this chapter I will provide some basic explanations about what a script is, and the two items that make up a script: variables and functions. I'll use general, everyday examples to try an explain these concepts to show how really simple they are. In the next chapters we will get into the details of how to actually use these simple concepts

Unity GameObjects Link

http://unity3d.com/support/documentation/Manual/GameObjects.html

Class: a general explanation

Classes play a major part in Unity. We'll learn what a Class is and how all the other words we'll learn are related to Classes.

What is a Class?

Simple answer - it's a definition of something, just like in a dictionary.

For example, look up "dog" or "duck" in the dictionary and you can read what a dog or duck is. Is the definition in the dictionary a dog? Of course not, but it provides some detail of what a dog looks like and some of a dog's behaviors.

Unity answer - a definition that:
1) Defines the variables that will be used to store information (data); and
2) Defines the functions that are used to perform operations using the data stored in the variables.

In other words, a Class defines:
1) A few words - variable names; and
2) Some things it can do - functions.

Components and Scripts: a little magic

While working in Unity, you wear two hats:
1) A **Game-Creater** hat; or
2) A **Scripting** (programmer) hat.

When we wear our **Game-Creator** hat, you can select a GameObject and view the Properties of each Component attached to that GameObject;

When we put on our **Scripting** hat our terminology changes:

- A script is written to give a GameObject the ability to do things.
- When a script is saved, its filename is the name of a Class - a container of variables and functions;
- This Class (script file) is stored in your Project folder, sitting there, dormant and useless.
- Take off your **Scripting** hat;

The magic happens when we put our **Game-Creator** hat back on and we attach the Class (script file) to a GameObject;

- Wave the magic wand - ZAP - the Class (script file) is now called a Component, and the public variables of the Class (script file) are the Properties you see in the Inspector.

5: Variables in Scripts

What is a variable? Technically, it's a tiny section of your computer's memory that will hold any information you put there. While a game runs, it keeps track of where the information is stored, what value is kept there, and the type of information. However, all you need to know is how a variable works in a script, and it's very simple.

A variable is similar to a mailbox

What's usually in a mailbox, besides air? Well, usually nothing, but occasionally there is something in there. Sometimes there's money (a paycheck), bills, a picture from Aunt Mable, a spider, etc. The point is, what's in a mailbox can vary. Therefore, let's call each mailbox a variable instead, since what may be in it can vary.

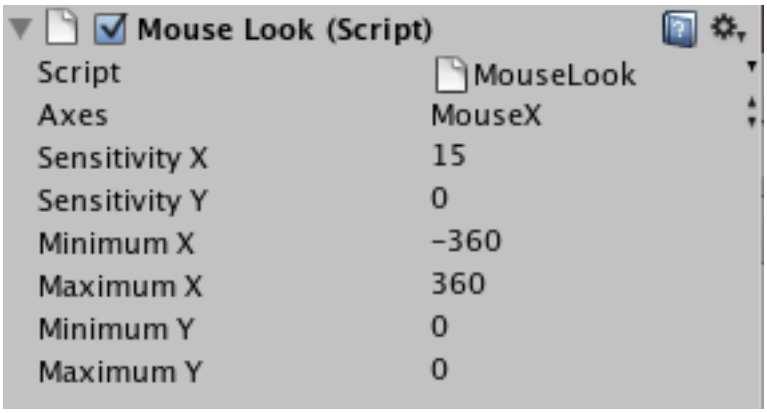
The name of a variable

Using the picture of the country mailboxes, if I asked you to see what is in the mailbox, the first thing you'd say is "which one?" If I said in the Smith mailbox, or the brown mailbox, or the round mailbox, you'd know exactly which mailbox to open to retrieve what is inside.

In scripts, we also have to name variables with unique names so that I could ask you what is in the variable named *myNumber,* or whatever cool name you used?

For those of you more technically curious, variables are names we make up for memory addresses in the computer. When we use a variable in a script, the computer knows exactly where to go in its memory to retrieve, or store, the information we need.

Variables are just substitutes



Huh??? What???

See those names in the left column: Sensitivity X, Sensitivity Y, etc.? Those are the names of variables in a script. See the numbers in the right column? That's the information those variables (mailboxes) hold. So how does it work?

Let's pretend you have a variable (mailbox) named *myNumber.* Stored in *myNumber* is the number 9 (in the mailbox is the number 9). Now, if you saw this, it normally wouldn't make any sense, unless you took algebra in high school:

2 + *myNumber* = Huh???

You can't add words and numbers and get an answer. But you can when a word is a variable in a script, because whatever is in the variable gets substituted for the word (the name of the variable). So in our little math equation, 9 is substituted where *myNumber* is shown, therefore:

2 + 9 = 11

Bottom line, as you write scripts, a variable is simply a place-holder, or substitute, for the actual information you want to use.

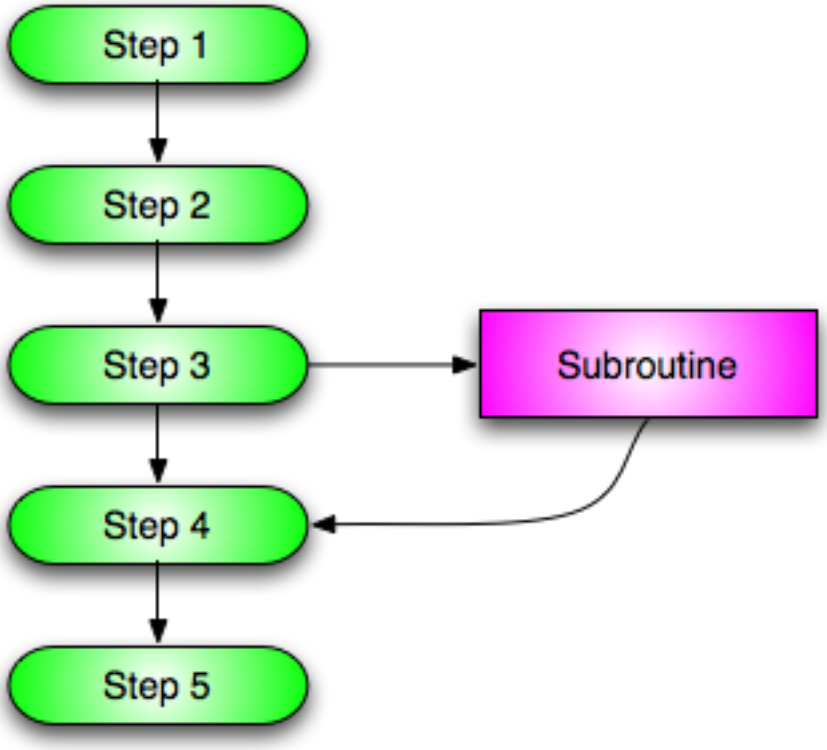
More later

That's the very basics of variables, they are simply a convenient way to hold any information you'll need in your script. Variables can hold different types of information, too, such as words, or even whole GameObjects. There are more finer details related to variable syntax and how we get values in and out of variables, which we'll learn later.

6: Functions in Scripts

Functions is where the action is, decisions are made, tasks are performed. It's the working part of a script. In my original example of talking to a friend and having to answer the phone, talking to your friend was a function you were performing when you had to change and perform a different function of talking on the phone.

Step, step, sidestep, step...



When a script is running, each line of code is run in sequence.

As the script runs, there may be interruptions, such as answering a phone call in the "talking to a friend" example. "Talking to a friend" was the function you were performing when an interruption was detected by you, the ringing of the phone. You had to do something different,you had to perform a different function, which also has its own series of steps. When completed, you returned to the steps of "talk with a friend," right where you left off.

It's only logical to use functions

GameObjects can have many types of input, such as detecting when they run into things, or something runs into it, or the player may press certain keys to direct gameplay. These inputs will call specific functions that do specific tasks, that you created in your script.

These functions are created because they are a series of steps that are used over and over many times. So instead of retyping the same series of steps again and again, we place these steps inside a container of code, and give it a name. When we want to use that container of code in our script, simply write the functions name in our script and that calls the function in to action.

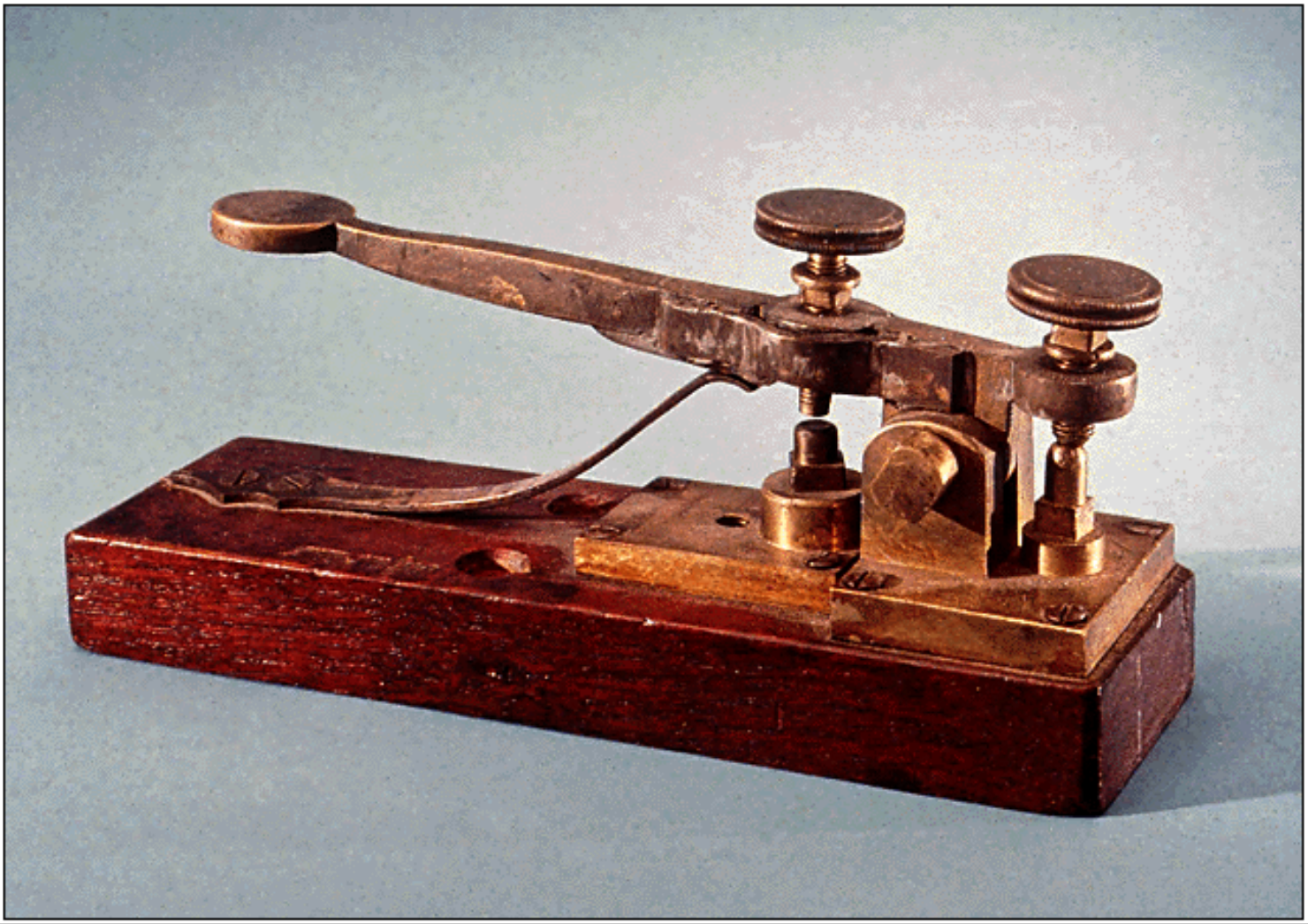
More Later

We will discover later how to properly call functions when needed. For now, all you need to understand is a function is a series of steps that perform a specific task, and the reason a function is even created is because the specific task will need to be performed many times. So it makes sense to put the series of steps in its own section of code, give it a name, then we can call it when we need that task to be performed.

7: Dot Syntax in Scripts: Communicating

Our script has variables, which contains information or data, and the script has functions, which allow for tasks to be performed. I now want to introduce the concept of communicating with other Components in a GameObject. I will just touch on this here, as communication between Components is a vital part of scripting. It's what makes interaction possible.

What's with the Dots?



When you look at code written by others, you'll see words with periods separating them. What the heck is that? Looks complicated as all get-out, doesn't it.

Here's an example from the Unity documentation:

```
transform.position.x
```

Don't concern yourself about what the above means, that comes later, I just want you to see it.

That's called **Dot Syntax.**

Since UnityScript is like a foreign language, there are grammar rules that need to be followed.

Here's another example. It's the address of my house, sorta:

USA.Vermont.Essex.22 myStreet

Looks funny, doesn't it? That's because I used the syntax (grammar) of UnityScript instead of the Post Office. However, I'll bet if you look closely, you can easily figure out how to find my house.

Here's another example. If we met at some random place on the earth, and I asked you to go get my sunglasses using only this:

USA.Vermont.Essex.22 myStreet.2nd floor.office.desk.center drawer.sunglasses

Would you have any problem locating them?

It's an Address?

Yes, that's all it is. Dot Syntax is simply the way you specify any GameObject, or any Component, variable (property) or function in any GameObject in your game.

8: Conclusion

The Next Step

These previous lessons should allow to to proceed to more detailed study of UnityScript now that you understand the fundamental pieces used to make scripts.

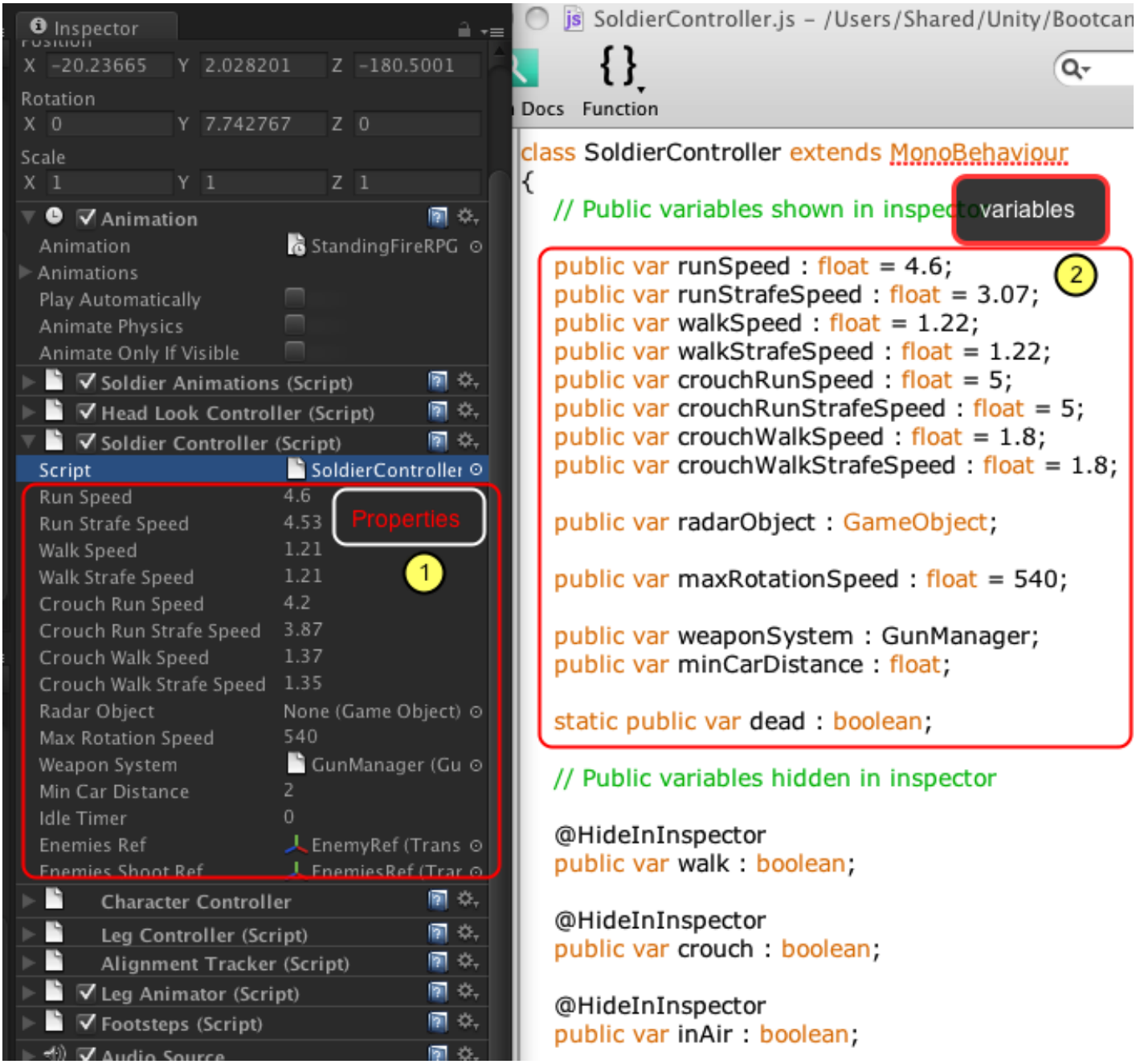
With the concepts of variables, functions, and Dot-Syntax under your belt, now we can proceed to learn some of the details of creating scripts.

Variables in Detail

9: Variables & Component Properties

GameObjects you create will have certain Properties that make them what they are. When you add a Component to a GameObject, you are changing, or adding, its behaviors. This means, of course, that the specific information a particular Component adds to a GameObject has to be stored somewhere within the Component. That information, or data, is stored in **variables**.

Variables become Component Properties

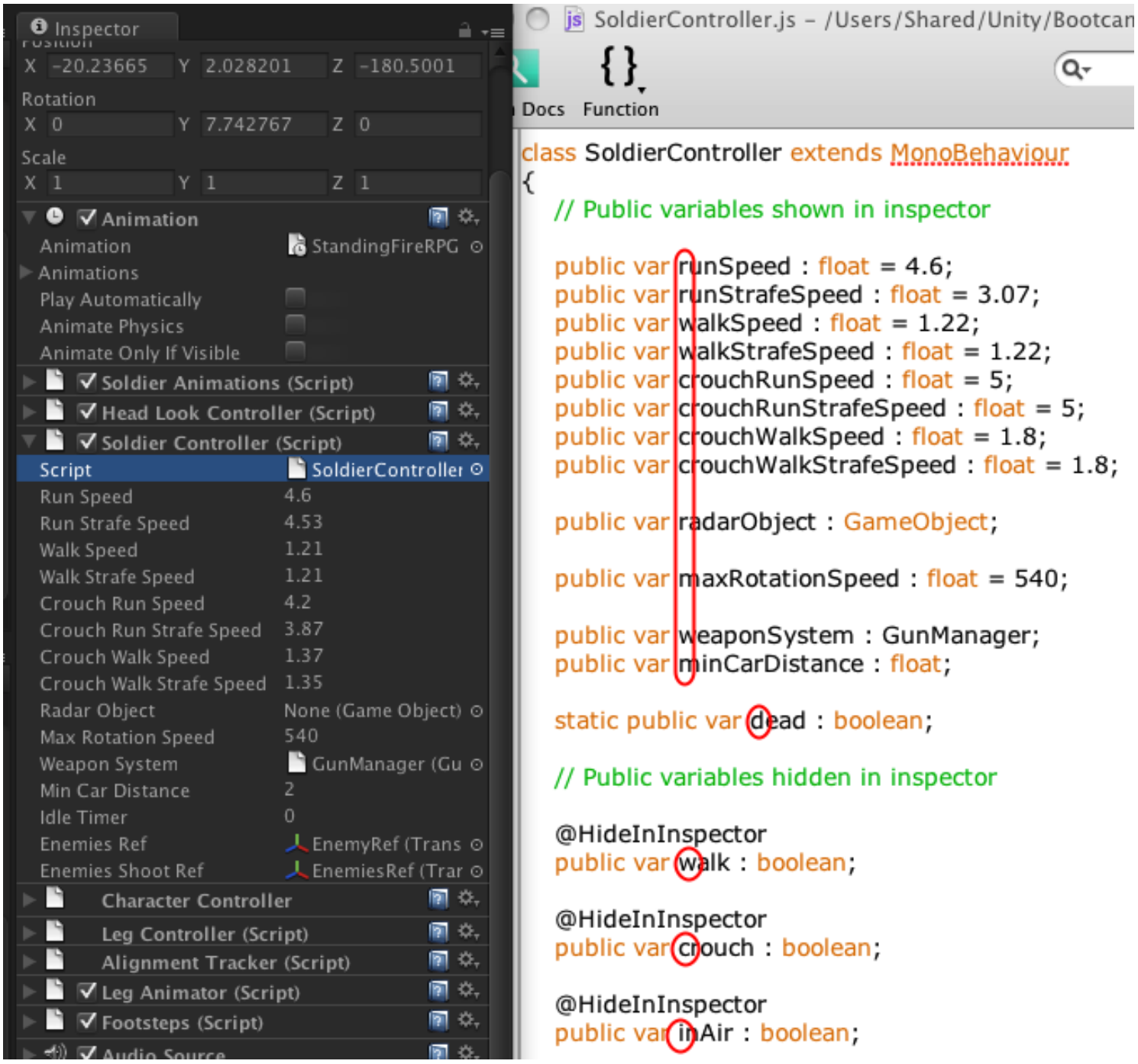


This is a portion of the Soldier Controller script (SoldierController.js)

1. On the left is the Component Properties as shown in the Unity Inspector panel.

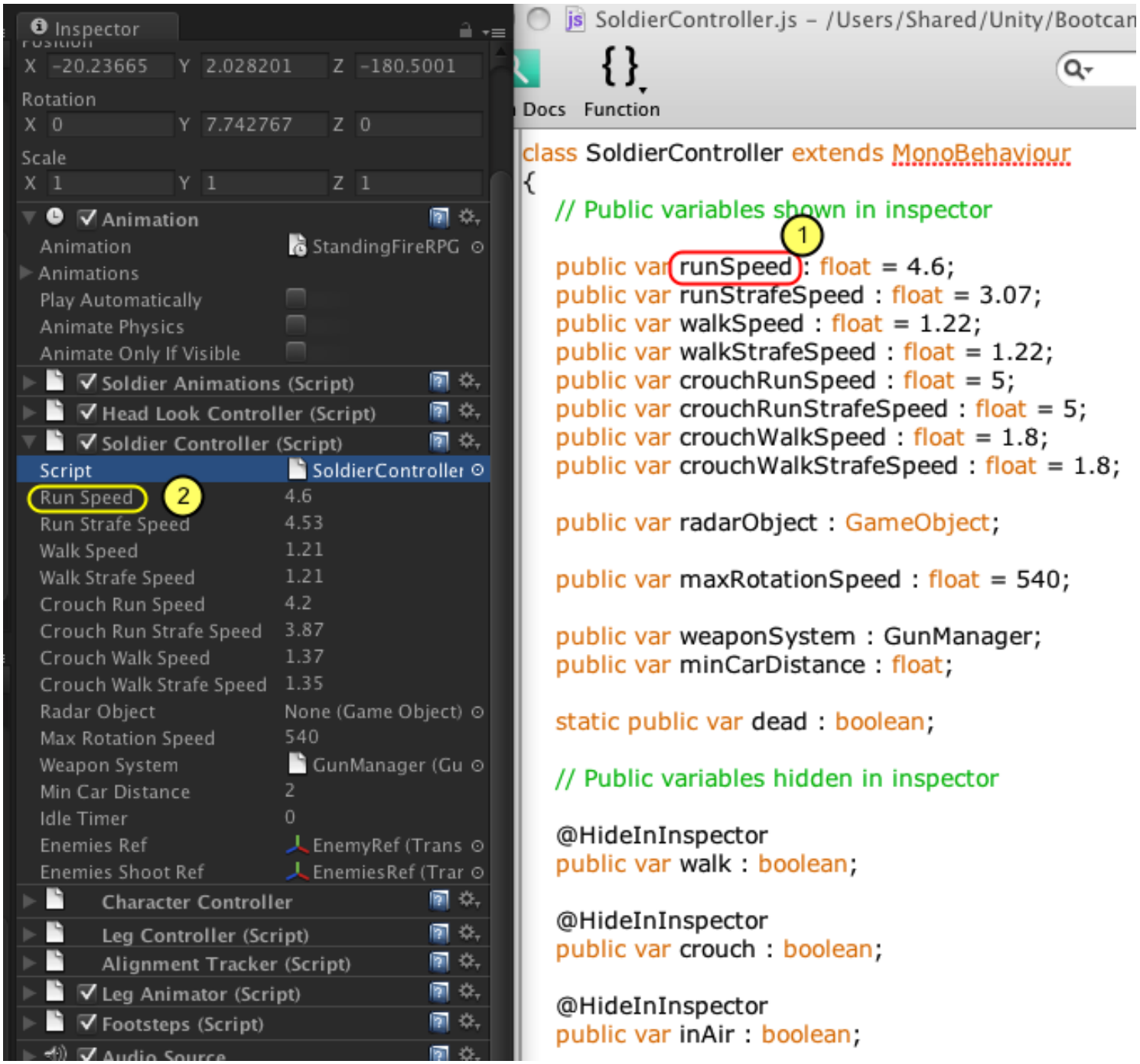
2. On the right is the script in the script editor Unitron.

Begin variable name with lowercase letters



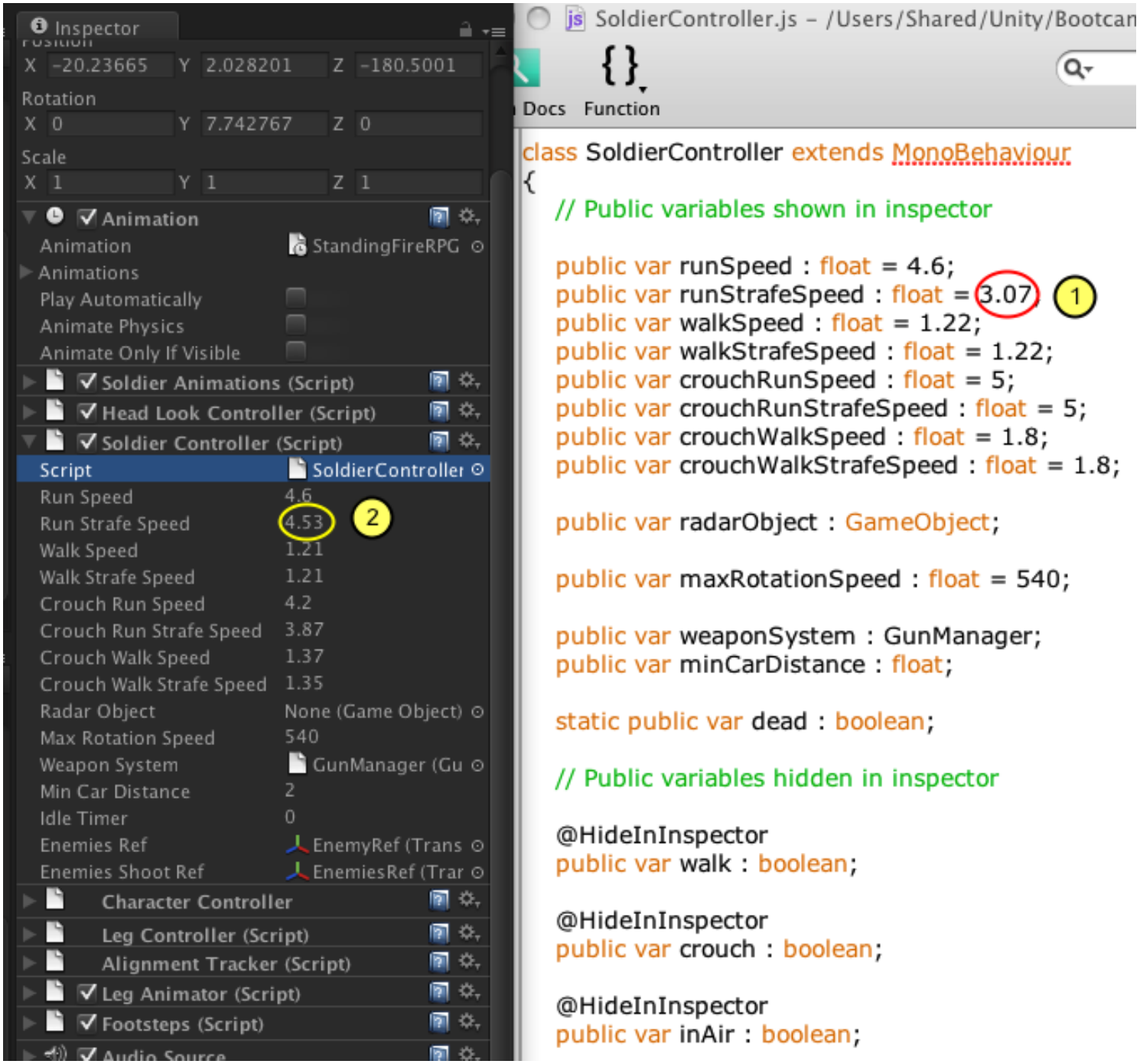
As you can see, every variable begins with the first letter of its name with a lowercase letter.

Multi-word variable names



1. **Variable** names can only be one word long, no spaces. Therefore, to make **variable** names that you want to be more than one word in length requires you to squish all the words together in your script, like **runSpeed**. The name begins lowercase, and each additional word begins uppercase.

2. Unity performs a little magic and turns **runSpeed** into the Property name **Run Speed**.



1. In the script the value is 3.07

2. In the Inspector the value is 4.53

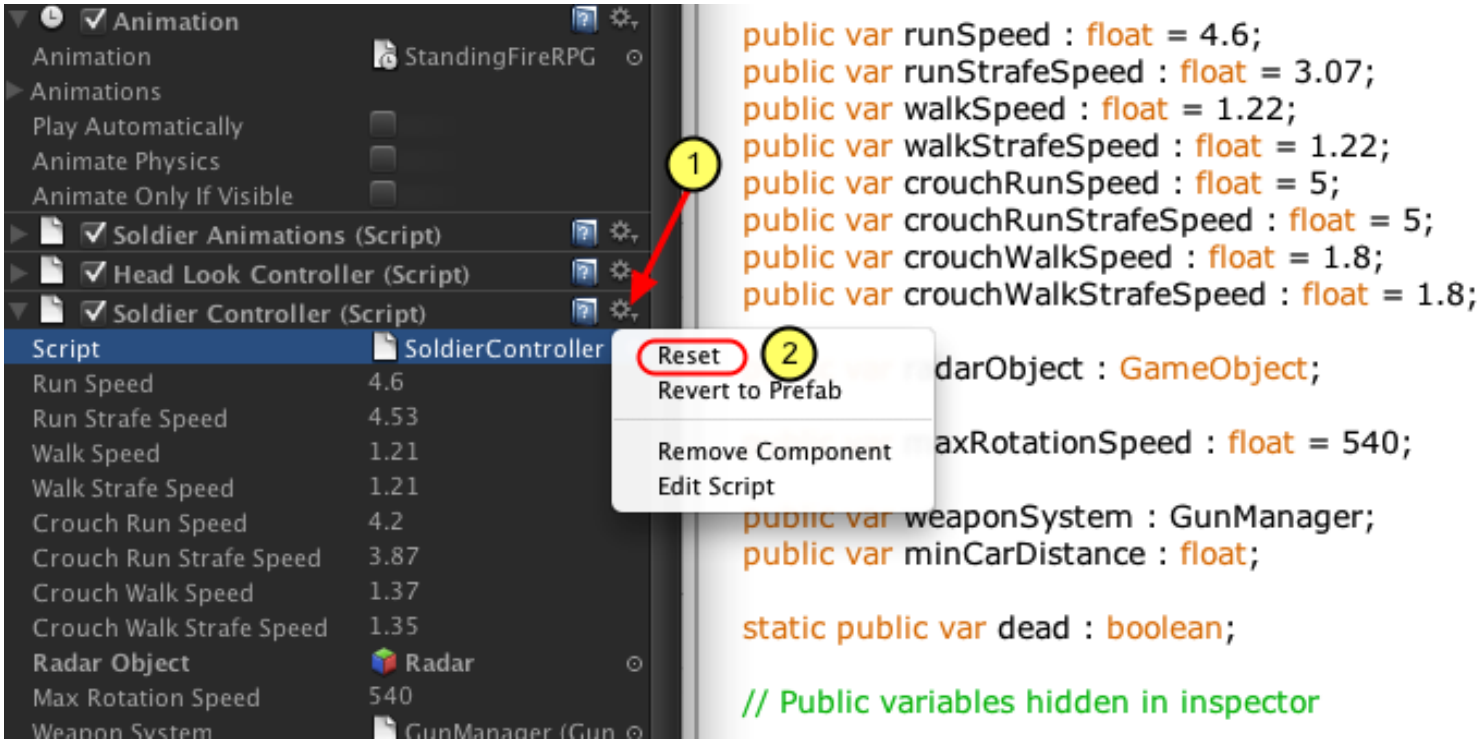
Did something go wrong? No. Values you assign to **variables** in scripts are considered a **default** value. You can tweak the values in the Inspector panel to get your GameObjects to behave just right. Plus, these tweaks you make are retained, meaning you don't need to be constantly modifying your script.

Note: If you make changes in the Inspector while in Play mode, the changes are lost. So while you're tweaking the Property values in Play mode, write down any setting you wish to apply once you exit Play

mode.

Just keep in mind, your tweaking doesn't change the script at all, it remains exactly how you wrote it.

You can reset values in the Inspector



To reset the values in the Inspector panel to the original values in the script:

1. Click the gear;

2. Click Reset

This will change the Property values in the Inspector back to the original values specified in the script.

10: Variable names and "type"

Like the mailbox explanation I gave earlier showing how different types of things can be in a mailbox, **variables** also can contain different types of data. They can contain anything from a simple number, such as the number 3, to a whole GameObjects and Prefabs.

To add **variables** to your scripts, there is some simple grammar you need to learn.

Declaring variables in a script

Every **variable** you want to use in a script must be **declared** in a statement. What does that mean? Tell Unity what you want to create so Unity knows what it is when you use it in your script. Look at line 4.

I've created a **variable** called **easyName**. Right in front of it is the *keyword* **var,** which of course is an abbreviation for **variable**. Putting **var** there tells Unity I'm creating a new **variable**.

Therefore, before Unity can use any **variable** that we want to use, we have to tell Unity about it first. Telling Unity about the **variable** is called **declaring** the **variable**, and you only have to do it once. Now **easyName** can be used elsewhere in the script.

Notice the **declaration** ends with a semicolon. All statements have to end with one, otherwise, Unity won't know where the statement ends and a new one starts. Just like using a period at the end of a sentence. Ever tried to read a whole bunch of sentences without any periods? Can be tough. You could figure it out since you're smart. Unity can't, it's just a lowly computer program.

One more very important thing you have to remember, and it's probably going to bug you (pun). In all this code you see quite a few equal signs (=). In programming and scripting, that is not an equal sign, it's an **assignment operator**. So when you see:

```
easyName = 10;
```

That doesn't mean easyName equals 10. It means **easyName is assigned the value 10**.

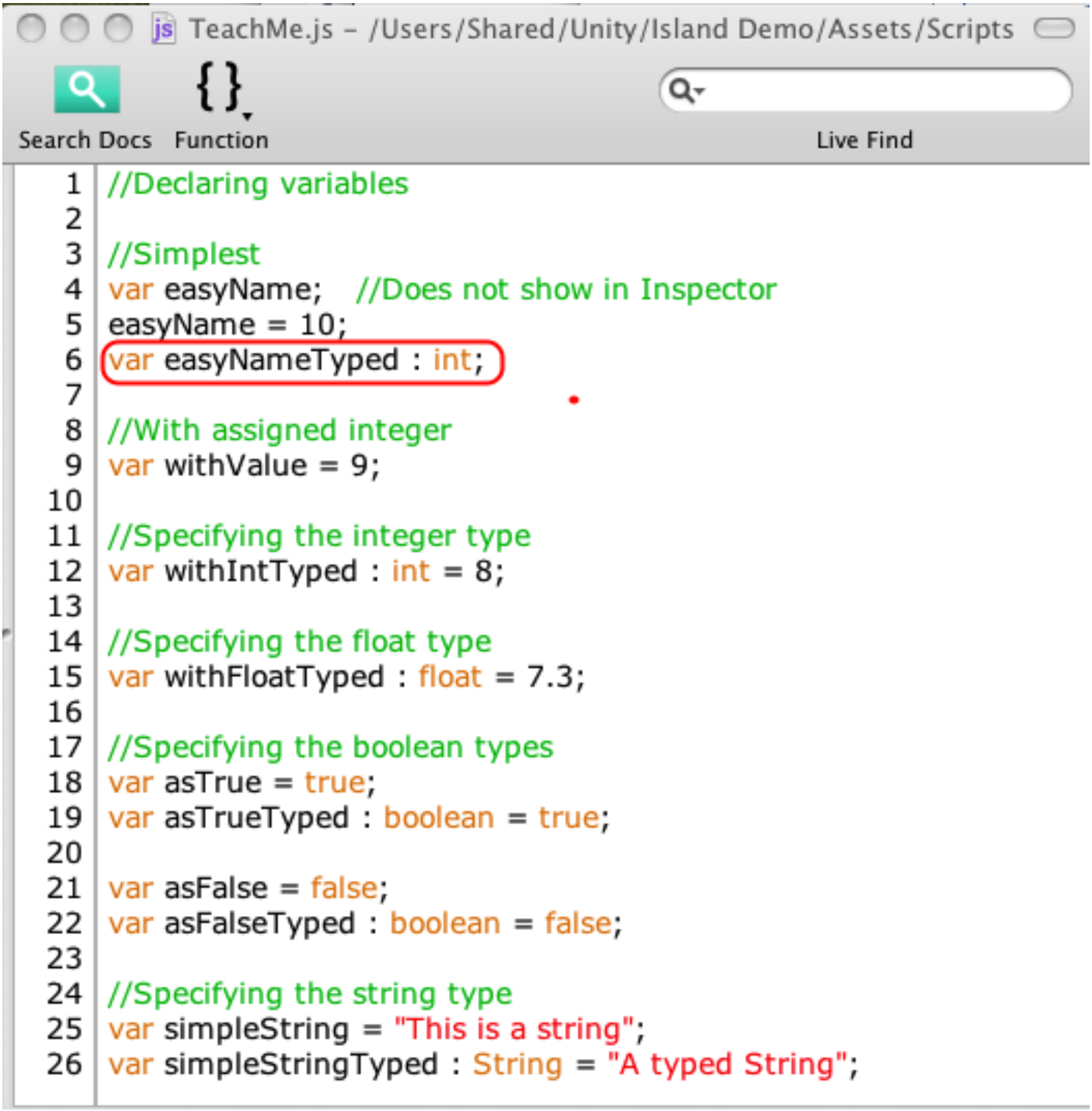
I'll talk about equality in a later lesson.

Common variable types

Type	Contents of the variable
int	A simple integer i.e. the number 3
float	A number containing a decimal point i.e. the number 3.14
String	Characters in double quotes i.e. "Watch me go now."
boolean	Either **true** or **false**

These are the four most common types of **variables** in just about any programming language.

The 'Type' of variable



Look at line 6. This declaration statement has something added at the end. What is that **int** thing?

Well, look at the table above and you see it means **integer**.

I created a **variable** named **easyNameTyped**, followed by a colon, then **int**.

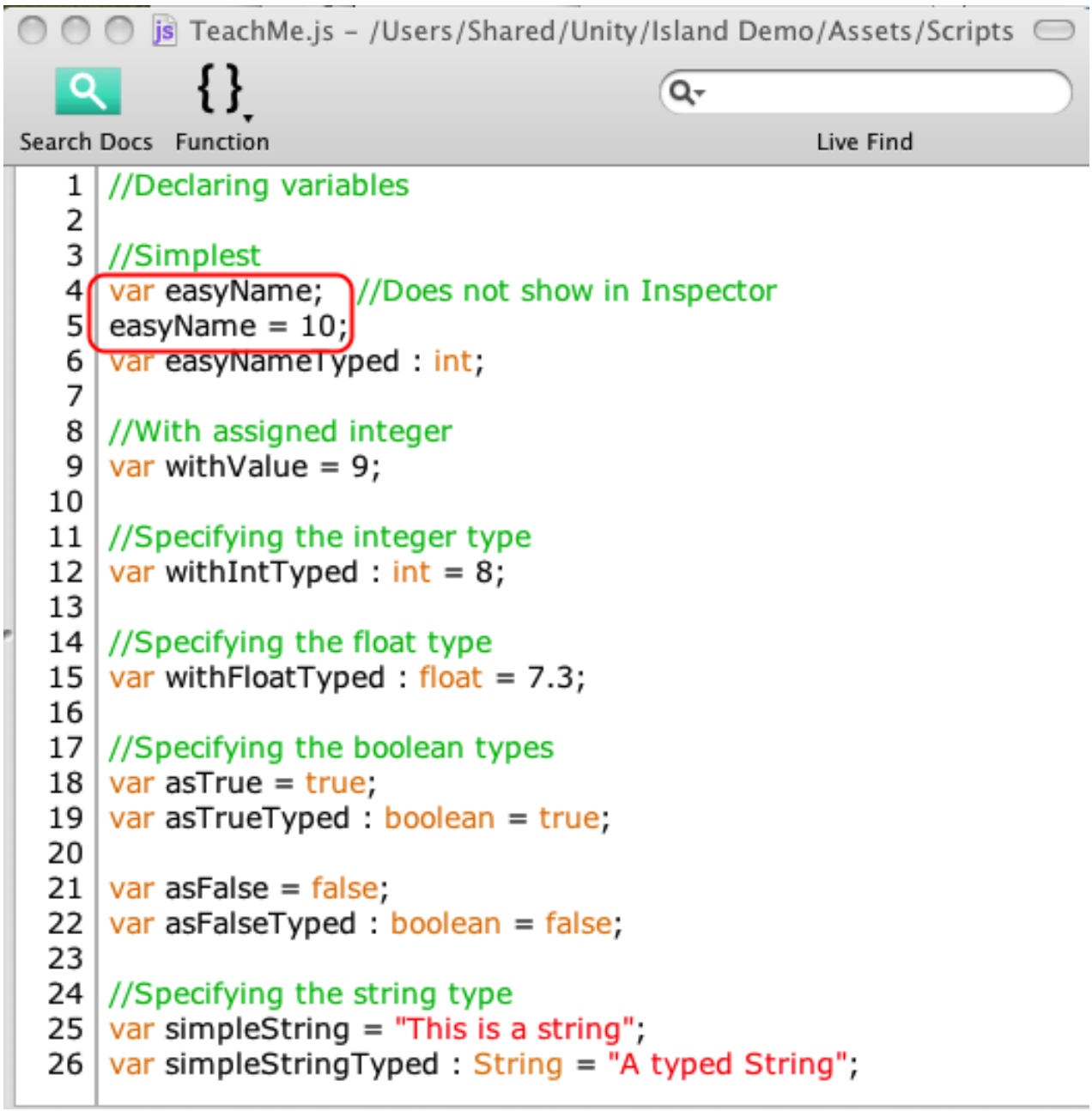
In UnityScript, that's how you tell Unity that you want a **variable** to hold a specific **type** of data. Use a colon followed by the **type**. In this case I've specifically told Unity that this **variable** is to hold an integer

value, a whole number.

OK, so now you probably want to know why the other **variable** **easyName** didn't specify a particular **type**. UnityScript allows you to be lazy and not write so much code as other coding languages require.

When writing scripts with UnityScript, Unity can usually figure out what **type** of **variable** it will be by the value assigned to it. Read on.

Type inference



Back to line 4, and line 5 as well.

When I declared **easyName**, I didn't specify its **type**, and Unity doesn't really care at this point because

easyName isn't being used in the script yet, It's only being declared that it's a **variable**. In line 5 **easyName** is being used for the first time, and it's being assigned an integer, the number 10. Now Unity knows what **type** of values **easyName** will be allowed to hold.

That's how **Type inference** works. The **type** is determined by how it's used the very first time in the script. From now on, as the game runs, any number that the script puts into **easyName,** even decimal numbers, such as 2.3, Unity will strip away the .3 part and only allow the integer part of the number, in this case 2.

While learning UnityScript, I suggest you always specify your **variable** types to help find potential bugs in your code. After you become more knowledgeable, then you can become lazy.

The way to force you to specify types is to add **#pragma strict** to your script as the very first line.

NOTE: Any time a **variable** appears in a script after it's been declared by using **var**, that's a use of that **variable**, such as line 5.

float Type

Line 15 is a **float** type of **variable**. That means the **variable** will store a decimal number, one containing a decimal point.

In this example I have specified **withFloatTyped** as a **float** in the declaration statement, so Unity knows right away what **withFloatTyped** can hold. I also assigned it a value in the declaration statement.

Had I just assigned it the value 7, to Unity it's actually 7.0

boolean Type

What the heck is this **boolean** type thing?

In the Unity Inspector it's displayed as a checkbox. So a **boolean variable** can be either true or false, checked or unchecked.

We will learn about using **true** or **false** later when we get into decision-making in scripts.

String Type

TeachMe.js – /Users/Shared/Unity/Island Demo/Assets/Scripts

Search Docs Function Live Find

```
1  //Declaring variables
2
3  //Simplest
4  var easyName;   //Does not show in Inspector
5  easyName = 10;
6  var easyNameTyped : int;
7
8  //With assigned integer
9  var withValue = 9;
10
11 //Specifying the integer type
12 var withIntTyped : int = 8;
13
14 //Specifying the float type
15 var withFloatTyped : float = 7.3;
16
17 //Specifying the boolean types
18 var asTrue = true;
19 var asTrueTyped : boolean = true;
20
21 var asFalse = false;
22 var asFalseTyped : boolean = false;
23
24 //Specifying the string type
25 var simpleString = "This is a string";
26 var simpleStringTyped : String = "A typed String";
```

Line 26

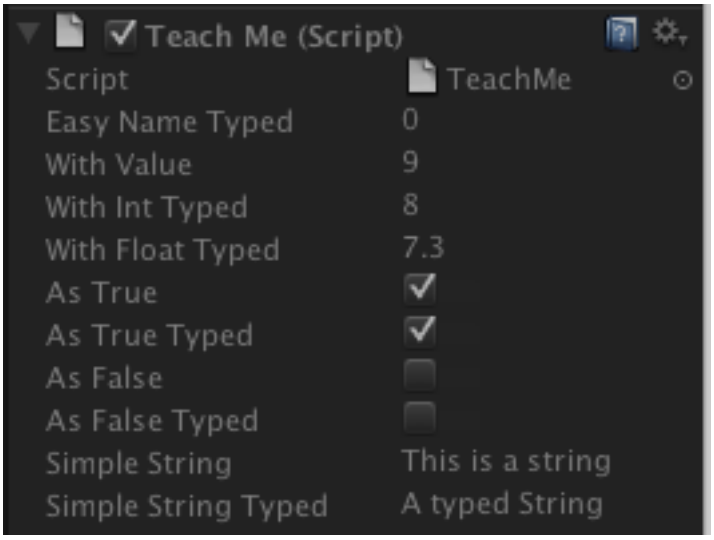
Notice that specifying **simpleStringTyped** as a **String variable** that the word **String** begins with an uppercase letter instead of lowercase. This is because **String** is a class. The others, **int**, **float**, and **boolean** just built-in as part of the UnityScript language.

http://unity3d.com/support/documentation/ScriptReference/String.html

All class names begin uppercase. We will see more of declaring **variables** as class types later, such as the **Transform** class.

Enclose characters in double quotes to form a string of characters.

Public variables



The **variables** I created are public **variables** which means the values stores in them are accessible from other scripts. They also appear in the Unity Inspector. The only exception was line 4. I did not declare a type for the **variable** **easyName**, so Unity couldn't add it to the Inspector. Unity has no clue what type of **variable** add, so it doesn't.

Note: I'll talk about this later when I talk about functions, but for now, **variables** that are declared inside functions are never public.

Private variables

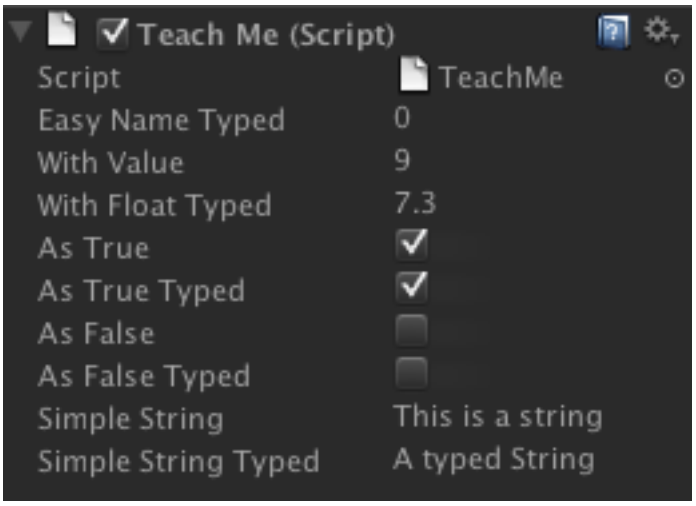
TeachMe.js – /Users/Shared/Unity/Island Demo/Assets/Scripts

Search Docs Function Live Find

```
1  //Declaring variables
2
3  //Simplest
4  var easyName;   //Does not show in Inspector
5  easyName = 10;
6  var easyNameTyped : int;
7
8  //With assigned integer
9  var withValue = 9;
10
11 //Specifying the integer type
12 private var withIntTyped : int = 8;
13
14 //Specifying the float type
15 var withFloatTyped : float = 7;
16
17 //Specifying the boolean types
18 var asTrue = true;
19 var asTrueTyped : boolean = true;
20
21 var asFalse = false;
22 var asFalseTyped : boolean = false;
23
24 //Specifying the string type
25 var simpleString = "This is a String";
26 var simpleStringTyped : String = "A typed String";
```

I made a slight change on line 12. I added the word **private** before **var**. This means the **variable withIntTyped** is not accessible from other scripts. Only the TeachMe script can use it.

Private variables are not in the Unity Inspector



As you can see, making **withIntTyped** a **private variable** does not allow it to show in the Inspector.

Conclusion

Seems like a lot of information to know about **variables**. However, it's just a few simple steps:

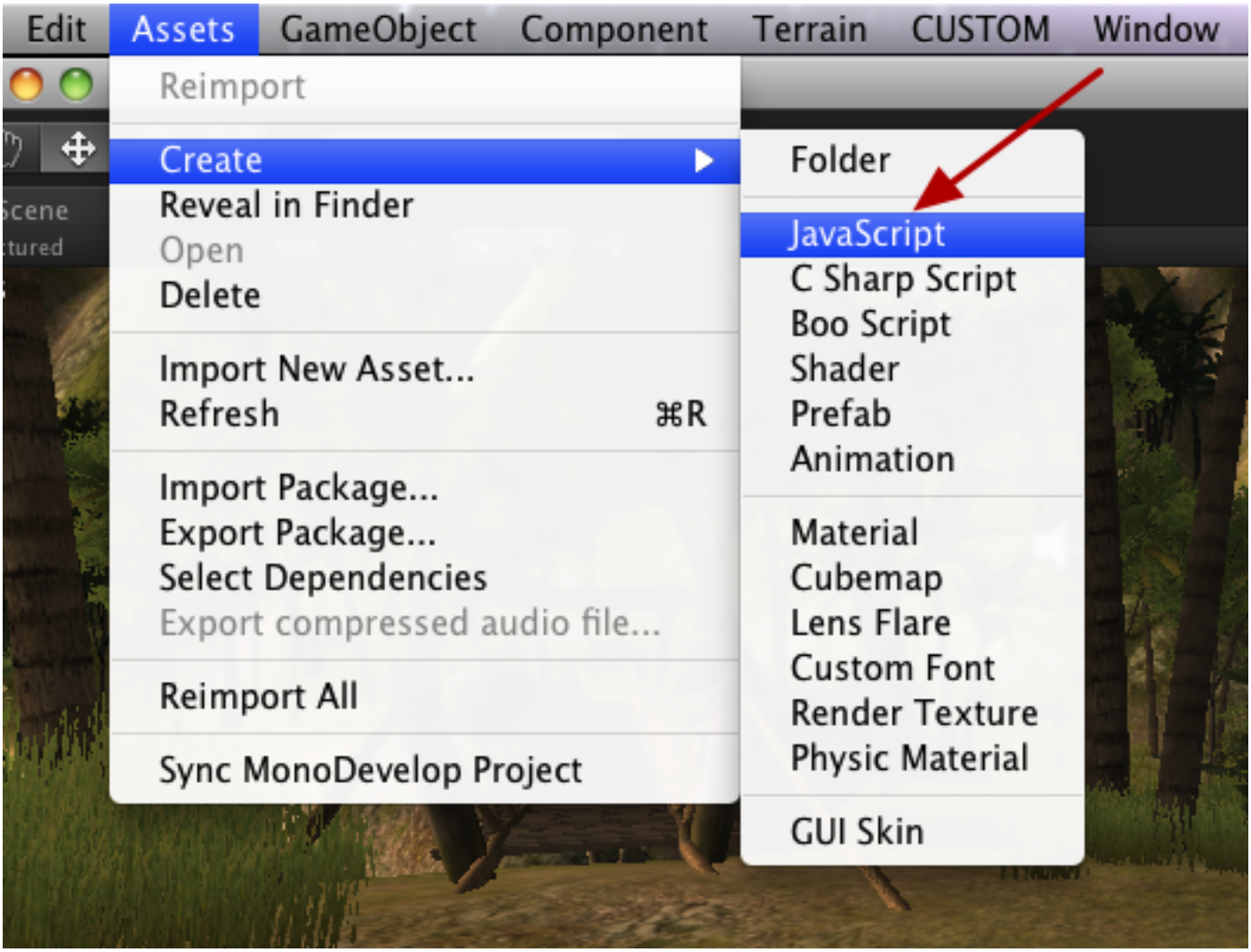
1. Do I want it to be public or private?
2. Write **var**
3. Create a name for the **variable**
4. Specify the **type**
5. Do I assign a value now, or later in the script or the Inspector?
6. End with a semicolon.

Functions in Detail

Unity's Update() & Start() functions

There are functions that Unity provides that make your scripts run, so that's where we will begin our study. The most common function provided by Unity is the Update() function. Every time you create a new script, you will see this function automatically pre-entered for you. Therefore, it must be pretty darn important. Have a look.

Create a new script



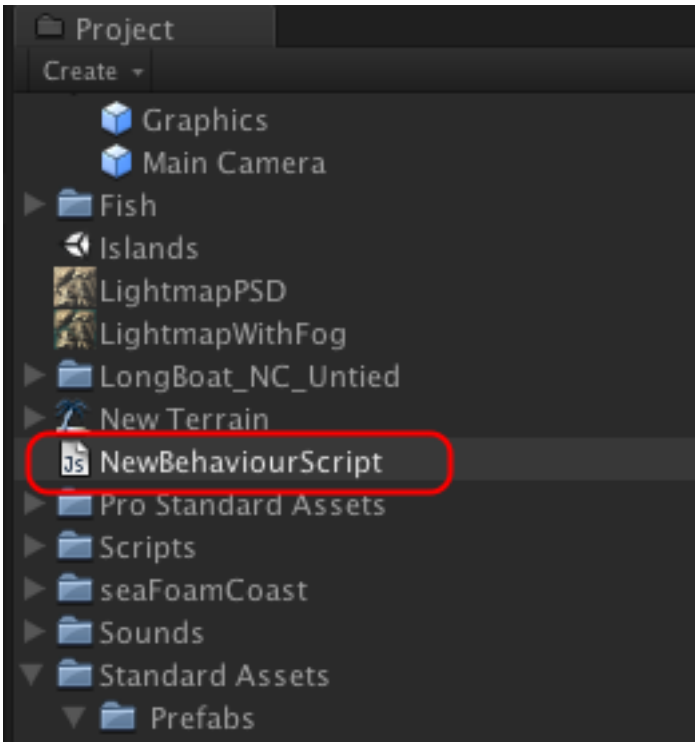
In the menu select Assets I Create I JavaScript.

Why JavaScript? There are many discussions on the Unity forum about the JavaScript version that Unity uses for scripting. Many prefer to call it UnityScript since it's not truly JavaScript, just very similar. I also prefer to use the term UnityScript.

Notice that the menu item selected was Assets. Remember, when scripts are attached to GameObjects, the script magically turns into a Component of that GameObject. Look up the word asset

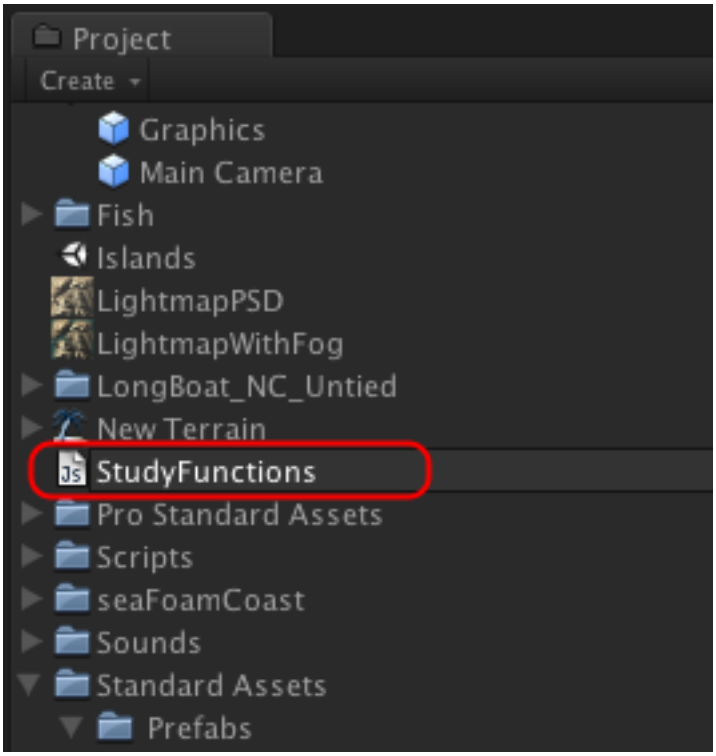
in a dictionary and it's basically "*a useful or valuable thing or quality.*" Sure enough, a script will make a GameObject more useful, hopefully.

A script named NewBehaviourScript is created



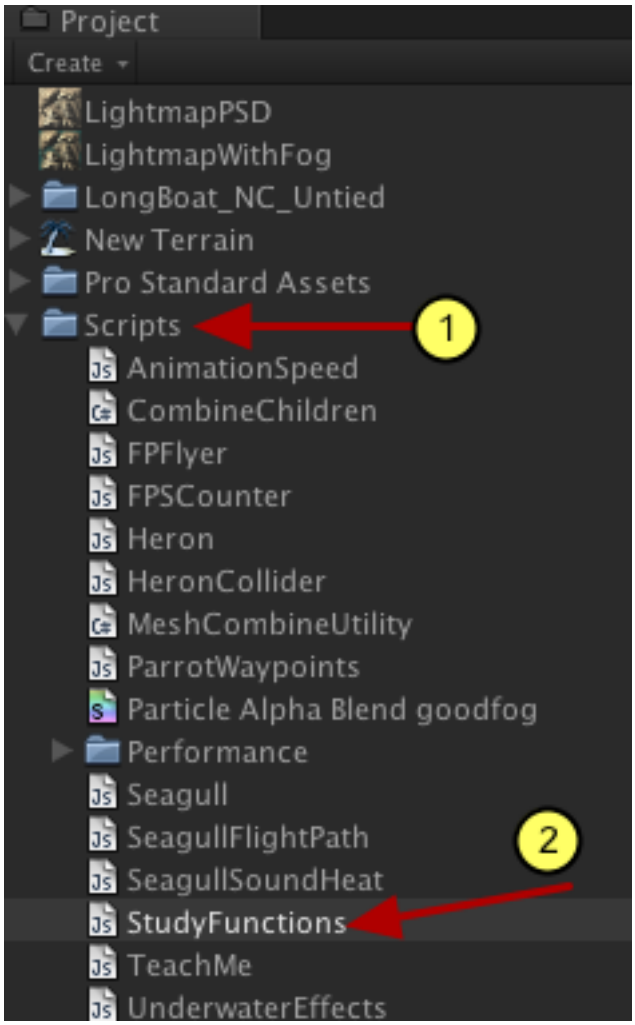
This new UnityScript file is created in your Project panel. By default it's given the name NewBehaviourScript.

Change it's name



Select the file and press the Enter key to edit its name. Script files should be given names that describe what they do. Since this is for studying functions, I'm naming it StudyFunctions. Press Enter again to save the new name.

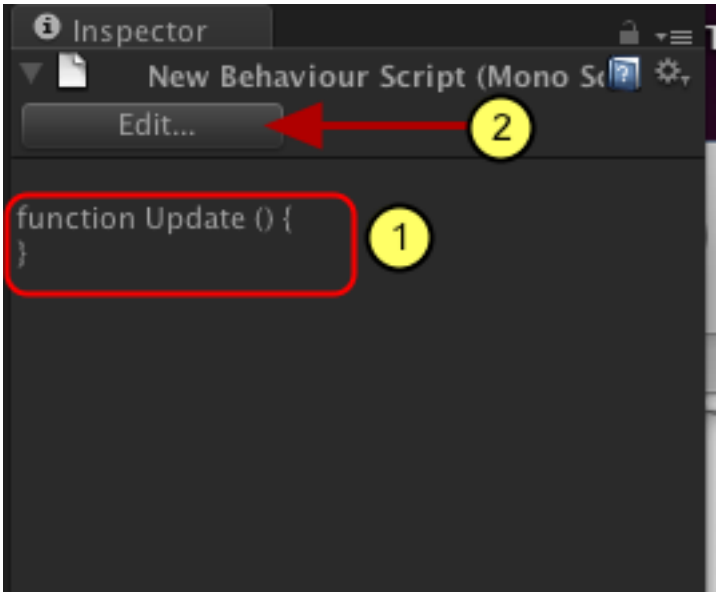
Move the script to the Scripts folder



1. Just drag the file to the Scripts folder.

2. There's my script.

Look at Unity's Inspector panel

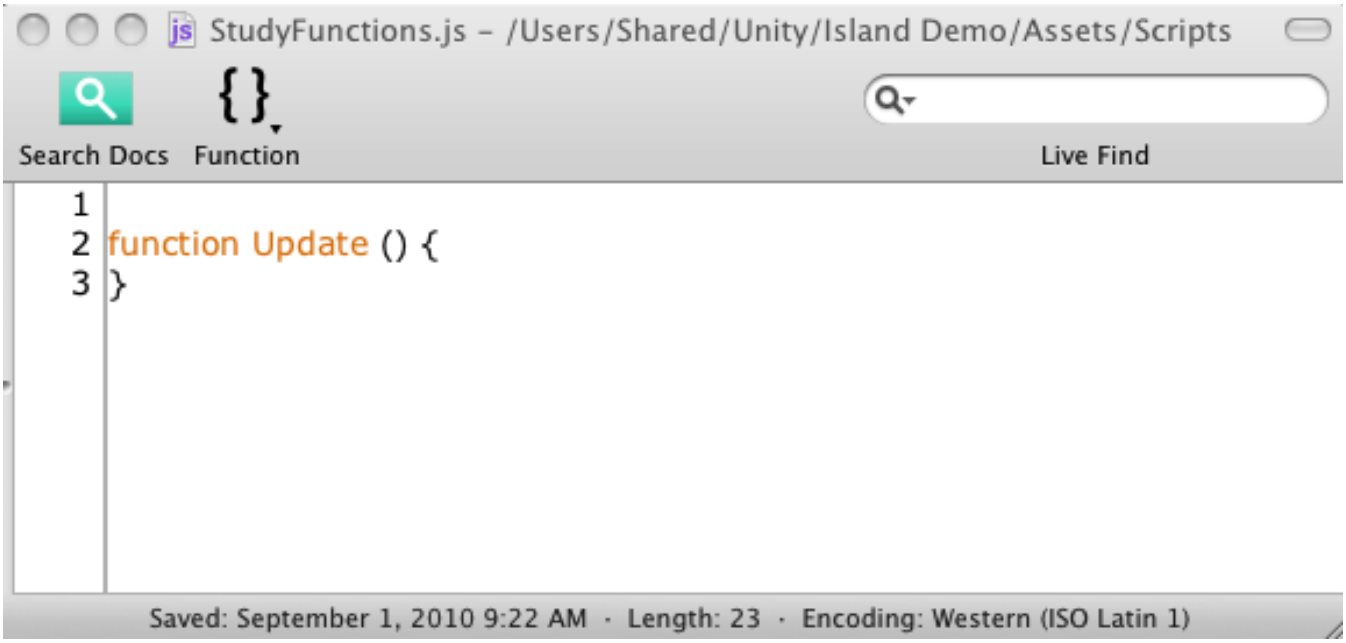


While the script is selected (highlighted) in the Project panel, the Inspector panel shows the contents of the script.

1. There's the Update() function that's automatically inserted.

2. Press the Edit button to open the file in the script editor.

Update(): the "Do-It" again, and again, and again function



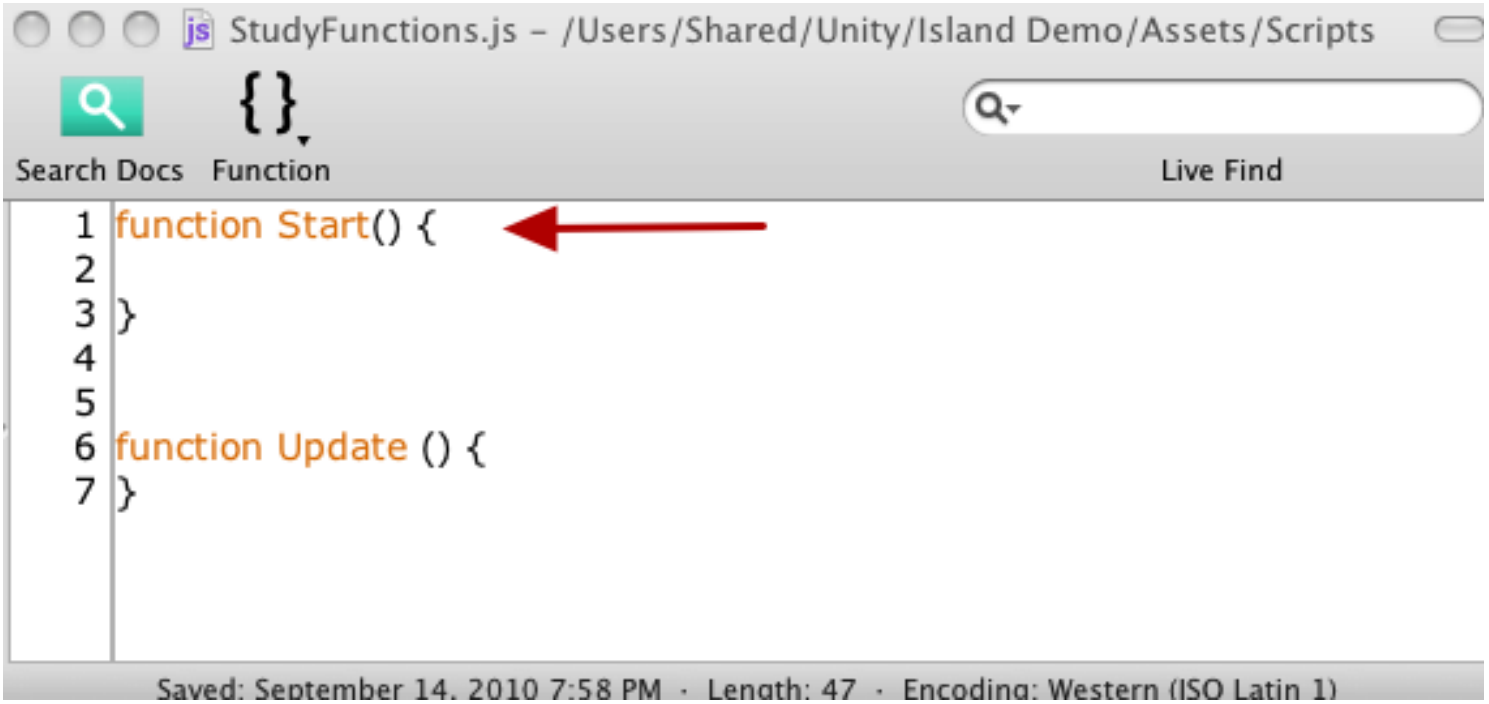
Why is this particular function, Update(), automatically inserted into new scripts?

Unity automatically calls this function every frame which is what makes your code execute. (Update()

Reference Link)

As you study the Scripting Reference Manual, you notice that a vast majority of the sample code is enclosed in the Update() function.

Start(): the "Do-It" just once time function



Another very common function supplied by Unity is the Start() function. This function is called only one time.

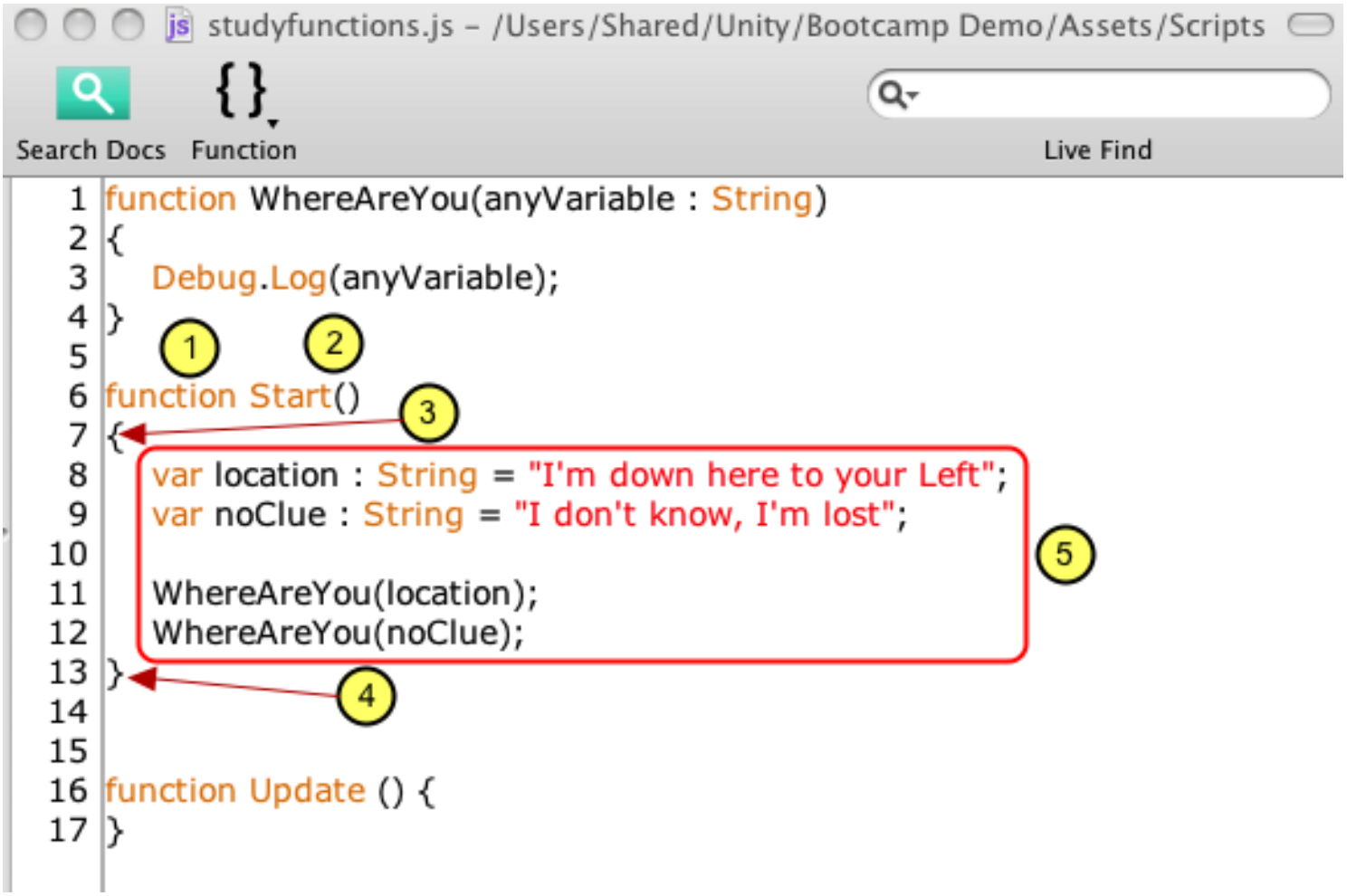
For any script, that is a Component of a GameObject, and has the Start() function in it, Unity automatically calls Start() when the GameObject is initially used in a scene.

Function names

I already went over what functions are, and the reason to have them (link).

What I show now is how to create functions, explaing the parts.

Declaring functions



Every function you, or Unity, want to use in a script must be **declared**. What does that mean? Tells Unity that you created a specific series of code steps and gave the whole group a name. Unity will then know what it is when you, or Unity, call for it to be used in your script.

Let's take a look at the Start() function. This is a functtion that Unity itself will call, not you. However, it's still up to you what code steps will be in the function.

1. Right in front of **Start()** is the *keyword* **function**. Putting the *keyword* **function** there tells Unity you're creating a new function in this script.

2. Immediately following **Start** are the round brackets, **()**. What's the purpose of those round brackets?

Besides being part of the name of a function, it's the way to pass data into the function's code.

I can just see you scratching your head, wondering what that means. Besides, the Start() function doesn't have anything between those round brackets, so what data is being passed to the function? Actually nothing here, but fear not, I will explain in the following steps.

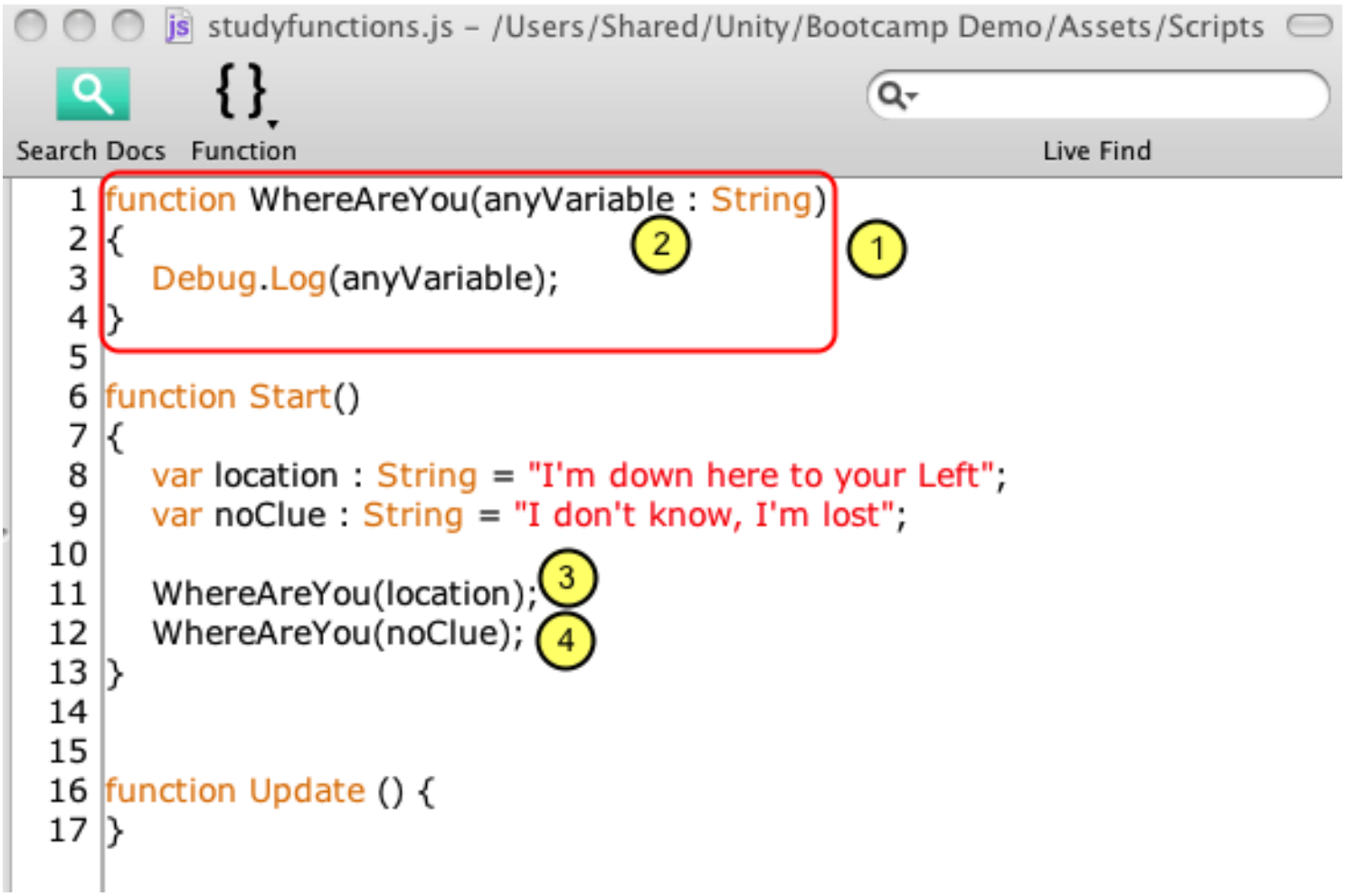
3. This is the beginning curly brace,
4. This is the ending curly brace. **NOTE: function declarations to not end with a semicolon.** The compiler already understands that the final curly brace is the end.

5. Everything between these curly braces (3 & 4) is the **body** of the function, the code this function will execute when the Start() function is called upon to do its thing.

NOTE: A bit of personal information. When I first started learning about functions many moons ago, I had a hell of a time trying to understand this stuff. I simply couldn't grasp what was happening. I understood that functions were containers of re-useable code, that was easy. However, calling a function and passinng data into the function just wouldn't sink in, primarily because geeks wrote the books, and they don't explain in enough detail. That's why I'm writing these lessons, because I remember the agony I went through trying to figure this stuff out.

This next step will delve into the brain-freeze I had.

My functional brain-freeze



1. This is a function declaration for a function I made, called **WhereAreYou()**. As you can see, it has all the parts described in the previous step.

2. Something extra is in this function. Right between the round brackets is a special variable called a *parameter.* I named this variable **anyVariable : String**. This tells Unity that I want to send some String data into the function so that the function's code can do something with it.

Functions can have from zero to any amount of parameters you need to get the data you want into the function. Each parameter is separated by a comma. The Start() function has no parameter, therefore, nothing is passed to the function. **WhereAreYou()** has one parameter.

Here's the part that threw me for a loop for quite a while:

Looking at the parameter **anyVariable : String**, a person just learning might look at that and say "Ok, it's going to pass some data into the function using the variable named **anyVariable**, and the type of data in it is a **String**."

BUT, when I see the **WhereAreYou()** function actually being called:

3. WhereAreYou(location);
4. WhereAreYou(noClue);

What the heck is going on? #**3** has a variable named **location** as the parameter, and #**4** has a variable named **noClue** as the parameter. How doest that work? Neither **location** nor **noClue** are in the function declaration.

After searching long and hard in books (no Google back then), I still didn't find the simple answer. Nobody explained it. It was through trial and error, messing with coding that eventually the lights turned on. In this example, the variable **anyVariable** is just being assigned the value from the variable **location** during the first call, and **noClue** when the function is called the second time.

In otherwords, it's doing this behind the scenes:

anyVariable = **location**;

and then:
anyVariable = **noClue**;

The value in **location** is copied to **anyVariable**. Same thing with **noClue**. Then the variable **anyVariable** is used within the function.

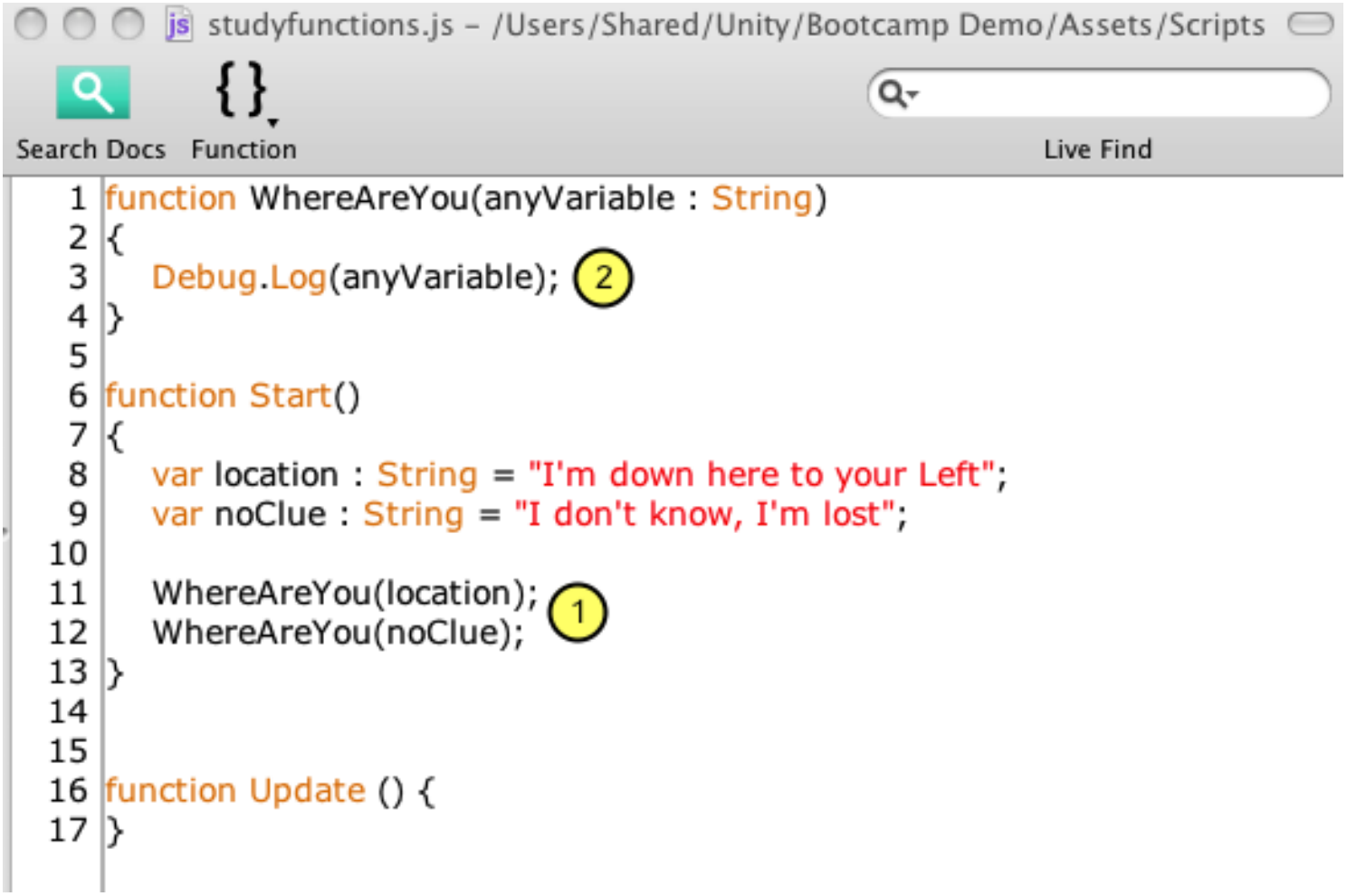
So, when I call the function, as in #**3** or #**4**, any String type variable can be the parameter. It's that simple.

Also, because a function works with a copy, the values in the variables, such as **location** and **noClue**, remain intact while the function can manipulate or change only the copy. These two examples don't make any changes, but functions you create will more than likely be doing all sorts of manipulating.

Function calls and flow

When functions are called, it's a simple sequence of steps with detours to other functions.

Let's look at function calls



1. In **Start()**, there are two calls to the **WhereAreYou()** function;

2. This is what the **WhereAreYou()** will do - **Debug.Log()**, another function, will provide output to Unity's console.

Script flowchart

On the left side is the Start() function executing its steps;
On the right is the WhereAreYou() functiion executing its steps.

1. In **Start()**, there are two calls to the **WhereAreYou()** function;

2. This is the first time **WhereAreYou()** is called and executed;

3. **Debug.Log()** is call to provide output to the console;

4. This is the second time **WhereAreYou()** is called and executed;

3. **Debug.Log()** is called to provide output to the console, again;

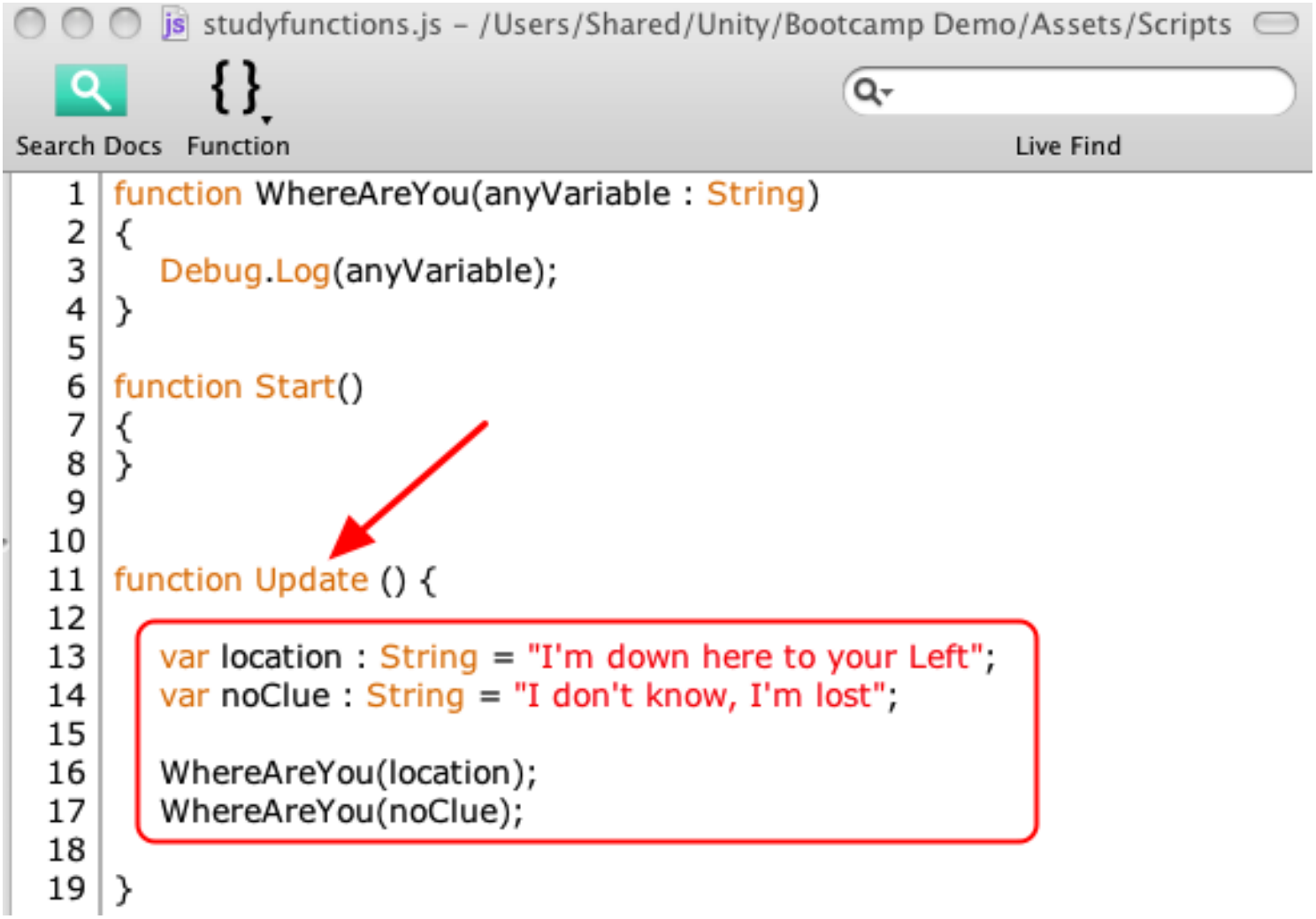
Console output

1. Result of first call to **WhereAreYou()**;

2. Result of second call to **WhereAreYou()**.

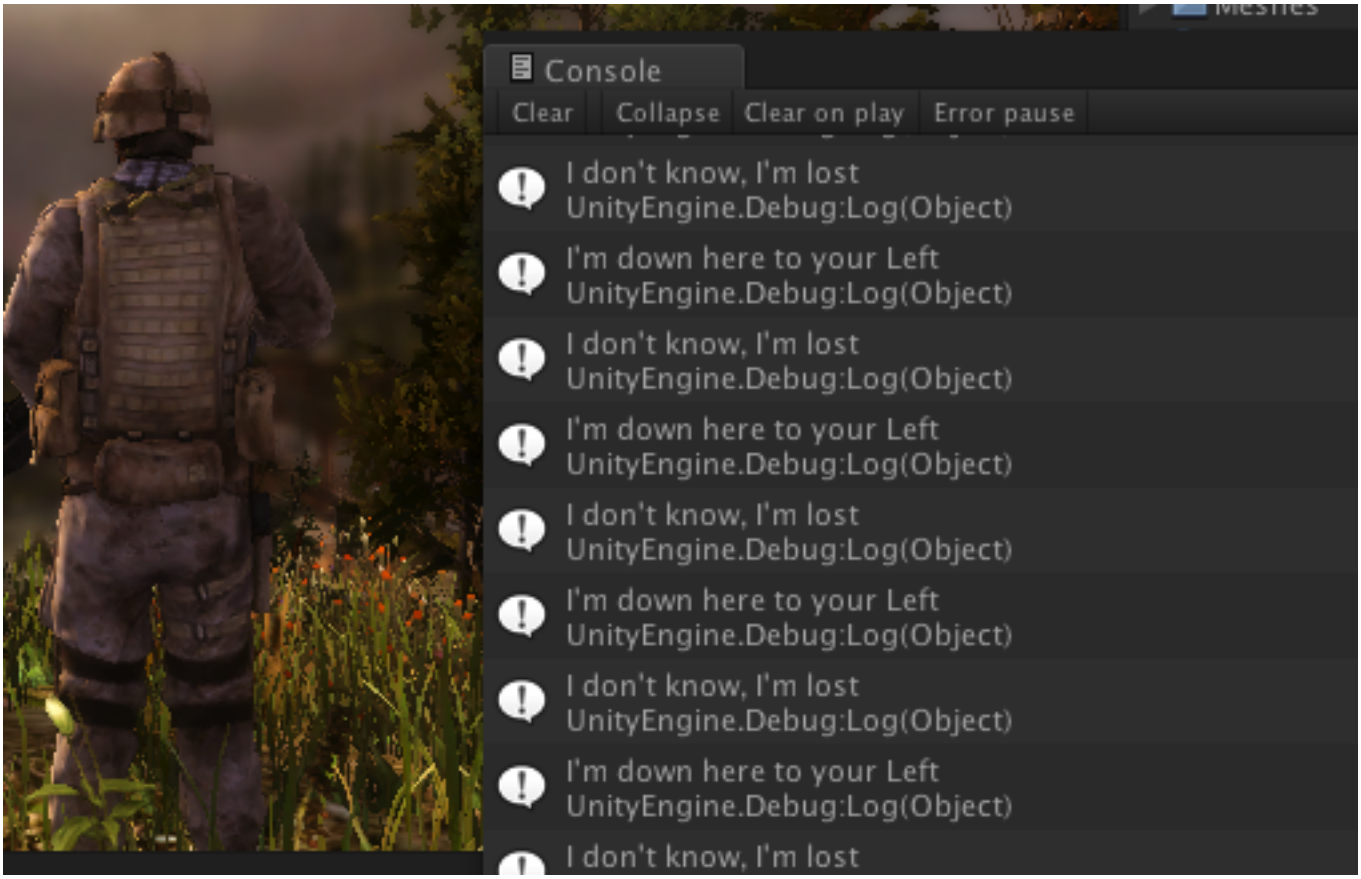
To have this script work, it had to be attached to a GameObject. I added an empty GameObject to the scene and attached the script. Pressed Play. This was the output.

What happens using Update() instead?



I moved my cose from Start() to Update().

Result of using Update()



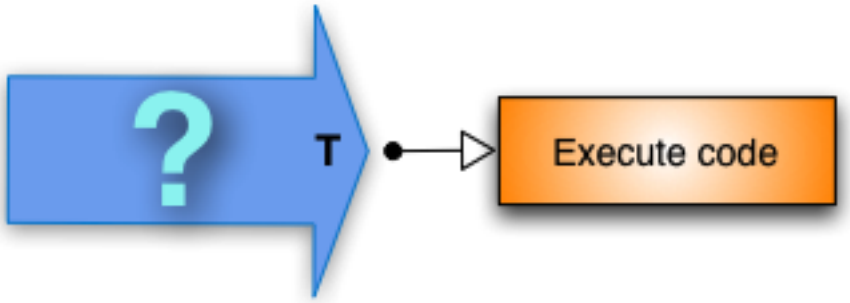
As expected, since Update() is called every frame as the game runs, output to the console continues repeatedly until I stop the game.

Decisions, decisions, always having to make decisions

if?

If, if, if. If I do thist... if I do that... What happens if... Certainly you've used the word **if** in a sentence, or a thought, in your daily life. More than likely several hundred times a day. So you already know how to use the word **if**. It's pretty much how you make decisions.

if Statements, true or false

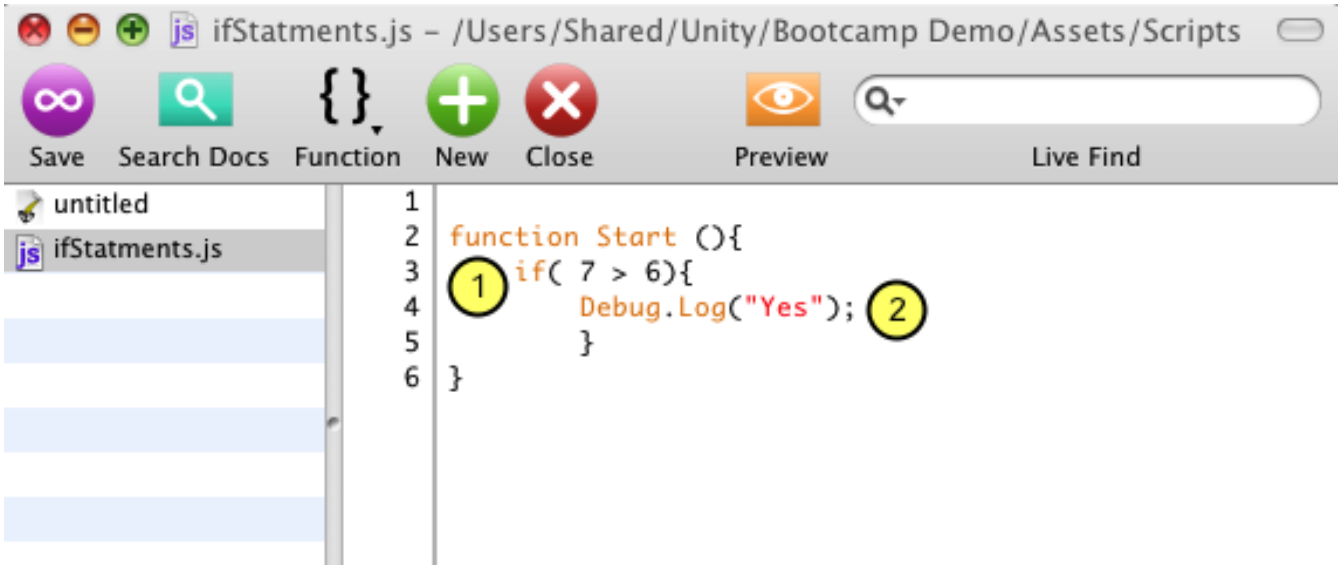


The player pressed a button. The door is locked. The character died. 7 > 6 (7 is greater than 6)

Gemeral statements, like these above, are either **true** or **false**. For example: The bear made big potty in the woods is either **true**, the bear did make a big potty in the woods, or **false**, the bear didn't make big potty in the woods.

Where does **if** enter the picture? Let's look at two examples written as it might appear in a script:

Conditions



I've created a new UnityScript (JavaScript), ifStatements.js, and attached it to an empty GameObject in the scene.

1. Write **if**, then place the statement that will be **true** or **false** in round brackets, followed by the code we want to execute in curly brackets.

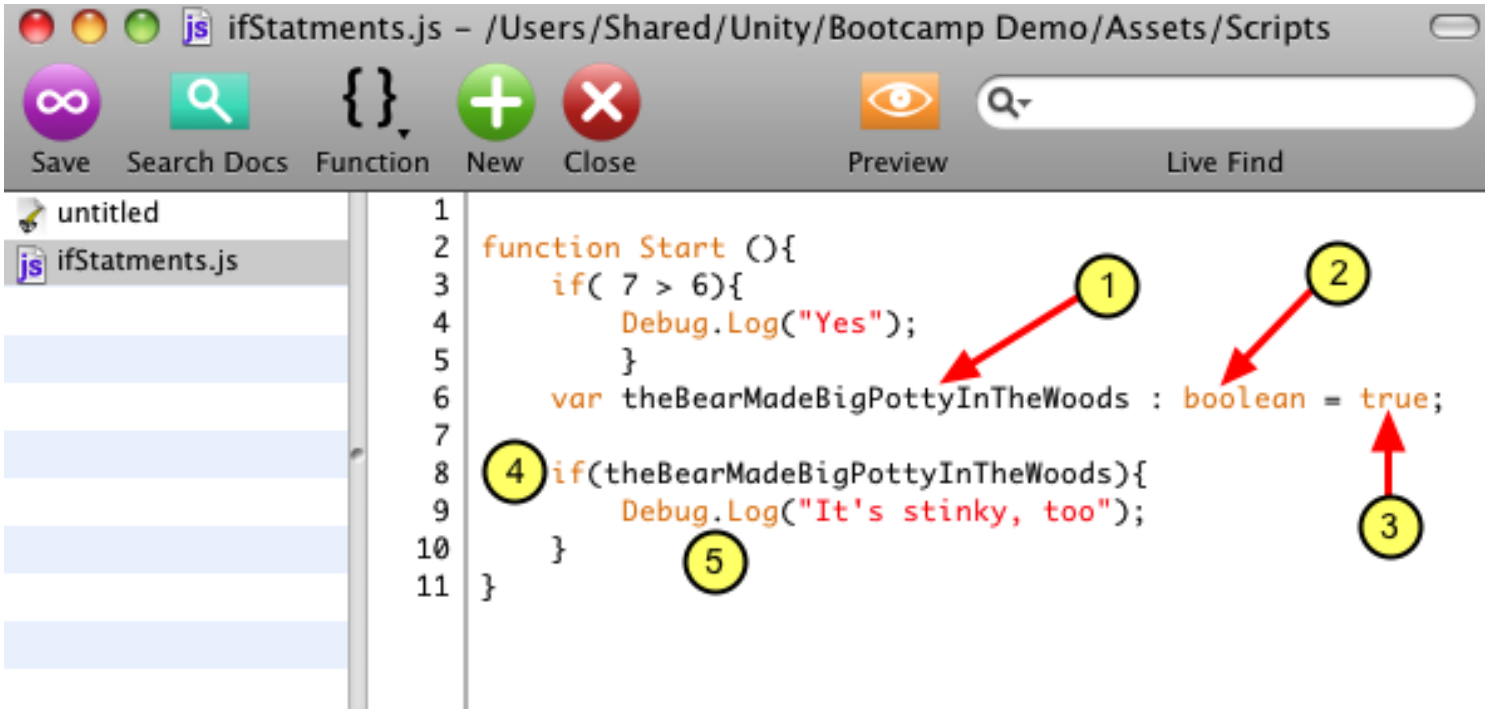
The statements we put in the round brackets, that we are checking to be **true** or **false**, are called **conditions**. So an **if** statement says:

If my **conditions** are met, then execute the code that follows

In this case, 7 > 6 is always **true**, so "Yes" is displayed in the console.

Note:
Unity will give you a warning that this is always **true**. I don't care since it's just for demonstration purposes, and it works. The reason it's always **true** is because 7 & 6 are **constants**, not variables. They can never change. 7 will always be equal to 7.

Checking for "true"



Then how do I check a condition such as The bear made big potty in the woods?

It has to be written in a way that can be tested as being **true** or **false**. We will use **boolean** values for now (link).

I'm using a **boolean** type of variable because a **boolean** can only be **true** or **false**.

1. So let's create a variable with a name of **theBearMadeBigPottyInTheWoods**. Of course I could

have used a much shorter name, but I bet you remember this one better.

2. I've specified the type as **boolean**.

3. I assigned this variable the value of **true**.

4. I created an **if** statement with the variable between the round brackets.

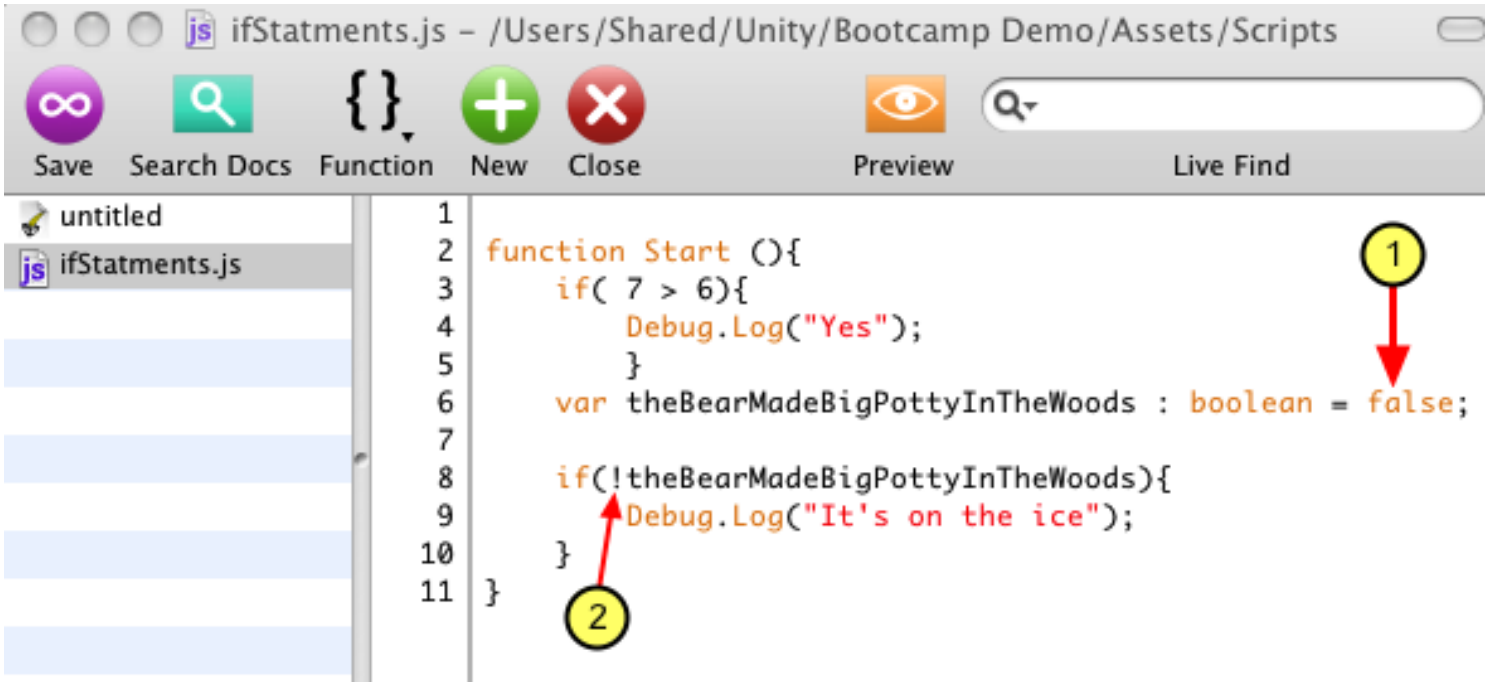
Of course, in this example, we already know the **condition** will be met since the variable was assigned the value **true.**

5. As a result "It's stinky, too" will appear in the console when I press play.

As a refresher, a variable name is just a substitute name for the value it actually stores, therefore, wherever the variable **theBearMadeBigPottyInTheWoods** is used in the script, it's replaced with the value **true**. That means the **if** statement conditions are met, so it executes the code to call Debug.Log().

Had **theBearMadeBigPottyInTheWoods** been asigned **false**, then the condition would not be met, so the Debug.Log() would not have been called.

"Not" Checking?



Not checking? Yup. This is going to seem like cheating, but in this particular example we are going to turn the **false** value into **true** so that the **if** statement will execute.

1. Notice I changed the variable assignment to **false**. I did this because the bear is a polar bear, and

they don't do their potty in the woods.

Through a little bit of reverse logic, the **if** statement is still going to execute.

2. Look very closely at the beginning of the **theBearMadeBigPottyInTheWoods** variable between the round brackets. There's an **exclamation mark** in front of the variable.

What the heck does that do?

It means **Not**.

Try to stay with me here. It's saying "**Not** the value in the variable **theBearMadeBigPottyInTheWoods**."

The value stored in **theBearMadeBigPottyInTheWoods** is **false**.

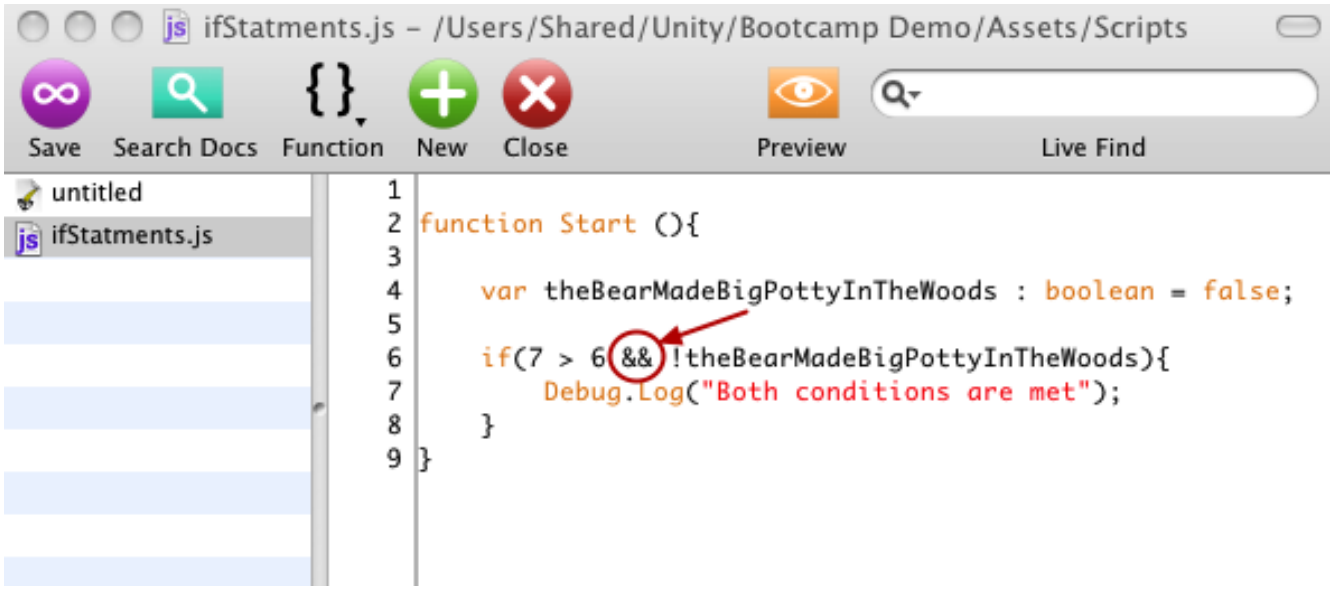
Therefore, it's saying **Not theBearMadeBigPottyInTheWoods**, or **Not false**.

Not false is the same as saying **true**.

So, this meets the condition of the **if** statement and the output to the console will be "It's on the ice".

Variables don't have to be the boolean type to work

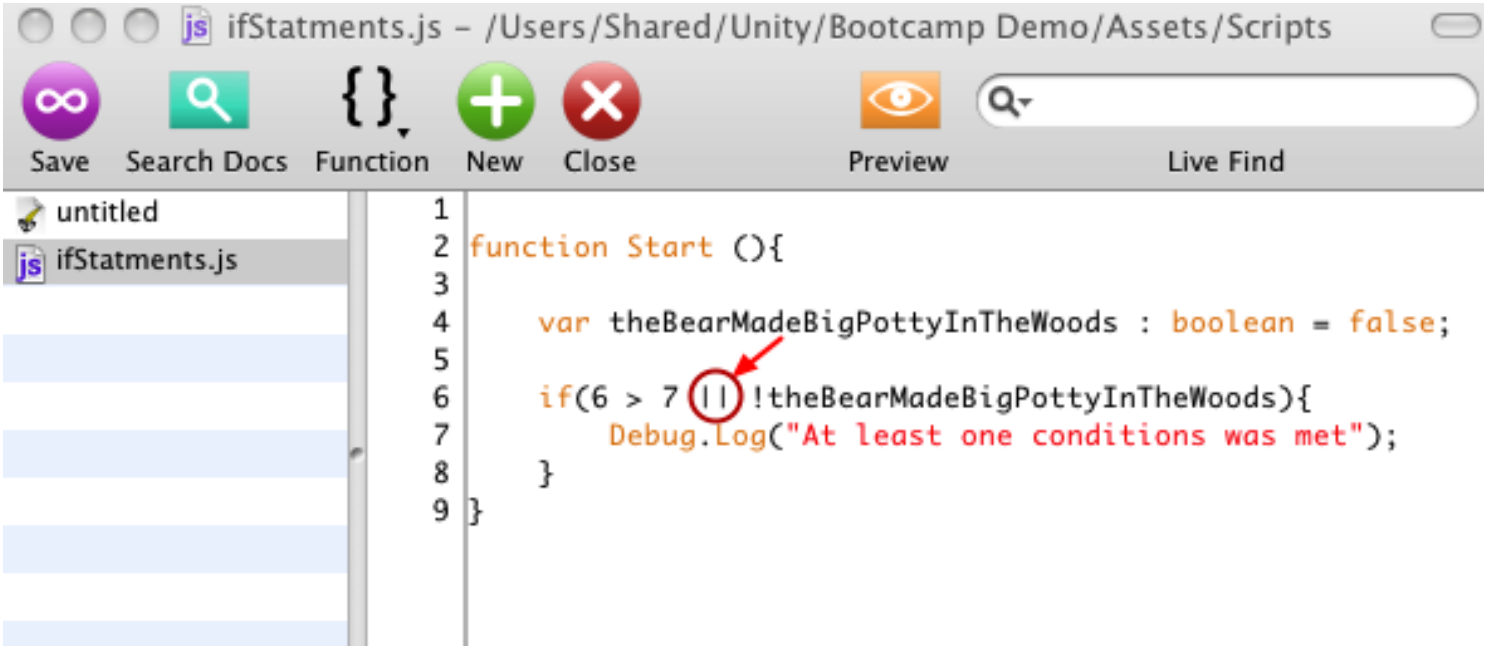
How about multiple condition?



You may have many conditions you want to have met before an **if** statement will execute. You can combine several conditions being checked by using two ampersand symbols, **&&**, which means **AND**.

This means that all conditions have to be **true** to make the **if** statement execute.

Multi conditions, but only one has to be true

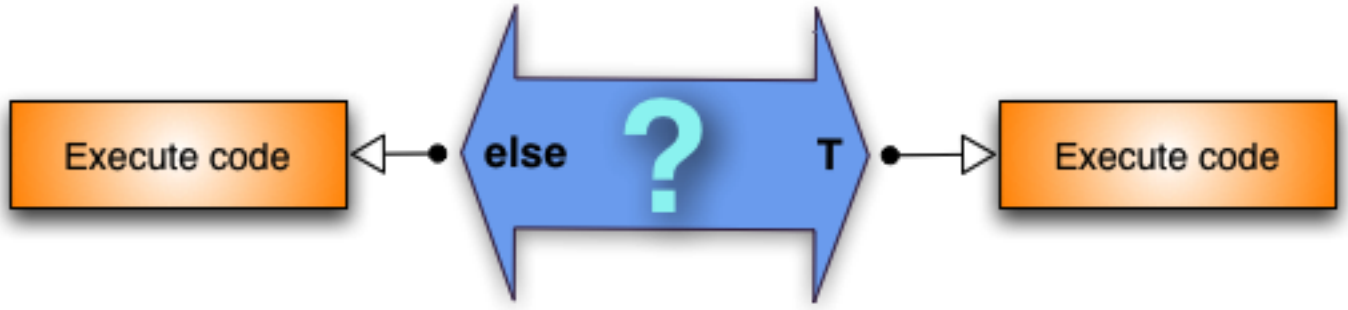


Here I've made one condition **false**, 6 > 7. I also have two verticle line characters, **||**, between the conditions being checked. This means **OR**.

This means that **only one** condition has to be **true** to make the **if** statement execute, 6 > 7 **OR** !theBearMadeBigPottyInTheWoods.

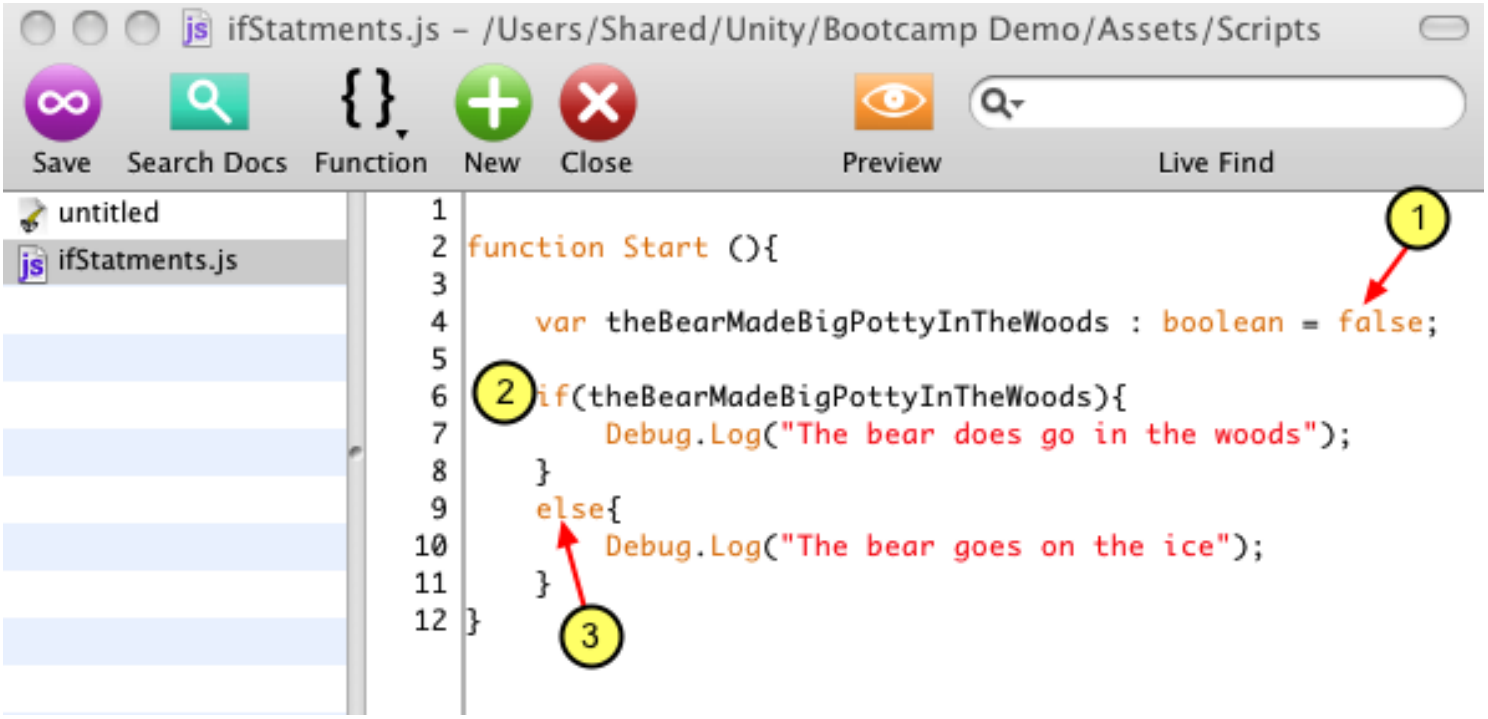
In this case 6 > 7 is **false**, but !theBearMadeBigPottyInTheWoods is **true**.

Optional code to execute



The **if** statements I've shown only executed code if **conditions** were met. Now we see that there is an option to execute some alternate code when **conditions** are not met.

if-else Statement



1. The variable is assigned **false**

2. This is the usual part of the **if** statement

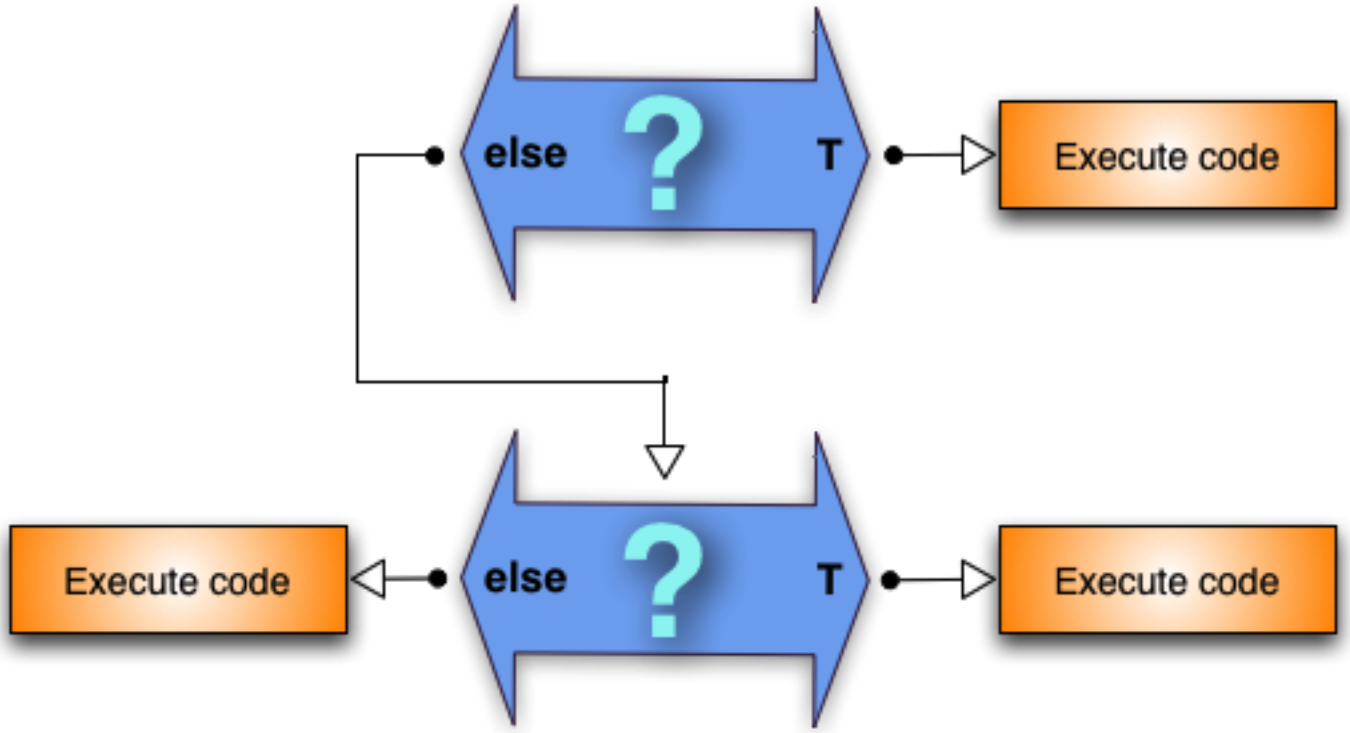
3. **else** is added along with its own code to execute between curly brackets.

This is very simple. Since the variable value is **false**, the **if** statement conditions are not met, so the code in **else** is executed instead.

What this **if** statements says is:

> **If** my **conditions** are met, then execute the code that follows, **else** execute the alternative code

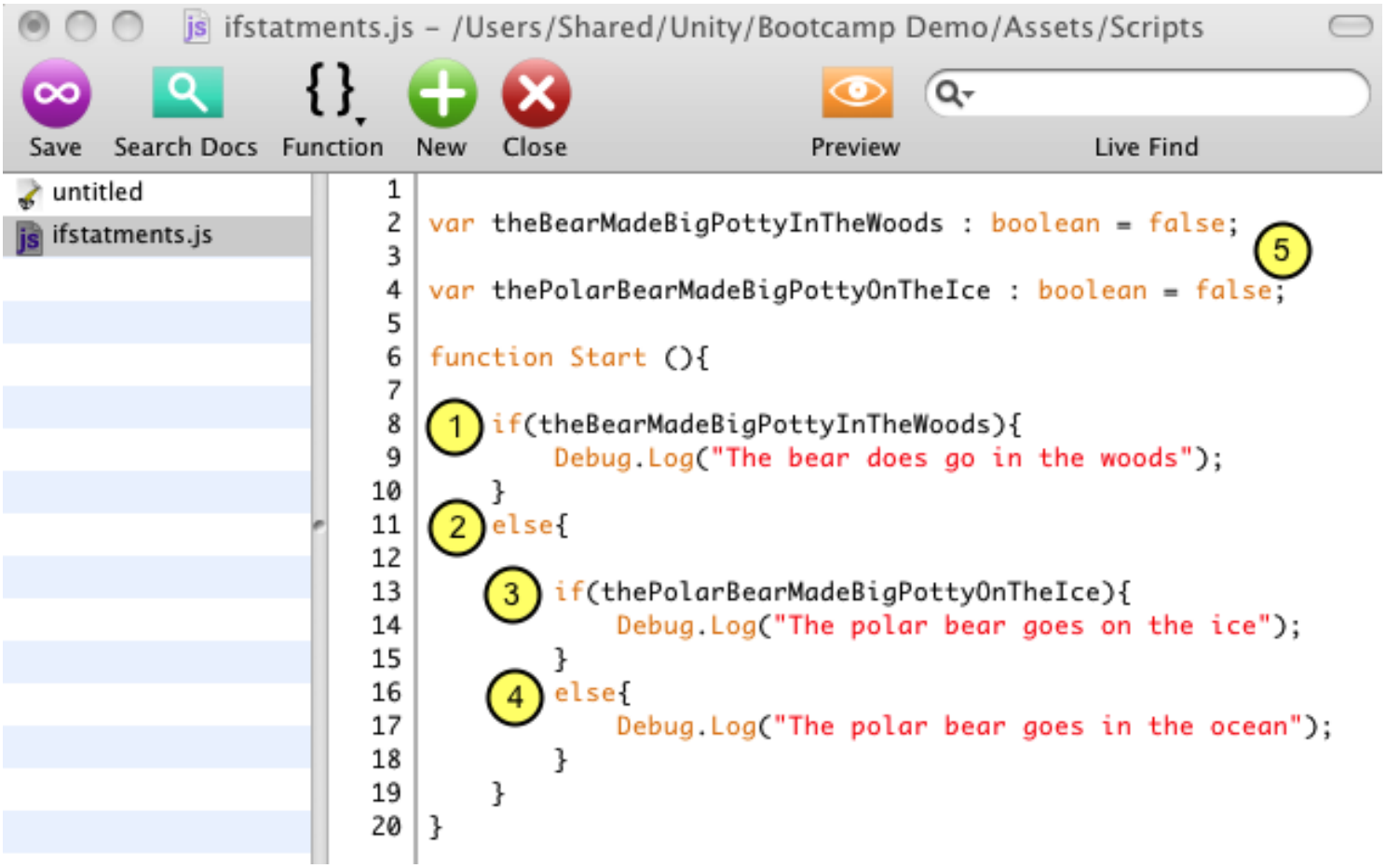
What about more than two options?



When choices need to be made, you will discover there are many situations that can have more than just one or two possible outcomes. That's easy to do by simply inserting another **if-else** statement in the else code of the previous **if-else** statement.

This is called "nesting."

if-else-if-else...



You can nest **if-else** statements as deep as you need, and the **conditions** for each **if** statement can be whatever you need.

1. First **if**

2. First **else**

3. Second **if**

Notice this second **if** statement is in the code for the first **else** (#2)

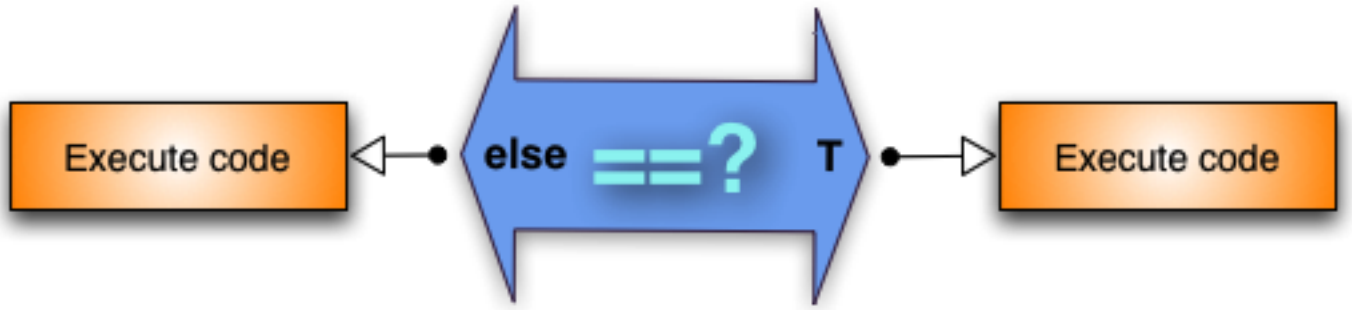
4. Second **else**

In this example, the output to the console will be "The polar bear goes in the ocean" because the **conditions** in both if statements are not met, they're both **false**. Why? Because both variables, **theBearMadeBigPottyInTheWoods** and **thePolarBearMadeBigPottyOnTheIce**, are assigned values of **false**.

Note:

5. I moved the variables out of the Start() function. I did this so they would become **public** variables and show up in the Unity Inspector. This will allow you to change their values, **true** and **false**, to see the different outputs to the console without having to change the script.

Variables other than the boolean type

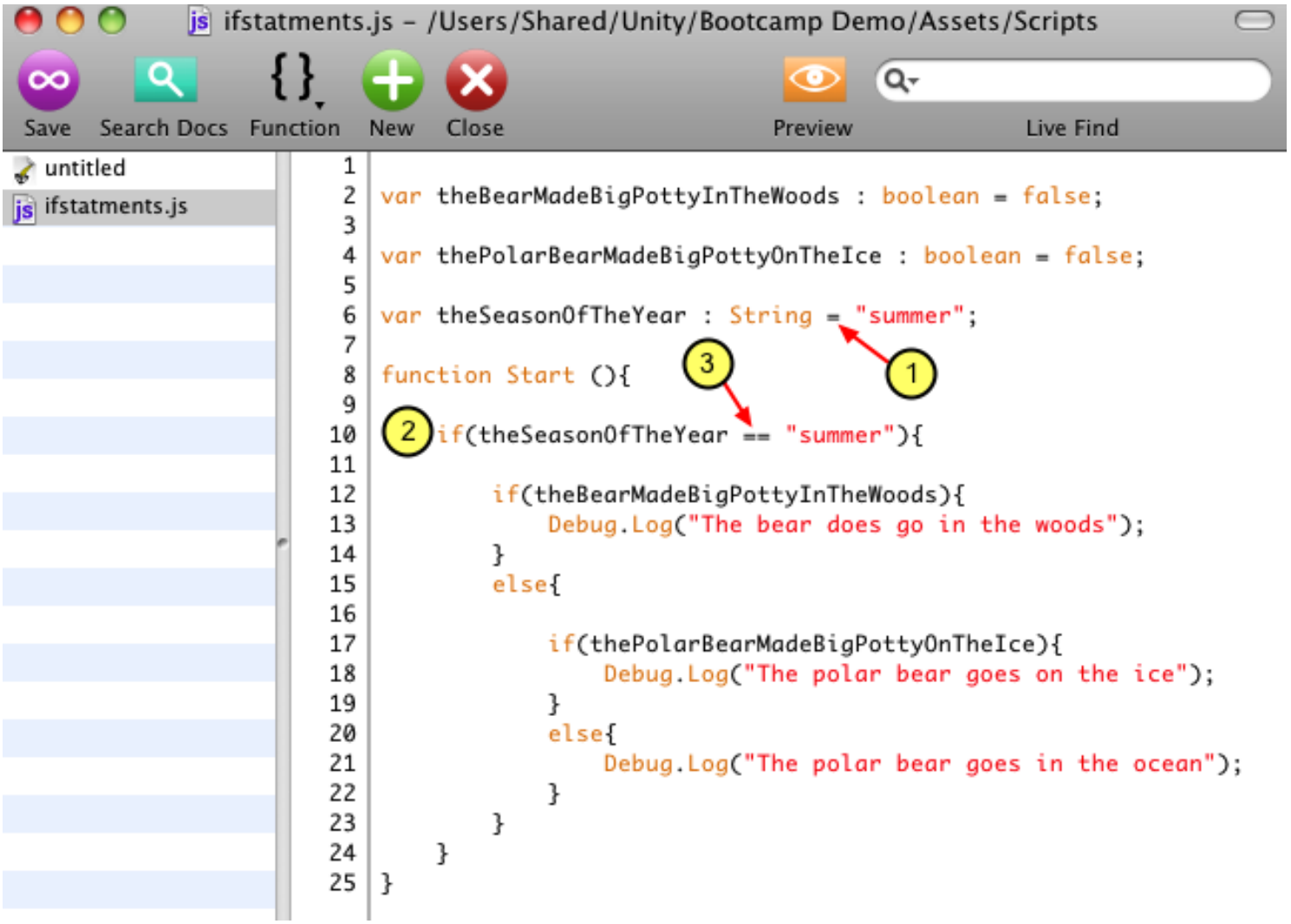


All the examples I've shown use **boolean** variables, the assigned value could only be **true** or **false.** This made it pretty easy to make sure the if statement was checking for the condition of **true** or **false**.

Any variable type can be used in an **if** statement, all we have to do is phrase the condition statement so that the answer will be **true** or **false**. How do we do this?

Have you ever asked a person a question that could be answered with yes or no? Of course you have. That's what we need to do here.

"= = "is your equal sign



1. I created another variable called **theSeasonOfTheYear** and assigned it a String of "summer."

2. I created a new **if** statement using **theSeasonOfTheYear**

3. Notice I used two equal signs. That how you write "equals" in scripting code.

So exactly how does **theSeasonOfTheYear** == "summer" made the conditiom **true** for the **if** statement? Let's start with an easier example:

6 == 6

That says 6 equals 6. Would you say thats a true statement? Sure. How about:

"summer" == "summer"

I'd say that's true as well, wouldn't you? Ok, so what's the value stored in the variable

theSeasonOfTheYear? According to #1 in the code I wrote it's "summer", therefore

theSeasonOfTheYear == "summer"

is also a true statement, which is exactly what's needed for an **if** statement to execute its code.

So any type of variable can be used as part of a condition as long as the values on both sides of the equal sign (==) are, infact, equal to each other.

Note:

theSeasonOfTheYear will also appear in the Inspector. If you change the value to anything besides "summer" the **if** statement conditions won't be met. As a result, none of the code will execute, and noting will show in the console. Why?

Because #2 is only an **if** statement, not an **if-else** statement. There is no alternative code to execute.

Another Note:

The whole example is pretty simple code, but as you can see, when you start nesting just a few **if** statements it can become visually complex looking, even when it's very simple. I haven't used any comments in the script files so they wouldn't clutter my examples. When you start creating your own scripts, use comments generously so theat when you look at them later you don't sit there scratching you head wondering what it was you were trying to do.

Dot Syntax: Component Communication

Communicating outside of your script

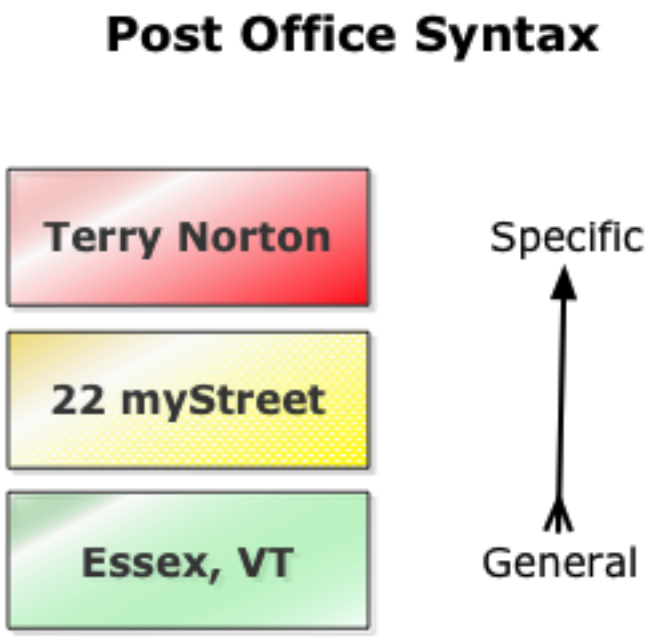
Scripts can do many things, and the real power comes when scripts perform functions when GameObjects interact with each other and the user. That means there has to be some way for GameObjects to communicate. This is where **Dot Syntax** comes in.

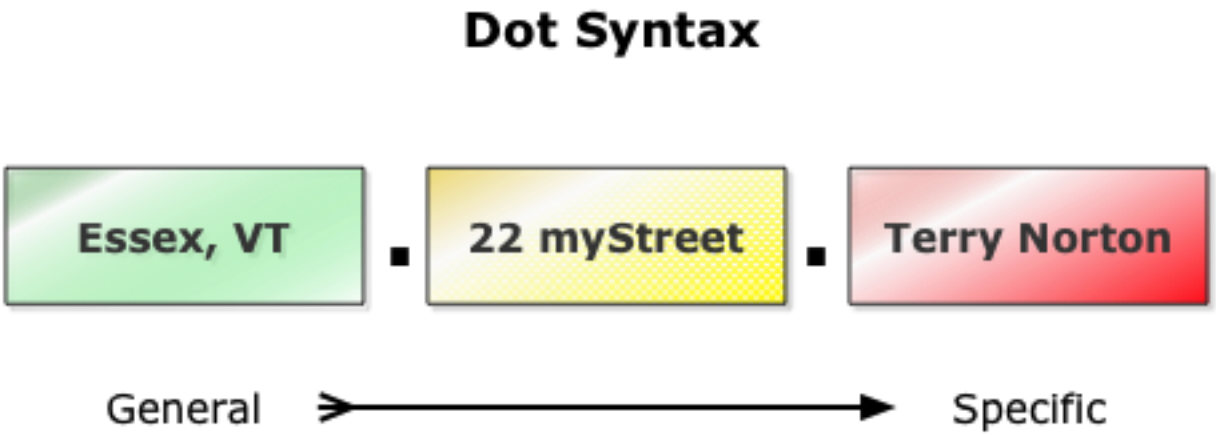
Dot Syntax is an address

(Click here) to review the basic concept I presented earlier

Simplicity of Dot Syntax

Terry Norton
22 myStreet
Essex, VT



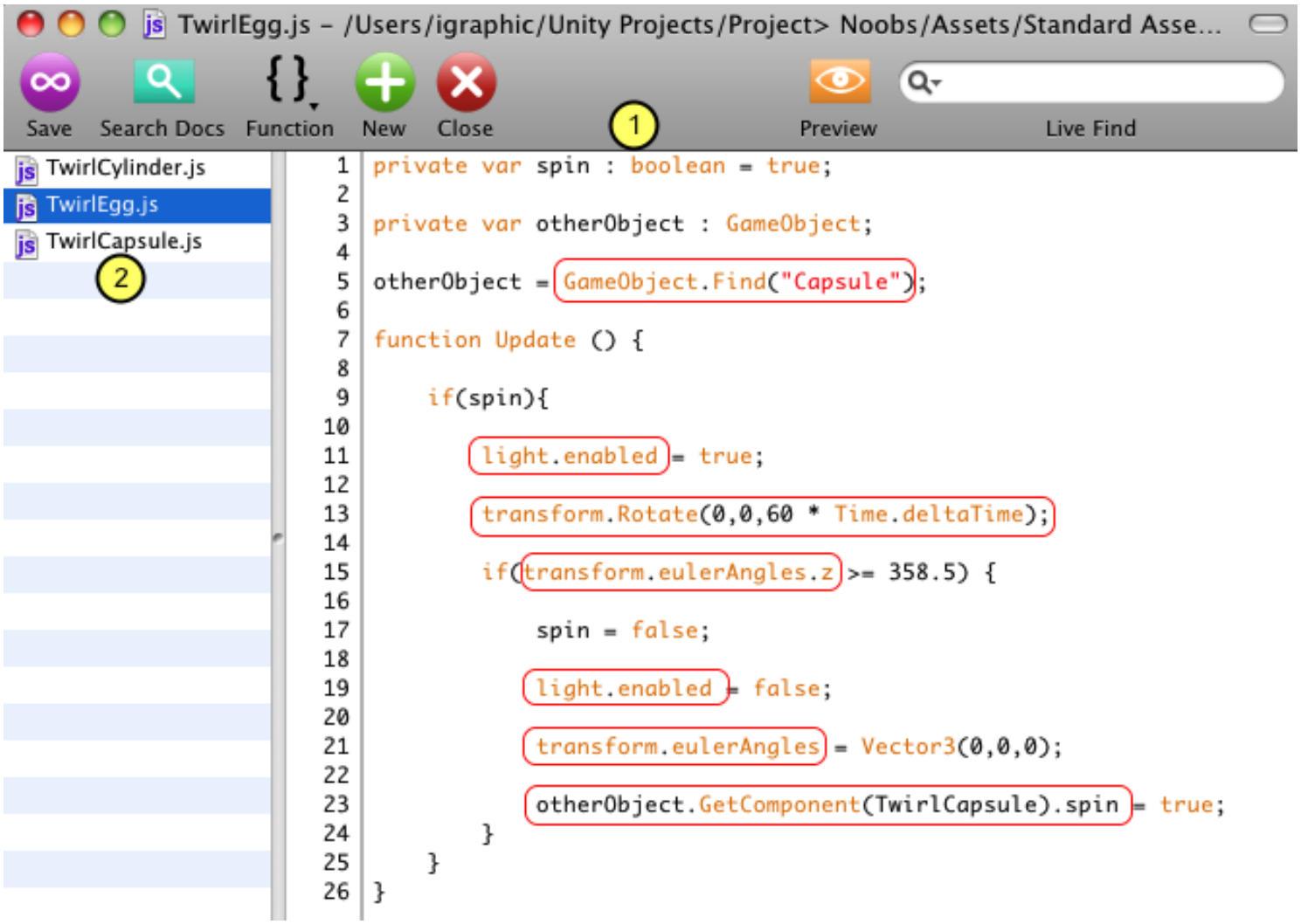


You are very accustomed to seeing a person's address written in the format the postal service requires. UnityScript also has a required format to locate variables and functions in other components, whether they are scripts you create or components provided by Unity.

Above is a comparison of the two formats. When you see or use **Dot Syntax**, you are using a variablle

or function located outside your current script.

Dot Syntax in a script



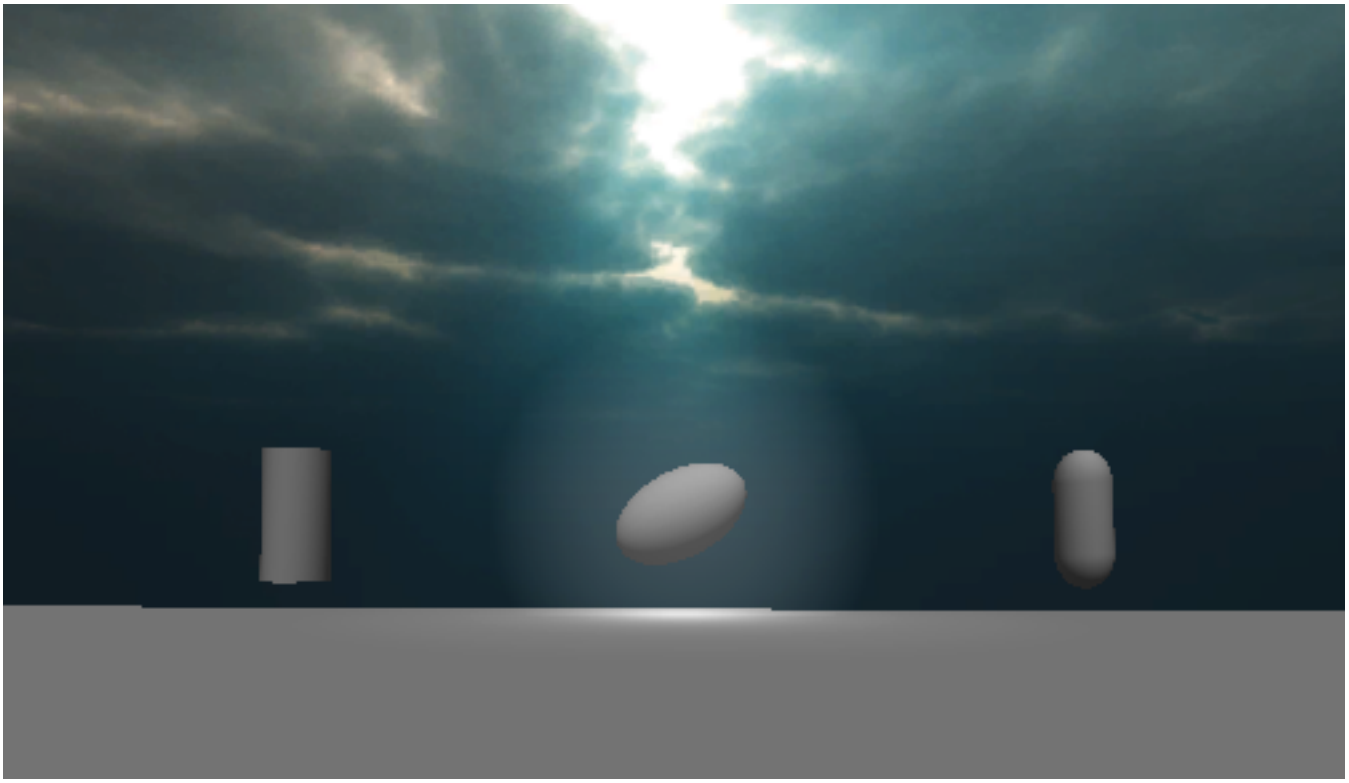
1. Here's one of three scripts that rotates a GameObject one revolution. When done, it allows another GameObject to rotate by changing the value of a variable in the other GameObject's script.

2. This is a list of the three scripts for the three different GameObjects: an egg, a capsule, and a Cylinder. These are just arbitrary shapes I used, just for fun.

As you can see, a majority of this little script makes use of **Dot Syntax** to read variables, change variable, and call functions. The three scripts are very similar, but there are a few differences I want to show you.

I will show you how to make use of Unity's Reference so you understand where this stuff comes from.

Sample view of project



This is a snapshot of the GameObjects in action in the Game window. The egg is in the process of rotating. Each GameObject takes a turn, then the process repeats. As each item rotates it's surrounded by a halo of light.

If viewing this lesson on the Web, click the image to play the video. If viewing this lesson as a PDF, click the image to play the video in a YouTube window.

The Egg script, lines 1 - 5

I will explain each line, why it's used and how it works.

Line 1: setting the controlling variable

TwirlEgg.js – /Users/igraphic/Unity Projects/Project> Noobs/...

```
1  private var spin : boolean = true;
2
3  private var otherObject : GameObject;
4
5  otherObject = GameObject.Find("Capsule");
6
7  function Update () {
8
9      if(spin){
10
11         light.enabled = true;
12
13         transform.Rotate(0,0,60 * Time.deltaTime);
14
15         if(transform.eulerAngles.z >= 358.5) {
16
17             spin = false;
18
19             light.enabled = false;
20
21             transform.eulerAngles = Vector3(0,0,0);
22
23             otherObject.GetComponent(TwirlCapsule).spin = true;
24         }
25     }
26 }
```

This is a private variable called **spin**. It's a boolean which means it's value can be either **true** or **false**.

spin is used to control whether the Egg is allowed to rotate. When **true** it will rotate, when **false** it won't.

spin is assigned the value of **true** because the Egg will be the first GameObject to rotate when Play is pressed.

It's private so it doesn't appear in the Inspector. You can remove private if you want to play with the checkbox in the Inspector.

Line 3: a place to store the next GameObject

TwirlEgg.js – /Users/igraphic/Unity Projects/Project> Noobs/...

```
1  private var spin : boolean = true;
2
3  private var otherObject : GameObject;
4
5  otherObject = GameObject.Find("Capsule");
6
7  function Update () {
8
9      if(spin){
10
11         light.enabled = true;
12
13         transform.Rotate(0,0,60 * Time.deltaTime);
14
15         if(transform.eulerAngles.z >= 358.5) {
16
17             spin = false;
18
19             light.enabled = false;
20
21             transform.eulerAngles = Vector3(0,0,0);
22
23             otherObject.GetComponent(TwirlCapsule).spin = true;
24         }
25     }
26 }
```

This is a private variable I named **otherObject**. It's a GameObject type of variable which means it's value is the name of another **GameObject**.

otherObject is used to store the next GameOject that will rotate after the Egg is finished.

It's private so it doesn't appear in the Inspector because I don't want its value to be allowed to be changed in the Inspector. In the other two scripts this variable isn't private. I will explain later.

Line 5: specifying the next GameObject to rotate

TwirlEgg.js – /Users/igraphic/Unity Projects/Project> Noobs/...

```
1  private var spin : boolean = true;
2
3  private var otherObject : GameObject;
4
5  otherObject = GameObject.Find("Capsule");
6
7  function Update () {
8
9      if(spin){
10
11         light.enabled = true;
12
13         transform.Rotate(0,0,60 * Time.deltaTime);
14
15         if(transform.eulerAngles.z >= 358.5) {
16
17             spin = false;
18
19             light.enabled = false;
20
21             transform.eulerAngles = Vector3(0,0,0);
22
23             otherObject.GetComponent(TwirlCapsule).spin = true;
24         }
25     }
26 }
```

The **otherObject** varaible is being assigned the **Capsule** GameObject.

Here is our first use of Dot Syntax: **GameObject.Find()**

(Click here) to go to the Unity website for this function.

GameObject.Find()

So what information do we have here? First of all, looking at the description we see that it finds a game object and returns it. That's exactly what is needed in line 5 of our TwirlEgg script. We need to specify the GameObject that will start rotating after the Egg is finished.

So why does the Find() function have to be written with **GameOject** and a **dot** in front of it?

The Find() function isn't declared in our TwirlEgg script, so we have to locate it in order to use it. It's declared automatically for you somewhere else in Unity's code. Remember, **Dot Syntax** is an address format, it tells us where the Find() function is located.

Well, this documentation tells us it's located in the **GameObject** class. It's the same thing as saying the Find() function is declared in the **GameObject** script. Is it a script? I don't know, nor do I care. All I need to know is the class (script) it's located in, and this document tells me that information.

And of course, how do I even know to begin looking in the **GameObject** class (script) for this Find() functions? For the simple reason that further down in the TwirlEgg script, line 23, I need to set a variable in a another script (Component) in another **GameObject**. That means I have to look for the needed function in the **GameObject** class (script).

OK then, how did I know to use the Find() function? By reading the list of things available in the **GameObject** class (script).

GameObject variables & functions I can use

Class Functions	
CreatePrimitive	Creates a game object with a primitive mesh renderer and appropriate collider.
FindWithTag	Returns one active GameObject tagged tag. Returns null if no GameObject was found.
FindGameObjectsWithTag	Returns a list of active GameObjects tagged tag. Returns null if no GameObject was found.
Find	Finds a game object by name and returns it.

(click here) to go to the GameObject page at Unity. In about the middle of the page you will see the Class Functions. The Find() is there and tells me what it does. All I did was go to this web page and start reading until I found what I needed.

How do I use the GameObject.Find() function?

First thing I see is that it's a **static** function. That means it's a global function that can be called from any script I create anywhere.

Then the name of the function is **Find()**. Inside the round brackets we will write the name of the GameObject we want to find. The GameObject we want is the **Capsule**.

Also notice the name is a **String** type, so we have to surround the name with double-quotes.

Finally, there is a colon followed by GameObject. This is how UnityScript specifies what the function will return to the code where Find() was called. That's the whole point of calling GameObject.Find(), to find the **Capsule** GameObject and put into the variable **otherObject**.

More info about GameObject.Find() in the documentation

For performance reasons it is recommended to not use this function every frame Instead cache the result in a member variable at startup or use GameObject.FindWithTag.

JavaScript

```
var hand : GameObject;
// This will return the game object named Hand in the scene.
hand = GameObject.Find("Hand");
```

It suggest that GameObject.Find() not be placed in the Update) function.

Then there is an example shown. This is exactly how our code is written, except for the variable name and the GameObject name.

The Egg script, lines 7 - 11, The Update() function

The Update() function is where the action happens

TwirlEgg.js – /Users/igraphic/Unity Projects/Project> Noobs/...

```
1  private var spin : boolean = true;
2
3  private var otherObject : GameObject;
4
5  otherObject = GameObject.Find("Capsule");
6
7  function Update () {
8
9      if(spin){
10
11         light.enabled = true;
12
13         transform.Rotate(0,0,60 * Time.deltaTime);
14
15         if(transform.eulerAngles.z >= 358.5) {
16
17             spin = false;
18
19             light.enabled = false;
20
21             transform.eulerAngles = Vector3(0,0,0);
22
23             otherObject.GetComponent(TwirlCapsule).spin = true;
24         }
25     }
26 }
```

Line 7:

The code enclosed in the Update() function is executed automatically by Unity every frame when Play is pressed to start.

Line 9:

This **if** statement is checking to see if the variable **spin** is true or false. It was declared back in line 1 as **true**, so that means the **if** statement code is going to execute every frame, over and over, until **spin** is set to **false**, which will will be later in line 19.

The reason I declared **spin** as **true** in line 1 is because one of the three GameObject has to the first one rotate. I all three are set as **false,** then nothing would happen when Play is pressed.

spin is like a switch. When it's on (true), the Egg will rotate. When off (false), the Egg stops.

Line 11:

Here is another statement using **Dot Syntax**.

Line 11: turns on the Light component

Light Inherits from Behaviour

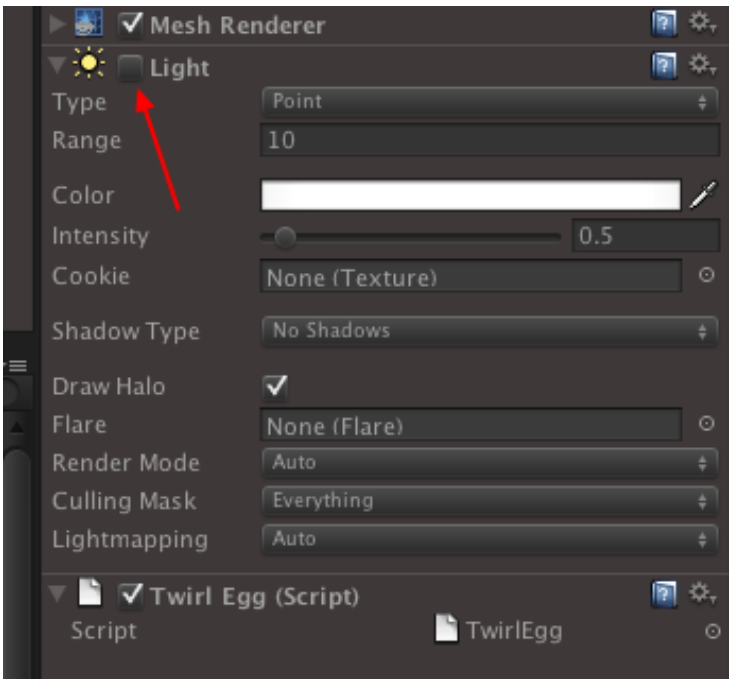
Script interface for light components.

Use this to control all aspects of Unity's lights. The properties are an exact match for the values shown in the Inspector.

Usually lights are just created in the editor but sometimes you want to create a light from a script:

(Click here) to view the Light Class at Unity. About half way down the page are the Inherited Variables. The **enabled** variable is the one being used to simply enable or disable the whole Light component in the TwirlEgg script.

The Light checkbox



This is the checkbox that is being enabled by line 11. As the games plays, when the Egg rotates and stops, the checkbox will be checked, then unchecked.

The Egg script, line 13, Rotating the Egg

Rotating the Egg using the Rotate() function

TwirlEgg.js – /Users/igraphic/Unity Projects/Project> Noobs/...

```
1  private var spin : boolean = true;
2
3  private var otherObject : GameObject;
4
5  otherObject = GameObject.Find("Capsule");
6
7  function Update () {
8
9      if(spin){
10
11         light.enabled = true;
12
13         transform.Rotate(0,0,60 * Time.deltaTime);
14
15         if(transform.eulerAngles.z >= 358.5) {
16
17             spin = false;
18
19             light.enabled = false;
20
21             transform.eulerAngles = Vector3(0,0,0);
22
23             otherObject.GetComponent(TwirlCapsule).spin = true;
24         }
25     }
26 }
```

Once again we are making use of **Dot Syntax**, this time to loacte a function in the **Transform** Class.

We have to use **Dot Syntax** to call the Rotate() function because it isn't declared in the TwirlEgg script. It's declared somewhere else in Unity's code.

Transform.Rotate

function **Rotate (eulerAngles :** Vector3**, relativeTo :** Space **= Space.Self) :** void

Description
Applies a rotation of eulerAngles.z degrees around the z axis, eulerAngles.x degrees around the x axis, and eulerAngles.y degrees around the y axis (in that order).

If relativeTo is left out or set to Space.Self the rotation is applied around the transform's local axes. (The x, y and z axes shown when selecting the object inside the Scene View.) If relativeTo is Space.World the rotation is applied around the world x, y, z axes.

JavaScript

```
function Update() {
    // Slowly rotate the object around its X axis at 1 degree/second.
    transform.Rotate(Vector3.right * Time.deltaTime);

    // ... at the same time as spinning relative to the global
    // Y axis at the same speed.
    transform.Rotate(Vector3.up * Time.deltaTime, Space.World);
}
```

1

function **Rotate (xAngle :** float**, yAngle :** float**, zAngle :** float**, relativeTo :** Space **= Space.Self) :** void

Description
Applies a rotation of zAngle degrees around the z axis, xAngle degrees around the x axis, and yAngle degrees around the y axis (in that order).

If relativeTo is left out or sot to Space.Self the rotation is applied around the transform's local axes. (The x, y and z axes shown when selecting the object inside the Scene View.) If relativeTo is Space.World the rotation is applied around the world x, y, z axes.

2

JavaScript

```
function Update() {
    // Slowly rotate the object around its X axis at 1 degree/second.
    transform.Rotate(Time.deltaTime, 0, 0);
```

Every GameObject has a Transform component. It's what determines where it is in the Scene, and how it's oriented. So we have access to the Rotate() function within the Egg. We just need to call it in the Transform component. Therefore, the Rotate() function has **transform** and a **dot** in front of it to locate it properly.

1. Shows the parameters for the Rotate() function.

2. An example showing how to write the function. This is very close to what I wanted to accomplish. On line 13 I'm specifying to rotate around the **Z** axis 60 degrees per second.

Essentially what **Time.deltaTime** means is "1 per second." So in line 13 I'm multiplying times 60, so I'm saying rotate 60 degrees/second. Therefore, the Egg takes 6 seconds to make one full rotation of 360 degrees.

The Egg script, line 15, Watching the Rotation

Line 15 monitors how much the Egg has rotated

TwirlEgg.js – /Users/igraphic/Unity Projects/Project> Noobs/...

```
1  private var spin : boolean = true;
2
3  private var otherObject : GameObject;
4
5  otherObject = GameObject.Find("Capsule");
6
7  function Update () {
8
9      if(spin){
10
11         light.enabled = true;
12
13         transform.Rotate(0,0,60 * Time.deltaTime);
14
15         if(transform.eulerAngles.z >= 358.5) {
16
17             spin = false;
18
19             light.enabled = false;
20
21             transform.eulerAngles = Vector3(0,0,0);
22
23             otherObject.GetComponent(TwirlCapsule).spin = true;
24         }
25     }
26 }
```

This is an **if** statement that checks how many degrees the Egg has rotated.

Of course the infomation that needs to be monitored is stored in the Transform component. As **eulerAngles**?? What are **eulerAngles**?

I had to look this up on Wikipedia. It says "The **Euler angles** are three angles introduced by Leonhard Euler to describe the orientation of a rigid body." Well now, isn't that just peachy-keen. Oh well, I looked at some examples in the Unity docs, and sure enough, it seems to do what I want, so I used it.

eulerAngles store the rotation angles

Transform.eulerAngles

var **eulerAngles** : Vector3

Description
The rotation as Euler angles in degrees.

(1)

The x, y, and z angles represent a rotation z degrees around the z axis, x degrees around the x axis, and y degrees around the y axis (in that order).

(2)

Only use this variable to read and set the angles to absolute values. Don't increment them, as it will fail when the angle exceeds 360 degrees. Use Transform.Rotate instead.

JavaScript

```
// Print the rotation around the global X Axis
print (transform.eulerAngles.x);
// Print the rotation around the global Y Axis
print (transform.eulerAngles.y);
// Print the rotation around the global Z Axis
print (transform.eulerAngles.z);

// Assign an absolute rotation using eulerAngles
var yRotation : float = 5.0;
function Update () {
    yRotation += Input.GetAxis("Horizontal");
    transform.eulerAngles = Vector3(10, yRotation, 0);

}
```

(3)

Do not set one of the eulerAngles axis separately (eg. eulerAngles.x = 10;) since this will lead to drift and undesired rotations. When setting them to a new value set them all at once as shown above. Unity will convert the angles to and from the rotation stored in Transform.rotation

1. Just what I need to know, the degrees of rotation around the **z** axis.

2. Yup, that's what I want to do, read the angle.

3. This example isn't exactly what I'm trying to do in the **if** statement, but if the value can be printed, then I also know I can test if the value is equal to, or greater than 358.5 degrees.

Why 358.5 instead of 360? Because 360 is really zero, 359 would be a good test. However, I noticed that rotation didn't happen in whole numbers. There's some fraction part as well. I'm sure it's beause of using Time.deltaTime. So I picked a number that's almost 359 degrees. This isn't critical science, so 358.5 is close enough, and it works.

Once the rotation gets to 358.5 degrees or higher, the **if** statement becomes true and its block of code executes.

The Egg script, lines 17 - 23, Stop Rotating

Now that the Egg has made one rotation, it's time to stop it and tell the next GameObject to rotate.

Lines 17 & 19: Stop the spinning and turn off the halo

TwirlEgg.js – /Users/igraphic/Unity Projects/Project> Noobs/...

```
1  private var spin : boolean = true;
2
3  private var otherObject : GameObject;
4
5  otherObject = GameObject.Find("Capsule");
6
7  function Update () {
8
9      if(spin){
10
11         light.enabled = true;
12
13         transform.Rotate(0,0,60 * Time.deltaTime);
14
15         if(transform.eulerAngles.z >= 358.5) {
16
17             spin = false;
18
19             light.enabled = false;
20
21             transform.eulerAngles = Vector3(0,0,0);
22
23             otherObject.GetComponent(TwirlCapsule).spin = true;
24         }
25     }
26 }
```

Line 17:

Back in line 9 the variable **spin** was **true**. This allowed the Egg to rotate and the halo light was turned on. Now **spin** is being set to **false.**

Line 19:

Back in line 11 the halo light was turned on. Now it's been set to **false** to turn it off.

Line 21

TwirlEgg.js – /Users/igraphic/Unity Projects/Project> Noobs/...

```
1  private var spin : boolean = true;
2
3  private var otherObject : GameObject;
4
5  otherObject = GameObject.Find("Capsule");
6
7  function Update () {
8
9      if(spin){
10
11         light.enabled = true;
12
13         transform.Rotate(0,0,60 * Time.deltaTime);
14
15         if(transform.eulerAngles.z >= 358.5) {
16
17             spin = false;
18
19             light.enabled = false;
20
21             transform.eulerAngles = Vector3(0,0,0);
22
23             otherObject.GetComponent(TwirlCapsule).spin = true;
24         }
25     }
26 }
```

Here we are back with the **eulerAngles** again. All this is doing is making sure the **z** axis is reset to the starting point of zero degrees before it rotates again.

Resetting the rotation to zero

```
// Assign an absolute rotation using eulerAngles
var yRotation : float = 5.0;
function Update () {
    yRotation += Input.GetAxis("Horizontal");
    transform.eulerAngles = Vector3(10, yRotation, 0);

}
```

1

2

Do not set one of the eulerAngles axis separately (eg. eulerAngles.x = 10;) since this will lead to drift and undesired rotations. When setting them to a new value set them all at once as shown above. Unity will convert the angles to and from the rotation stored in Transform.rotation

1. This is the same document previously shown, I've cropped it to only show what I need. This is an example of setting specific angles for all three axes, x, y & z. I'm making sure they are all set to zero before thhe next rotation starts.

2. This is a warning not to set an individual axis. This is actually what I had done before reading this section. It worked fine, but I changed the code anyway to the preferred method now shown on line 21.

Line 23: telling the next GameObject to start rotating

TwirlEgg.js – /Users/igraphic/Unity Projects/Project> Noobs/...

```
1  private var spin : boolean = true;
2
3  private var otherObject : GameObject;
4
5  otherObject = GameObject.Find("Capsule");
6
7  function Update () {
8
9      if(spin){
10
11         light.enabled = true;
12
13         transform.Rotate(0,0,60 * Time.deltaTime);
14
15         if(transform.eulerAngles.z >= 358.5) {
16
17             spin = false;
18
19             light.enabled = false;
20
21             transform.eulerAngles = Vector3(0,0,0);
22
23             otherObject.GetComponent(TwirlCapsule).spin = true;
24         }
25     }
26 }
```

There's quite a bit happening in this one line of code.

1. The purpose is to assign the value **true**

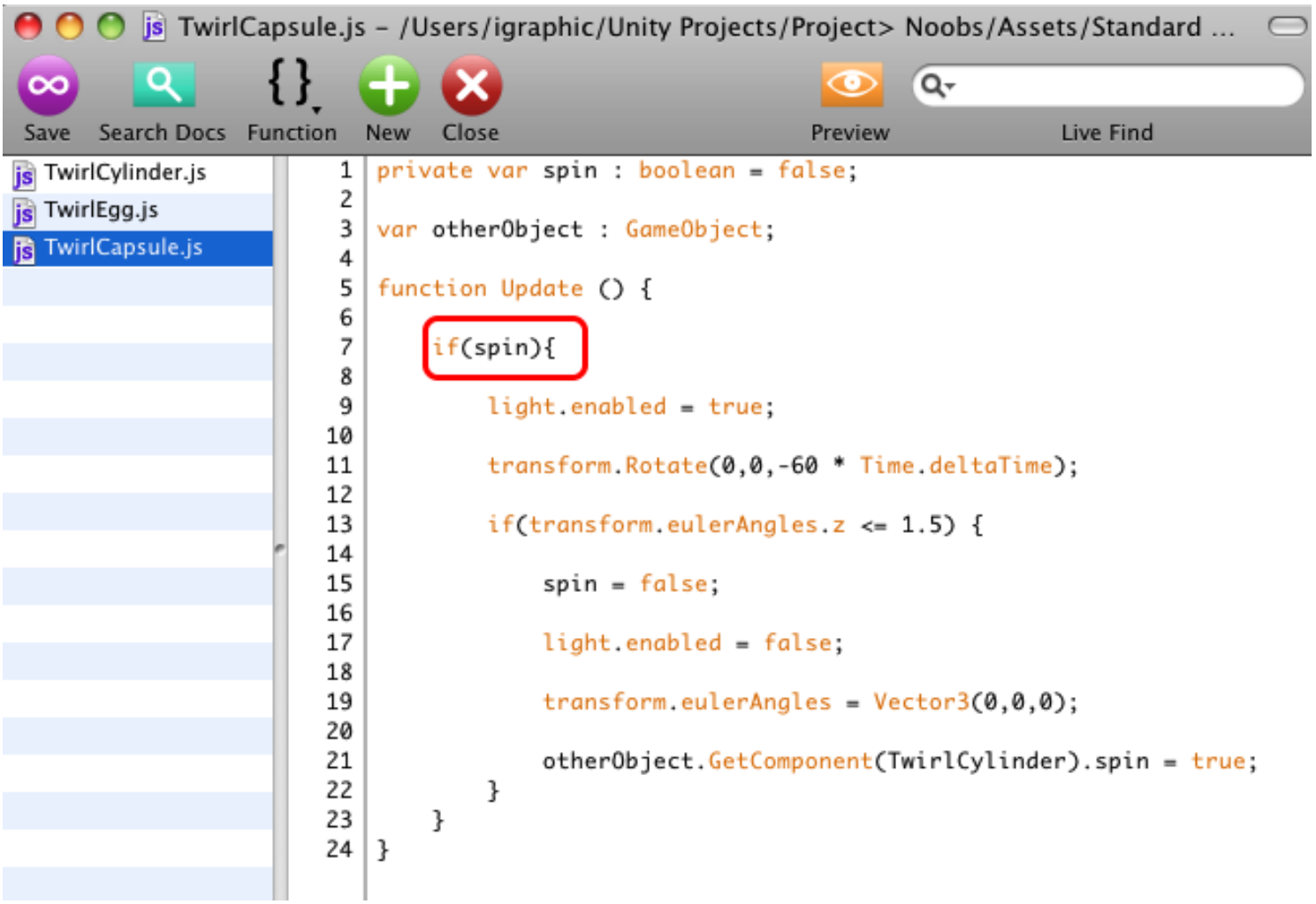
2. to the variable **spin**

3. that is located in the **TwirlCapsule** script

4. that's in the **Capsule** GameObject (see line 5)

Here again **Dot Syntax** is used to locate a variable in another script in another GameObject, then assign it a value.

spin in the TwirlCapsule script

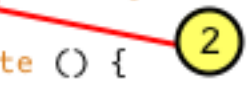


This is the variable **spin** that's being set to **true**. This script is very similar to the TwirEgg script we've been studying, and it should since it does the same thing

Back to the TwirlEgg script

TwirlEgg.js – /Users/igraphic/Unity Projects/Project> Noobs/...

```
1  private var spin : boolean = true;
2
3  private var otherObject : GameObject;
4
5  otherObject = GameObject.Find("Capsule");
6
7  function Update () {
8
9      if(spin){
10
11         light.enabled = true;
12
13         transform.Rotate(0,0,60 * Time.deltaTime);
14
15         if(transform.eulerAngles.z >= 358.5) {
16
17             spin = false;
18
19             light.enabled = false;
20
21             transform.eulerAngles = Vector3(0,0,0);
22
23             otherObject.GetComponent(TwirlCapsule).spin = true;
24         }
25     }
26 }
```



1. First thing we had to do in the **Dot Syntax** address was specify the GameObject. This variable, **otherObject,**

2. stores the GameObject called **Capsule**.

This means that anywhere the variable **otherObject** is used in this script, its stored value, **Capsule,** is used.

3. Next in the **Dot Syntax** address is the component that's in the Capsule, which is the **TwirlCapsule** script.

GameObject.GetComponent

function **GetComponent (type : Type) :** Component

Description
Returns the component of Type type if the game object has one attached, null if it doesn't. You can access both builtin components or scripts with this function.

GetComponent is the primary way of accessing other components. From javascript the type of a script is always the name of the script as seen in the project view.
Example:

JavaScript

```
function Start () {
    var curTransform : Transform;

    curTransform = gameObject.GetComponent(Transform);
    // This is equivalent to:
    curTransform = gameObject.transform;
}

function Update () {
    // To access public variables and functions
    // in another script attached to the same game object.
    // (ScriptName is the name of the javascript file)
    var other : ScriptName = gameObject.GetComponent(ScriptName);
    // Call the function DoSomething on the script
    other.DoSomething ();
    // set another variable in the other script instance
    other.someVariable = 5;
}
```

(Click here) to view GetComponent at Unity.

1. Here is an example that is close to what line 23 is doing. In this example they are simply retrieving the script and assigning it to a variable called **other**.

Our line 23 is not assigning the script to a varaible.

2. **GetComponent(TwirlCapsule)** is a function that returns a Component, which, in our case, is the TwirlCapsule script. What this means is that instead of assigning it to a variable, we are just using the Component returned from the GetComponent() function directly as part of the **Dot Syntax** address, a direct substitution instead of going through the process of assigning it to a varaible, then using the varaible. There's no point in creating extra steps.

Assigning using the Inspector

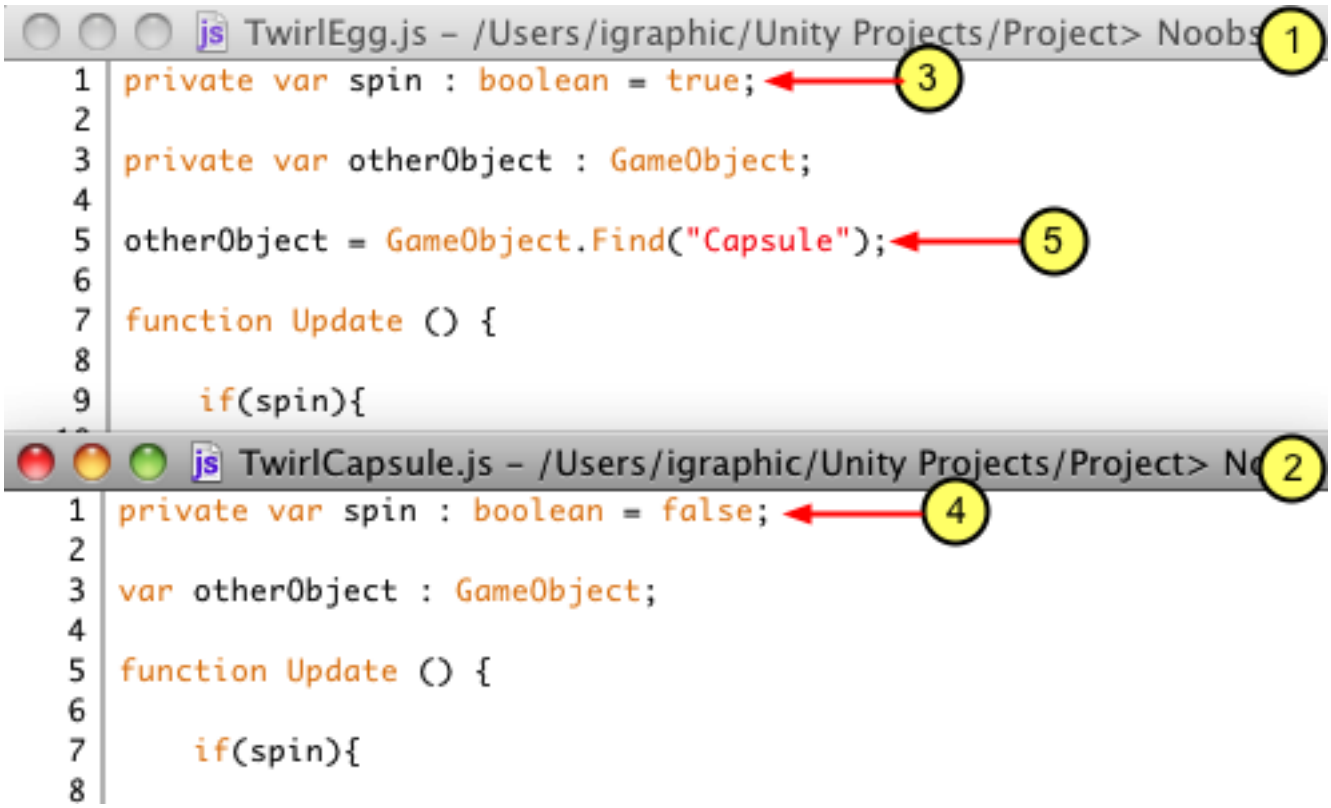
A neat Unity feature

In the TwirlEgg script, line 5, the Capsule GameObject was found using the GameObject.Find() function. The other two scripts, TwirlCapsule and TwirlCylinder, don't use the GameObject.Find() function. In fact, neither script has any code at all to find the other GameObjects that will be needed for the scripts to work properly. Yet, they still work because of the ability of using the Inspector to accomplish this task of assigning objects.

I purposely created these other two scripts to demonstrate this feature of drag-&-drop in the Inspector.

Note:
Ideally, I would have just created a single script and used it for all three GameObjects, and then used the Inspector to do the needed assignments. However, the purpose of this exercise is for teaching, so some differences were intentionally made.

The differences in the scripts



This images shows the beginning lines of two scripts:

1. TwirlEgg script

2. TwirlCapsule script

Next is the declaration of the **spin** variable:

3. In the Egg it's srt to **true**. One of the GameObjects has to be set to rotate when Play is pressed.

4. In the Capsule it's set to **false**

Now for assigning GameObjects to the **otherObject** variable:

5. The Capsule is assigned as the next GameObject that will rotate after the Egg is finished rotating.

Notice there is no assignment statement in the TwirlCapsule script.

private vs. public variable

TwirlEgg.js – /Users/igraphic/Unity Projects/Project> Noobs/...

```
1  private var spin : boolean = true;
2
3  private var otherObject : GameObject;
4
5  otherObject = GameObject.Find("Capsule");
6
7  function Update () {
8
9      if(spin){
```

TwirlCapsule.js – /Users/igraphic/Unity Projects/Project> No...

```
1  private var spin : boolean = false;
2
3  var otherObject : GameObject;
4
5  function Update () {
6
7      if(spin){
8
```

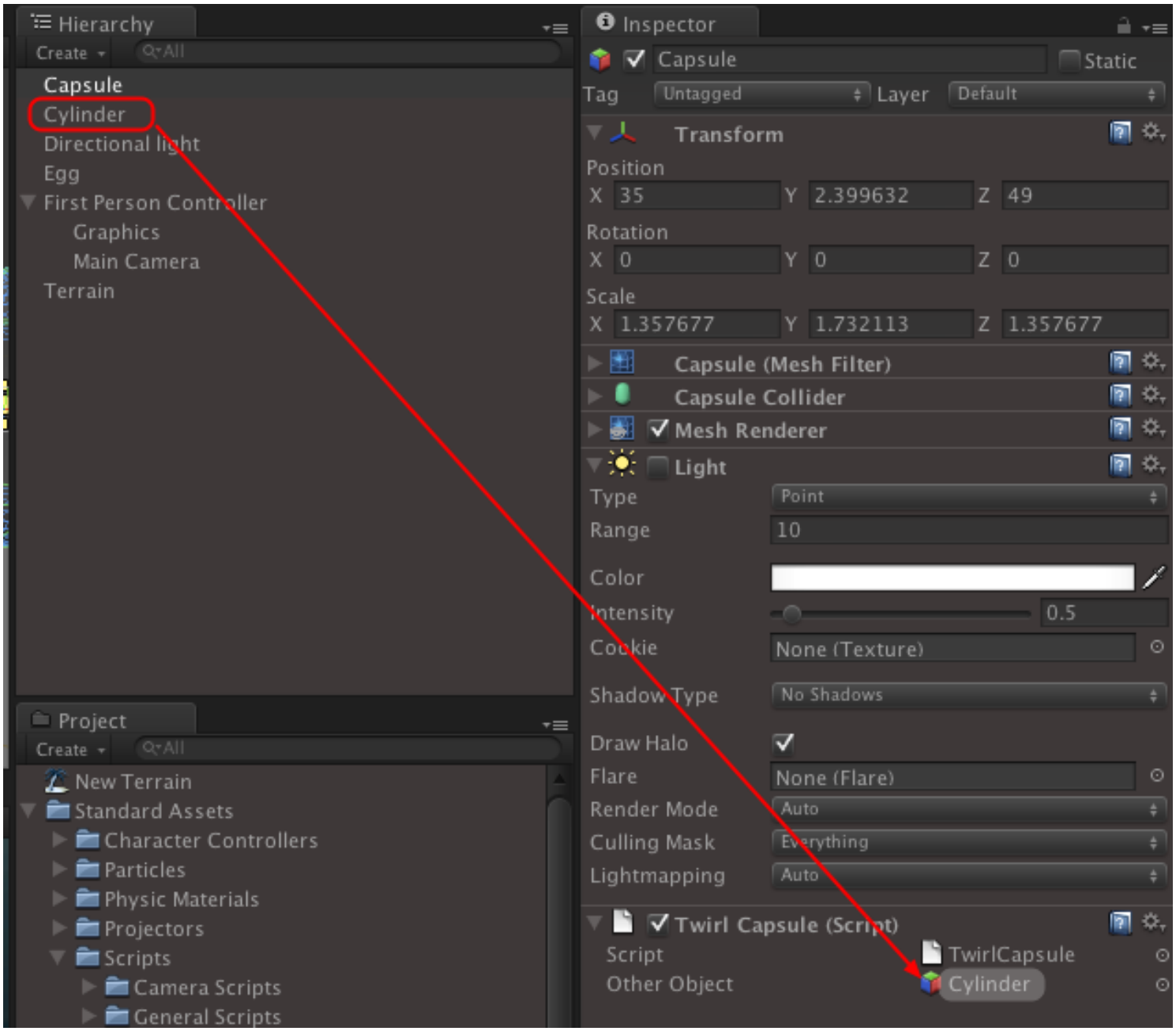
Why is there no assignment statement for the **otherObject** variable in the TwirlCapsule script? Because the assignment is taking place manually in the Inspector. This provides flexibility to control game action or make changes quickly without constantly having to edit scripts.

To have this ability to use the Inspector to make changes means the variable needs to be seen in the Inspector.

1. In the TwirlEgg script, **otherObject** is private which means it won't be visible in the Inspector

2. There is no private specified which means it's automatically public. The **otherObject** variable will be listed in the Inspector

Drag to make the assignment



After the Capsule is finished rotating, the Cylinder will rotate next.

The Cylinder is assigned to the **otherObject** variable by simply selecting the Cylinder GameObject in the Hierarchy panel and dragging it to the **otherObject** variable in the Inspector panel. Notice that Unity displays **otherObject** as two separate words, Other Object.

That's it! No need to even mention the Cylinder GameObject in the script. Also, even if the assignment

had been made in the script, dragging any other GameObject to **otherObject** would override what's written in the script

Using one script and the Inspector

In this section I have removed the three different scripts form the three GameObject, and replaced them with one script, Twirl.js

I've also made the **spin** variable public so that it's available in the Inspector. This allows selecting any of the GameObjects to be selected to be the first one to rotate.

Twirl.js – /Users/igraphic/Unity Projects/Project> Noobs/Ass...

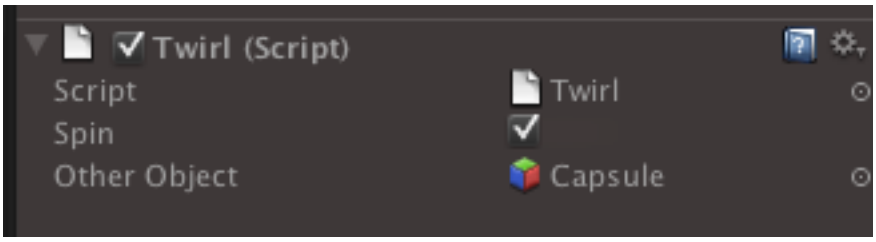
```
1  var spin : boolean = false;
2
3  var otherObject : GameObject;
4
5  function Update () {
6
7      if(spin){
8
9          light.enabled = true;
10
11         transform.Rotate(0,0,-60 * Time.deltaTime);
12
13         if(transform.eulerAngles.z <= 1.5) {
14
15             spin = false;
16
17             light.enabled = false;
18
19             transform.eulerAngles = Vector3(0,0,0);
20
21             otherObject.GetComponent(Twirl).spin = true;
22         }
23     }
24 }
```

1. The **spin** variable is now public and appears in the Inspector. It's also assigned the value of **false**

2. The **otherObject** is public and appears in the Inspector

3. A single script named Twirl is now used instead of each GameObject using a different script.

Inspector settings



This is a view of the Twirl component (script) of the Egg GameObject. Notice that Spin is checked. This overrides the **false** setting in the script and means the Egg will rotate when Play is pressed.

When Egg stops rotating, then Capsule will rotate next since it has been assigned to **otherScript**.

Congratulations, you now understand the major basics of scripting

What next?

With this basic knowledge of variables, functions, and communication principles, most of your fears should now be removed.

More learning to do

You should be able to look at just about any script, now, and not tremble too much. Some scripts can be quite long, but they still just variables, functions, and Dot Syntax. Now you can actually dissect even the longest scripts into these basic parts.

This basics manual did't touch many of the other things like loops, arrays, etc., but that wasn't the intend of this manual. However, you should be able to partake of just about any scripting tutorial that discusses many of the other programming fundamentals with new found confidence.

I do plan to do more

Since this site is dedicated to UnityScript, I will be creating many more lessons getting deeper into the subject. Many will be videos as well as manuals like this. Be aware that some parts will be free, some will require purchase.