

5

Fundamentals

Now that we've covered most of the basic concepts and terms that are commonly taught to new programmers, it's time to learn some of the fundamental concepts that are required to write functioning code. So far, we've learned about the small bits and pieces that are required to understand how to look at code; now it's time to start writing it.

I also find it is important that you compare C# to other programming languages. This comparison helps to contrast C# against languages that might do a similar task in a slightly different way. With such comparison results in mind, we're better able to understand why C# looks the way it does and we'll be better prepared to understand a concept if we look something up on the Internet exemplified in a different language.

Many of the projects that Unity 3D provides as example projects are written with JavaScript. So too are many modules available on the Unity 3D Asset Store. When necessary, you might want to read the code to understand how to use one of the downloaded assets. If you've never seen JavaScript before, you might notice how some of the code will seem familiar, but you might also find much of it to be quite alien.

In the world of computer programming, it's best to have an open mind and be able to at least read through other programming languages. To that end, some examples in this chapter will have code written in other languages shown along with the C# example.

5.1 What Will Be Covered in This Chapter

Learning how to add complexity to your code takes time. We'll cover how to make classes communicate with one another on a more global scope. Then we'll learn how to control the flow of our data in more comprehensive ways. Finally, we'll want to understand one of the more useful constructs in C#, the array, and all of its related forms.

- Review of classes
- Class inheritance
- The keyword *static*
- Jump statements, return, continue, break
- Conditional operators
- Arrays of various types, array lists, jagged arrays, dictionaries
- Strings

5.2 Review

The topics covered thus far have a lot in common with programming languages like JavaScript, UnrealScript, Lua, and Java, to name a few. Some of the syntax might look different, but the basic operation of how variables are created and assigned mechanically works the same. For instance, we understand assigning a variable to look like the following statement.

```
int a = 10;
```

In Lua, another commonly used game programming language, you might see something more like

```
local a = 10
```

In JavaScript, we might see something like the following:

```
var a = 10;
```

A language might not use some of the tokens we're used to seeing in C#, but use of a keyword, an identifier, and an assignment operator is the same. This rule also goes for something like a `for` loop. In C#, and many other C-style languages, we're used to using the following:

```
for(int i = 0 ; i < 10 ; i++) {
    //code here
}
```

JavaScript varies slightly, using `var` in place of `int`, but the rest remains the same. However, in Lua, again, we would see something more like the following:

```
for i = 0, 10, 1 do
    print(i)
end
```

Lua tends to omit many of the tokens used to help show how code statements are encapsulated; some prefer the brevity of the language. For a C-style language, the use of curly braces helps visually separate the statements more clearly.

Often, getting to this point is a difficult and unforgiving task. I hope you had some classmates, coworkers, friends, or family members to help you work out some of the exercises. If you're still a bit foggy on some of the concepts, you may want to go back and review. We've covered many cool things so far. For instance, you should be able to identify what the following code is.

```
class SomeNewClass
{
}
```

If you understood this to be a class declaration for `SomeNewClass`, then you'd be right. We should also know that for a class to be useful to Unity 3D, we need to add some directives that will look like the following. To add in directives, we start before the class declaration. We use the keyword `using` followed by a library path and add in both `UnityEngine` and `System.Collections`.

```
using UnityEngine;
using System.Collections;
class SomeNewClass
{
}
```

Directives allow our class to have access to the tools that Unity 3D has to offer. For a Windows app to have access to your computer, you need to use similar directives. For instance, if you want to work with the Microsoft XNA libraries, you'll need to add the following directives.

```
using System.Windows.Threading;
using Microsoft.Xna.Framework;
```

We're still a couple keywords away from a usable class in Unity 3D, though. First, we need to make this class visible to the rest of Unity 3D. To change a class's visibility, we need to add some accessibility keywords.

```
using UnityEngine;
using System.Collections;
public class SomeNewClass
{
}
```

We're almost there, and now we get to a part that might be confusing.

```
using UnityEngine;
using System.Collections;
public class SomeNewClass : MonoBehaviour
{
}
```

What is this : `MonoBehaviour` stuff that's coming in after our class identifier? You should be asking questions like this. We'll get around to answering this in Chapter 6. `MonoBehaviour` is another class that the programmers at Unity 3D have written. `SomeNewClass` is now inheriting from the class `MonoBehaviour`; we'll dive into what this all means in Chapter 6.

A class is the fundamental system for C#. You're allowed to write classes with nothing in them; for instance, `class NothingClass{}` is a complete class structure. You can add a single type of data in a class and give it an identifier. In this case, `class MyData{int SomeNumber;}` is an entire object with one integer value called `SomeNumber` in it. On the other hand, a file without anything other than `int NothingInParticular;` is meaningless.

A class can also be a complex monster, with many different behaviors. It's not uncommon for a class to contain several hundred or even a few thousand lines of code. The general practice is for a class to be as small as possible. Once a class grows past a few hundred lines, it's often time to consider breaking the class into several smaller classes of reusable code.

What all of this means in relation to Unity 3D is that we need to write several short classes and add them to the same `GameObject`. The same goes for any other programming project. The more modular your code is, the easier it is to create complex behavior by creating different combinations of code.

5.2.1 Modular Code

In a player character, you might have a `MovementController` class separated from a `WeaponManager` class. On top of this, you can choose either a `humanAI` class or a `monsterAI` class if you want the computer to take control. A more modular approach provides a wider variety once you've written different classes to manipulate a `GameObject` in the scene.

The same goes for functions. Rather than having a long algorithm operating in the `Update ()` function, it's better to modularize the algorithm itself. First, you might have some sort of initialization where you obtain the character's mesh and animation system. This should be done in a separate function call rather than being repeated at the beginning of every frame. We'll take a look at a better way to do this later in Section 7.18.



For now, we're reviewing all of the different things we've learned. Next, we are going to want to add in the entry points left in by the Unity 3D programmers called `Start ()` and `Update ()`; to do this, we'll simply add them into the class code block.

```
using UnityEngine;
using System.Collections;
public class SomeNewClass : MonoBehaviour
{
    void Start ()
    {
    }
    void Update ()
    {
    }
}
```

The two functions need to have a return type; in this case, both `Start ()` and `Update ()` return type `void`. The empty parentheses, `()`, indicate that they don't accept any arguments. Lastly, both `Start ()` and `Update ()` have nothing between their associated curly braces `{ }` and will not execute any code. To extend this with our own code, we can add a new function.

```
using UnityEngine;
using System.Collections;
public class SomeNewClass : MonoBehaviour
{
    void Start ()
    {
        Debug.Log(MyFunction(7));
    }
}
```



```

    void Update ()
    {
    }
    int MyFunction (int n)
    {
        return n ;
    }
}

```

If we examine the function we added, we would find we created a new function called `MyFunction`. This function returns an `int`, and accepts an arg with type `int`. In our `Start ()` function, we're using a `Debug.Log ()` function that comes from the `Debug` class in the `UnityEngine` library. The `Start ()` function then passes an `int 7` to `MyFunction ()`. The `MyFunction` returns a value `n`, which is then sent back up to the `Debug.Log ()`, which sends 7 to the Console panel in Unity 3D.

```

using UnityEngine;
using System.Collections;
public class SomeNewClass : MonoBehaviour
{
    void Start ()
    {
        for (int i = 0; i < 10; i++)
        {
            Debug.Log(MyFunction(i));
        }
    }
    void Update ()
    {
    }
    int MyFunction (int n)
    {
        return n ;
    }
}

```

Before moving on, we'll add in one last `for` loop. If you can guess that this code will print out a 0 through 10 in the Console output panel in Unity 3D, then you're doing great. If you were able to follow along with this example, then you've come quite a long way in understanding the basics behind C#.

Now that we've got most of the basics under our belt, it's time to move onto more interesting things that make C# a powerful language. What differentiates C# from older programming paradigms is the fact that it's an object oriented programming language. Objects make programming powerful, and—if you're not fully informed—confusing.

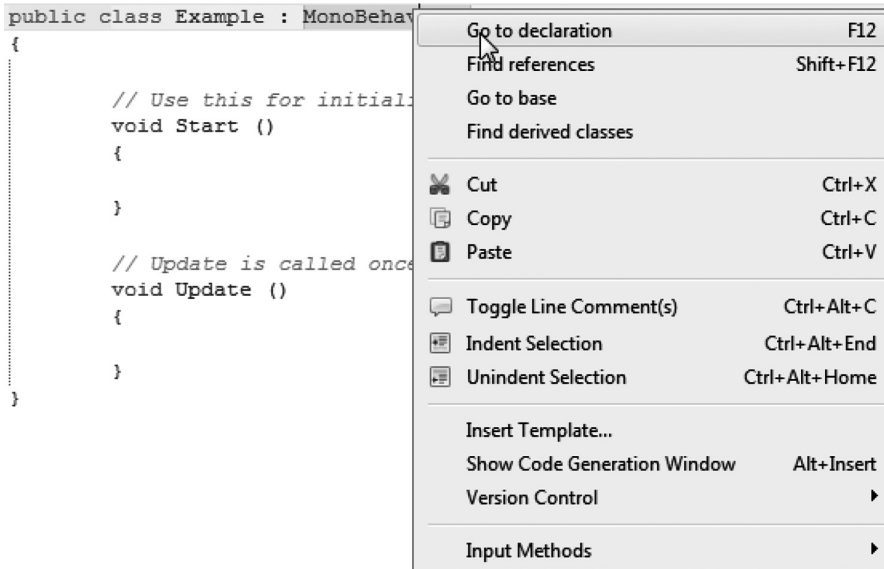
From here on out, you're going to get into the more interesting capabilities behind C#. The basics we've covered have similar utility in most programming languages. Declaring functions and variables will look the same in many different languages. Order of operation and logic both behave similarly in most modern programming languages.

Try not to limit yourself to learning a single programming language. The science of computers is always changing. New languages are being written, and new business models rely on people who know about the latest developments.

5.3 Inheritance: A First Look

Once we've written a new class, we should be thinking toward generalizing what a class can do. When we started our class declarations, we always had `: MonoBehaviour` added after the class name. This means that our class is inheriting from another class named `MonoBehaviour`.

Let's take a look at an inheritance project: Open the `Example.cs` file in MonoDevelop and right click on `MonoBehaviour`.



A pop-up with some interesting options appears over the word. Select `Go to declaration`, which brings you to the Assembly Browser. Declaration is used here with the same meaning as when we use a declaration statement to create a new identifier. We've covered declarations before, but in case you forgot, a declaration statement is used to assign a variable a new identifier.

This point should indicate that `MonoBehaviour` is a class that programmers at Unity 3D wrote for us to use. This might be a bit too much information right now, but this is showing you all of the public members that you can access within the class `MonoBehaviour`.

`MonoBehaviour` is a class that is tucked away in a dynamically linked library (DLL), or a dynamically linked library. Unlike the class we will be writing in a moment, a DLL is a prebuilt set of code that cannot be changed. Technically, you can change the contents of a DLL by various hacking methods. You could also get source code from Unity 3D to rebuild the DLL after making changes. However, a DLL is usually something that needs to be updated by its original authors.

The keyword `using` allows us access to the contents of the library where MonoDevelop is living inside of. To allow us to inherit from `MonoBehaviour`, we need to add in the `UnityEngine` library with the statement `using UnityEngine;`

To summarize, `UnityEngine` is a library written by Unity's programmers. This library contains `MonoBehaviour`, a class that is required for our C# code to communicate with Unity's functions. To use the `MonoBehaviour` functions, we tell our class to inherit from it.

5.3.1 Class Members

When we write our own classes, they can often be used as an object. There are some cases in which a class wouldn't be used as an object, but for the following examples, we're going to use them as objects. To understand what this means, it's best to see this in practice.

When you declare a class, its identifier is an important name to identify the class by. The keyword `class` tells the computer to hold onto the following word as a new type of data. Each class's identifier basically becomes a new word, much like a keyword, with special meaning.

In addition to the name of the class, all of the contained variables and functions should be considered special components of that new object. Encapsulation determines how easily these contained objects can be used.

Members of a class are not restricted to its functions. Any public variable or function becomes an accessible member of that class. To see how all of this works, we'll need to write some really simple classes in MonoDevelop.

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

After `void Update ()`, we're creating a public class `Cat`. After that, we're creating a new `PianoCat` class that inherits from `Cat`. In programming terms, `PianoCat` inherits from the base class `Cat`. So far, nothing interesting is happening, at least anything worthy of posting to the Internet.

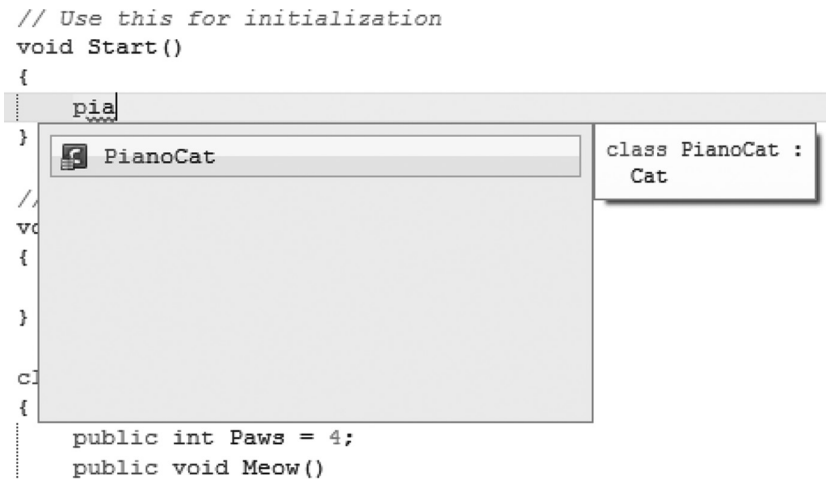
To see what it means to inherit properties, we need to add some properties to `Cat`. Let's give our base `Cat` the ability to meow. Start with `Debug.Log("Meow");` in our `Cat` class. The `Debug` function is a part of `UnityEngine`, which is declared way at the top of this file. Our class `Cat` is still able to reference the libraries that are declared at the beginning of this file.

```
public class Cat
{
    public int Paws = 4;
    public void Meow()
    {
        Debug.Log("meow");
    }
}
public class PianoCat : Cat
{
    public void PlayPiano()
    {
        Meow();//inherited from Cat
    }
}
```

So while `PianoCat` plays piano in a new function, he can also meow (`Meow();`), which is a function inherited from `Cat`.

5.3.2 Instantiating

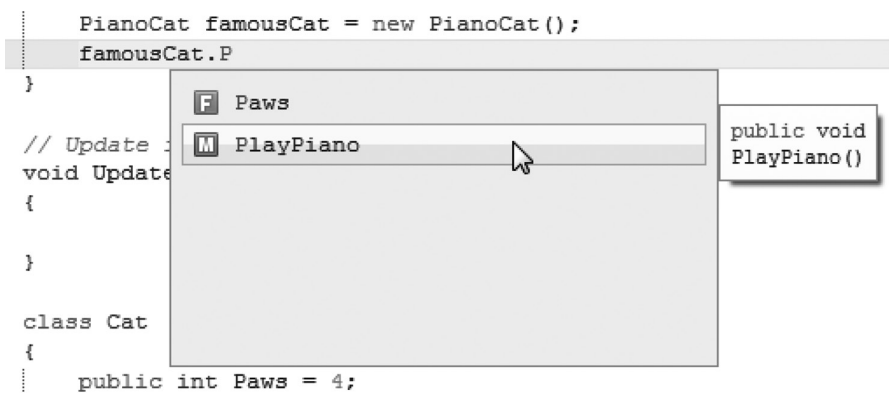
To make use of these two classes, we're going to need to do something useful with them.



In the first part of our `Example.cs` class, we've still got our `void Start ()` we can begin to add code to. When we start to enter `PianoCat`, the auto complete popup provides us with a `PianoCat` option.

```
//Use this for initialization
void Start () {
    PianoCat famousCat = new PianoCat();
}
```

To make use of a class, we need to create an instantiation of the class. Instanting a class means to construct it from a blueprint. Each instance is a new object created from the class; any changes you do to the instantiated version is unique to that instance.



We've created a `famousCat` with all of the properties of a `PianoCat`. To make this new cat (`famousCat`) play piano, we call on the member function found in the `PianoCat` class.

```
void Start () {
    PianoCat famousCat = new PianoCat();
    famousCat.PlayPiano();
}
```

`famousCat.PlayPiano();` is easily added through the handy pop-ups that `MonoDevelop` gives us. As soon as we add in the dot operator after `famousCat`, we're given several options, one of which is `PlayPiano`, but we also get `Meow ()` function.

```
void Start ()
{
    PianoCat famousCat = new PianoCat();
    famousCat.PlayPiano();
    famousCat.Meow();
}
```

`Meow` is a member function of `PianoCat` because `PianoCat` inherited from `Cat`. Likewise, if we made a `NyanCat` or `HipsterCat` based on `Cat`, they would also inherit the `Meow` function from `Cat`. Running the code in Unity 3D returns an expected meow in the Console panel.



We set the number of paws on `Cat` to 4; we can use that number through `famousCat`.

```
public class Cat
{
    public int Paws = 4;
    public void Meow() {
        Debug.Log("meow");
    }
}
```

In our `Start ()` function, in the first class we started with, we can use `print ()` to show how many Paws the `famousCat` inherited from `Cat`.

```
void Start ()
{
    PianoCat famousCat = new PianoCat();
    famousCat.PlayPiano();
    famousCat.Meow();
    Debug.Log(famousCat.Paws);
}
```

We've gotten to a point where we'd like to search the entire scene for a specific object. In this case, we should look for every player in the scene. Looking through `GameObject`, we find the following functions.

- ◆ `Find (string) : GameObject`
- ◆ `FindGameObjectsWithTag (string) : GameObject[]`
- ◆ `FindGameObjectWithTag (string) : GameObject`
- ◆ `FindWithTag (string) : GameObject`
- ◆ `GetComponent<T> () : T`

We can find an object using its name or tag. If we want to find any object that happens to have a `player.cs` component, we need to know which `GameObject` it's attached to first. This is quite a puzzle. It's hard to know the name of the object we're looking for if we only know one thing about it. However, there is a more general way to find things.

Of course, we could just use Unity 3D's editor and assign tags, but we wouldn't learn how to do things with code alone. In a more realistic situation, we could just have our game designers assign tags to everything, use the `FindWithTag()` function, and be done with it.

To give ourselves a better understanding of objects and types, we should go through this process without tags. This way, we will gain a bit more understanding as to what defines objects and types, and how to use `null`. The `null` keyword basically means “nothing”; we’ll see how that’s used in a moment.

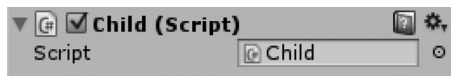
```
public sealed class GameObject : Object
```

5.3.2.1 A Basic Example

Programmers use the version of the word *inheritance* from genetics, inheriting traits and ability rather than wealth and status. `GameObject` is the child of `Object`; we know this because of the operator separating `GameObject` from `Object`. In a general sense, all of the code that was written in `Object` is a part of `GameObject`.

5.3.3 Parent and Child

To show you a basic example of how inheritance works with code, we’ll write two new classes. The first is `Parent.cs` and the second is `Child.cs`. Create both of these in the Unity 3D project we’ve been working with. Using the usual right click in the project panel and then the path `Create → C# Script` is usually the best way. Name the new files according to this tutorial.



Assign the `Child` script to a cube in the scene. The class declaration in the `Child.cs` should look like the following. If there are other scripts on the object in the scene, then you can right click on the script’s title and remove the component from the pop-up.

```
public class Child : Parent
{
}
```

Observe that the usual `MonoBehaviour` following the colon (`:`) is replaced with `Parent`. Now to prove that `Child` inherits the functions found in `Parent.cs`, we should add a function that the `Child.cs` can use. This code should appear in the `Parent` class.

```
public void ParentAbility()
{
    Debug.Log("inheritable function");
}
```

I added this after the `Update ()` function that’s created by default when you use the Unity 3D editor to create classes for your project. To see that the function can be used by `Child.cs`, we’ll use it in the `Start ()` function in the `Child.cs`, as in the following.

```
public class Child : Parent
{
    void Start ()
    {
        ParentAbility();
    }
}
```

ParentAbility() is a public class found in the Parent class, not in the Child class. Running this script when it's attached to an object in the game produces the expected *inheritable function* printout in the Console panel. Remember, there's nothing inside of the Child class that has yet to have any new member functions or variables added. If we keep adding new public functions to Parent, Child will always be able to use them.

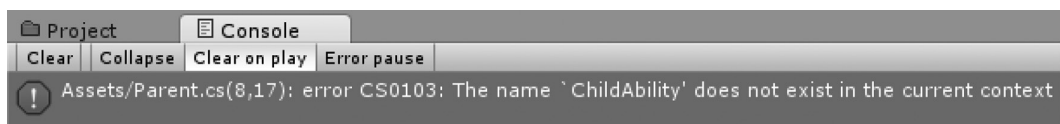
Next, add a function to the Child class like the following.

```
public void ChildAbility()
{
    Debug.Log("My parent doesn't get me.");
}
```

Then, check if the Parent class can use that function.

```
using UnityEngine;
using System.Collections;
public class Parent : MonoBehaviour
{
    //Use this for initialization
    void Start ()
    {
        ChildAbility();
    }
    //Update is called once per frame
    void Update ()
    {
    }
    public void ParentAbility()
    {
        Debug.Log("inheritable function");
    }
}
```

Doing so will produce the following error.



Even though the function is public, the parent doesn't have access to the functions that the Child class has. This is an important difference. The other main takeaway from this tutorial is to remember that Parent inherited from MonoBehaviour, as did all the other scripts we've been working with to this point.

The significance behind this discussion is that Child.cs has all of the functions found not only in Parent.cs but also in MonoBehaviour. One more thing should be noted:

```
public class Parent : MonoBehaviour
{
    public int SomeInt;
    ///other code...
}
```

Adding a public `int` to the `Parent` class will also allow the `Child` class to use the `SomeInt` variable.

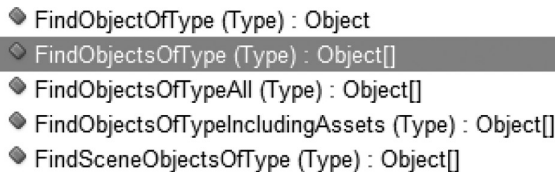
```
using System.Collections;
using UnityEngine;
public class Child : Parent
{
    void Start ()
    {
        Debug.Log(SomeInt);
        ParentAbility();
    }
    public void ChildAbility()
    {
        Debug.Log("My parent doesn't get me.");
    }
}
```

`SomeInt` is available as though it were declared as a part of `Child`, because it is. It's important to know that changing anything in the `Child` class will never affect the `Parent` class. Inheritance only works one way: The children of a parent can never change the functions in a parent. Such an occurrence rarely happens in real life as well as in C#.

5.3.4 Object

`GameObject`, the class we're looking at, inherits from `Object`. Then `GameObject` adds new functions to on top of anything that already existed in `Object`. As we have just observed from our simple inheritance example, if we find a function in `Object`, we can use it in `GameObject`.

What this means to us is that we may use any interesting functions inside of the `Object` in the Assembly Browser in `Object` as well.



- ◆ FindObjectOfType (Type) : Object
- ◆ FindObjectsOfType (Type) : Object[]
- ◆ FindObjectsOfTypeAll (Type) : Object[]
- ◆ FindObjectsOfTypeIncludingAssets (Type) : Object[]
- ◆ FindSceneObjectsOfType (Type) : Object[]

Stepping through the thought process, we begin with “I’d like to find an object in the scene, but I might not know what it’s called.” From that, I would likely look for something called `FindObjects`. The objects I need to find are going to be `GameObjects`, so that does narrow things down a bit. Inside of `Object`, we find a function called `FindObjectsOfType()`, which is probably what we’re looking for.

```
GameObject[] gos = GameObject.FindObjectsOfType(typeof(GameObject)) as
GameObject[];
```

In the `Start ()` function, we need to set up an array for every game object in the scene. That’s done with `GameObject[]`; the square brackets tell the variable it’s an array. We’ll cover arrays in more depth in Section 5.9. Then we name it something short, like `gos`, for game objects. Then we can have `Object` run its `FindObjectsOfType` function to fill in our array with every `GameObject` in the scene.

Here’s the interesting part, where inheritance comes in.

```
GameObject[] gos = Object.FindObjectsOfType(typeof(GameObject)) as GameObject[];
```

`GameObject.FindObjectsOfType()` works just as well as `Object.FindGameObjectsOfType()`, which should be interesting. There is significance to this finding, as inheritance in C# makes things complex and clever. Keep this behavior in mind in the following example.

Now that we have a list of objects, we need to find the `GameObject` with the `Child.cs` attached to it. To do this, we're going to reuse the `foreach` loop we just covered in Chapter 4.

```
//Use this for initialization
void Start ()
{
    PianoCat famousCat = new PianoCat();
    famousCat.PlayPiano();
    famousCat.Meow();
    Debug.Log(famousCat.Paws);
    GameObject[] gos =
    GameObject.FindObjectsOfType(typeof(GameObject)) as GameObject[];
    foreach (GameObject go in gos)
    {
        Debug.Log(go);
    }
}
```

So now we're up to an array of `GameObjects` followed by a way of looking at each `GameObject`. Right now, each monster in the scene knows about everything else in the scene. The `Child.cs` is a component of the `GameObject` cube in the scene, as is everything else that the `GameObject` is made of.

```
Component comp = go.GetComponent(typeof(Child));
```

To find the `Child.cs` in the `ComponentsList` of the `GameObject` called `go` in the `foreach` statement, we need to create a variable for it. In this case, we have `Component comp` waiting to be filled in with a component from the `GameObjects` in the `gos GameObject[]` array. That component also inherits from `Object`.

```
public class Component : Object
```

NOTE: You could enter `Component.FindObjectsOfType()`, and this will work the same as though it were called from `Object` or `GameObject`. Inheritance is a complex topic, and right now, we're only observing how functions can be shared from the parent to children, but there's a problem when children try to share. We've used `GameObject.CreatePrimitive()` to make cubes and spheres. And we know `Object` has `FindObjectsOfType`, which both `GameObject` and `Component` can use because of inheritance. Therefore, does that mean `Object.CreatePrimitive` will work?

Actually, no; `Object` doesn't inherit functions of variables from its children. In programming, parents don't learn from their kids. Suppose `GameObject.CreatePrimitive()` is a function found in `GameObject`, and `GameObject` and `Component` both inherit from `Object`; does that mean `Component.CreatePrimitive()` will work? Again, no; in C#, children don't like to share. C# is a pretty tough family, but rules are rules.

Therefore, now we have a `Component` in place for the player, and we're setting `comp` to the `go.GetComponent(typeof(Player))`, which will set `comp` to the `Child.cs` component found in the game object, if there is one. We are relying on the fact that not all objects have that component. When the `GetComponent(typeof(Player))` fails to find anything, `comp` is set to `null`, and it does so reliably.

5.3.4.1 A Type Is Not an Object

Type, as we've mentioned, is the name given to a class of an object. Like a vowel, or noun, type refers to the classification of a thing, not the actual thing itself. Semantics is important when dealing with programming, and it's very easy to miss the meanings behind how words are used. For instance, if we need to tell a function the type of a class, we can't simply use the class's name.

```
GetComponent(Player);
```

If the function expected the type `player` and not the object `player`, you'd get an error. There's a subtle difference here. The type of an object is not the object. However, it's difficult to say "this is the type `Child`" and not the "`Child`" class object. The two things use the same word, so it's easy to confuse.

There is a function that will get the type of an object and satisfy functions that need types and not objects. And that's where `typeof(Child)` comes in. If we have the object `Child` `typeof(Child)`, we can put `Child` into the `typeof()` function and use that function as the type of `Child` and not the class `Child`.

Give the `FindObjectsOfType()` function a `typeof(Child)` and it returns an array with every instance of `Child` in the scene. We can give the `Example.cs` a `GameObject ChildObject;` to chase after. Just so we don't get confused what version of the word `Player` we're looking for we should make it tremendously clear what we're looking for.

5.3.5 != null

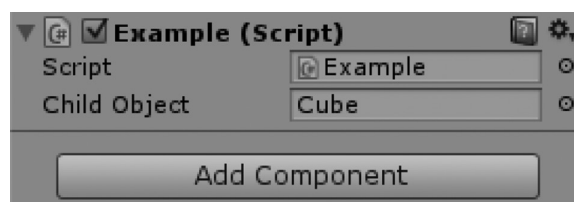
Add in a `GameObject` variable for the player at the class scope. The not null check is important. We'll find out why in Section 5.7.4, but for now we'll just observe how it's best used.

```
public class Example : MonoBehaviour
{
    public GameObject ChildObject;
    ///other code...
}
```

Then, in the `Start ()` function, we finish our code with the following:

```
void Start ()
{
    PianoCat famousCat = new PianoCat();
    famousCat.PlayPiano();
    famousCat.Meow();
    Debug.Log(famousCat.Paws);
    GameObject[] gos =
    GameObject.FindObjectsOfType(typeof(GameObject)) as GameObject[];
    foreach (GameObject go in gos)
    {
        Debug.Log(go);
        Component comp = go.GetComponent(typeof(Child));
        if (comp != null)
        {
            ChildObject = go;
        }
    }
}
```

The new section is `if (comp != null)`; our `ChildObject` is the game object with the `go` where `comp` is not null. Sometimes, phrases like "component is not null" are unique to how programmers often think. Their vocabulary may seem unfamiliar at first, but they will grow accustomed to it soon enough. As we start to use little tricks like "`!= null`," we'll start to understand why programmers often seem to use a language of their own.



We can run the game, and check the `ChildObject` variable. We'll see that it's `Cube`, and if we check, we'll find that the cube with the `Child.cs` attached to it is the one that we're looking for.

`[something] != null` is commonly used to know when something has been found. When the something is not empty, you've found what it is you're looking for. This is a useful trick to apply for many different purposes.

5.3.6 What We've Learned

Using data to find data is a commonplace task in programming. With Unity 3D, we have plenty of functions available which make finding objects in a scene quite easy. When monsters go chasing after players, we need to have a simple system for allowing them to find one another.

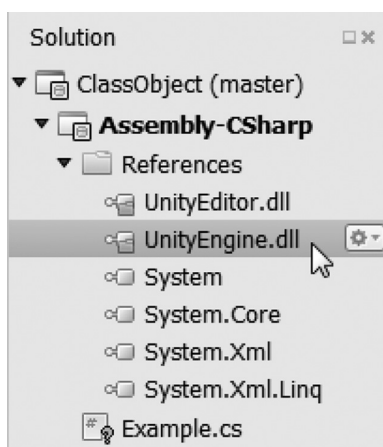
Computers don't use senses like we do. They have another view of the game that can't be compared to how we, as players, perceive the world. By searching through data, and using algorithms, the computer-controlled characters require a different set of parameters to operate.

In one sense, the computer has complete knowledge over the game world. How that world is perceived by the computer requires our instruction. A monster's ability to understand its environment depends on our code.

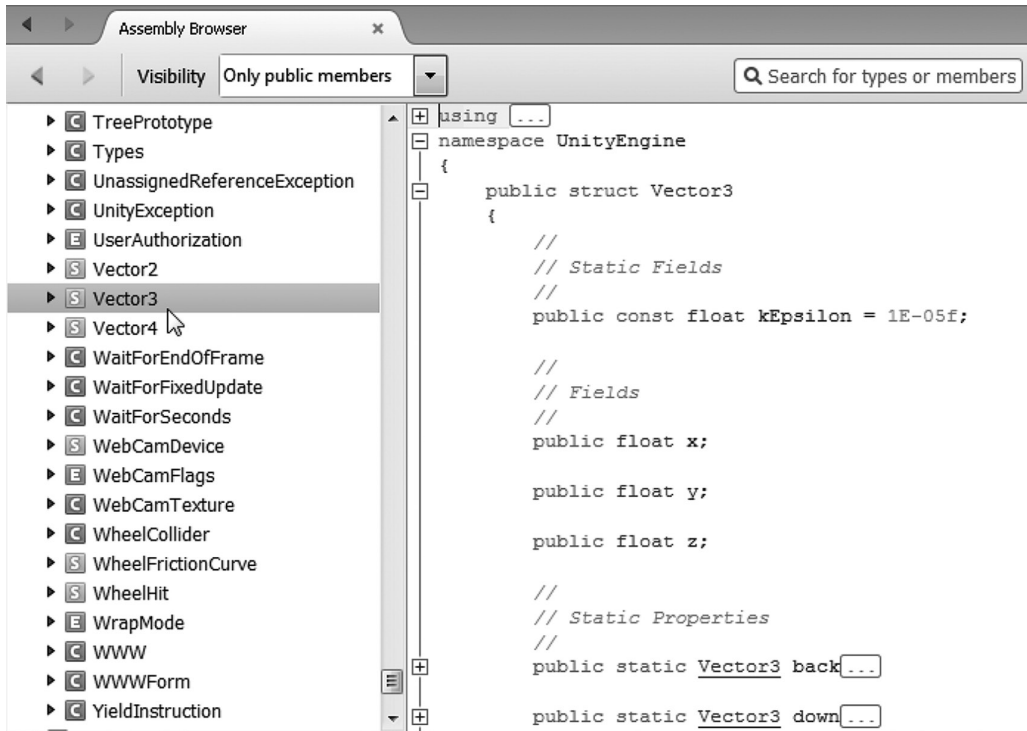
5.4 Instancing

An important feature of C# and many C-like programming paradigms is the use of a class as a variable. When we use a class as a variable, it's created as a new object. Some of these classes are values, which means that the function itself can be used to assign to variables, and many classes are used as values. Using a class as an object which is itself a value is at the heart of object oriented programming.

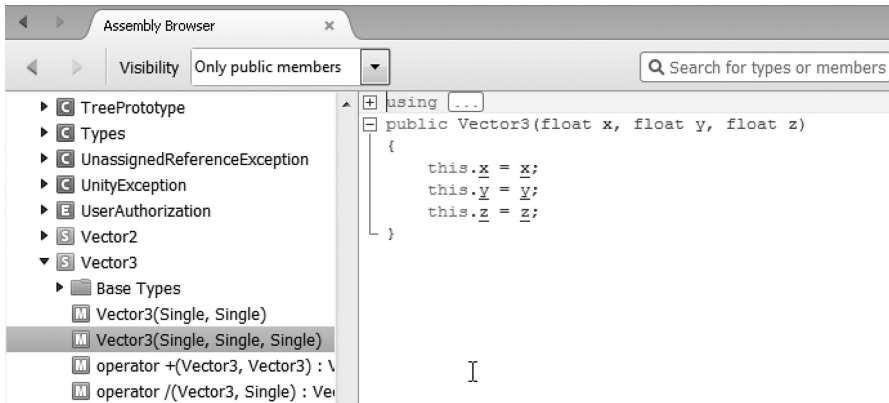
To make use of a class, or object, variable, we often need to create a new instance of the class. Instancing is basically creating space in memory for a class of a more complex data type. We do this in the same way we create a variable for a more simple data type. However, when assigning a value to the variable of this sort, we need to change things up a bit.



The `Vector3` class is located in the guts of the `UnityEngine` library. We can also see what else the `UnityEngine` library has by double clicking on the `UnityEngine.dll` located on the Solution panel in MonoDevelop. This opens the Assembly Browser. Inside there, you'll find the `UnityEngine.dll`, which you can then expand to find the contents of the DLL.



Expand this again and scroll down; you will find that the contents are arranged alphabetically to make things easier to find. Expand the **Vector3** class to reveal its contents; you'll find many additional objects inside of the **Vector3** class.



Don't let this cacophony of information bewilder you. We only need to deal with a small portion of these objects to get started. Later on, we'll be able to browse the classes in each one of these DLLs and look for classes we can include in our code.

5.4.1 Class Initialization

The reason why we don't need to create an instance for an **int** or a **bool** is that they are built-in types found in the system library. We can also create our own types for our own game. Yes, we get to invent new types of data. For instance, you'll need a player type and a monster type.

For these new types, we'll want to store the status of their health and possibly their ammunition. These bits of information can be any value; however, they are usually `ints` or `floats`. Of course, to store a player's location in the world, we'll want to include a `Vector3`, inferring that a type can contain many other non-primitive types. Eventually, everything in the game will have a class and a type of its own.

In C#, the built-in types do not need a constructor. We'll get into constructors when we start instantiating objects later in this chapter. Plain old data types (PODs) include not only the built-in types but structures and arrays as well. The term came from the C++ standards committee, and was used to describe data types that were similar between C and C++. This term can be extended to C# that shares the same POD types.

5.4.2 New

The `new` keyword is used to create a new instance of a class. Some data types that fulfill a variable, do not need the `new` keyword. For example, when you declare an `int`, you simply need to set it to a number, as in `int i = 7;`. This declaration creates space in memory for an `int` named `i`, and then sets the value for `i` to 7. However, for more complex data types, like a vector, you need to give three different float values. Unity 3D calls a 3D vector a `Vector3`, because there are three values stored in it: `x`, `y`, and `z`.

NOTE: Unity 3D also has a `Vector2` and a `Vector4` type. These are instantiated in the same way as a `Vector3`, but have more specific uses. `Vector2` only has an `x` and `y` value; `Vector4` has an additional vector, `w`, after `x`, `y`, and `z`.

Normally, in 3D space, you'd need only a `Vector3`, but when dealing with something called quaternions, you'll need the extra fourth number in a vector; we'll get into this a bit later on.

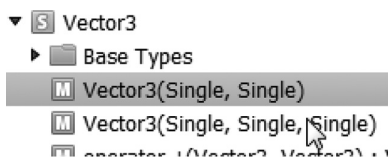
Remember, declaring a variable starts with the type followed by the name we're assigning to the variable. Normally, you'd start with something like `Vector3 SomeVector;`. Where the difference comes in is assigning a value to the variable. To do this, we start with the `new` keyword, followed by the `Vector3`'s initialization.

5.4.3 Constructors

To follow along, we'll use the `ClassObject` Unity 3D project and examine the `Example.cs` file in `MonoDevelop`. Constructors add an extra, but necessary, step when using a variable type that's derived from an object class. With many classes, there are different ways in which they can be initialized with a constructor. Most initializations can be done without any parameters, as in the following example.

```
void Start ()
{
    Vector3 vector = new Vector3();
}
```

The use of a pair of parentheses after the type refers to a constructor, or a set of parameters that can be used to build the instance when it's created.



Looking at the Assembly Browser again, we can see that there are two `Vector3()` functions under the `Vector3` class. The constructors are named the same as the class. There's always going to be an assumed `()` version, with no parameters for construction. After that, there can be any number of alternative constructors. In this case, there are two additional constructors created for `Vector3()`.

We should also note that `float` is another name for `single`, as seen in the Assembly Browser. The word *double*, which we've mentioned in Section 4.5, has twice the amount of space in memory to store a value than a single; a single is a 32-bit floating point, and a double is a 64-bit floating point value. More on that when we cover how numbers are stored.

We can add information, or parameters, when initializing a new `Vector3` type variable. Parameters, also known as arguments, are values that have been added to the parenthesis of a function. The different parameters are setting variables within the class we're instantiating. Inside of a `Vector3`, there is an `x`, a `y`, and a `z` variable. Like the variables we've been using, like `SomeInt`, `Vector3` is a class that has its own variables. When using parameters, we can get some guides from our integrated development environment (IDE).

If you look at the top right of the pop up, you'll see some indicators, you'll see an up and down triangle before and after a "1 of 3". The indicator numbers the different ways a `Vector3` can be created. The up and down arrow keys on your keyboard will change which methods we can use to create a `Vector3` class. The third one down has a `float x`, `y`, and `z` in the argument list for the `Vector3` type. This is related to the class's constructor.

```
// Use this for initialization
```

```
void Start()
```

```
{
```

```
    Vector3 vector = new Vector3()
```

```
}
```

```
// Update is called once per frame
```

```
void Update()
```

```
{
```

```
}
```

```
public Vector3(
```

```
    float x,
```

```
    float y,
```

```
    float z
```

```
)
```

After the first parenthesis is entered, `(`, a pop-up will inform you what parameters you can use to initialize the `Vector3` we're creating. This information comes from the `UnityEngine.dll` we looked at earlier. As you fill in each parameter, you need to follow it with a comma to move to the next parameter.

You should use the up and down arrows to switch between constructor options. The second option requires only an `x` and a `y`; this leaves the `z` automatically set to 0. As we start using other classes that need different initialization parameters, it's useful to know what sort of options we have available.

To use the vector we created, we'll assign the object in Unity's Transform, in the Position component, to the vector variable we just assigned. This assignment will move the object to the assigned `Vector3` position. We'll go into further detail on how to know what a component expects when assigning a variable when we start to explore the classes that Unity 3D has written for us.

```
void Start ()
```

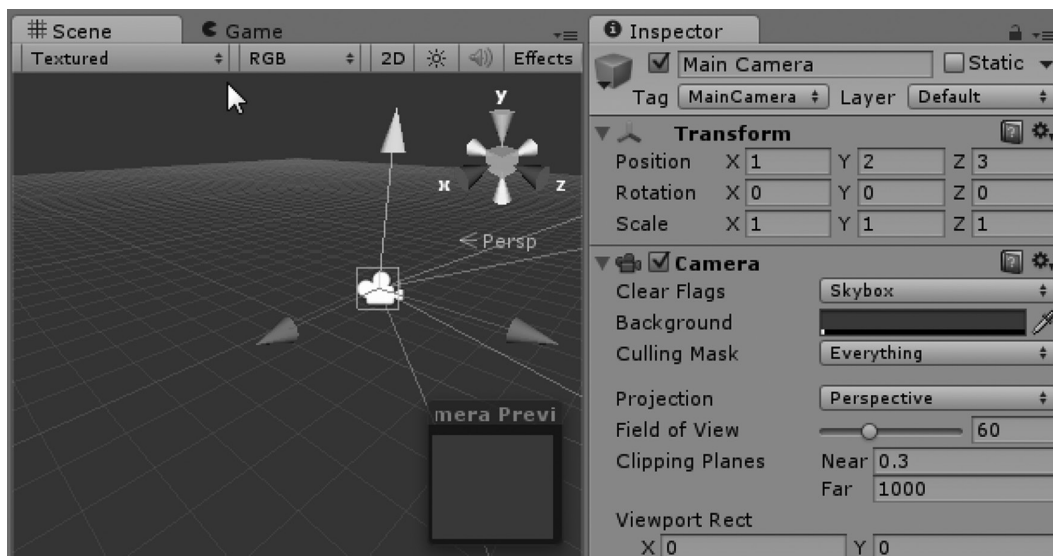
```
{
```

```
    Vector3 vector = new Vector3(1, 2, 3);
```

```
    transform.position = vector;
```

```
}
```

When this class is attached to the Main Camera in the scene and the Play in the Editor button is pressed, we will see the following in Unity 3D. Notice entries for Position in Transform roll out in Inspector.



The declaration and initialization of a new vector requires an extra step. For instance, this situation can be translated into English as follows: “I’m making a new `Vector3`, and I’m naming it to `vector`. Then I’m assigning the `x`, `y`, and `z` to 1, 2, and 3. Then, I’m setting the position in the Transform component to `vector`.” Likewise, the shorter version could simply be “I’m setting the Position in the Transform component to a `Vector3`, with `x`, `y`, and `z` set to 1.0f, 2.0f, and 3.0f.”

NOTE: Natural language programming, like the English translation above, has been tried before. However, the compiler’s job would be to both interpret and translate English into byte code, and this has proven to be quite tricky. Interpretation of English is part intuition and part assumption. Neither of these abilities can be easily programmed into software to work consistently and perfectly predictably.

Various ways to program computers using different formatting options to make code more human readable have been tried many times. In the end, the formatting almost always ends up using more and more keywords, which restrains the programmer, by taking away the options and freedoms that are allowed when fewer rules are applied.

You can also use an initialized vector without assigning it to a variable name first. This is perfectly valid; when Position in the object’s Transform component is set, it’s automatically assigned a `Vector3` that’s created just for the position.

```
void Start ()
{
    transform.position = new Vector3(1, 2, 3);
}
```

Once a new class has been initialized, you can modify the individual variables found inside of the `Vector3` class. A variable found in a class is referred to as a member variable of the class. We’ll go into more detail in Section 6.3.2 about what sort of variables and functions can be found in a class that

Unity 3D has written for your benefit. To change one of these variables inside of a class, you simply need to assign it a value like any other variable.

```
void Start ()
{
    Vector3 vector = new Vector3(1, 2, 3);
    vector.x = 1.0f;
    transform.position = vector;
}
```

There's an interesting difference here. An `f` has been used: the `1.0f` is assigned to the `x` member of `vector`. This informs the compiler that we're assigning a float type number to `vector.x` and not a double. By default, any number with a decimal with no special designation is assumed to be a double number type.

We'll dive into the significance of the difference later on. To tell Unity 3D we're assigning a float to the `x` member of `vector`, we need to add the suffix `f` after the number. This changes the number's type to float. This is related to type casting, which we will cover in Section 6.5.3.

We can also assign POD as an object.

```
int i = new int();
Debug.Log(i);
```

Adding this after the previous code block, we'll get 0 sent to the Unity 3D Console panel. We don't need to do things like this for all data types, but it's interesting to see that it's possible. Most data types can be initialized in this way. There are some tricky constructors, like `string`, which need some additional work for us, to understand the parameter list. The details here will have to wait till Section 6.4 in the book.

5.4.4 What We've Learned

The example in this section sets the `x` of the `vector` to `1.0`; the `y` and the `z` are left at `0.0`; however, the values for `y` and `z` were assigned by C# when the class was first initialized. Setting variables in the class is controlled by how the class was written. The visibility of the variables in the `Vector3` class is determined in the same fashion as we declare variables in classes that we have written ourselves. We can make changes to the `x`, `y`, and `z` float variables in the `Vector3` class because of their public declaration.

Each nuance and use of the C# language takes some time to get used to. When certain tasks are shortened into easier-to-write code, the code ends up being harder to read. However, most of the harder tasks often have a commonly used format. These shorthand formats are sometimes called idioms.

Idioms in English do not often make the most sense when logically analyzed, but they do have specific meanings. Much like in English, programming idioms might not make sense when you look at them, but they do have a specific meaning when in use. As we make further progress, you'll be introduced to some of the more commonly used idioms in C#.

5.5 Static

The `static` keyword is a keyword that ties all instances of a class together. This allows all instances of a class to share a common variable or function. When you see `static` next to a function, this means you don't need to make a new instance of the class to use it. Because `static` functions and variables are class-wide, we access them in a slightly different way than an instance of a variable or function.

There is a `static` function found inside of `Input` called `GetKey()`, which we will use. The last keyword is `bool`, after `public` and `static`. We've seen `bool` before, but this is used as a return type. When we write our own functions, we'll also include our own return type, but just so you know, return types are not limited to `bool`. Functions can return any data type, and now we'll take a look at how this all works.

5.5.1 A Basic Example

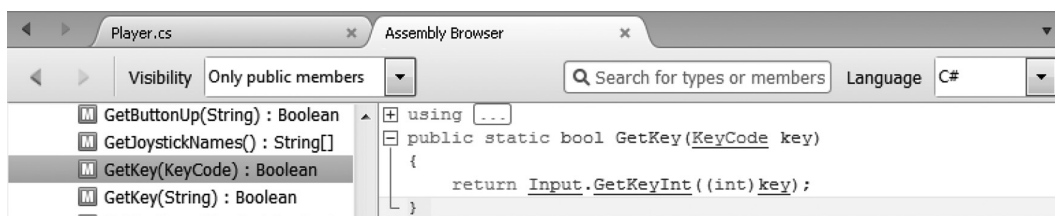
Let's start with the Static project; begin by opening the scene in the Unity 3D Assets Directory. The Main Camera in the scene should have the Player component attached.

```
using UnityEngine;
using System.Collections;
public class Player : MonoBehaviour
{
    //Use this for initialization
    void Start ()
    {
    }
    //Update is called once per frame
    void Update ()
    {
        //add in a check for the A key
        bool AKey = Input.GetKey(KeyCode.A);
        if (AKey)
        {
            Debug.Log("AKey");
        }
    }
}
```

The code here will send the following to the Console panel in Unity 3D.

```
AKey
UnityEngine.Debug:Log(Object)
Player:Update () (at Assets/Player.cs:15)
```

The KeyCode class has a public variable for each key on your computer's keyboard. Each variable returns true when the key on the keyboard is pressed. The `Input.GetKey()` function returns the `bool` value based on which the key is checked in the KeyCode class. We assign the `bool AKey` to the returned value from the `Input.GetKey(KeyCode.A);` function. The statement `if (AKey)` then executes the `Debug.Log("AKey");` which then prints out `AKey` to the Unity Console window. The `GetKey()` function is a member of the `Input` class and is accessed by using the `.` or dot operator. Now that we're reading libraries we need to know how to get to the members of the classes we're looking at.



Select the `GetKey()` and go to the definition of this function. You can do this by right clicking on the word and selecting `Go To Definition` in the pop-up. This will open the Assembly Browser, where we'll be shown the function's definition. We're shown `public static bool GetKey(KeyCode key)` as the function's definition.

This process provides us with the beginnings of a player controller script. Filling in the rest of the WASD keys on the keyboard will allow you to log the rest of the most common movement keys on your keyboard. The `Input` class is found in `UnityEngine` that was included in this class with the `using UnityEngine` directive at the top of the class.

From previous chapter lessons, you might imagine we'd need to create an instance of `Input` to make use of its member functions or fields. This might look like the following:

```
void Update ()
{
    Input MyInput = new Input();
    bool AKey = MyInput.GetKey(KeyCode.A);
    Debug.Log(AKey);
}
```

While this syntactically makes sense, you're better off not creating new instances of the `Input` class. The `GetKey()` function is `static`, so there's no need to create an instance of `Input` to use the function. This will make more sense once we write our own static function later in this chapter. However, should we actually try to use `MyInput` as an object, we'd have some problems.

```
// Update is called once per frame
void Update()
{
    Input MyInput = new Input();
    MyInput.
    bool
    Debug
}
```

Equals
GetHashCode
GetType
ToString

```
public virtual bool
Equals(
    object obj
)
```

Summary

Determines whether the specified *Object* is equal to the current *Object*.

When we check for any functions in the `MyInput` object after it's been instanced, we won't find anything useful. The only things available are universal among all things inheriting from the object class. We'll get to what the object class is in Section 6.13.4, about inheritance, but for now, just assume that these functions will appear and act the same for pretty much anything you are allowed to make an instance of.

The lack of functions or variables is due to the fact that all of the functions and variables have been marked as `static`, so they're inaccessible through an *instance* of `Input`. Statically marked functions and variables become inaccessible through an instance. This inaccessibility is related to an object's interface and encapsulation. This doesn't mean that they are completely inaccessible, however; but to see them, we'll need to write our own class with our own functions and variables.

5.5.2 Static Variables

Let's start off with a simple concept: a `static` variable. Using our previous analogy of toasters, each new toaster built in a factory has a unique identity. Each unit built would have a different serial number. Once in someone's house, the number of times the toasters have been used, when they are turned on and begin to toast, and their location in the world are all unique to each toaster. What they all share is the total number of toasters that exist.

The `static` keyword means that the function or variable is global to the class it's declared in. Let's make this clear; by declaring a variable as `static`, as in `static int i;`, you're indicating that you want to use the `int i` without needing to make an instance of the class that the variable is found in.

Variables that are unique to each instance of a class are called *instance variables*. All instances of the class will share the static variable's value regardless of their instanced values.

If we create a mob of zombies, we might find it important to maintain a certain number of zombies. Too many and the computer will have problems keeping up. Too few and the player will go around with nothing to shoot at. To give ourselves an easier time, we can give each zombie a *telepathic* connection with all other zombies. A programmer would call this a *static variable* or a *static function*.

As we've seen, variables usually live a short and isolated life. For instance, the variable used in the ever-powerful `for` loop exists only within the `for` loop's code block. Once the loop is done, the variable in the `for` loop is no longer used and can't be accessed outside of the loop.

```
for (int i = 0; i < 10; i++)
{
    print(i); //prints 1 to 10
}
//i no longer exists
print(i); //error, i is undefined
```

Now that we're creating objects based on classes, we need to make those classes interact with one another in a more convenient way. To keep track of the number of undead zombies in a scene, we could have the player keep track; as each zombie is created or destroyed, a message could be sent to the player.

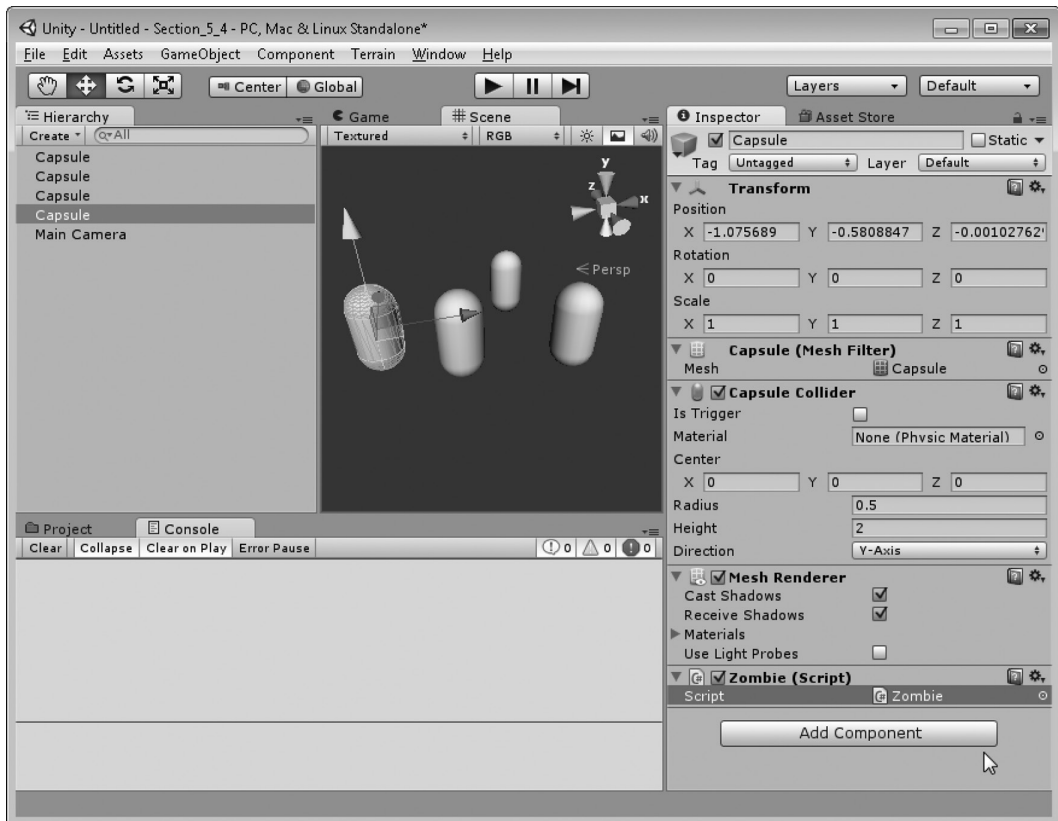
The Player class keeping track of important zombie data is awkward. Zombie-related data should remain in the zombie class, unrelated classes should not depend on one another in this way. The more spread out your code gets, the more problems you'll have debugging the code. Creating self-contained classes is important, and prevents a web of interdependent code, sometimes referred to as spaghetti code.

Remember that when an object is created it's instanced from a single original blueprint, which can include some variables that are shared between each instance of the class. With static variables instanced cases can talk to the blueprint for a common connection between any new instances made from it.

5.5.2.1 A Basic Example

```
using UnityEngine;
using System.Collections;
public class Zombie : MonoBehaviour
{
    static int numZombies;
    //Use this for initialization
    void Start ()
    {
        numZombies++;
    }
}
```

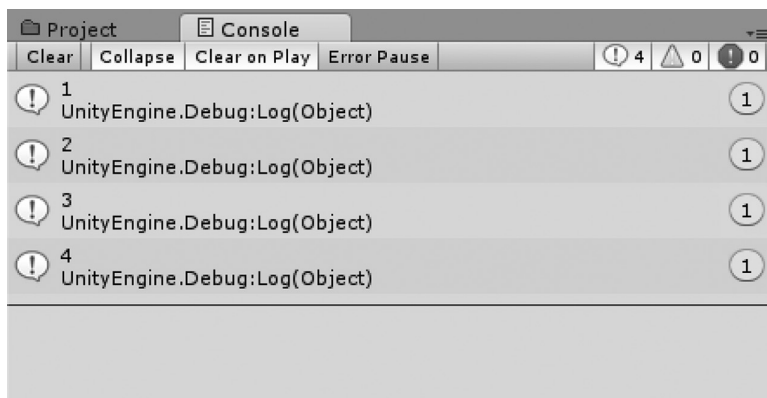
Here, in a pretty simple-looking zombie class, we've got a `static` keyword placed before an `int` identified as `numZombies`. This statement turns `int numZombies`, which would normally be confined to this class alone, to an `int` that's shared among all zombies. As Unity 3D creates more zombies, each zombie has its `Start ()` function called once the class has been instanced: `numZombies` increments by 1.



With the above code attached to a collection of four different capsules, we'll be better able to see how many zombies there are in the scene through code. Just be sure that each capsule has the zombie script attached.

```
void Start ()
{
    numZombies++;
    Debug.Log(numZombies);
}
```

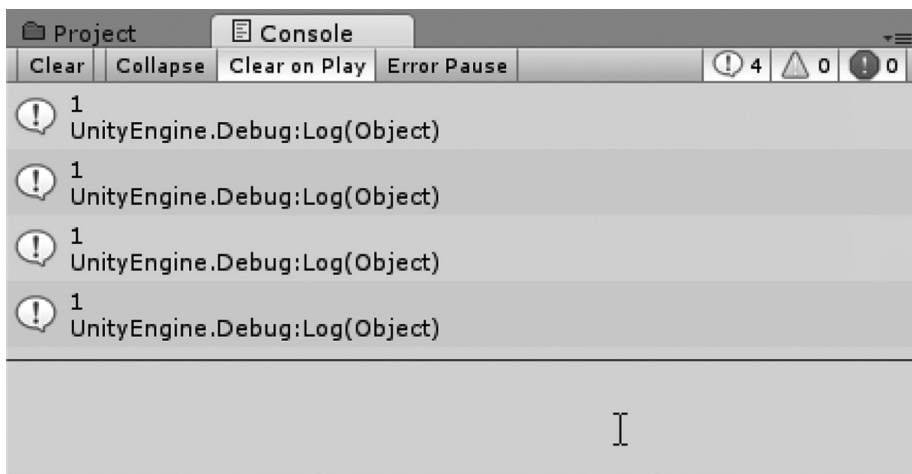
The `Debug.Log()` function in each Zombie's `Start ()` function prints out the `numZombies` int value. Each new instance of a Zombie increments the same `numZombies` variable.



Based on how many zombies are around, you can change a zombie's behavior, making them more aggressive, perhaps. Maybe only after five zombies have been spawned, they'll join one another and perform a dance routine. Having a shared property makes keeping track of a similar variable much easier.

```
using UnityEngine;
using System.Collections;
public class Zombie : MonoBehaviour {
    int numZombies;//no longer static
    //Use this for initialization
    void Start () {
        numZombies++;
        Debug.Log(numZombies);
    }
}
```

When the same code is run without `static` before the `numZombies` variable, the following output is the result:



The `numZombies` value is independent to each instance of the zombie class. When each new `Zombie` instance is created its `numZombies` `int` is initialized independently to 0. When the `Start ()` function is called in each zombie instance the `numZombies` value is incremented independently from 0 to 1. Therefore when white, rob and stubbs call their own `Start ()` function their own instance of the `numZombies` `int` is incremented, `numZombies` doesn't get incremented for all zombies because it's not static.

This is an interesting concept here. Up to this point, we've been thinking of classes as encapsulated independent entities. With the `static` keyword, there's a way to keep the separated objects connected to one another through a variable. This raises the question: Can this be done with functions?

5.5.3 Static Functions

Static functions act on behalf of all instances of the class they are a member of. They work only with static variables of the class. If we extend our zombie class with a new static function, we can use the zombie

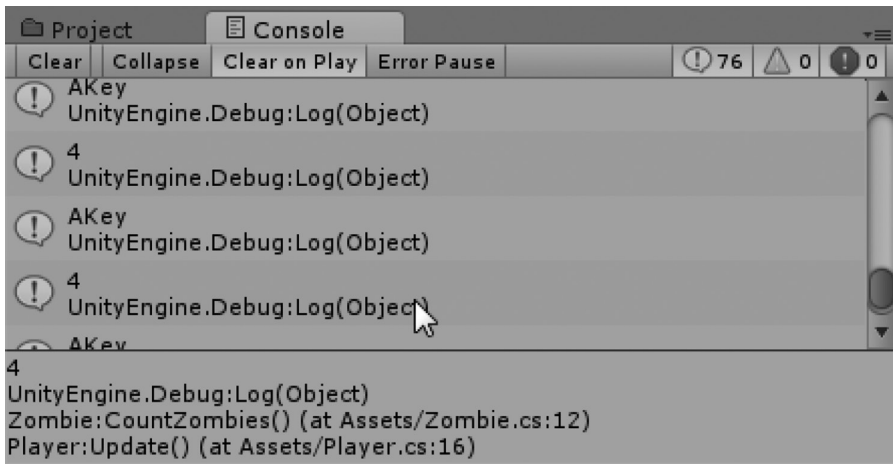
class to call a static function that can read static variables. To make the static function accessible to other classes we need to make the static function public as well.

```
using UnityEngine;
using System.Collections;
public class Zombie : MonoBehaviour
{
    static int numZombies;
    //int numZombies;//no longer static
    //Use this for initialization
    void Start ()
    {
        numZombies++;
        Debug.Log(numZombies);
    }
    //Logs number of zombies to the console
    public static void CountZombies()
    {
        Debug.Log(numZombies);
    }
}
```

Then we'll add a call to `Zombie.CountZombies()`; to find out how many zombies have been created in the `Update ()` function of the `Player` class, by adding it to the `if (AKey)` code block.

```
using UnityEngine;
using System.Collections;
public class Player : MonoBehaviour
{
    //Use this for initialization
    void Start ()
    {
    }
    //Update is called once per frame
    void Update ()
    {
        //add in a check for the A key
        bool AKey = Input.GetKey(KeyCode.A);
        if (AKey)
        {
            Debug.Log("AKey");
            //calls the static function in Zombie
            Zombie.CountZombies();
        }
    }
}
```

Now when the A key on the keyboard is pressed, we'll get a count printed to the Console panel indicating the number of zombies in the scene along with the `Debug.Log("AKey");` console log.



The 4 being printed out comes from the `public static void CountZombies();` function call being made to `Zombie`, not from any of the instances of the zombie class attached to the capsules in the scene.

To call static functions, you use the class name, not the name of an instanced object made from the zombie class. As you can see, there's more than one way the class identifier is used. Up to this point, we've been using the class name mostly as a way to create an instance of that class. With a static function or variable, the class identifier is used to access that static member of the class.

5.5.4 Putting It All Together

To complete our `Player` class we should add the rest of the key presses required to move. And add in a few places to store and use the position of the player.

```
public class Player : MonoBehaviour {
    //store the position of the player
    Vector3 pos;
    //Use this for initialization
    void Start () {
        //set the position to where we start off in the scene
        pos = transform.position;
    }
    //Update is called once per frame
    void Update () {
        bool WKey = Input.GetKey(KeyCode.W);
        bool SKey = Input.GetKey(KeyCode.S);
        bool AKey = Input.GetKey(KeyCode.A);
        bool DKey = Input.GetKey(KeyCode.D);
        if(WKey) {
            pos.z += 0.1f;
        }
    }
}
```

```

        if(SKey) {
            pos.z -= 0.1f;
        }
        if(AKey) {
            pos.x -= 0.1f;
        }
        if(DKey) {
            pos.x += 0.1f;
        }
        gameObject.transform.position = pos;
    }
}

```

At the class level, we need to create a new `Vector3` variable to update each frame. Therefore, `Vector3 pos;` will store a new `Vector3`, using which we'll be able to update individual values. In the `Start ()` function of the player class, we'll want to set the initial value of `pos` to where the object starts in the scene. The value for that is stored in `transform.position`; this goes the same for almost any `GameObject` that Unity 3D has in a scene.

With the start of any task, we're in need of collecting and holding onto data. Therefore, in the `Zombie.cs` class, we're going to add in a few new variables.

```

public static int numZombies;
public bool die;
public GameObject player;
public float speed = 0.01f;

```

We're going to use `bool die` to destroy a zombie when he gets too close to the player. Perhaps the zombies are self-destructive zombies; we'll just go with a simple method to decrement the `numZombies` value for now. Then, we're going to need to keep track of the player with a `GameObject player` variable. Making the `bool die` public, we'll be able to test this from the editor.

To fill in the data that we created, we're going to need to use the `Start ()` function of `zombie`.

```

void Start ()
{
    player = GameObject.Find("Main Camera");
    numZombies++;
    Debug.Log(numZombies);
}

```

So we're adding `player = GameObject.Find("Main Camera");` to use the `GameObject`'s static function called `Find()`. It's not important at the moment to know how or why this function works. We're more interested in how to use the `player` object, or rather the `Main Camera` in the scene.

Once we've added those lines, it's time to add some work to the `Update ()` function in the `zombie` class.

```

void Update ()
{
    Vector3 direction =
        (player.transform.position - transform.position).normalized;
    float distance =
        (player.transform.position - transform.position).magnitude;
}

```

We're going to want to get a couple of bits of more information that need to be updated in each frame. First is `direction`; we get the `Vector3 direction` by subtracting the player's position from the zombie's position. This value is then `.normalized`; to give us a value constrained to a value which doesn't exceed a value of 1 in any direction.

After that, we need to get the distance from the zombie to the player. This is done by calculating the player's position minus the zombie's position; we then use `.magnitude;` to convert that `vector3` to a `float distance`. We'll go further into how these sorts of functions work later on.

After building up some data for the rest of the function we use, we can then set up some statements to take action.

```
Vector3 move =
transform.position + (direction * speed);
transform.position = move;
```

Next, we'll take `direction` and then multiply it by one of the variables we set up ahead of time. `float speed = 0.01f;` is being used to multiply `direction`, such that `move` is slowed down. Therefore, we create a new variable called `move`, then set the `transform.position` of the zombie to `transform.position + (direction * speed);`, which will move the zombie toward the player, slowly.

From there, we can then use the distance to check for when to make the zombie die.

```
if(distance < 1f)
{
    die = true;
}
```

We'll just make an arbitrary distance of `1f`. Therefore, if the distance is less than this number, we set `die` to `true;` to complete the statement. When `die` was declared at the beginning of the class `public bool die;` it was automatically initialized to `false`. To be more clear, we could have used `public bool die = false;`, but it's unnecessary as new `bool` values are always initialized to `false` when they're created, unless dynamically initialized otherwise.

```
if(die)
{
    numZombies--;
    Destroy(gameObject);
}
```

Last, setting `if (die)` is true, we will then decrement `numZombies` by 1 using `numZombies--;`. This takes the previous value of `numZombies`, reduces it by 1, and then sets `numZombies` to the new value of `numZombies - 1`. After that, we use the `Destroy()` function to destroy the `gameObject` that the script is attached to.

The complete `Update ()` function in `Zombie.cs` should look like the following.

```
void Update ()
{
    Vector3 direction =
    (player.transform.position - transform.position).normalized;
    float distance =
    (player.transform.position - transform.position).magnitude;
    Vector3 move = transform.position + (direction * speed);
    transform.position = move;
    if (distance < 1f)
    {
        die = true;
    }
    if (die)
    {
        numZombies--;
        Destroy(gameObject);
    }
}
```

This code creates a movement that pushes the capsule toward the Main Camera in the scene. To show that `numZombies` static is accessible to any new script in the scene, we'll create a new `ZombieSpawner.cs` file and drop it into the scene with a new cube object.

I've created a new cube by selecting `GameObject → Create Other → Cube`. After the cube is dropped into the scene, I moved it back to the center of the world by setting `x`, `y`, and `z` to 0 under the Transform component in the Inspector panel. After that, I added a new script to it called `ZombieSpawner.cs`.

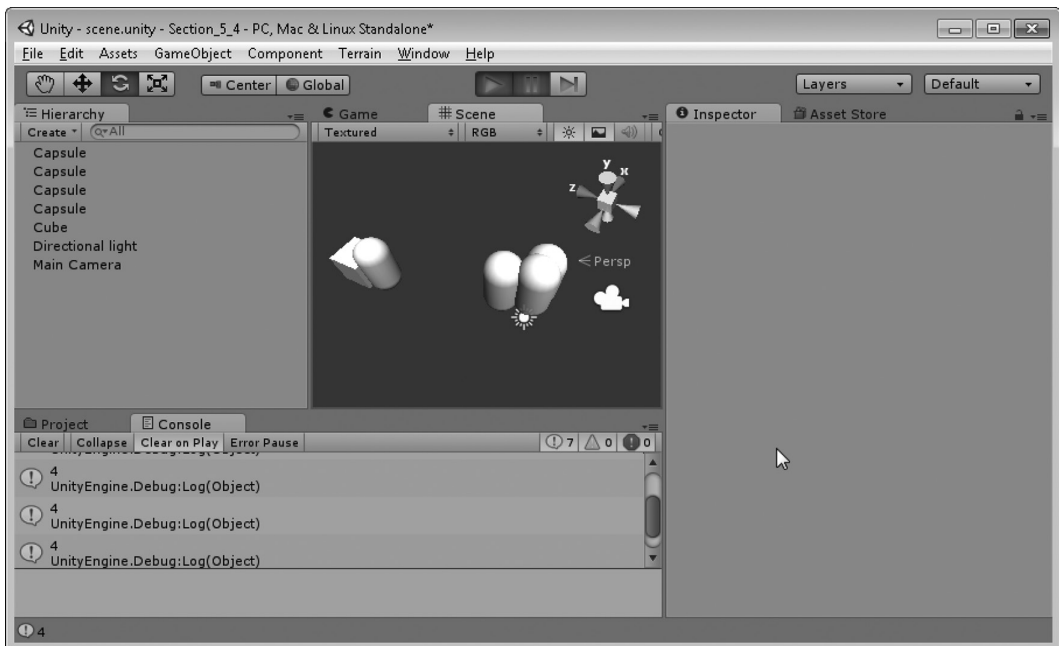
In the new `.cs` file, I've added the following code to the `Update ()` function.

```
void Update ()
{
    if (Zombie.numZombies < 4)
    {
        GameObject go =
        GameObject.CreatePrimitive(PrimitiveType.Capsule);
        go.AddComponent(typeof(Zombie));
        go.transform.position = transform.position;
    }
}
```

In the `if(Zombie.numZombies < 4)` statement, check the static `numZombies` value in the zombie class. This is a direct access to that value from `ZombieSpawner.cs`, which is possible only because the value is set to `static`, because of which we've got the ability to check its value from any other class.

When the value is less than 4, we follow the statement that tells the script to create a new `GameObject` `go`. This creates a new `PrimitiveType.Capsule`, which then has a `Zombie` component added to it.

When you click the play icon at the top to begin the game in the editor, we'll get the following behavior.



When you check out the Scene panel with the game running, you'll be able to watch the cylinders move toward the Main Camera object. When they get close enough, they pop out of existence. As soon as the capsule is gone, another one is spawned at the cube object.

5.5.5 What We've Learned

Using static functions certainly layers on quite a distinctive level of complexity on an already complex understanding of what a class is. Not only do classes have functions talking to one another, they also have the ability to share information. It's important to know that you can have classes without any static variables or functions at all.

You should use a static variable only if all of the classes of the same type need to share a common parameter. This goes the same for a static function. We'll go further into how static functions can help in Section 6.11, when we get back to some of the more complex behaviors, but for now, we'll leave off here.

Moving forward, we'll want to keep in mind that C# has static variables and functions for a specific reason. There are efficiency and optimization considerations to keep in mind as well. For instance, the player's `transform.position` is the same for each zombie. Each zombie doesn't need to have its own instance variable for the player. We could have easily converted the `Vector3 pos;` in the player class to a static value.

```
public static Vector3 pos;
```

The static `Vector3 pos;` variable could then be used by the zombies in the following manner.

```
Vector3 direction = (Player.pos - transform.position).normalized;  
float distance = (Player.pos - transform.position).magnitude;
```

This certainly works, and it might be considered easier to read. This would even make the `GameObject player;` obsolete since `Player.pos` is a static value, and you don't need to use `GameObject.Find()` to begin with. This type of optimization should be thought of as you look at your code and discover new ways to make it more efficient and easier to read.

5.6 Turning Ideas into Code—Part 2

We've gotten pretty far. At this point, we're about ready for some everyday C# programming uses. To get to the point, beyond the lessons here, we're already capable of putting all of the lessons we've learned to this point to get the beginnings of a simple game going.

To start with, I've added a plane, a directional light, and some cubes to my scene found in the FPSControl project. Each object was positioned using the editor so that the Main Camera has something to look at. The objects were added using the GameObject menu.



This gives us a basic scene to get started with. To build the very basics of a first-person camera controller for the Main Camera, we'll start off by adding a `FPSCaim.cs` to the Main Camera object. I like using the Add Component button in the Inspector panel.



This will get us a new C# script to get started with. To move the camera, we'll want to get a feel for the sorts of data we'll be using to make the camera move around. First, we'll want to check out what data the Input class will give us. Starting off with `Input.mousePosition`, we can see that we'll want a `Vector3`. Input has many static variables and functions we can use; we'll look at the other available variables and functions in a moment.

```
// Update is called once per frame
void Update()
{
    Vector3 mousePosition = Input.mousePosition;
}

public static Vector3 mousePosition { get; }
```

Therefore, a `Vector3 mousePosition` will be useful here. However, a mouse is a 2D device, so what does the data coming out of the mouse look like?

```

void Update ()
{
    Vector3 mousePosition = Input.mousePosition;
    Debug.Log(mousePosition);
}

```

Add in a `Debug.Log()` for the `mousePosition` that we have assigned to the `Input.mousePosition`. Remember that the variable `mousePosition` we're assigning to the `Input.mousePosition` and the latter need to have matching types. With this `Debug.Log()` running, we get many numbers streaming out on the x and the y, with no z values coming in other than 0.0. This is pretty much what I'd expect; my mouse doesn't have any way to detect how far from the mouse pad the mouse is, so I wouldn't expect any numbers from the z value. Therefore, it's time to isolate the data into a more simple-to-use variable.

```

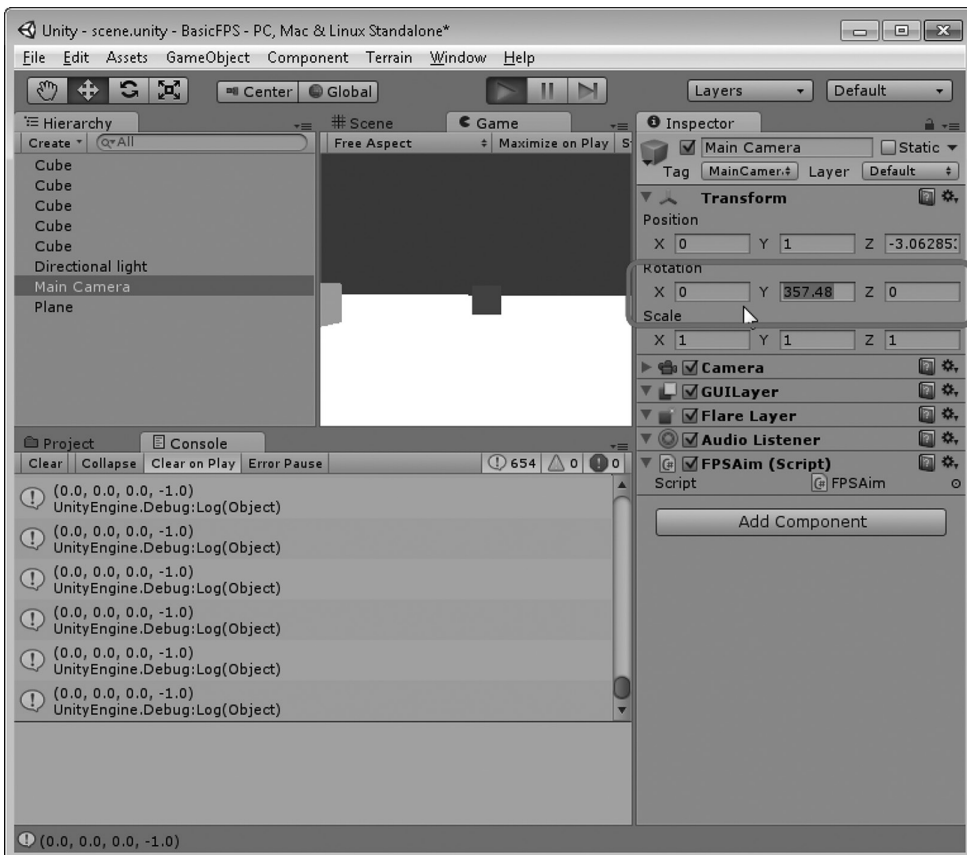
void Update ()
{
    Vector3 mousePosition = Input.mousePosition;
    float mouseX = mousePosition.x;
    float mouseY = mousePosition.y;
}

```

Therefore, now, I've boiled down the incoming `mousePosition` to a new `mouseX` and a `mouseY` so that I can more easily deal with each axis on its own. Now, we should look at the local rotation of the camera.

```
Debug.Log(transform.localRotation);
```

This gives us an interesting output. Thanks to our working within a game editor, we can fiddle with the camera in the game. Click the play icon and we'll switch over to the Game view. In the Inspector panel, we can play with the different values in the Transform component of the camera.



When we drag the Rotation Y value, we see the numbers streaming out of the Console panel. The rotations of an object are stored as a quaternion rotation value. Quaternions are special types of vectors that have four values. These values are used to both rotate and orient an object. With an xyz rotation, you're left with an ambiguous upward direction. Without going too deep into quaternion mathematics and topics like the gimbal lock, we'll digress and get back to using Euler rotations, something much easier to deal with.

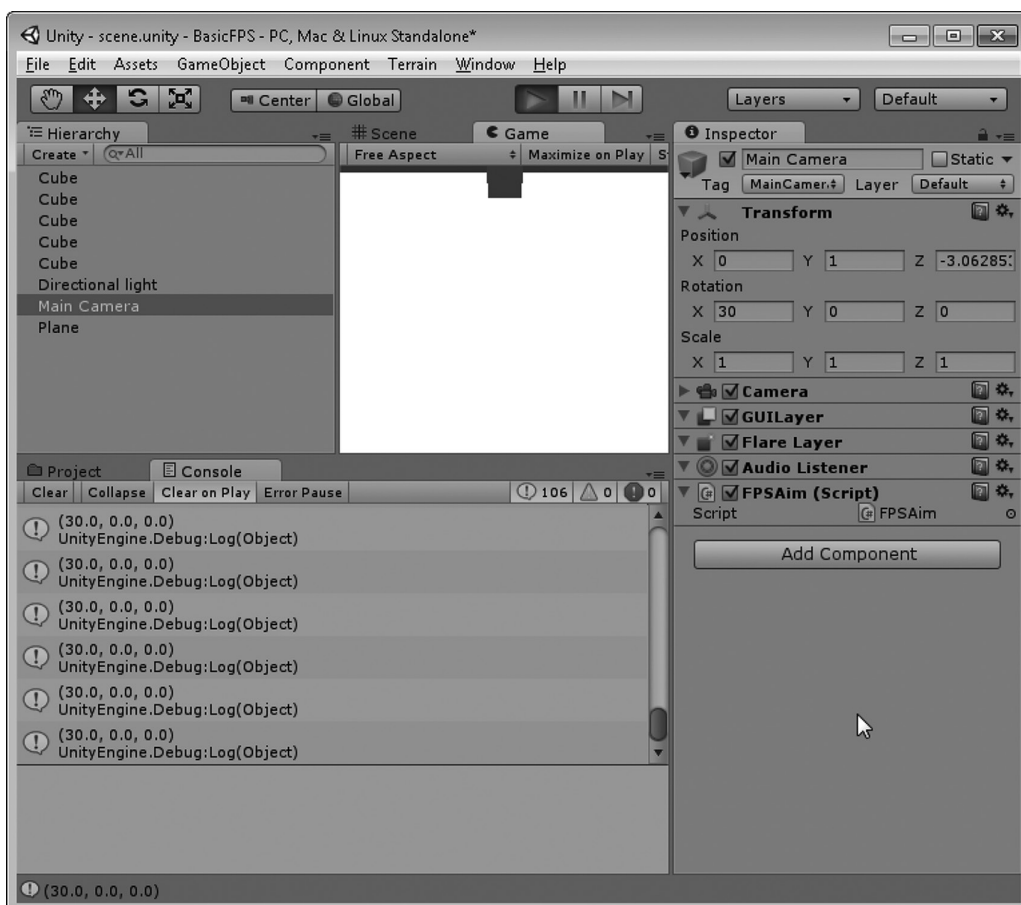
Thankfully, Unity 3D programmers know that not everyone understands quaternions so well.

```
Debug.Log(transform.eulerAngles);
```

There's a `transform.eulerAngles` value that we can read instead of quaternions. From this data, we get numbers that reflect the values, which we see in the Transform component in the Inspector panel. Putting the mouse together with the `transform.eulerAngles` of the camera, we're able to aim the camera. However, first, how can we effect the camera's rotation using `eulerAngles`?

```
transform.eulerAngles = new Vector3(30,0,0);
```

To test the above function, we'll make a simple statement to set the `transform.eulerAngles` to a new `Vector3(30, 0, 0)`; to see if this has the desired effect.



Running again tells us that yes we can aim the camera up and down using this number. We could simply map the `mouseY` to the `eulerAngles X` and see what this does.

```
transform.eulerAngles = new Vector3(mouseY,0,0);
```

However, this makes the camera move up and down a lot more than what we might want. We could reduce the amount of movement the `mouseY` has by multiplying it by a small value.

```
transform.eulerAngles = new Vector3(mouseY * 0.1f, 0, 0);
```

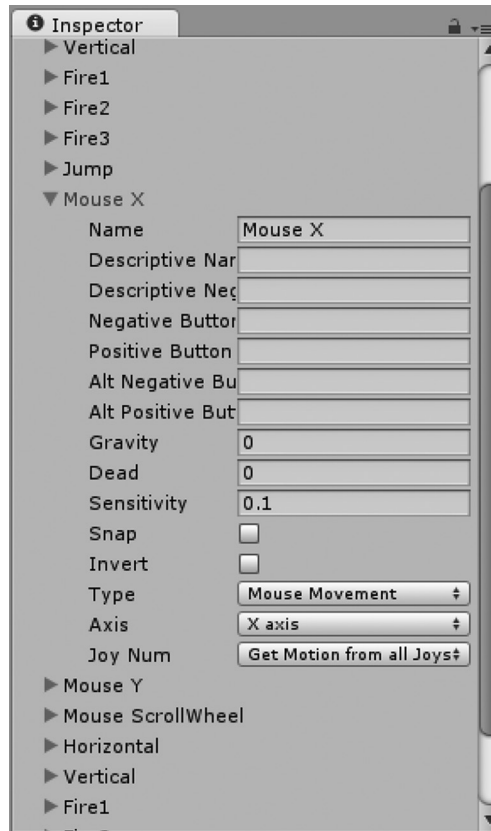
However, this doesn't have the right effect either. With this, we're limited by the size of our monitor. When the mouse gets to the top of the monitor, we're stuck and can't look up or down any further than our monitor lets us. Perhaps we've got the wrong data!

5.6.1 Input Manager

Where is Input coming in from anyway?



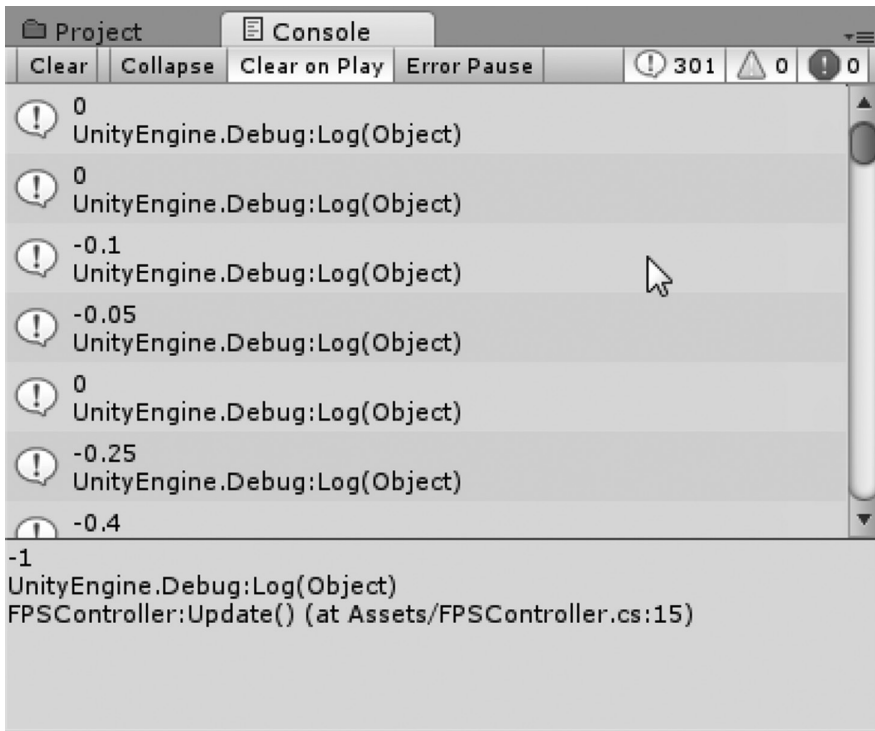
Hunting around in the Unity 3D editor, we find the Input manager at following menu path: `Edit` → `Project Settings` → `Input`; with this selected, we get an update in the Inspector panel. This shows us some interesting bits of information.



Hey, it's a `Mouse X`, as well as several other types of input. Perhaps we should be looking at this for our Input data. Therefore, before we get too far, it should be clear that most of this is findable with a few short web searches for Unity 3D Input manager queries. We find in the Unity 3D manual an example showing us how to use `Input.GetAxis("Mouse X");`, so we should start there.

```
void Update ()
{
    float mouseX = Input.GetAxis("Mouse X");
    Debug.Log(mouseX);
    transform.eulerAngles = new Vector3(0, mouseX, 0);
}
```

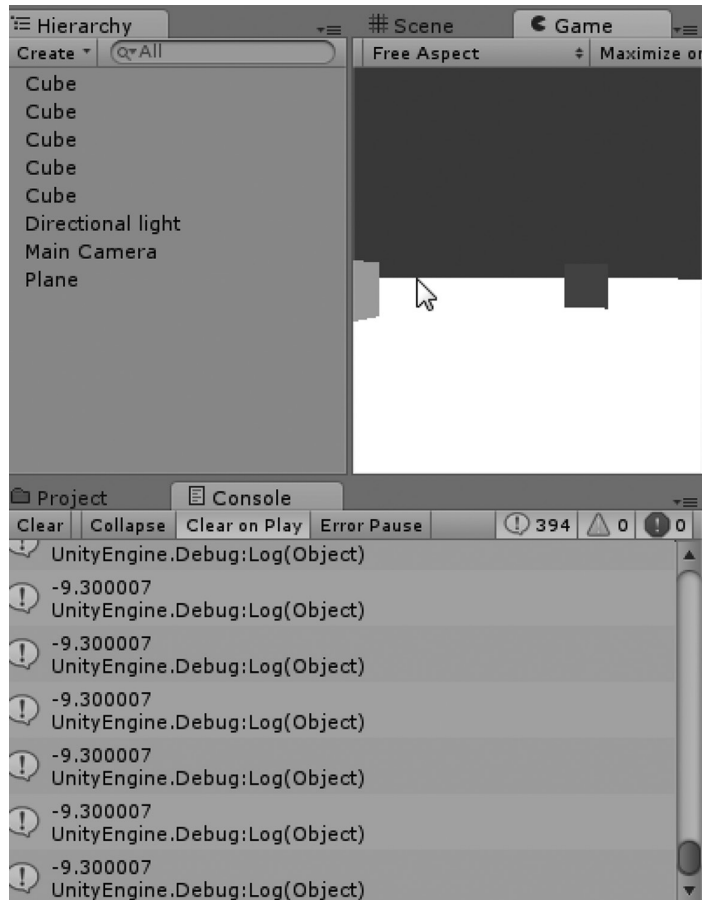
If we try to use the code as is, we'll get some sort of jittery movement in the camera as we move the mouse left and right. Looking at the Console panel gives us an interesting set of numbers coming out of the `mouseX` variable.



We see a fluctuation between 0 and -0.1 or more when moving in one direction and positive values when moving in the other. Perhaps this means that we should be adding the number to the rotation of the camera rather than trying to use it as a set value. To make this change, we'll keep the `mouseX` value around beyond the scope of the `Update ()` function.

```
float mouseX;  
void Update () {  
    mouseX += Input.GetAxis("Mouse X");  
    Debug.Log(mouseX);  
    transform.eulerAngles = new Vector3(0, mouseX, 0);  
}
```

This means we'll have a class variable that hangs around and isn't reset every time the `Update ()` function is called. Then we change the `mouseX = Input` to `mouseX += Input.GetAxis("Mouse X");`, which will add the value of the `Mouse X` input to `mouseX`.



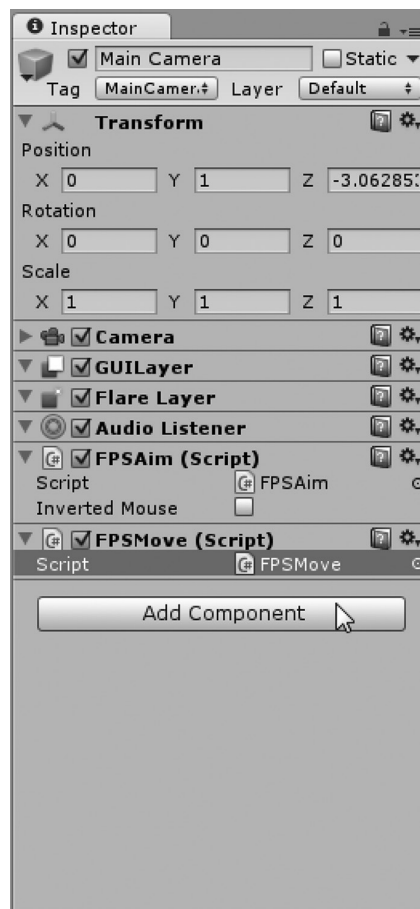
Testing this again gives us a more likeable behavior. Now that we have this working, we should make the same setup for mouseY.

```
float mouseX;
float mouseY;
void Update ()
{
    mouseX += Input.GetAxis("Mouse X");
    mouseY += Input.GetAxis("Mouse Y");
    transform.eulerAngles = new Vector3(mouseY, mouseX, 0);
}
```

So far so good. However, now we have up and down inverted. If you're cool with this, then there's no problem, but universally, there's bound to be a preference one way or the other. To avoid this, its best you add in an option for the player to pick using an inverted mouse or not.

```
float mouseX;
float mouseY;
public bool InvertedMouse;
void Update ()
{
    mouseX += Input.GetAxis("Mouse X");
    if (InvertedMouse)
    {
        mouseY += Input.GetAxis("Mouse Y");
    } else
    {
        mouseY -= Input.GetAxis("Mouse Y");
    }
    Debug.Log(mouseX);
    transform.eulerAngles = new Vector3(mouseY, mouseX, 0);
}
```

Let's add in an option to allow the user to invert the "Mouse Y" movement. We're still in need of some sort of movement that can be controlled with the keyboard. However, we should get used to the idea of separating our code into different classes, so we'll create a new class just for taking in input from the keyboard.



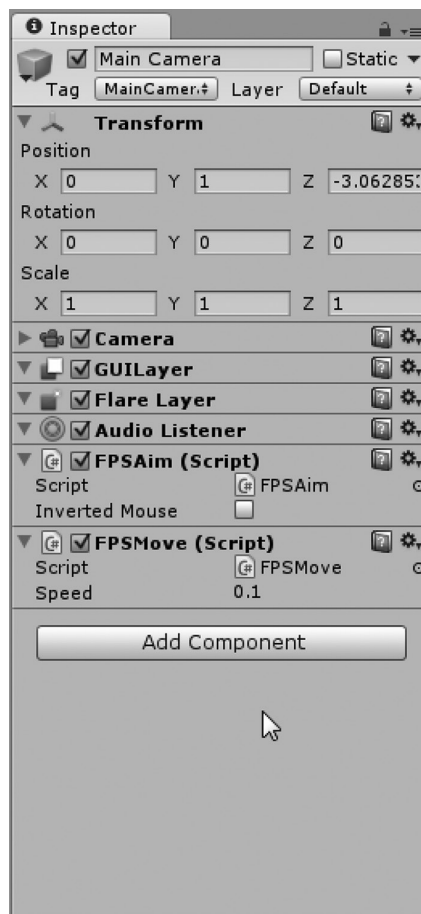
Using a preferred method, I've added in a new C# class called `FPSMove.cs` to the Main Camera. Within `FPSMove.cs`, we can check for an `Input.GetKey()` to check for any desired keyboard input.

```
void Update ()
{
    if (Input.GetKey(KeyCode.W))
    {
        transform.position += transform.forward;
    }
}
```

We'll use this to add to our `transform.position` a `transform.forward`, but wow, we are moving fast! We should fix this with some sort of speed multiplier.

```
public float speed;
void Update ()
{
    if (Input.GetKey(KeyCode.W))
    {
        transform.position += transform.forward * speed;
    }
}
```

By making the speed variable public, we can play with the value in the editor.

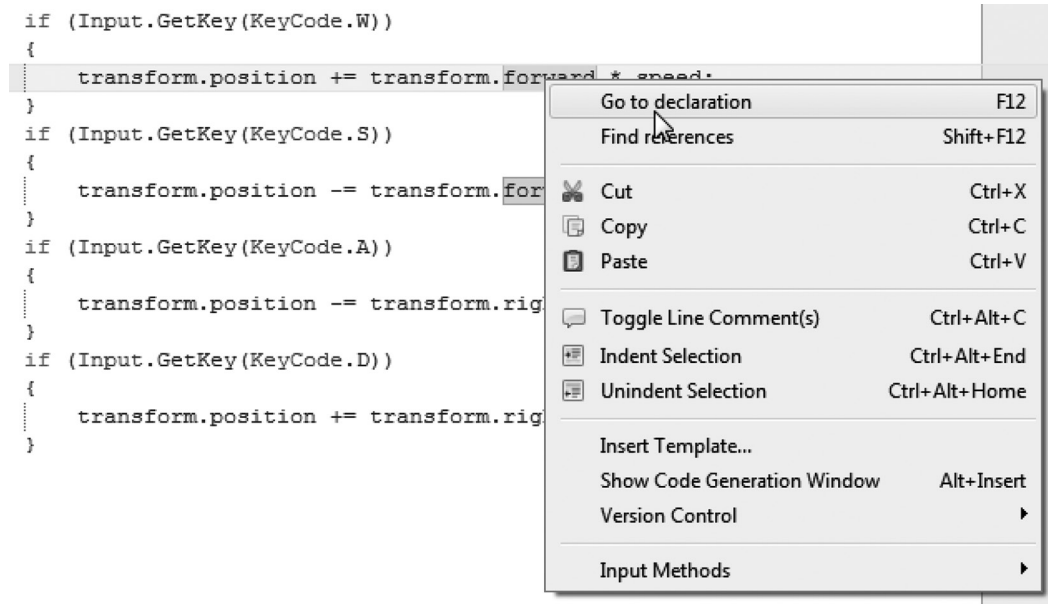


This code will allow us to slowdown the movement to a more controllable rate. With a bit of fiddling, we can get to the final FPSMove.cs code in the Update () function.

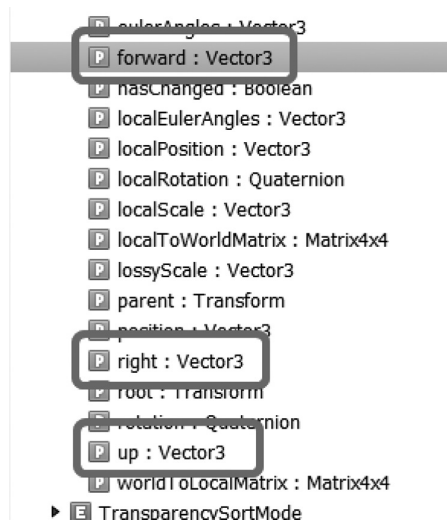
```
public float speed = 0.1f;
void Update ()
{
    if (Input.GetKey(KeyCode.W))
    {
        transform.position += transform.forward * speed;
    }
    if (Input.GetKey(KeyCode.S))
    {
        transform.position -= transform.forward * speed;
    }
    if (Input.GetKey(KeyCode.A))
    {
        transform.position -= transform.right * speed;
    }
    if (Input.GetKey(KeyCode.D))
    {
        transform.position += transform.right * speed;
    }
}
```

We've taken the transform.forward and used - = to move backward and + = to move forward. Likewise, since there's no transform.left, we need to use transform.right. Then, use transform.position - = to move left and + = to move to the right.

Therefore, where did transform.forward and transform.right come from? If we look at the transform.forward static value in Transform, we can go to the forward declaration.

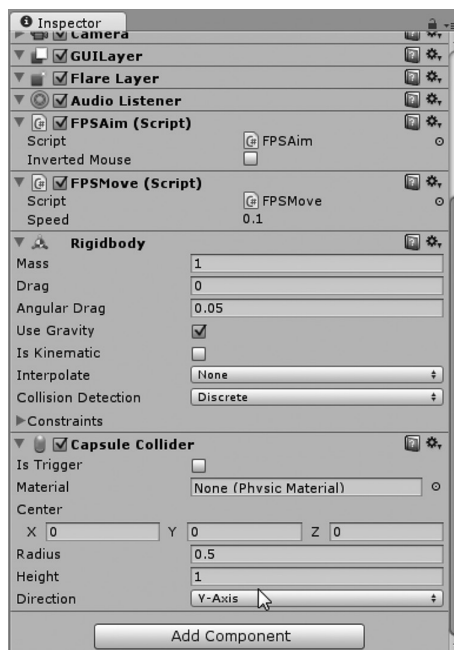


From here, we'll be taken to the Assembly Browser. This works for transform.rotation as well.



We'll notice that there's also a `right` and an `up`. Many other static values are found here. It's important to learn how Unity 3D has organized the data in each class such that we can make use of them. Remember, this is a game engine, and as such we have access to many of the commonly required forms of data that game designers need. This data structure is an example of useful data, common only in well-written game engines.

Therefore, now we have the beginnings of a fairly useful pair of C# files that we can attach to any object in a scene. However, first, we should limit the camera from going through the ground. If we add a `Rigidbody` and a `Capsule Collider` to the Main Camera, the object will collide with the ground and anything else with a physics collider.



Physics colliders are complex data structures that solidify objects in a game. We can use these structures to make objects feel and act heavy. You might notice that with our new setup, we're sliding around a little bit like we're walking around on ice. This is because our `Rigidbody` has a `Drag` of 0 by default. Changing this to 1 will keep the camera from sliding around so much.

At this point, we've got a pretty usable beginning to a first-person camera. We can keep adding in new variables to make this more sophisticated, but for now, we'll leave off here. I'll leave the cool stuff up to you. Perhaps a speed variable on the `FPSAim` would be called for to make the mouse aim quicker.

5.6.2 What We've Learned

The process of writing code often takes a bit of getting used to. This exercise involves a great deal of looking at and testing various sources for data and trying to find results that we can use. By setting up and testing `Debug.Log()`, we can check to see if the data we want is going to be the data we can use.

Before we get too far, it's important to search the Internet for solutions that others have come up with. From their code, we can extract some ideas we can use for our own purposes. Even something as simple as finding the name of a function can help us test and try out a function to see if it's something we can use.

The process might be repetitive, but it's a lot like drawing or painting on a canvas. The first line drawn is usually covered by iterative layers of paint covering one idea with another. One concept to come away with here is the fact that we separated the mouse aim from the keyboard movement. Why was this done, and is it the best way to write C# in general?

There are always going to be arguments for writing a single large set of code versus many smaller more modular pieces of code. One argument for smaller files is the fact that each one can stand on its own. This means you can try out different forms of aim controls and mix them with the keyboard controls without having to rewrite the keyboard control.

However, should you need to have the two interact with one another, you might need to find more clever ways to have separate objects communicate with one another. In this case, a single file is more straightforward. However, there are simple ways for objects to talk to one another.

This also means you can add in a new script for, say, `FPSPrimaryWeapon.cs` that handles the left mouse button. Perhaps an `FPSJump.cs` and maybe even a `FPSSecondaryWeapon.cs` can be added. This means that as you add in new functions, you add in new CS files. As you come up with new versions of each function, you can handle the updates and changes without having to make changes to the other files—make your code modular.

5.7 Jump Statements

Suppose you're in the process of looking for a specific color of crayon in a large assortment of art tools. After grabbing handfuls of different pencils and pens, you find the crayon you want and stop looking through the rest of your bin. You've just used a jump statement, and returned to your work with the colored crayon of your desire.

The keywords `break` and `continue` come in particularly handy when sorting through a list of objects. Like `return`, these keywords stop the function from further execution or they jump to a different part of the same function. `return` has the additional feature of coming out of a function with a value. We have looked at these keywords briefly before, but it's important to get a better understanding of `break`, `continue`, and `return` in use.

`continue` in particular restarts a loop from the top and then continues the function again, whereas `break` stops the block of code it's in altogether.

5.7.1 Return

We need to love the `return` keyword. This keyword turns a function into data. There are a couple of conditions that need to be met before this will work. So far, we've been using the keyword `void` to declare the return type of a function. This looks like the following code fragment.

```
void MyFunction()
{
    //code here...
}
```

In this case, using `return` will be pretty simple.

```
void MyFunction()
{
    //code here...
    return;
}
```

This function returns `void`. This statement has a deeper meaning. Returning a value makes a lot more sense when a real value, something other than a `void`, is actually returned. Let's take a look at a function that has more meaning.

The keyword `void` at the beginning of the function declaration means that this function does not have a return type. If we change the declaration, we need to ensure that there is a returned value that matches the declaration. This can be as simple as the following code fragment.

```
int MyFunction()
{
    //code here...
    return 1; //1 is an int
}
```

This function returns `int 1`. Declaring a function with a return value requires that the return type and the declaration match. When the function is used, it should be treated like a data type that matches the function's return type.

5.7.1.1 A Basic Example

Here is a quick review class that has a function declared as `int MyNumber()` in the `Start ()` function.

```
using UnityEngine;
using System.Collections;
public class Example : MonoBehaviour {
    int MyNumber() {
        return 7;
    }
    //Use this for initialization
    void Start () {
        int a = MyNumber();
        print (a);
    }
    //Update is called once per frame
    void Update () {
    }
}
```

When this code block is attached to the Main Camera in the scene, 7 is printed to the Console panel. The function `MyNumber()` returned 7 when it was called. When we add some parameters to the function, we can make the `return` statement much more useful.

```
int MyAdd(int a, int b) {
    return a + b;
}
//Use this for initialization
void Start () {
    int a = MyAdd(6, 7);
    print (a);
}
```

In this fragment, we have `int MyAdd(int a, int b)`, which we then assign to `int a` in the `Start ()` function. This prints 13 to the console when run. We can skip a step to make the code a bit shorter.

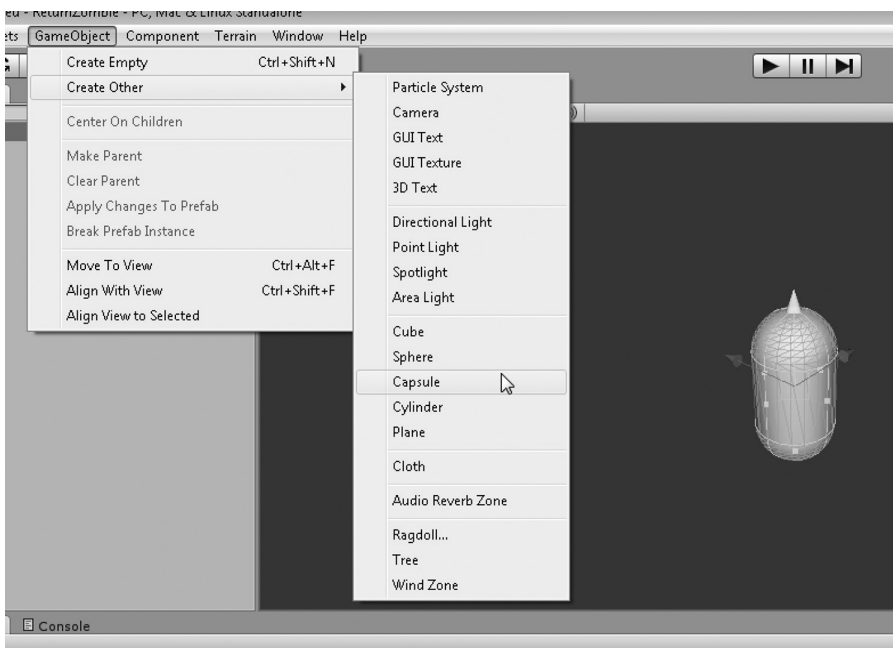
```
void Start () {
    print (MyAdd(6, 7));
}
```

This code produces the same 13 being printed to the Console panel and should make it clear that the function can easily be used as a value.

5.7.2 Returning Objects

We can get something more interesting out of a function once it begins to return something more substantial than a simple number—this would be a better way to use `return`. Therefore, in the case of getting a zombie from a function, we'd want to define a zombie first.

As a simple example, select `GameObject` → `Create Other` → `Capsule` to drop a simple capsule in the scene.



From here, we're going to want to add a script component to the cylinder. In the Inspector, select Add Component → New Script, and then enter *Zombie*, to name the new script and make sure the type is set to CSharp. This will represent what will eventually turn into zombie behavior. Of course, the work of writing proper artificial intelligence for zombies is something that will have to wait till after this exercise. This creates a new *Zombie.cs* class, which is now a component of the capsule *GameObject* in the scene.

Open the *JumpStatements* project and open the scene found in the Assets Directory in the Projects panel. On the Main Camera, attach a new script called *ReturnZombie*, which will serve as the example for testing a function that will give us the zombie in the scene. In the *ReturnZombie* script, we'll add a function that returns a *Zombie*.

```
using UnityEngine;
using System.Collections;
public class ReturnZombie : MonoBehaviour
{
    //Use this for initialization
    void Start ()
    {
    }
    //Update is called once per frame
    void Update ()
    {
    }
    //returns a zombie
    Zombie GetZombie()
    {
        return (Zombie)GameObject.FindObjectOfType(typeof(Zombie));
    }
}
```

5.7.3 A Class Is a Type

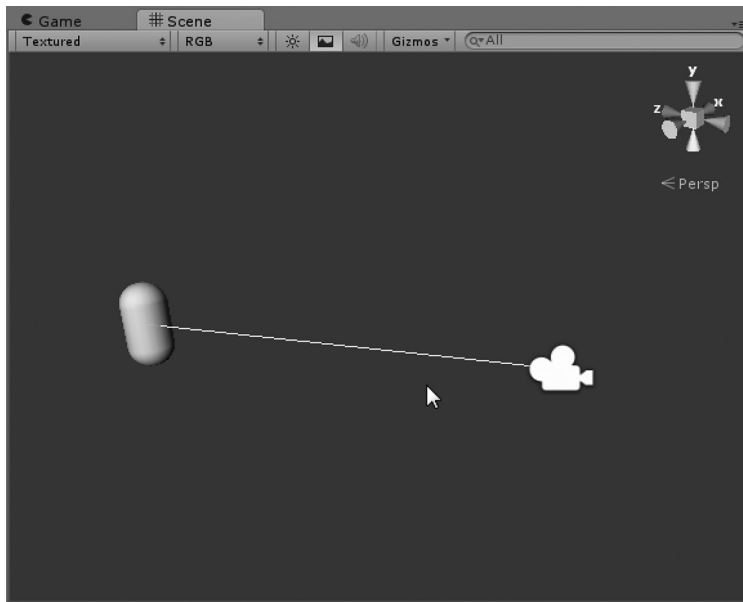
Our function that returns a *Zombie*, is at the end of our Unity 3D C# default template. We've called our function *GetZombie()*, and we're using a simple function provided in *GameObject* called *FindObjectOfType()*, which requires a *type* to return. The different classes we create become a new type. Much as an *int* is a type, a *Zombie* is now a type based on the fact that we created a new class called *Zombie*. This goes the same for every class we've created, which are all types.

To inform the *FindObjectOfType()* function of the type, we use the function *typeof(Zombie)*, which tells the function that we're looking for a type *Zombie*, not an actual zombie. There's a subtle difference between telling someone we'd like a thing that is a kind of zombie, not a specific zombie. We'll get into the specifics of types in Section 6.5.3 that focuses on dealing with types, but we're more interested in return at the moment, so we'll get back to that.

After making sure that we're getting a *Zombie* from the function, we use *return* in the function to fulfill the *Zombie GetZombie()* return type. We're not using *void* since our return actually has something to return. Using this becomes quite interesting if we add a use to the function. In the *ReturnZombie Update ()* function, add in the code to draw a debug line from the camera to the function.

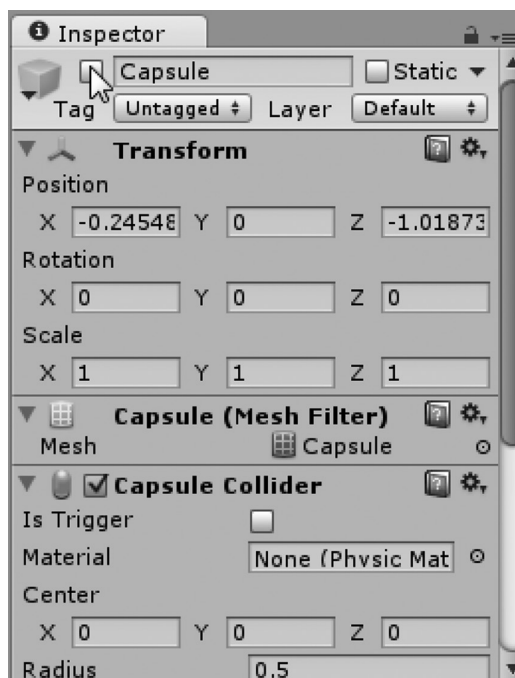
```
//Update is called once per frame
void Update ()
{
    Debug.DrawLine(transform.position,
        GetZombie().transform.position);
}
```

`Debug.DrawLine()` starts at one position and goes to another position. The second position is a function's `transform.position`. If you look into the Scene panel, you'll see a white line being drawn between the camera and the capsule with the Zombie component added to it.



This remarkably means that the `GetZombie().transform.position` is returning the value as though the function is the Zombie itself, which has some interesting consequences. Having classes, or rather objects, communicate between one another is a particularly important feature of any programming language. Accessing members of another class through a function is just one way our objects can talk to one another.

This reliance on the function breaks if there are no zombies in the scene. If we were to deactivate the capsule by checking it off in the Inspector panel, we'd effectively remove the capsule from existence.



Unchecking here while the game is running will immediately bring up an error.

```
NullReferenceException: Object reference not set to an instance of an object  
ReturnZombie.Update () (at Assets/ReturnZombie.cs:12)
```

A `NullReference` is caused when the return type of the function has nothing to return. Errors aren't good for performance and we should handle the problem more effectively. We need to make our use of the function more robust.

```
void Update ()  
{  
    Zombie target = GetZombie();  
    if (target != null)  
    {  
        Debug.DrawLine(transform.position,  
            target.transform.position);  
    }  
}
```

If we add in a check to see if `Zombie` is null, we'll be able to avoid giving the `Debug.DrawLine()` function nonexistent data. Therefore, we create a local variable to the function. Add in `Zombie target = GetZombie();` and now we have a variable that can be either null or a zombie.

5.7.4 Null Is Not Void

There is conceptual difference between void, which is *nothing*, and null, which is more like *not something*. The word *void* means there is no intention of anything to be presented, whereas *null* implies that there can be something. We may add `if(target != null)` and can say that target can be empty or it can be `Zombie`. When target isn't a zombie, target is null; when it is there, it's `Zombie`.

We may add `if(target != null)`—in more readable English, “if the target is not null”—and then conditionally execute the following code block. In the case where we have a target, we can draw a line from the camera's `transform.position` to the target's `transform.position`. This code alleviates the error, and thus we don't have to worry about sending `Draw.DebugLine` an error-inducing null reference.

5.7.5 What We've Learned

Functions that return class objects are quite useful. All of the properties that are associated with the class are accessible through the function that returns it. There are various jump statements that have different results.

The `return` statement has a clear purpose when it comes to jumping out of a function. Not all `return` jump statements need to be followed by a value. If the function is declared as `void MyFunction()`, the `return` statement only needs to be `return;` without any value following it.

We'll cover more jump statements in a following chapter.

5.8 Operators and Conditions

Conditional operators allow you to ask if more than one case is true in an `if` statement. If you decided to go for a swim on a cold day on the sky being clear, then you might end up freezing. Your *jump in a pool* `if` statement should include temperature in addition to the sky.

5.8.1 Conditional Operators && and ||

The `if` statement in pseudocode might look something like the following: If (temperature is hot and sky is clear) {go swimming}. To translate that into something more C# like, you might end up with the following.

```
void Start ()
{
    float temp = 90f;
    bool sunny = true;
    if (temp > 60 && sunny)
    {
        print("time to go swimming!");
    }
}
```

If the temperature is above 60 and it's sunny, then it's time to go swimming. Zombies might not have that sort of thought process, but your player might. Silly code examples aside, the `&&` operator is used to join together multiple conditions in the `if` statement shown. This operator is in a family of symbols called conditional operators, and this particular operator is the *and* operator.

5.8.1.1 A Basic Example

To get a better understanding how this operator works, we'll break things down into a simplified example. To follow along, start with the Conditional project and open the scene from the Assets Directory. We could start with two different `if` statements. This means we have to use more curly braces and lines of code to complete a fairly simple task.

```
//Use this for initialization
void Start ()
{
    if (true)
    {
        if (true)
        {
            print("this could be more simple.");
        }
    }
}
```

Using more than one `if` statement should look a bit clumsy. The reason for using conditional operators is to simplify the number of `if` statements you can use in a single line. Rather than spreading out various test conditions across multiple `if` statements, you can reduce the `if` statement into a smaller more clear statement.

```
void Start ()
{
    if (true && true)
    {
        print("both sides of the && are true");
    }
}
```

Inside of the `if` statement are two different boolean cases. Only if both sides of the *and* conditional operator are true will the `if` statement will execute. If either side is false, then the `if` statement will not be evaluated.

```
void Start ()
{
    if (false && true)
    {
        print("I won't print...");
    }
}
```

The above statement has one false statement and will not be evaluated. All of the conditions in the `if` statement's arguments need to be true for the encapsulated instructions to execute. This logic can be further extended with more than one conditional operator.

```
void Start ()
{
    if (true && true && true)
    {
        print("I will print!");
    }
    if (true && true && true && true && true && true && false)
    {
        print("I won't print!"); //one false at the end blew it
    }
}
```

Of course, there are no limits to how many arguments will be accepted in a single `if` statement. This absence of limit allows for more complex decision making, but we can make things even more interesting by adding in the other conditional operator *or*.

```
void Start ()
{
    if (true || false)
    {
        print("I will print!");
    }
}
```

The `||`, or double bar, is used to indicate the conditional *or* that is used to evaluate an `if` statement if either boolean is true. The only case where the *or* will not allow the `if` statement to evaluate is when both sides are false.

```
void Start ()
{
    if (false || false)
    {
        print("I won't print!");
    }
    if (false || false || true)
    {
        print("I will print!");
    }
    if (false || false || false || false || false || false || true)
    {
        print("I will print!"); //just needs one true to work!
    }
}
```

When both sides of the or operator are false, then the contents of the `if` statement will not be evaluated. If there are more than two statements, any one of the statements being true will execute the contained instructions. The and operator and the or operator can work together, but the logic may get muddy.

```
void Start ()
{
    if (false || true && true)
    {
        print("I will print!");
    }
}
```

It's hard to immediately guess what might happen when you first look at the code. To make things more clear, we can use parentheses in pairs in the `if` statement, like we did with numbers earlier.

```
void Start ()
{
    if ((false || true) && true)
    {
        print("I will print!");
    }
}
```

Evaluating this `if` statement works in a way similar to how math works. The `(false || true)` is evaluated to result in `true`. As long as there is at least one `true`, the `||`, or *or*, returns `true`. This then turns into `[true] && true` being evaluated for the `&&` and the conditional operator. This means that since both sides of the `&&`, or *and*, operator are true, the `if` statement will evaluate the code between the two curly braces.

Conditional operators are usually used when comparing numbers. Let's take a very contrived scenario, where we have an `enemyHealth` and `myHealth`.

```
void Start ()
{
    int enemyHealth = 10;
    int myHealth = 1;
    bool ImStronger = MyHealth > EnemyHealth;
    if (ImStronger)
    {
        print("I can win!");
    }
}
```

With the code above, the relational operator tells us we are clearly at a disadvantage. However, we can add some additional information to make a better-informed decision.

```
void Start ()
{
    int enemyHealth = 10;
    int myHealth = 1;
    bool imStronger = myHealth > enemyHealth;
    int enemyBullets = 0;
    int myBullets = 11;
    bool imArmed = myBullets > enemyBullets;
    if (imStronger || imArmed)
    {
        print("I can win!");
    }
}
```


If we are better armed than our enemy, then we should have a better chance at winning. Therefore, in this case, if `imStronger` or `imArmed` then it's possible "I can win!". There's just a simple mental leap to read "`||`" as "or" as well as "`&&`" as "and."

5.8.2 What We've Learned

Conditional operators are the controls used for executing your instructions. It's often easier to break out the logic into different parts before using them. For instance, consider the following conditions when looking at some vectors. If we're going to execute a set of instructions only when a target object is above the player, we could create a simple boolean ahead of time for use later on.

```
void Update ()
{
    bool isAbove = target.transform.position.y > transform.position.y;
}
```

The `bool isAbove` will be `true` when our target object is higher than the `transform.position` of the class we're working in. Then we might add the following code to check if the target is in front of our position (`bool isInFront`).

```
void Update ()
{
    bool isAbove = target.transform.position.y > transform.position.y;
    bool isInFront = target.transform.position.z > transform.position.z;
}
```

We're simplifying this, so that we're assuming anything with a greater `z` is in front of the object we're working in. Either way, we've got two nicely named booleans that offer us something we can then use to set another boolean called `isInFrontAndAbove`.

```
void Update ()
{
    bool isAbove = target.transform.position.y > transform.position.y;
    bool isInFront = target.transform.position.z > transform.position.z;
    bool isInFrontAndAbove = isAbove && isInFront;
}
```

With this one boolean, we can then use a third parameter to check if our target is to the left or right of our object. We can set both with one check.

```
void Update ()
{
    bool isAbove = target.transform.position.y > transform.position.y;
    bool isInFront = target.transform.position.z > transform.position.z;
    bool isInFrontAndAbove = isAbove && isInFront;
    bool isLeft = target.transform.position.x < transform.position.x;
    bool isInFrontAndAboveAndLeft = isInFrontAndAbove && isLeft;
    bool isInFrontAndAboveAndRight = isInFrontAndAbove && !isLeft;
}
```

We can use the `!`, or not, in our conditionals as well. The last `bool` that checks for `isInFrontAndAbove`, also checks for *is not left*, which means that the object must be *in front and above* but not *left*. This leaves the only other possibility, which is that the target must be to the right.

Getting all of the logic straight is much harder when we try to bunch all of the possibilities into a single if statement. For example, for the `isInFrontAndAboveAndRight` boolean, we might end up with a statement a bit like the following:

```
void Update ()
{
    bool isInFrontAndAboveAndRight = (target.transform.position.y >
transform.position.y && target.transform.position.z > transform.position.z &&
! target.transform.position.x < transform.position.x);
}
```

A single statement so long might be considered poor form. Though there are other ways to make this easier, it's best to keep things short and readable.

5.9 Arrays: A First Look

Arrays are nicely organized lists of data. Think of a numbered list that starts at zero and extends one line every time you add something to the list. Arrays are useful for any number of situations because they're treated as a single hunk of data.

For instance, if you wanted to store a bunch of high scores, you'd want to do that with an array. Initially, you might want to have a list of 10 items. You could in theory use the following code to store each score.

```
int score1;
int score2;
int score3;
int score4;
int score5;
int score6;
int score7;
int score8;
int score9;
int score10;
```

To make matters worse, if you needed to process each value, you'd need to create a set of code that deals with each variable by name. To check if `score2` is higher than `score1`, you'd need to write a function specifically to check those two variables before switching them. Thank goodness for arrays.

5.9.1 Fixed-Sized Arrays

An array can be initialized as a fixed-sized array, or an array can be created to be a resizable; we'll get to these types of arrays in Section 5.12. A fixed-sized array is easier to create, so we'll cover that first. Dynamic arrays require a more interesting step, where an array object is instantiated before it's used.

Arrays are a fundamental part of computer programming, in general. Without arrays, computers wouldn't be able to do what they're best at: repeating a simple task quickly and consistently. We'll be covering a few different types of arrays in the next few Sections 5.11 and 5.12, and we'll be starting with the simplest type of array: the fixed-sized array.

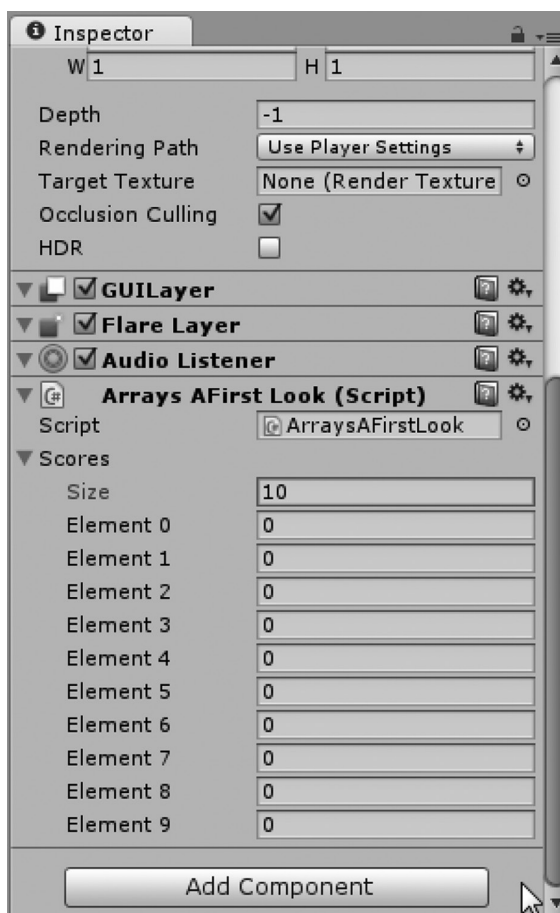
5.9.1.1 A Basic Example

Let's start with the Arrays project and open the scene. The concept is simple in theory, but it's so much easier to see what's going on if you use the following code and check out what's going on in the Unity 3D

Inspector panel. Attach this script to the Main Camera in an empty scene, and then take a look at what shows up.

```
using UnityEngine;
using System.Collections;
public class ArraysAFirstLook : MonoBehaviour
{
    public int[] scores = new int[10];
}
```

The inclusion of the square brackets tells the compiler you're creating an array of ints and naming it `scores`. Because we're working with a built-in type of an int, each value stored in the array is initialized to 0. A new `int[]` array needs to be created before it's assigned to the `scores` array.



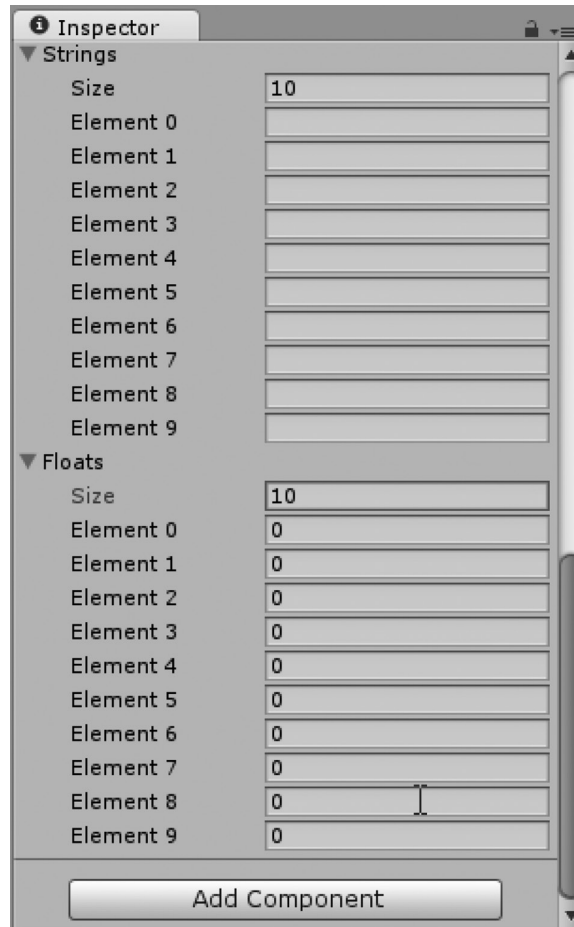
The best part is that the `scores` array is handled as a single object; we'll get into what that means in a moment. Rather than write a function to process one number value at a time we can write a function to deal with the array and process all of the number values at once. We'll take a look at a simple sorting method in Section 7.9.4.

Assigning scores to `int[10]` creates an array with 10 items. As you might imagine, you can make larger or smaller arrays by changing the number in the square brackets. There aren't any limitations to the size you can assign to an array. Some common sense should apply, however; an array with many trillions of values might not be so useful.

Each chunk of data in the array is located by what is called an *index*. The index number is an integer value since it's not useful to ask for a value between the first and second entry in the array. Arrays can be created from something other than ints as well.

```
public string[] strings = new string[10];  
public float[] floats = new float[10];
```

Both these statements create valid types of arrays. You are also allowed to create new data types and make arrays of those as well. An *n* array can be created for any type, not just numbers and strings. This isn't necessarily a requirement. We can use any valid identifier to use for an array.



It's important to notice the difference between types. Floats, ints, and numbers in general will have 0 as the default value in the array when they are created. Strings, on the other hand, are initialized with nothing in the array; they are *null*, or, rather, they have no value assigned. You'll need to remember that an array can contain only a single type across all of its entries. Therefore, an array of strings can contain only strings, an array of ints can contain only ints, and so on.

```
public string[] TopScoreList = new string[10];
```

The convention of using a plural does make for a more easily read variable name, but by no means is it necessary for identifying an array. Again, it's up to you to come up with readable variable names, and it's nice to use plurals for identifying an array.

```
public class MyClass
{
}

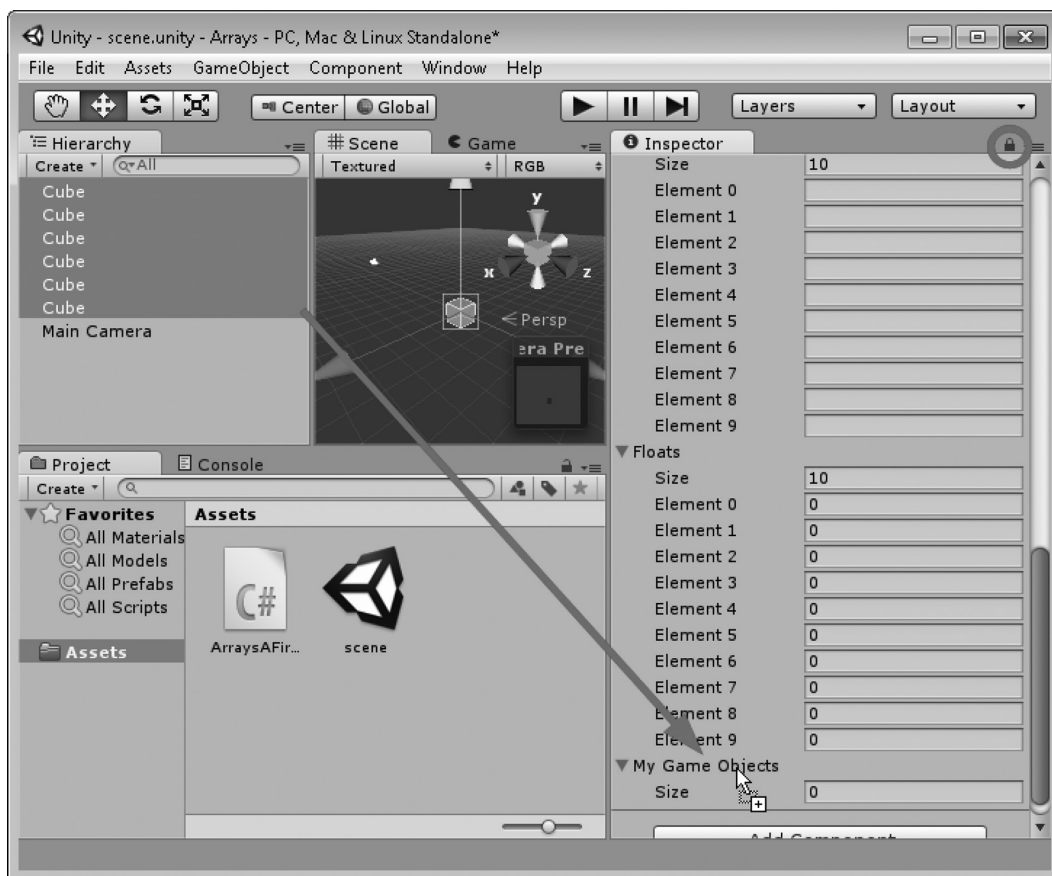
public MyClass[] MyClasses = new MyClass[10];
```

This creates a new class called `MyClass`, which for this example is a class of nothing. It's important that both sides of the `MyClasses` statement match. In the example above an array was created for the `MyClass` type. Unfortunately, because Unity 3D isn't aware of what to do with `MyClass`, the array doesn't show up in the Inspector panel. Later on, we'll figure out some ways to make this sort of information show up in Unity 3D.

What makes an array in Unity 3D more interesting is when we do not assign a number to set the size of the array. We can add the following statement to the list of other variables we've already added.

```
public GameObject[] MyGameObjects;
```

If we simply leave an array unassigned, we're free to set the size afterward. Select the Main Camera and click on the lock icon at the top right of the Inspector panel. This will prevent the Inspector from changing when you select something else in the scene. Then, select the other objects in the scene and drag them down to the `MyGameObjects` variable in the Inspector.

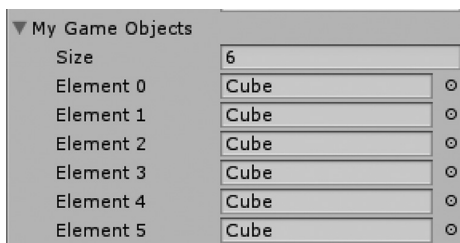


This action sets the size of the array once objects have been dragged into the array. However, this doesn't mean that the code is aware of the size of the array. The size of an array cannot easily be changed once it's been created. Therefore, how do we know how many objects are in an array? This can be accessed by the `.Length` property of the array type.

In the `Start ()` function, we can use the following statement:

```
void Start ()
{
    Debug.Log(MyGameObjects.Length);
}
```

Click the play icon and observe the Console panel. The `Debug.Log` command will print out how many objects had been dragged into the array.



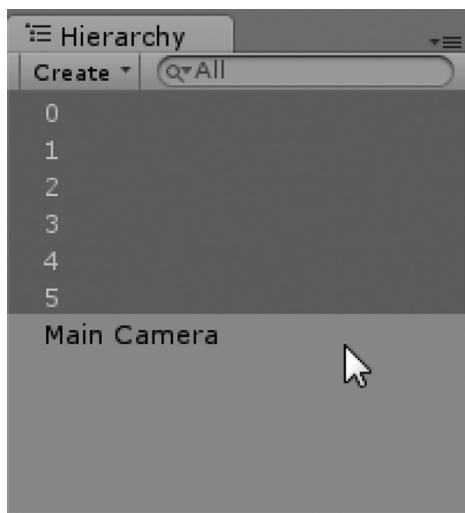
The number should reflect what is being observed in the Inspector panel, where we dragged objects into the array.

```
6
UnityEngine.Debug:Log(Object)
ArraysAFirstLook:Start () (at Assets/ArraysAFirstLook.cs:16)
```

So far so good, but how do we use this? Now that we have an array of game objects from the scene, we're able to manipulate them in a `for` loop.

```
void Start ()
{
    Debug.Log(MyGameObjects.Length);
    for (int i = 0; i < MyGameObjects.Length; i++)
    {
        MyGameObjects [i].name = i.ToString();
    }
}
```

This code changes the names of the game objects in the array to a number. The property of the array `.Length` returns an integer value which we can use for a few different reasons, the most practical use being to set the number of iterations in a `for` loop.



5.9.2 Foreach

This loop also happens to reflect how the objects are organized in the array. As we had experimented before with loops, we can use the array in a multitude of ways. We can also iterate over an array using the `foreach` loop. Once we've changed the names of each object, we can tell one from the other by its name.

5.9.2.1 A Basic Example

```
void Start ()
{
    Debug.Log(MyGameObjects.Length);
    for (int i = 0; i < MyGameObjects.Length; i++)
    {
        MyGameObjects [i].name = i.ToString();
    }
    foreach (GameObject go in MyGameObjects)
    {
        Debug.Log(go.name);
    }
}
```

The `foreach` statement is dependent on the type of data found on the inside of the array. Therefore, we use `GameObject go` to set up a place to hold each member of the `MyGameObjects` array while in the `foreach` loop. If we wanted to iterate over an array of a different type, we'd have to change the parameters of the `foreach` loop. Therefore, to iterate over an array of ints which was declared `int[] MyInts;`, we'd need to use `foreach(int i in MyInts)` to iterate over each member of that array.

Of course, the variable's name can be anything, but it's easier to keep it short. To use `foreach(int anIntegerMemberFromAnArray in MyInts)`, for example, would be a bit of work to key in if we wanted to do a number of operations on each int found in the array. Then again, there's nothing preventing us from spending time being verbose. We'll cover the `foreach` loop in more detail in Section 6.11.5.

5.9.3 Dynamic Initialization

```
void Start ()
{
    float[] DynamicFloats = new float[10];
}
```

We can also initialize a new array in a function. This means that the array exists only while in the scope of the function and can't be accessed from outside of that function. It's important to know that the size of an array is determined ahead of its use. When the array is declared in this way, the size or rather the number of objects the array can contain is set.

Here, we split the initialization into two statements. The first line tells C# we are creating a `float[]` variable identified as `DynamicFloats`. Then, we need to populate this floats array with a new array. The array floats is then assigned a new float array of 10 indices. We cannot switch this assignment to a different type.

```
Float[] DynamicFloats;
DynamicFloats = new int[10];
```

Changing the type creates an error, which we'll have to fix. There are clever ways to avoid this, but we'll get into that in Section 6.14.

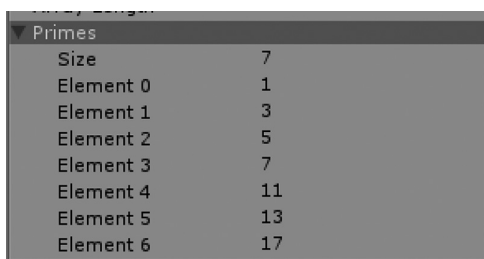
We can also use a variable to set the length of the array.

```
public int ArrayLength;
//Use this for initialization
void Start ()
{
    float[] DynamicFloats = new float[ArrayLength];
}
```

This code sets the length of the dynamically defined array. This can be helpful once we need to make our code more flexible to change for various game design settings. On the other hand, we can populate an array with predetermined information.

```
public int[] Primes = new int[] {1, 3, 5, 7, 11, 13, 17};
```

The above statement declares and sets an array that has seven members and assigns each number to a low prime number value. *These even show up in the Inspector panel in Unity 3D.*



5.9.4 Using the While Loop with Arrays

Iterating through our array is pretty simple. We've covered some basic loops that handle arrays quite well. If we use a simple while loop, we can process each value stored in a fixed array.

```
void Start ()
{
    int[] scores = new int[10];
    int i = 0;
    while(i < 10)
    {
        print(scores[i]);
        i++;
    }
}
```

At this point, all of the values are indeed zero, so we get 10 zeros printed to our Console panel. However, there are a few interesting things to point out here. First of all, `int i` is initialized to 0 ahead of the while loop. We'll get into what this means later on, but remember for now that arrays start at zero. The next interesting thing is how the numbers stored in `scores[]` are accessed.

5.9.4.1 Setting Array Values

```
scores[0] = 10;
```

We can set the value of each index of `scores` to a specific value by accessing the scores by their index. When the scores array was initialized as `int[10]`, `scores` now has 10 different `int` number spaces. To access each value, we use a number 0 through 9 in square brackets to get and set

each value. The while loop starts at 0 because we started with an `int i = 0;` before entering the while loop.

```
void Start ()
{
    int[] scores = new int[10];
    int i = 0;
    while(i < 10)
    {
        scores[i] = Random.Range(0, 100);
        print (scores[i]);
        i++;
    }
}
```

With this code, we're using a function called `Random` and using its member function `Range`, which we're setting to a value between 0 and 100. The index is picked by using the `i` that was set to 0 before the loop started. The loop starts with `scores[0]` being set to a random number between 0 and 100.

At the end of the while block, the `i` is incremented by 1 and the while loop begins again. The next time, though, we are setting `scores[1]` to a random number between 0 and 100.

5.9.4.2 Getting Array Values

Each time, we're getting the value from `scores[i]` to print. We can make this a bit more clear with the following example.

```
void Start () {
    int[] scores = new int[10];
    int i = 0;
    while(i < 10)
    {
        scores[i] = Random.Range(0, 100);
        int score = scores[i]; //getting a value from the array
        print (score);
        i++;
    }
}
```

If we add in the line `int score = scores[i];`, we'll get the score to the value found in `scores[i]`. Each value remains independent of the other values stored in the array. Because we're able to use the entirety of `scores[]` as a single object with index values, we're able to accomplish a great deal of work with fewer lines of code.

Arrays are simple objects with lots of uses. We'll get into a pretty interesting use in Section 5.11.3, but for now, we'll have to settle with playing with some numbers in the array.

5.9.5 What We've Learned

Arrays are useful for more than just storing scores. Arrays for every monster in the scene will make it easier to find the closest one to the player. Rather than dealing with many separate variables, it's easier to group them together. Arrays are used so often as to have special loops that make dealing with arrays very simple.

5.10 Jump Statements: Break and Continue

Loops are often used to iterate through a series of matching data. In many cases, we're looking for specific patterns in the data, and we'll change what we want to do depending on the data we're sifting through. In most cases, a regular `if` statement is the easiest to use. When our logic gets more detailed and we need to add more complex behaviors to our code, we need to start adding in special keywords.

When we come across the one thing we're looking for, or maybe the first thing we're looking for, we might want to stop the `for` loop that might change our result. In this case we use the `break;` keyword.

5.10.1 A Basic Example

To start with, we'll want to begin with the `Example.cs` file, fresh from the `JumpStatementsCont` Unity 3D project. To the `Start ()` function of the `Example.cs` file, we'll add the following code.

```
//Use this for initialization
void Start ()
{
    for (int i = 0; i < 100; i++)
    {
        print(i);
        if (i > 10)
        {
            break;
        }
    }
}
```

When we run this code, we'll get a printout from 1 to 11, and since 11 is greater than 10, the `for` loop will stop. In the `for` loop's second argument, we've got `i < 100`, because of which we would assume that the `print(i);` would work till we hit 99 before the `for` loop exits normally. However, since we have the `if` statement that breaks us out of the `for` loop before `i` reaches 100, we only get 1 to 11 printed to the console. Of course, there are reasons for using `break` other than cutting for loops short.

5.10.1.1 Continue

```
void Start ()
{
    for (int i = 0; i < 100; i++)
    {
        print(i);
        if (i > 10)
        {
            print("i is greater than 10!");
            continue;
        }
    }
}
```

5.10.2 ZombieData

The keyword `break` is often used to stop a process. Often, when we go through a group of data, we might be looking for something specific; when we find it, we would need to process it. If we create an array of

zombies, we'll need to assign them some specific zombie behaviors. Therefore, we might create a new `ZombieData.cs` class that could include some zombie information.

```
using UnityEngine;
using System.Collections;
public class ZombieData : MonoBehaviour
{
    public int hitpoints;
}
```

Here's a very simple zombie class that has nothing more than some hitpoints. I'll leave the rest up to your game design to fill in. Then, in a new `Example.cs` script, I've added in some logic to create a bunch of game objects. Some of them have `ZombieData`; some don't.

```
using UnityEngine;
using System.Collections;
public class Example : MonoBehaviour
{
    public GameObject[] gos;
    //Use this for initialization
    void Start ()
    {
        gos = new GameObject[10];
        for (int i = 0; i < 10; i++)
        {
            GameObject go =
                GameObject.CreatePrimitive(PrimitiveType.Cube);
            Vector3 v = new Vector3();
            v.x = Random.Range(-10, 10);
            v.z = Random.Range(-10, 10);
            go.transform.position = v;
            go.name = i.ToString();
            if (i%2 == 0)
            {
                go.AddComponent(typeof(ZombieData));
            }
            gos [i] = go;
        }
    }
    //Update is called once per frame
    void Update ()
    {
    }
}
```

The for loop in `Start ()` creates 10 new game objects temporarily stored as `go`. This is done with `GameObject go = GameObject.CreatePrimitive(PrimitiveType.Cube);`; we're creating 10 cube primitives. Then, we create a new `Vector3()`, with `Vector3 v = new Vector3();`. Once `v` is created, we give `x` and `z` a new value between `-10` and `10` using `Random.Range(-10,10);` and then assign the `v` to the new game object `go` with `go.transform.position = v`. For clarity, we give each `go` a new name after its iterative number from the for loop by using `go.name = i.ToString();`. Once the new game object `go` is created, we assign it to an array at the class level called `gos`.

Then, we get to `if (i%2 == 0) {go.AddComponent(typeof (ZombieData));}`, which assigns a new `ZombieData` object to each `go` if `i%2` is 0. The `i%2` is a clever way to check if a number is even or odd. Therefore, in this case, if the number is even, then we assign a new `ZombieData`; otherwise, no `ZombieData` is assigned. This means half of the cubes are not zombies.

5.10.3 Foreach—Again

In a normal game setting, you might have many different types of objects in a scene. Some objects might be zombies, whereas others might be innocent humans running away from the zombies. We're creating a number of game objects to simulate a more normal game setting.

```
//Update is called once per frame
void Update ()
{
    foreach (GameObject go in gos)
    {
        ZombieData zd =
(ZombieData)go.GetComponent(typeof(ZombieData));
        if (zd == null)
        {
            continue;
        }
        if (zd.hitpoints > 0)
        {
            break;
        }
        print(go.name);
        zd.hitpoints = 10;
    }
}
```

Now, just for the sake of using both `continue` and `break` in some logical fashion, we use the array to check through our list of game objects using `foreach(GameObject go in gos)`. The first line, `ZombieData zd = (ZombieData)go.GetComponent(typeof(ZombieData));`, assigns a `ZombieData` to the variable `zd`. The next line does something interesting.

Here, `null` is useful; in this case, if the object in the array has no `ZombieData`, then `zd` will not be assigned anything. When `zd` is not assigned anything, its data remains `null`. Therefore, in this case, if there's no zombie data assigned to the object in the array, we're looking at *then continue*, or in this case, *go back to the top of the loop, and move on to the next item in the array*. `Continue` means stay in the loop but skip to the next item.

If `zd` exists, then we move to the next line down and we don't hit `continue`. Therefore, we can check if `zd.hitpoints` is greater than 0; if it is, then we'll stop the loop altogether. Otherwise, we'll go and print out the game object's name and then assign 10 hitpoints.

The result of this loop prints out the even-numbered named game objects, and assigns them 10 hitpoints, but it does this printout and assignment only once. If a zombie's hitpoints were to fall below 0, then the zombie's name would be printed out and his hitpoints would be reassigned to 10.

For an AI character, it would be useful to know what objects in the scene are zombies. And your zombies should know what objects in the scenes are players. In your zombie behavior code, you might use something like the following.

```
if (playerController == null) {
    continue;
}
```

This code is used to skip on to the next object in the scene, which might be a player. Likewise, you could do the following with `null`.

```
if (playerController != null) {
    attackPlayer();
}
```

This statement would be just as useful. Likewise, you can do things before using `continue`. If your zombie needed a list of characters in the scene, you would want to add them to an `Array List`, which is different from an `Array` and you would want to add them to an `array first`.

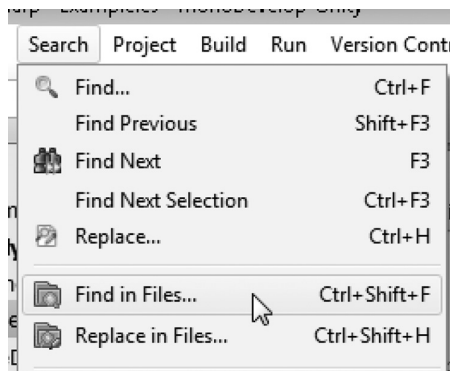
```
HumanData hd = (HumanData)go.GetComponent(typeof(HumanData));
If (hd != null) {
    allHumans.Add(go);
continue;
}
```

In the above example, we'd check the `go` if it's got a `HumanData` component. If it does, add it to an array list called `allHumans`, and then continue to the next object in the list.

5.10.4 What We've Learned

The previous section was a simple use of `break` and `continue`. As more complex situations arise, then using the `JumpStatements` comes in more handy. The jump statements are often used while searching through an array of many different kinds of objects.

When reading through other people's code, it's often useful to do a search for various keywords when you're interested in seeing how they are used. By searching for `continue`; or `break`;, you might find several different uses that might have not come to your mind immediately.



If you've downloaded some code assets from the Asset Store, you can use the following path to sort through their code base to find various places where more unique keywords are used: Search → Find in Files. Much about what is covered in this book is to introduce minimal uses for various features of the C# language.

Discovering the many different ways code can be used comes with experience, and it is something that cannot be taught in a single volume of text. The process of learning different programming features is like looking at a tool box and knowing what a screw driver looks like. It's an entirely different process to understand how it's best used.

5.11 Multidimensional Arrays

```
Int[,] TwoDimensionalArray;
```

An array is a single list of objects. Each entry is identified in the array by an index. By adding an additional index, we're allowed to create an additional depth to the array. When working with a single-dimensional array, we can get several simple attributes about the contents of that array.

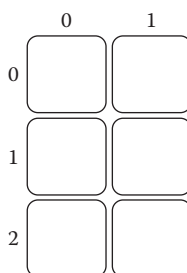
```
void Start () {
    GameObject[] oneDimension = new GameObject[5];
    for(int i = 0; i < oneDimension.Length; i++) {
        Debug.Log(i);
    }
}
```

With the above statement, we get an output, showing us 0 to 4 printed to the Console panel in Unity 3D. Using `oneDimension.Length`, we get a value representing the number of items in the array. At the moment, though, we have not added anything to the contents of each index. This situation is altered with the following change to the array's declaration.

```
GameObject[,] twoDimension = new GameObject[2,3];
for(int i = 0; i < twoDimension.Length; i++) {
    Debug.Log(i);
}
```

With the above statement, we get 0 to 5 printed to the Console panel in Unity 3D. The `Array.Length` parameter simply returns the total number of items in the array, but we're not able to get any specific information on how the indices are arranged. To get a better feel for the contents of the array, we might consider the `TwoDimensionalArray`; as a grid of two columns by three rows.

5.11.1 Columns and Rows



As shown in the above image, we have a 2 by 3 grid as a representation of the `GameObject[2, 3]` array. Each box holds onto a `GameObject`. A multi dimensional array has its place in programming, though it is rare, and the coding is usually better handled using a couple of single-dimensional arrays. However, it's more convenient to pass a single multi dimensional array to a function than it is to pass two different single-dimensional arrays.

To utilize this array, we'll want to populate the array with some `GameObjects`.

5.11.1.1 A Basic Example

Let's begin with the `MultiDimensionalArray` project in Unity 3D and open the scene.

```
void Start ()
{
    GameObject a = new GameObject("A");
    GameObject b = new GameObject("B");
    GameObject c = new GameObject("C");
    GameObject d = new GameObject("D");
    GameObject e = new GameObject("E");
    GameObject f = new GameObject("F");
    GameObject[,] twoDimension =
        new GameObject[2, 3]{{a, b, c}, {d, e, f}};
}
```

Notice how the array is assigned. There are two sets of curly braces, a pair of curly braces in a set of curly braces. We group three items into two subgroups, and assign them to the 2D array. The notation `{{}, {}}` is used to assign a 2 by 3 array with 6 `GameObjects`.

Next, we'll add a function to sift through the 2D array.

```
void InspectArray(GameObject[,] gos)
{
    int columns = gos.GetLength(0);
    Debug.Log("Columns: " + columns);
    int rows = gos.GetLength(1);
    Debug.Log("Rows: " + rows);
    for (int c = 0; c < columns; c++)
    {
        for (int r = 0; r < rows; r++)
        {
            Debug.Log(gos [c, r].name);
        }
    }
}
```

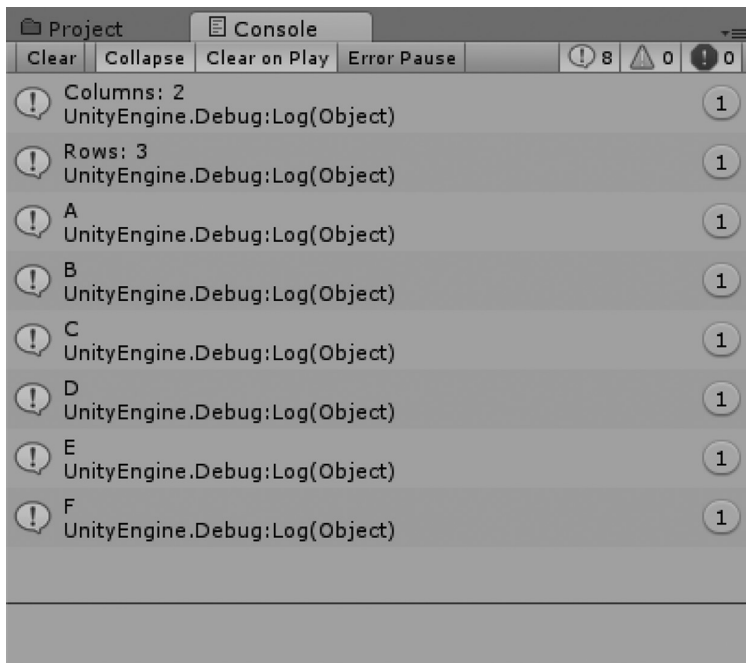
With this code, we'll get an idea of what types of loops will work best with a 2D array. First off, we have a function with one argument. The function `InspectArray(GameObject[,] gos)` takes in a 2D array of any size. Therefore, a `GameObject[37,41]` would fit just as well as the `GameObject[2, 3]` that we are using for this tutorial.

We're then using `gos.GetLength(0);` to assign our columns count. The `GetLength()` array function takes a look at the dimensions of the array. At index 0 in `[2, 3]`, we have 2, which is assigned to `columns`. Next, we use `GetLength(1);` to get the number of rows in the 2D array.

Using two `for` loops, one inside of the other, we're able to iterate through each one of the objects in the 2D array in a more orderly manner. Without a system like this, we're in a more difficult situation, not knowing where in the array we are.

```
void Start ()
{
    GameObject a = new GameObject("A");
    GameObject b = new GameObject("B");
    GameObject c = new GameObject("C");
    GameObject d = new GameObject("D");
    GameObject e = new GameObject("E");
    GameObject f = new GameObject("F");
    GameObject[,] twoDimension =
    new GameObject[2, 3]{{a, b, c}, {d, e, f}};
    InspectArray(twoDimension);
}
```

At the end of the `Start ()` function, we can make use of the `InspectArray()` function and get a log of the items in each position in the 2D array.



5.11.2 A Puzzle Board

Puzzle games often require a great deal of 2D array manipulation. To start a project, we can use the `Grid2D` project. The code in the `Grid2D.cs` class will begin with the following:

```
using UnityEngine;
using System.Collections;
public class Grid2D : MonoBehaviour
{
    public int Width;
    public int Height;
    public GameObject PuzzlePiece;
    private GameObject[,] Grid;
    //Use this for initialization
    void Start ()
    {
        Grid = new GameObject[Width, Height];
        for (int x = 0; x < Width; x++)
        {
            for (int y = 0; y < Height; y++)
            {
                GameObject go =
                    GameObject.Instantiate(PuzzlePiece) as GameObject;
                Vector3 position = new Vector3(x, y, 0);
                go.transform.position = position;
                Grid [x, y] = go;
            }
        }
    }
}
```


The `PuzzlePiece` on the script will need a prefab assigned to it. This means you'll have to drag the `Sphere` object in the `Project` panel to the variable in the `Inspector` panel. Select the `Game` object in the `Hierarchy` panel and drag the `Sphere` object in the `Project` panel to the `puzzle piece` variable in the `Inspector`, as shown below:



In the `Grid2D.cs` class, the first variables we're going to look at is the `public int Width` and `public int Height`. These two variables are made visible to the `Inspector` in the `Unity 3D` editor. I've set both of these to 6 in the `Inspector` panel. This is followed by a `GameObject`, which we'll fill the grid with. The last variable is a 2D array, which we will fill with the `GameObject PuzzlePiece`.

In the `Start ()` function, we'll add an initialization for the grid with `Grid = new GameObject[Width, Height];` to set up the 2D array so we can use it. Every fixed-sized array, whether 1D, like a `GameObject[]`, or 2D, which looks like the above `GameObject[,]`, needs to be initialized before it's used. Before being initialized, the array is `null`, which means it's lacking any size or any place for us to put anything into the array.

To fill in the array, we use the following code block added to the `Start ()` function.

```
for(int x = 0; x < Width; x++) {
    for(int y = 0; y < Height; y++) {
        GameObject go = GameObject.Instantiate(PuzzlePiece) as GameObject;
        Vector3 position = new Vector3(x, y, 0);
        go.transform.position = position;
        Grid[x, y] = go;
    }
}
```

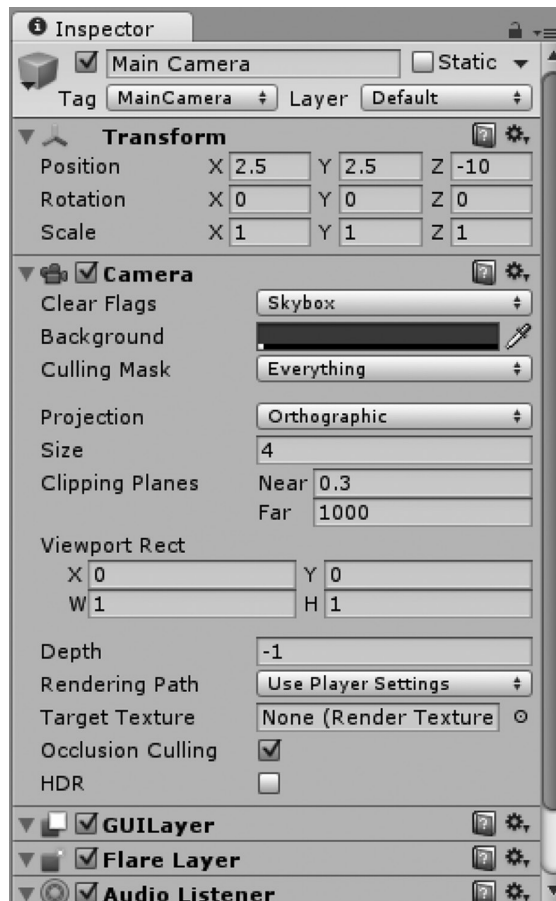
This code has two functions; the first for loop iterates through each position along the x; inside of each x position, we make a column for y with another for loop. At each position, we create a new `GameObject.Instantiate(PuzzlePiece) as GameObject;` to assign to `GameObject go`. When `GameObject.Instantiate()` is used, an object of type `object` is created. To be used as a `GameObject` type it must be cast. Without the cast, you'll get the following warning in the Unity 3D Console panel:

```
Assets/Grid2D.cs(14,28): error CS0266: Cannot implicitly convert type
'UnityEngine.Object' to 'UnityEngine.GameObject'. An explicit conversion
exists (are you missing a cast?)
```

This error is telling us we need to cast from `Object` to `GameObject`, quite a simple fix even if you forget to do this ahead of time.

After making a new instance, we need to use the `x` and `y` to create a new `Vector3` to set the position of each puzzle piece. Use `Vector3 position = new Vector3(x, y, 0);` to create a new `Vector3` for each position. Once the new `Vector3` position has been created and assigned to `position`, we can tell the `GameObject go` where it should be in the world. This is done with the statement that follows: `go.transform.position = position;`

Once the `GameObject` has been created and positioned, we can assign it to the `GameObject[,]` array with `Grid[x, y] = go;`, where we use `x` and `y` to pick which index in the 2D array the `GameObject` is to be assigned to. Once this is done, we can use the `Grid[x,y]` to get a reference to the puzzle piece. To pick a puzzle piece with the mouse, we're going to need to modify the camera's settings.

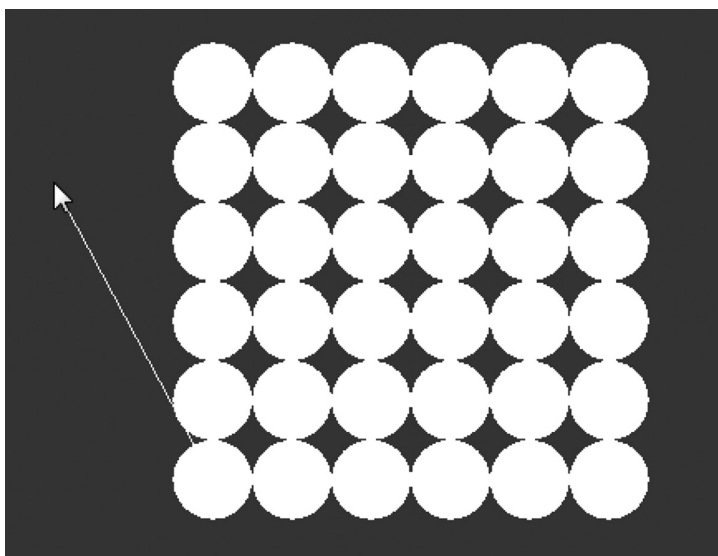


Three changes are required; the first is Projection, which we'll change from Perspective to Orthographic. This will mean that lines will not converge to a point in space. *orthographic* projection means that the line where the mouse appears over the game will be parallel with what the camera is looking at. We'll also want to resize the camera to match the grid. Setting this to 4 and repositioning the camera to 2.5, 2.5, and 10 allows us to see the grid of puzzle pieces.

```
void Update ()
{
    Vector3 mPosition =
        Camera.main.ScreenToWorldPoint(Input.mousePosition);
    Debug.DrawLine(Vector3.zero, mPosition);
}
```

In the `Update ()` function on the `Grid2D.cs` class, we can use the following statement to convert `mousePosition` on the screen into a point in space near the puzzle pieces. To help visualize this scenario, we'll draw a line from the origin of the scene to the mouse point with `Debug.DrawLine()`. Here, the grid started off with the `for` loop `int x = 0;` and the inner loop started with `int y = 0;`, where the `x` and `y` values were also used to assign the position of the puzzle pieces.

The first puzzle piece is at position 0, 0, 0, so the `Debug.DrawLine` starts at the same position. We should have a line being drawn from the origin to the end of the cursor.



Now, we need to do a bit of thinking. To pick one of the puzzle pieces, we have a `Vector3` to indicate the position of the mouse cursor; the position correlates roughly to the `x` and `y` position of each puzzle piece. The `Grid[x,y]` index is an integer value, so we should convert the `Vector3.x` into an `int` and do the same for the `y` value as well.

We can add the following two lines to the `Update ()` function after getting the mouse position.

```
int x = (int)(mPosition.x + 0.5f);
int y = (int)(mPosition.y + 0.5f);
```

This will do two things. First, it takes the `mPosition` and adds `0.5f` to the value coming in. When casting from a float to an `int`, we lose any values past the decimal point. Therefore, float 1.9 becomes `int 1`. The desired behavior would be to round the value up to the next whole number if we're close to it. Therefore, adding `0.5f` to the float value will ensure that we will get a better approximation to an expected integer value.

These int values can now be used to pick the PuzzlePiece with `Grid[x,y]`; This can be done with the following code.

```
GameObject go = Grid[x,y];
go.renderer.material.SetColor("_Color", Color.red);
```

The statement `GameObject go = Grid[x,y];` works, but only when the cursor is over a puzzle piece. When the cursor is over a puzzle piece, we turn its material color to red. When the cursor is no longer over a puzzle piece, we get a bunch of warnings informing us we're selecting an index that is out of range.

5.11.3 Checking Range

IndexOutOfRangeException: Array index is out of range.
Grid2D.Update () (at Assets/Grid2D.cs:27)

This is true when the cursor is at -4, 7; there is no assigned object to the `Grid[-4,7]`; we've only assigned from 0 to 6 on the x and 0 to 6 on the y. These errors should be resolved before as move on. This can be done with an if statement or two.

```
if(x >= 0) {
    if(y >= 0) {
        if(x < Width) {
            if(y < Height) {
                GameObject go = Grid[x,y];
                go.renderer.material.SetColor("_Color", Color.red);
            }
        }
    }
}
```

This code checks first that x is at least 0, and then does the same for y. After that, we check that x is less than the width we chose initially, and do the same for y. However, this is a messy bit of code. It works just fine but we can clean this up. There's nothing happening if only `x <= 0`; none of the if statements require any special actions to occur. They all contribute to the same assignment and color change statements at the end. That's how we know they can be reduced to one if statement.

The reduced statement looks like the following:

```
if(x >= 0 && y >= 0 && x < Width && y < Height)
{
    GameObject go = Grid[x,y];
    go.renderer.material.SetColor("_Color", Color.red);
}
```

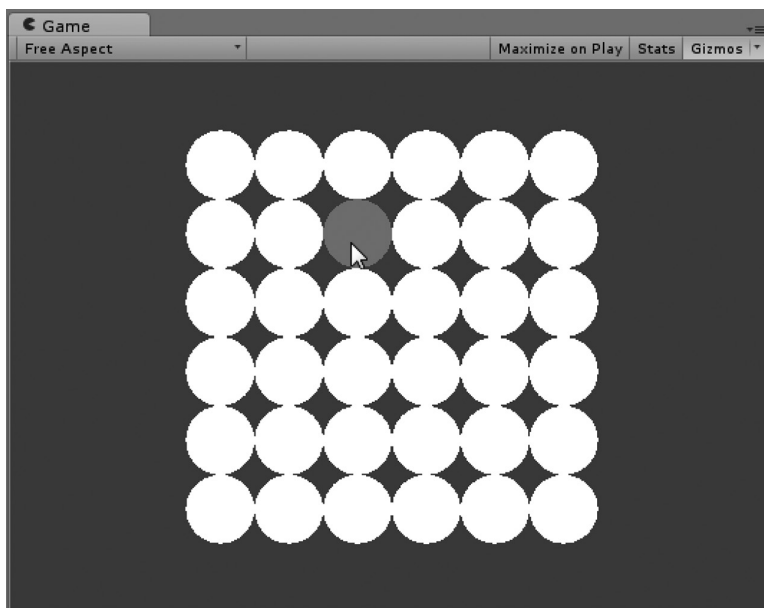
Much better. Therefore, this code works to highlight every game object we touch, but the objects remain colored red. The puzzle piece should return to white when the mouse is no longer hovering over them; how would we do that? The logic isn't so simple. We could pick every other object and set it to white, but that's impractical. Strangely, the solution is to set all of the objects to white, and then color the current object red.

```
for(int _x = 0; _x < Width; _x++)
{
    for(int _y = 0; _y < Height; _y++)
    {
        GameObject go = Grid[_x, _y];
        go.renderer.material.SetColor("_Color", Color.white);
    }
}
```

Iterate through all of the game objects in a similar way to how we instantiated them. Because this code is sharing a function with `x` and `y`, where we converted the float to an int, we need to make new versions of `x` and `y` for this function; therefore, we'll use `_x` and `_y` to create a new variable for the for loops. With this, we set them all to white. After resetting all of the puzzle pieces to white, we move to the next function that sets them to red. The completed `Update ()` function looks like the following:

```
void Update () {
    Vector3 mPosition =
    Camera.main.ScreenToWorldPoint(Input.mousePosition);
    int x = (int)(mPosition.x + 0.5f);
    int y = (int)(mPosition.y + 0.5f);
    for(int _x = 0; _x < Width; _x++)
    {
        for(int _y = 0; _y < Height; _y++)
        {
            GameObject go = Grid[_x, _y];
            go.renderer.material.SetColor("_Color", Color.white);
        }
    }
    if(x >= 0 && y >= 0 && x < Width && y < Height)
    {
        GameObject go = Grid[x,y];
        go.renderer.material.SetColor("_Color", Color.red);
    }
}
```

We're not so interested in what our scene looks like in the middle of the `Update ()` function, just how the scene looks like at the end. None of the intermediate steps appears in the scene, until the `Update ()` function has finished. This means we can do all sorts of unintuitive actions during the course of the `Update ()` function, so long as the final result looks like what we need.



Now, we have a very simple way to detect and pick which puzzle piece is selected in a grid of objects using a 2D array and a `Vector3`. Of course, the strategy used here may not fit all situations. In addition, applying various offsets for changing the spacing between each object might be useful to make

differently proportioned grids. Later on, you will want to add in additional systems to check for color to determine matches, but that will have to wait for Section 5.11.4.

To finish off this exercise, we'll want to move the code we wrote into a function, which would keep the `Update ()` function tidy, keeping a regularly updated function as simple looking as possible. This approach to coding also allows a function to be called from outside of the `Update ()` function.

```
//Update is called once per frame
void Update () {
    Vector3 mPosition =
        Camera.main.ScreenToWorldPoint(Input.mousePosition);
    UpdatePickedPiece(mPosition);
}
void UpdatePickedPiece(Vector3 position)
{
    int x = (int)(position.x + 0.5f);
    int y = (int)(position.y + 0.5f);
    for(int _x = 0; _x < Width; _x++)
    {
        for(int _y = 0; _y < Height; _y++)
        {
            GameObject go = Grid[_x, _y];
            go.renderer.material.SetColor("_Color", Color.white);
        }
    }
    if(x >= 0 && y >= 0 && x < Width && y < Height)
    {
        GameObject go = Grid[x,y];
        go.renderer.material.SetColor("_Color", Color.red);
    }
}
```

With the code moved into its own function, our `Update ()` loop is made a bit more easy to read. We're very specific about what is required to update the picked piece. We just need to remember to rename `mPosition` to a position inside of the new function. We can also extend the new function to return the piece that has been picked. By breaking apart a long string of code into smaller, more simplified functions, we're able to gain more flexibility and allow for changes more easily later on.

5.11.4 What We've Learned

Multi dimensional arrays can be a bit of a pain to work with, but after some practice, the logic becomes more clear. Puzzle games that use a grid of colored objects to match up often use a 2D array to check for patterns.

We could expand the concept to more than two dimensions as well. Using the previously defined `GameObjects`, we could make a 3D array that looks like the following:

```
GameObject[, ,] threeDimension = new GameObject[4,3,2]
{
    { {a,b}, {c,d}, {e,f} },
    { {a,b}, {c,d}, {e,f} },
    { {a,b}, {c,d}, {e,f} },
    { {a,b}, {c,d}, {e,f} }
};
```

Something like this is easier to produce using code to fill in each value, but it's important to be able to visualize what a `[4,3,2]` array actually looks like. Though a bit impractical, these sorts of data structures are important to computing, in general. Outside of video game development, multi dimensional arrays become more important to data analysis.

Large database systems often have internal functions that accelerate sorting through arrays of data in more logical ways. Of course, if you're thinking about building a complex RPG, then multi dimensional arrays might play a role in how your character's items and stats are related to one another.

5.12 Array List

Fixed-sized arrays are great. When you create an array with 10 indices, adding an 11th score or more will be cause for some rethinking. You'd have to go back and fix your code to include another index. This would be very inefficient, to say the least. This is where an array list comes in handy. An `ArrayList` is initialized a bit differently from a more common fixed-sized array.

```
ArrayList aList = new ArrayList();
```

An `ArrayList` is a C# class that has special functions for building lists that allow for changes in the number of values stored. An `ArrayList` is not set to a specific size. Rather than having to pick an index to put a value into, we merely use the `ArrayList` function `Add()`; to increase the number of objects in the `ArrayList`. In addition, an array has a type associated with it.

An array like `int[] numbers` can store only a bunch of ints. An `ArrayList` can store any variety of object types. Another feature which we'll find out later is that an array, not an `ArrayList`, can have multiple dimensions, for instance, `int[,] grid = new int[8,8];`, which creates an 8 by 8 array. An `ArrayList` cannot be used like this.

The drawback with an `ArrayList` is the fact that we're not able to populate the list ahead of time. With the fixed-sized array, we've been getting to know thus far that we're able to assign each index a value ahead of time; for instance, consider the following:

```
public int[] numbers = new int[] {1, 3, 5, 7, 11, 13, 17};
```

We know that this statement will appear in the Inspector panel with seven numbers in a nice UI roll-out. An array list doesn't allow for this behavior, as it needs to be created and then populated after its creation. The reason why Unity 3D allows us to assign values to objects in a scene is the scene file. The scene itself has had special data or metadata added to the scene. The scene now has specific bindings created to tie the data you've created to the class attached to an object.

To simulate this in a more common Windows app, you'd have to create a secondary file that stores specific settings. When the app is started, the settings need to be read in and then assigned to the classes as they are instantiated into the scene. Unity 3D has done much of the specific bindings for you, which allows your game to have scenes with arranged objects and unique settings for each object.

```
int i = 3;  
aList.Add(i);
```

The identifier inherits the `ArrayList` member methods. The `Add` method is used to append a value to the end of the `ArrayList` object. The `aList` from the above code fragment is now an `ArrayList` with one item. With this, we're able to use the `aList` object.

```
print(aList[0]);
```

This statement will send 3 to the Console panel, as you might expect. In practice, the `ArrayList` type is often used to collect an unknown number of objects into something more manageable. Once we've gathered the objects we're looking for, it's easier to deal with the multitude of objects. Therefore, if there are between 0 and 100 zombies in a room, we can have different behaviors based on the number of zombies found. If we want to convert the `ArrayList` into a regular array, we'd need to do some copying from the `ArrayList` to the array.

In addition to allowing for any number of objects, we can also consume any variety of types in the array.

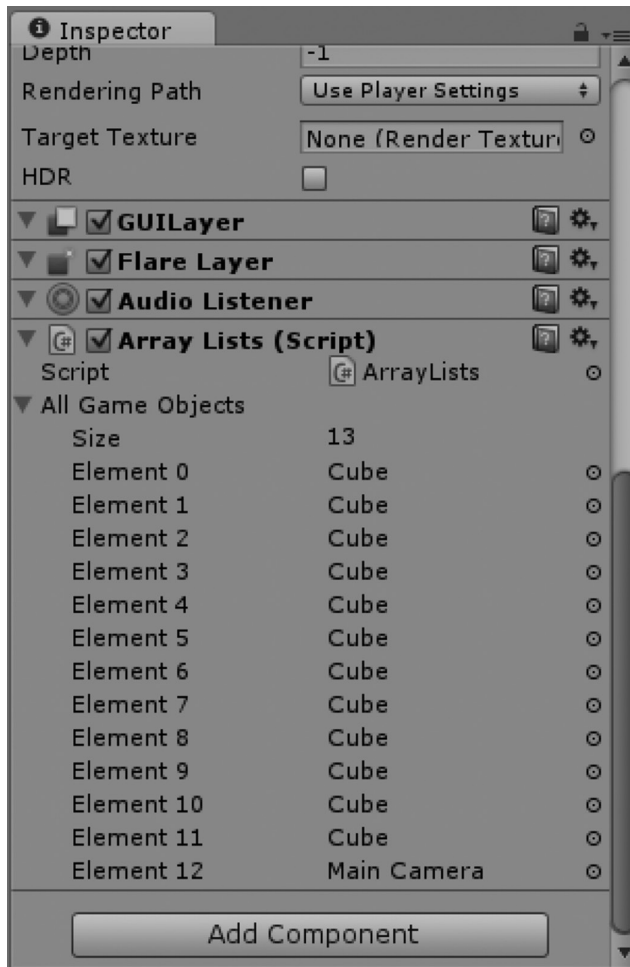
5.12.1 A Basic Example

To observe how an `ArrayList` is used, it's best to see it all in action; therefore, we'll start with the `ArrayLists` project in Unity 3D. In the scene, we can have any number of cube game objects. Should the number of objects be something that can't be predetermined, we'd want to use an `ArrayList` to store a list of them.

```
using UnityEngine;
using System.Collections;
public class ArrayLists : MonoBehaviour {
    //store all the game objects in the scene
    public GameObject[] AllGameObjects;
    //Use this for initialization
    void Start () {
        //creates an array of an undetermined size and type
        ArrayList aList = new ArrayList();
        //create an array of all objects in the scene
        Object[] AllObjects =
        GameObject.FindObjectsOfType(typeof(Object)) as Object[];
        //iterate through all objects
        foreach(Object o in AllObjects)
        {
            //if we find a game object then add it to the list
            GameObject go = o as GameObject;
            if(go != null)
            {
                aList.Add(go);
            }
        }
        //initialize the AllGameObjects array
        AllGameObjects = new GameObject[aList.Count];
        //copy the list to the array
        aList.CopyTo(AllGameObjects);
    }
}
```

Attached to the camera among all of the scattered cube objects is the above script. This behavior adds everything in the scene to an array. If it's an object, then we add it to the array. Afterward, we iterate through the array using `foreach (Object o in AllObjects)`, which allows us to check if the object in the `AllObjects` array is a `GameObject`. This check is done using `GameObject go = o as GameObject;`, and the following line checks if the cast is valid. If the cast is valid, then we add the object to our array list `aList`. This is done using the same `aList.Add()` function we used before.

After the iteration through the list, we end up with a final pair of steps. The first is to initialize a regular `GameObject` array to the size of the `ArrayList`. Then, we need to copy the `aList` to the freshly initialized `GameObject` array with `aList.CopyTo()`.



The final result is an `ArrayList` of each game object in the scene. If we skip the cast checking of whether `Object o` is a `GameObject`, then we get an invalid cast error. This happens only when we try to copy the `aList` to the `GameObject` array. We can do the following in the iteration to populate the `aList` without any problems.

```
//iterate through all objects
foreach(Object o in AllObjects)
{
    aList.Add(o);
}
```

This code simply adds everything that `GameObject.FindObjectsOfType()` finds to the `ArrayList aList`. This tells us that `aList` ignores the type that it's being filled with. To see what this `foreach` loop is doing, we can use the following modification:

```
//iterate through all objects
foreach(Object o in AllObjects)
{
    Debug.Log(o);
    aList.Add(o);
}
```

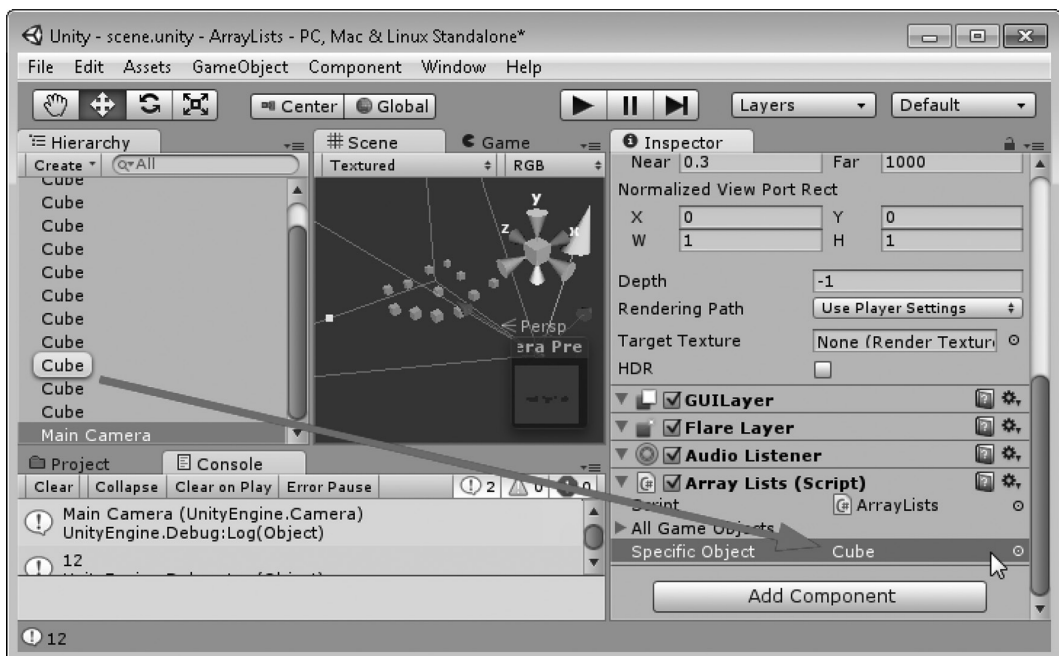
Here, we log each `o` to the console. We get many many more objects than you might have guessed. In the scene I've created, there are 76 different lines printed to the Console panel. The list includes the game objects as well as there are many dozens of other objects that are added to the `ArrayList`, too many to spell out in this case. A regular array can accommodate only one type at a time. And a `GameObject[]` array can have only `GameObjects` assigned to each index.

Since we might not necessarily know how many `GameObjects` reside in the final list, we need to dynamically add each one to the `ArrayList`, and wait till we stop finding new `GameObjects` to add. Once the list is done with iteration, we can then use the final count of the `aList`, using `aList.Count`, to initialize the `GameObject` array. The `ArrayList` has a function called `CopyTo`, which is designed to copy its contents to a fixed-sized array; therefore, we use it to do just that, with the last two statements:

```
//initialize the AllGameObjects array
AllGameObjects = new GameObject[aList.Count];
//copy the list to the array
aList.CopyTo(AllGameObjects);
```

5.12.2 ArrayList.Contains()

`ArrayList.Contains()` is a static function of `ArrayList`. The `ArrayList` type has several useful functions aside from just copying to an array. Say, we add a public `GameObject SpecificObject`; to the `ArrayLists` class. Then we can drag one of the cubes in the scene to the variable in the Inspector panel.



Now, we have an object and a populated array list. This allows us to search the array list for the object.

```
if (aList.Contains(SpecificObject))
{
    Debug.Log(aList.IndexOf(SpecificObject));
}
```

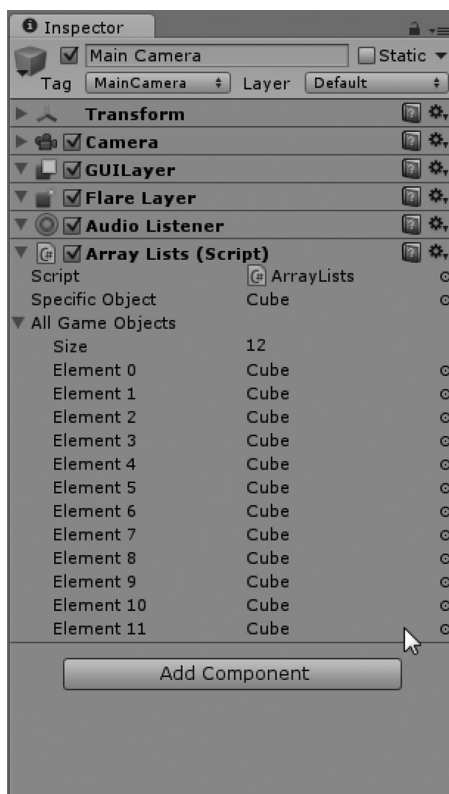
We can use two different array list functions. The first is `Contains()` and the second is `IndexOf()`. The `Contains()` function searches through the array list and returns `true` if it finds the object and `false` if the object isn't in the list. The `IndexOf()` function returns the index where the `SpecificObject` is found. Therefore, if the array list has the `specificObject` at index 12, the `Debug.Log` above returns 12.

5.12.3 Remove

The object that the script is a component of is found by `this.gameObject`, or simply, `gameObject`. Therefore, we may remove the object doing the searching from the list with the `Remove()` function. The statement that effects this behavior would look like the following:

```
if (aList.Contains(gameObject))
{
    aList.Remove(gameObject);
}
```

This statement reduces the size of the array list and takes out the Main Camera, which has the array list's component attached. With the camera removed, the copied array now has only cubes in it.



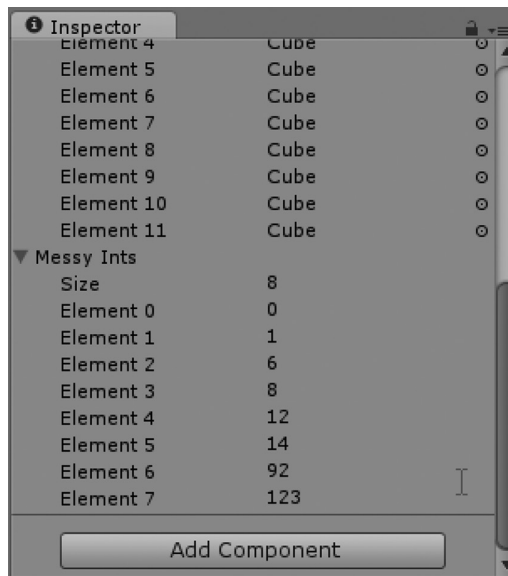
If the goal is to filter the scene for specific objects, then the array list serves as a great net to fill with the types of data you need. If you don't know ahead of time how many objects you are going to find, then an array list is a useful utility class that allows for some interesting tricks. Therefore, we've looked at removing and finding an object in an array list, but we don't need to stop there.

5.12.4 Sort and Reverse

Let's add `public int[] messyInts = {12,14,6,1,0,123,92,8};` to the class scope. This should appear in the Inspector panel, where we can see that `messyInts` is indeed a collection of messy ints. To copy the `messyInts` into a new `ArrayList`, we use `AddRange()` ;.

```
ArrayList sorted = new ArrayList();
//this adds the messyInts to the new ArrayList
sorted.AddRange(messyInts);
//command to sort the contents of the ArrayList
sorted.Sort();
//puts the new sorted list back into the messyInts array
sorted.CopyTo(messyInts);
```

With the above code added to the `Start()` function, we get the following result once the game is run.



The elements of the array have been sorted, starting at the lowest and ending with the highest. The sort function is a nicely optimized sort function. You might expect this from the experienced computer scientists working on the C# library we've been using. If we want to reverse the order in which the numbers are sorted, then we can use the `sorted.Reverse();` function.

```
ArrayList sorted = new ArrayList();
sorted.AddRange(messyInts);
sorted.Sort();
sorted.Reverse();//flips the array list over
sorted.CopyTo(messyInts);
```

Having some numbers rearranged is great, but there's little inherent value to the numbers if there are no associated objects in the scene that relate to the numeric values. The `Sort()` function is good for simple

matters where the data that's sorted has no association with anything in particular outside of the array. However, should we want to compare `GameObjects` in the scene, we might need to do something more useful.

5.12.5 What We've Learned

Making a custom sorting system requires some new concepts, which we have yet to get to. A specific type of class called an interface is required to create our own system of sorting. We could use this class in many different ways and take advantage of the power behind C#'s library.

A performance issue comes up with array lists. We may add objects to an `ArrayList` by simply using `MyArrayList.Add(someObject);`, which is easy. When we need to see what's in that array list, we begin to have a few problems.

As this was mentioned at the beginning of this chapter we didn't find out what happens if we use the following.

```
arraylist aList = new ArrayList();  
aList.Add(123);  
aList.Add("strings");
```

We have an issue with the above code fragment. If we assumed the `ArrayList` would only contain numbers; multiplying 123 by "strings" would result in an error. Of course we can't multiply these two very different types. We'd need to do a few tricks to check if we're allowed to multiply two objects from the `ArrayList`. First, we'd check if both are indeed numbers, if they are then we can proceed with a multiplication statement. If not, then we'd have to find another two objects which are numbers or we'd get an error.

This process takes time, and if the array list is big, then we'd be spending a great deal of time on checking object types. If we use a regular array, we can be sure that each object in the array is the same type, or possibly null. Checking against null for each object in the list is an extra step in the process. Reducing the number of steps your function need to work will speed up your functions and code will run faster.

When dealing with arrays, we might need to stuff an array into an array. Each array inside of an array can be of a different size. Imagine array A to be 10 items long. At `A[0]`, you can create another array B that is, for example, 3 items long. If each index of A is of a different size, you've created what is called a jagged array.

5.13 Strings

Strings are collections of letters. C# has no concept of what words are so a string is nothing more than a collection of meaningless letters. When you use a string, it's important to remember that the data stored in it is not like words.

5.13.1 Declaring a String

Basically, strings are presented to the computer by using the `"` or `'` operator. Strings act somewhat like an array, so we've held off till after we looked at arrays before getting to strings. Later on, strings will become a bit more useful, once we start naming `GameObjects`.

Strings can be used as user names and will be useful if you wanted to use them for dialog windows when talking to characters in a role-playing game. Being able to process strings is a useful skill to have for many general purpose programming tasks.

5.13.1.1 A Basic Example

Let's start with the Strings project in Unity 3D and open the scene in the Assets folder.

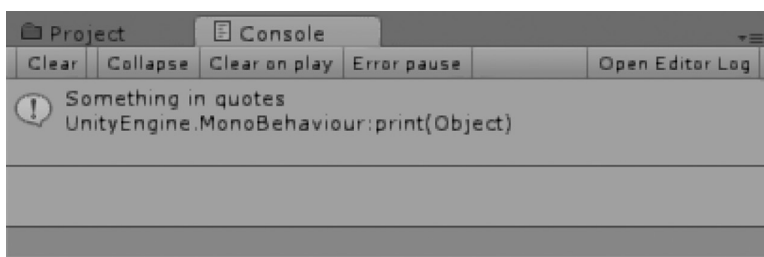
```
string s = "Something in quotes";
```

This statement creates a string with the identifier `s` and then assigns `Something in quotes` to `s`. Strings have several additional functions that are important. Be careful to not use smart quotes, which word processors like to use. Quotes that have directions are not the same as the quotes a compiler is expecting. Words in special quotes (“words”) actually use different characters when written in a word processor.

The compiler expects a different character. The character set that most word processors use is different from what C# uses to compile. For instance, if you hold down Alt and enter four numbers on the number pad, you’ll get a special character.

```
//Use this for initialization
void Start ()
{
    string s = "Something in quotes";
    print(s);
}
```

Printing the `s` results in a predictable output to the Console panel.



There’s not a whole lot unexpected happening here. However, we can do something with strings that might unexpected.

```
s += "more words";
```

We can add this statement just before the `print(s);` and get a different result. When we run the game, we get the following Console output.

```
Something in quotesmore words
UnityEngine.MonoBehaviour:print(Object)
Example:Start () (at Assets/Example.cs:8)
```

We forgot to add in a space before the *more* in `"more words"`, so it ran into the word quotes. White space is important to how we use strings. There’s no logic to what’s happening when we add more words to a string. To correct this, we need to add a space before *more* and we’ll get a more expected result: `"more words"`.

Strings do have some tricks that make them very useful. The string class has many member functions we can use, like `Contains`.

```
string s = "Something in quotes";
bool b = s.Contains("Something");
print (b);
```

This returns true; the word `"Something"` is contained in the string stored in `s`. Use any other set of letters which doesn’t appear in our string and the console will print false.

We can also do the following.

```
string s = "First word" + "Second word";
```

This statement will print out what you might expect.

```
First word Second word
UnityEngine.MonoBehaviour:print (Object)
Example:Start () (at Assets/Example.cs:7)
```

For argument's sake, we'll try another operator in the string declaration.

```
string s = "First word" - "Second word";
```

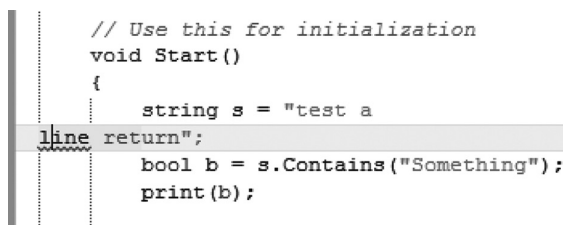
Here, we're going to try to subtract "Second word" from the first. Of course, this doesn't work and we get the following error.

```
Assets/Example.cs(6,23): error CS0019: Operator '-' cannot be applied to
operands of type 'string' and 'string'
```

So why does the + work and not the – in our declaration? The answer is operator overloading. Operators change their meaning depending on the context in which they appear. When the + is placed between two strings, we get a different result than when we put the + between two numbers. However, strings are rather particular. It's rather difficult to say for sure what it would mean to subtract one word from another. Computers are horrible at guessing.

5.13.2 Escape Sequences

Strings in C# need to be handled very differently from a regular word processor. Formatting strings can be somewhat confusing if you treat C# like any other text format.



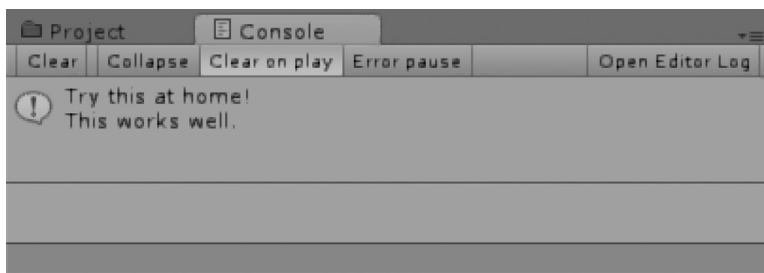
```
// Use this for initialization
void Start()
{
    string s = "test a
line return";
    bool b = s.Contains("Something");
    print(b);
}
```

If you want to add in a line return, you might end up with a mess as in the above image. Sometimes called line feeds, or carriage returns, these line returns tend to break C# in unexpected ways. To add in line feeds into the string, you'll have to use a special character instead.

I should note that the latest version of MonoDevelop will show an error when you make bad decisions like the above. Not all IDEs will try so hard to fight your bad formatting. Much of the time, software like Notepad++ or Sublime Edit will be more programmer friendly but will not alert you when you're doing something that will break the parser.

```
string s = "First line\nSecond Line";
```

Escape sequences work to convert two regular text characters into more specialized characters without needing to mess up your code's formatting. The \n creates a "new line" where it appears in the string. A \t adds a tab wherever it appears.



Unity 3D diverges a bit from most C# libraries as it doesn't implement all of the different escape sequences that are generally included. For instance, `\r` is carriage return, as is `\f` that are generally included in other common .NET development environments. The `\n`, which creates new line sequence, is more commonly known.

In old mechanical type writers, a carriage return is a lever that would both push a roll of paper and add in a new line at the same time: hardly something you see around today, unless you're a street poet. Not all of these escape characters work, but there are several others which you'll find useful.

```
print("\nI wanted quotes!\n");
```

To get double quotes to print in your console, you'll use the `\"` escape sequence. Here's a list of regular escape sequences.

<code>\a</code>	(beep)
<code>\b</code>	Backspace
<code>\f</code>	Formfeed
<code>\n</code>	New line
<code>\r</code>	Carriage return
<code>\t</code>	Tab
<code>\v</code>	Vertical tab
<code>\'</code>	Single quote
<code>\"</code>	Double quote
<code>\\</code>	Backslash
<code>\?</code>	Literal question mark

In addition to these escape sequences, we have additional three types that are used less often but are just as important. C# and Unity 3D will recognize some hexadecimal characters that are accessed with `\x` followed by two hexadecimal values. We'll find out more about these hex values and other escape sequences in a later chapter. Using hexadecimals is another way to get the more particular characters often found in other languages.

5.13.3 Verbatim Strings: @

In some cases, it's useful to use the `@` operator before a string. This tells C# that we're really not interested in formatting or not being able to make modifications to our string, like `string s = "this \nthat";`, which would result in the following:

```
this
that
```

The verbatim operator when used with strings—`string s = @"this \nthat";`—actually just prints out like the following:

```
this \nthat
```


This also means something as strange looking as

```
void Start () {  
    string s = @"this  
    that and  
    the other";  
    Debug.Log(s);  
}
```

prints to the console

```
this  
    that and  
    the other;
```

Notice that the console output includes all of the characters following the `@` symbol. This is allowed because of the verbatim operator. When using `Debug.Log()`, the verbatim operator can be used to format your strings to include new lines wherever you may need them.

5.13.4 String Format

Strings that contain information can easily be created; it's often the case when trying to track the behavior of a creature in the scene.

5.13.5 What We've Learned

We're not going to be building any word-processing software within Unity 3D. It's not something that sounds all that fun, and there are better environments for doing that. As far as strings are concerned, we're better off using them as seldom as possible.

Very few game engines dwell on their text-editing features, Unity 3D included. If you're planning on having your player do a great deal of writing and formatting to play your game, then you might have a tough time. For most purposes, like entering character names or setting up a clan name, the string class provided will have you covered.

5.14 Combining What We've Learned

We just covered some basic uses of loops. The `for` and the `while` loop have similar uses, but the `for` loop has some additional parameters that can make it more versatile. We also covered some relational, unary, and conditional operators.

Together with `if`, `else if`, and `else`, we've got enough parts together to do some fairly interesting logic tests. We can start to give some interesting behavior to our statements. For instance, we can make the placement of the zombies check for various conditions around in the environment before creating another zombie.

5.14.1 Timers

In the Timers project in the Unity 3D projects folder, we'll start with the `Example.cs` file attached to the Main Camera in the scene. Let's say we want to create a specific number of zombies, but we want to create them one at a time. To do this, we should use a timer; therefore, we should start with that first.

```
void Update ()  
{  
    print (Time.fixedTime);  
}
```

If we start with `Time.fixedTime`, we should see how that behaves by itself. We get a printout to the console starting with 0, which then counts up at one unit per second. Therefore, if we want some sort of action 3 seconds later, we can set a timer to 3 seconds from 0.

```
void Update ()
{
    if (Time.fixedTime > 3)
    {
        print("Time Up");
    }
}
```

If `Time.fixedTime` is greater than 3, "Time Up" is repeatedly printed to the console. However, we want that to print only once, since we want only one thing to happen when our `if` statement is executed. To do this, we should add in some way to increment the number we're comparing `Time.fixedTime` to.

```
float NextTime = 0;
void Update ()
{
    if (Time.fixedTime > NextTime)
    {
        NextTime = Time.fixedTime + 3;
        print("Time Up");
    }
}
```

With a class scoped variable called `NextTime`, we can store our next "Time up" time and update it when it is reached. Running this code prints "Time up" once the game starts. If `NextTime` was declared to 3, then we'd have to wait 3 seconds before the first `Time Up` is printed to the console.

Therefore, now, we have a timer. We should next add some sort of counter, so we know how many times the `if` statement was executed.

```
float NextTime = 0;
int Counter = 10;
void Update ()
{
    if (Counter > 0)
    {
        if (Time.fixedTime > NextTime)
        {
            NextTime = Time.fixedTime + 3;
            print("Time Up");
            Counter--;
        }
    }
}
```

First, add in `int Counter = 10` to store how many times we want to run our `if` statement. If the `Counter` is greater than 0, then we need to check our `Time.fixedTime` `if` statement. Once the timer is reset, we need to decrement our `Counter` by 1. Once the counter is no longer greater than 0, we stop. There's a cleaner way to do this.

```
float NextTime = 0;
int Counter = 10;
void Update ()
{
    if (Counter > 0 && Time.fixedTime > NextTime)
```

```

    {
        NextTime = Time.fixedTime + 3;
        print("Time Up");
        Counter--;
    }
}

```

We can reduce the extra `if` statement checking the counter value. To merge `if` statements, use the `&&` conditional operator. If either side of the `&&` is false, then the `if` statement is not evaluated. Because there's no code in the `if` statement checking the counter, there's no reason why it can't be combined with the `if` statement it's containing.

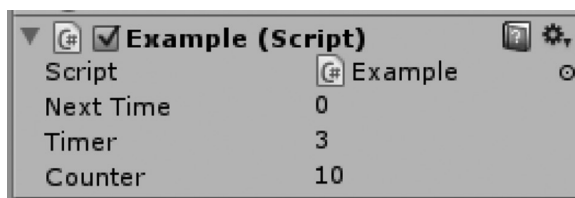
Now, we have a timer and a counter. We can make this more useful by exposing some of the variables to the editor.

```

public float NextTime = 0;
public float Timer = 3;
public int Counter = 10;
void Update ()
{
    if (Counter > 0 && Time.fixedTime > NextTime)
    {
        NextTime = Time.fixedTime + Timer;
        print("Time Up");
        Counter--;
    }
}

```

Add in `public` before some of the existing variables, and for good measure, let's get rid of the 3 which we're adding to `Time.fixedTime` and make that into a variable as well. This is common practice; use some numbers to test a feature, and then once the feature roughly works, change some of the numbers into editable parameters that may be handed off to designers.



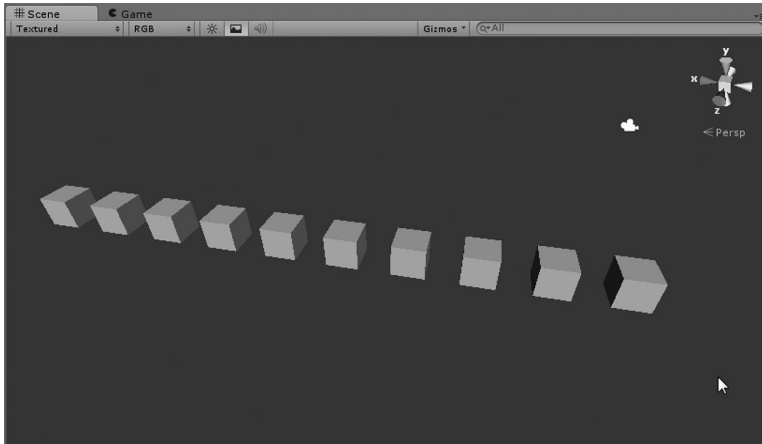
Now we can replace `print("Time Up");` with a more useful function. Once we've got the chops, we'll start replacing boxes with zombies, the code below serves as a good example for now.

```

public float NextTime = 0f;
public float Timer = 0.5f;
public int Counter = 10;
void Update ()
{
    if (Counter > 0 && Time.fixedTime > NextTime)
    {
        NextTime = Time.fixedTime + Timer;
        GameObject box =
        GameObject.CreatePrimitive(PrimitiveType.Cube);
        box.transform.position = new Vector3(Counter * 2f, 0, 0);
        Counter--;
    }
}

```

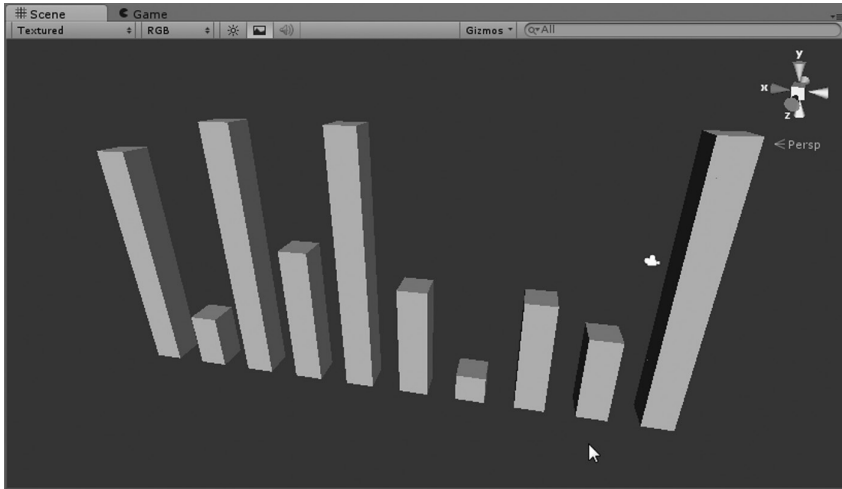
This little block of code will slowly pop a new cube into existence around the scene origin. We're using Counter to multiply against 2 so each box object will appear in a different place along the *x* coordinate.



You can speed up the process by lowering the timer value we just added. What if we wanted to have more than one box created at each counter interval? This objective can be accomplished using either a *while* or a *for* statement being run inside of the *if* statement.

```
public float NextTime = 0f;
public float Timer = 0.5f;
public int Counter = 10;
void Update ()
{
    if (Counter > 0 && Time.fixedTime > NextTime)
    {
        NextTime = Time.fixedTime + Timer;
        int randomNumber = Random.Range(1, 10);
        for (int i = 0; i < randomNumber; i++)
        {
            GameObject box =
                GameObject.CreatePrimitive(PrimitiveType.Cube);
            box.transform.position =
                new Vector3(Counter * 2f, i, 0);
        }
        Counter--;
    }
}
```

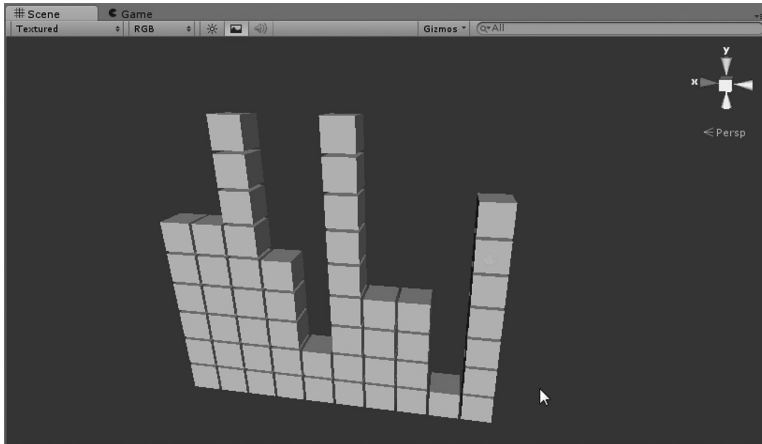
Adding in the *for* loop is the easiest way to do this. *Random.Range()*; is a new function that returns a number between the first argument and the second argument. We're just testing this out so we can use any number to start with.



Okay, looks like everything is working as we might have expected. We should take out the numbers we're using in the `Random.Range()` and make them into more public parameters for use in the editor.

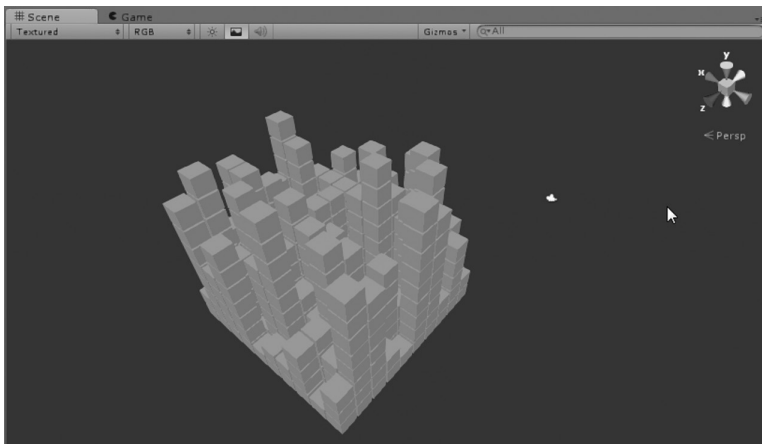
```
public float NextTime = 0f;
public float Timer = 0.5f;
public int Counter = 10;
public int MinHeight = 1;
public int MaxHeight = 10;
public float HorizontalSpacing = 2f;
public float VerticalSpacing = 1f;
void Update ()
{
    if (Counter > 0 && Time.fixedTime > NextTime)
    {
        NextTime = Time.fixedTime + Timer;
        int randomNumber = Random.Range(MinHeight, MaxHeight);
        for (int i = 0; i < randomNumber; i++)
        {
            GameObject box =
                GameObject.CreatePrimitive(PrimitiveType.Cube);
            box.transform.position = new Vector3(Counter *
                HorizontalSpacing, i * VerticalSpacing, 0);
        }
        Counter--;
    }
}
```

Adding more variables to control spacing will also make this more interesting.



By tweaking some of the parameters, you can build something that looks like this with code alone! The reason why we should be exposing so many of these variables to the editor is to allow for creative freedom. The more parametric we make any system, the more exploration we can do with what the numbers mean.

If you feel adventurous, then you can add in a second dimension to the stack of cubes.



Look at that. What started off as some cube-generating system turned into some 3D pixel art city generator.

```
public float NextTime = 0f;
public float Timer = 0.5f;
public int Counter = 10;
public int MinHeight = 1;
public int MaxHeight = 10;
public float HorizontalSpacing = 2f;
public float VerticalSpacing = 1f;
void Update ()
{
    if (Counter > 0 && Time.fixedTime > NextTime)
    {
        NextTime = Time.fixedTime + Timer;
        for (int j = 10; j > 0; j--)
        {
```

```

        {
            int randomNumber = Random.Range(MinHeight, MaxHeight);
            for (int i = 0; i < randomNumber; i++)
            {
                GameObject box =
                GameObject.CreatePrimitive(PrimitiveType.Cube);
                box.transform.position =
                    new Vector3(Counter * HorizontalSpacing,
                                i * VerticalSpacing,
                                j * HorizontalSpacing);
            }
        }
        Counter--;
    }
}

```

If we spend more time on this, then we can even start to add on little cubes on the big cubes by setting scale and position around each of the cubes as they are generated. We won't go into that right now, so I'll leave that on to you to play with.

5.14.2 Adding in Classes

The `GameObject` `box` is limited by what a box generated by `GameObject` can do. To learn a more complex behavior, we should change this from `GameObject` to a more customized object of our own creation.

After the `Update ()` function, add in a new class called `CustomBox`.

```

class CustomBox {
public GameObject box = GameObject.CreatePrimitive(PrimitiveType.Cube);
}

```

In this `CustomBox` class, we'll add in the same code that created the box for the previous version of this code; however, we'll make this one public. The accessibility needs to be added so that the object inside of it can be modified. Next, replace the code that created the original box in the `Update ()` function with the following:

```

CustomBox cBox = new CustomBox();
cBox.box.transform.position = new Vector3(Counter * HorizontalSpacing, i *
VerticalSpacing, j * HorizontalSpacing);

```

Note the `cBox.box.transform.position` has `cBox`, which is the current class object and then `.box`, which is the `GameObject` inside of it. The box `GameObject` is now a member of the `CustomBox` class. To continue, we'll add in a function to the `CustomBox` class to pick a random color.

```

void Update ()
{
    if (Counter > 0 && Time.fixedTime > NextTime)
    {
        NextTime = Time.fixedTime + Timer;
        for (int j = 10; j > 0; j--)
        {
            int randomNumber = Random.Range(MinHeight, MaxHeight);
            for (int i = 0; i < randomNumber; i++)

```

```

        {
            CustomBox box = new CustomBox();
            box.box.transform.position =
                new Vector3(Counter * HorizontalSpacing,
                    i * VerticalSpacing,
                    j * HorizontalSpacing);
            box.PickRandomColor();
        }
    }
    Counter--;
}
}

```

Check that the color-picking function is public; otherwise, we won't be able to use it. Then check that the color set is done to the `box` `GameObject` inside of the `CustomBox` class. Add in a `cBox.PickRandomColor();` statement to tell the `cBox` variable to pick a new color for the `box` `GameObject`.

```

CustomBox cBox = new CustomBox();
cBox.box.transform.position = new Vector3(Counter * HorizontalSpacing, i *
VerticalSpacing, j * HorizontalSpacing);
cBox.PickRandomColor();

```

The function now tells each box to pick a random color after it's put into position.

We're starting to do some constructive programming and getting some more interesting visual results. Next, we're going to need to make these things start to move around. To do that, we're going to need to learn a bit more about how to use our `if` statements and some basic math to get things moving.

5.14.3 What We've Learned

We're starting to build more complex systems. We're iterating through custom classes and putting them to some use. Adding functions to classes and using them in `for` loops is a fairly flexible and reusable setup.

We should be a bit more comfortable reading complex algorithms in the functions we read on forums and in downloaded assets. If we come across something new, then we should be able to come up with some search terms to find out what we're looking at.

5.15 Source Version Control

We've talked a bit about working with other programmers. However, we have yet to talk about how this is done. You could consider simply talking with one another and work together on a file at the same time. This could mean trading time at the keyboard, but this is simply not practical.

The best concept for working with a team of programmers is source control. Sometimes called revision or version control, the concept is basically being able to keep track of code changes and keeping a history of all of your source files.

Version control means retaining the state of an entire project, not just one file at a time. Tracking the history of a project also means that you can move forward and backward through the history of the tracked files.

Source code, or the code base on which your game is built, is, in essence, a collection of text files. Text remains ordered from top to bottom. Line numbers and code structures are easily recognized by software and where changes appear good, source control software can recognize the differences between revisions.

5.15.1 Modern Version Control

Today, there are many different version control systems used by software engineers. You may have heard of the term *Open source*, which is a general term used to describe hardware, or software that has been made publicly available for modification or use.

With open source software, a legally binding agreement is made between the creators of the original code base to the community at large to allow for use and distribution of the concepts and systems that have been made public. Open source software, in particular, is often distributed on what is called a repository.

Public repositories, or repos, like GitHub, SourceForge, and BitBucket, offer free storage space and version control options. GitHub uses a commonly used software called Git, so it has become a popular free service for many open and closed source projects.

Closed source projects are private repositories used by studios to retain the rights to their code and keep their project secret. Large companies and games are usually closed source projects, which protects their intellectual property and keeps their systems and methods out of the public domain.

5.15.2 The Repository

A repository is a central location where revisions of code and the most current version of the code can be found. A repo stores a code base. Many publicly available repositories allow for contributions by any number of users. These changes can be submitted and integrated into the code base to fix bugs and allow for the community at large to help solve complex problems.

A repository can also be made private on a local server for internal development. This limits access to those who can communicate with the server. Most game studios tend to keep their code and methods secret from the public.

A private repo can also be useful for a solitary developer. Source control is useful for many different situations. As a solitary developer, it's all too easy to make a change and break everything. It's also easy to forget what the code looked like before everything broke. If you had a backup to refer to, a fix will come easier, by reverting to a previous version of the project. Source control for an individual can be indispensable for this reason.

Your code base includes every C# file you've written along with any supporting text files that your code might reference. Source control is not limited to text. Binary files like audio clips and textures can also retain revision history.

Source control has been around for quite some time. It's difficult to say when the history of source control began, but you can assume it's been around for nearly as long as using text to write software on a computer has been around.

Today, many professionals use the likes of Perforce or possibly Microsoft's Team Foundation Server. Though capable, they're both quite expensive. On the free side, in terms of cost, there are many alternatives for the independent game developer. Among the most common are SVN, GIT, and Mercurial. Each system involves a slightly different philosophy, giving each one different advantages.

The general idea behind source control involves setting up another computer as a server to host your code base. The host can be on the Internet, usually provided free of cost. The host can also be a personal computer you've set up at home. If the host is connected to the Internet, the contents are usually available to the public, such as the files provided for this book.

It's also handy to keep a backup of your work on a separate computer. A simple NAS, or network attached storage, device might be all you need. A NAS is a small box that looks a bit like an external hard disk. You can plug it into your home WiFi router, stream music and video from it, and use it to host your source control software. A quick Internet search for "Git on a NAS" or "SVN on a NAS" will turn up some results on how this can be done.

Binary files are called so because they contain computer-formatted binary data. Binary data complicates source control because revisions of a binary file cannot be merged together. In general, source code works only on text files.

5.15.3 GitHub

For this book, we'll be using a popular system called GitHub; otherwise, I'd need to leave my laptop online so you can access my computer to download the files. I've got American broadband, so downloading anything significant from my computer is impossible anyway.

Git stores the history of your project locally. All of the changes, history, and revision information is stored for easy access. This means that every change is retained locally as well.

This doesn't mean much for text files, as each change means a few lines of code. Images and other binary files, on the other hand, can add up. Keep this in mind when working on a large game project that might involve many hundreds of binary assets.

With large projects, the Git repo can be several gigabytes in size. This will increase if there are many image source files, like multilayered Painter or many large, dense Blender files. In general, binary files like that might be better suited for something like Dropbox or Skydrive.

Dropbox retains a history for each file every time it's changed, you can go into the Dropbox history and revert or restore a file from its revision history. This isn't source control; you can't have two files merge together or multiple people working on a file at the same time. You can with Git, but only with text files.

Git is a vast topic that can span a nice thick book all to itself. It's unfortunate that all of the references on Git assume you're already familiar with source control topics in general. Complicate this with the fact that the authors of the Git documentation assume you're a pro with Linux-style terminal commands.

5.15.4 What We've Learned

We still have some work left to do. We've gotten through the setup of GitHub and are ready to do some work. To get your own repo set up, we'll need to go through a few more steps, but I promise this part is super easy.

After setting everything up, we need to learn how to commit changes and push them to your own repo.

5.15.5 Project Files

A project is the complete collection of all involved source files. A repo can host any number of projects or even a directory containing any number of projects. Each project includes the directory structure and all related assets.

When a new Unity 3D project is created, the Assets Directory should be under source control. This means any changes to a file found in the Assets Directory on your computer can be tracked and any number of revisions recorded.

After a source control system is set up—we will get to this in a moment—and source files have been written, it's time to check-in your work. A check-in is a submission of work to source control, which means that anyone with access can get an update of your work from the host.

It's easy to rename, move, or delete any file or directory from source control, so there's no need to worry about making major changes. Every revision is recorded, numbered, and kept in a database. This allows you to revert a file or the entire Assets Directory to an older version of the source code. In most cases, the database keeps track only of changes between revisions; otherwise, at the end of a project, your source control server might have an overwhelming amount of data to track.

5.16 Setting Up a Repository

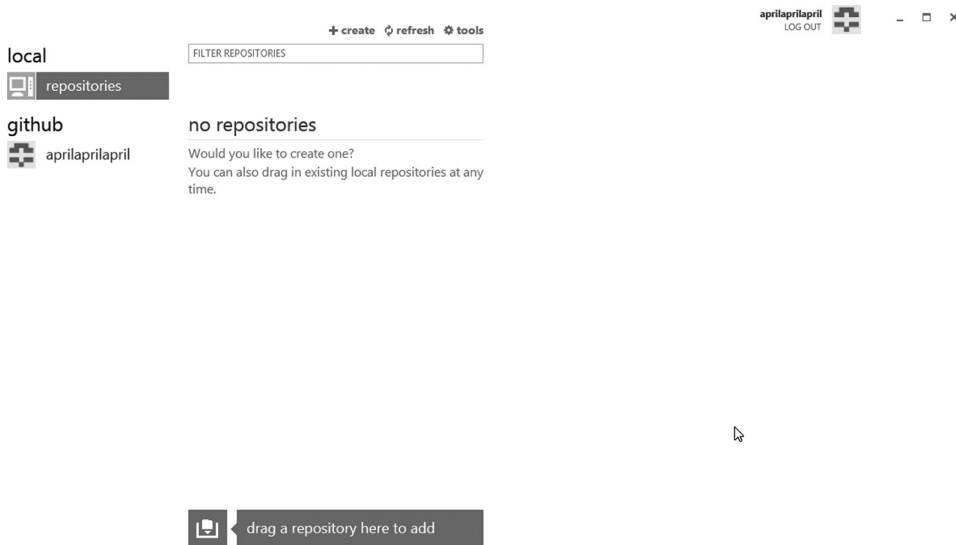
The Unity 3D Pro license has an option to add on an Asset Server module for more money. As an independent game developer, this might not be an option. We'll go through with the free, more commonly used source control system instruction for now.

To get started, you will want to grab the latest version of the GitHub software. This is located at <https://github.com/>. Here, you can follow the strange little mascot through the download and installation process of their software.

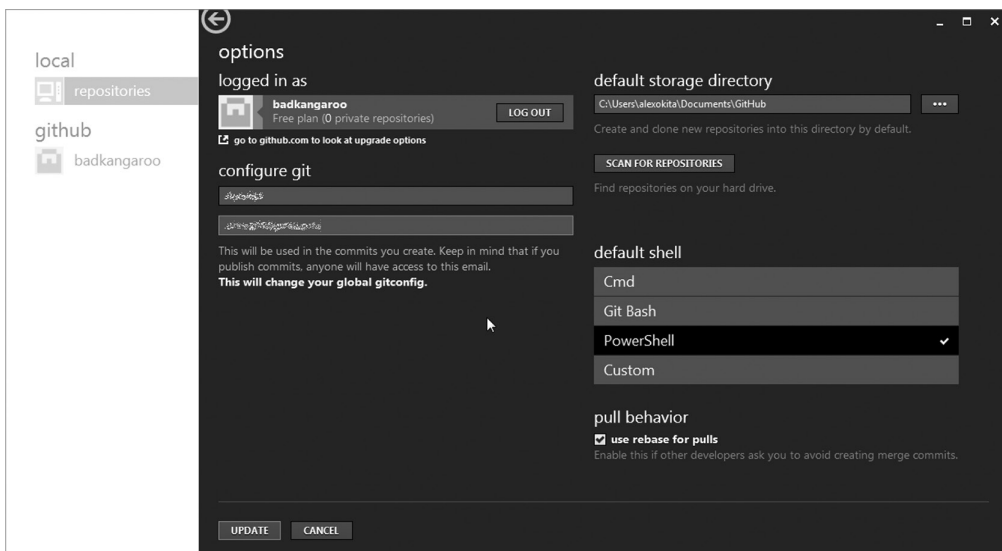
You may create a free account on github.com, which means that you'll be allowed to create as many projects as you like. You'll also be allowed to have an unlimited number of contributors to your project. You'll need to sign up with a user name and password on the GitHub website. Once this is done, you're just about ready to post your work from Unity 3D.

For a private repository, you'll need to sign up with a paid account. Since we're not exactly going to be building the next Call of Duty or Halo with our first project, there's really no reason to worry about anyone stealing our source code. Once you've started a more serious project, you might consider either a subscription with GitHub or building a private Git server.

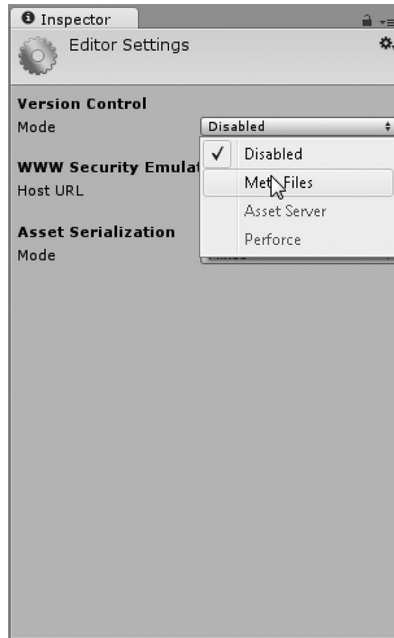
I'd recommend skipping the command line setup and use the client software for your operating system. Classically trained programmers may tell you this is a mistake, but I'd rather not go through how to operate source control through the command line. Computers have a graphic user interface for a reason.



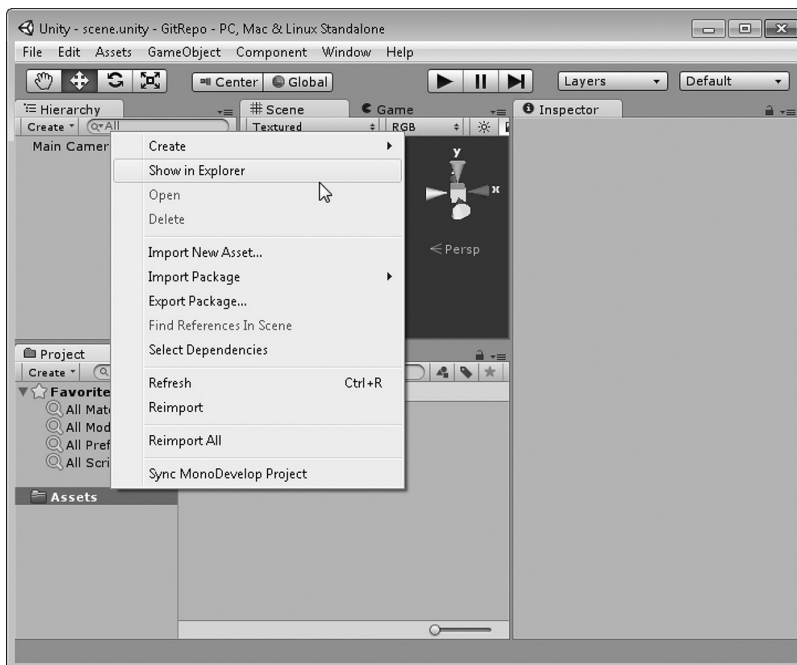
After the GitHub app has started, it's a good idea to double check the settings in the tools → options dialog. To make sure you'll be able to push your files to GitHub, you'll need to check that your user name and email are both filled in.



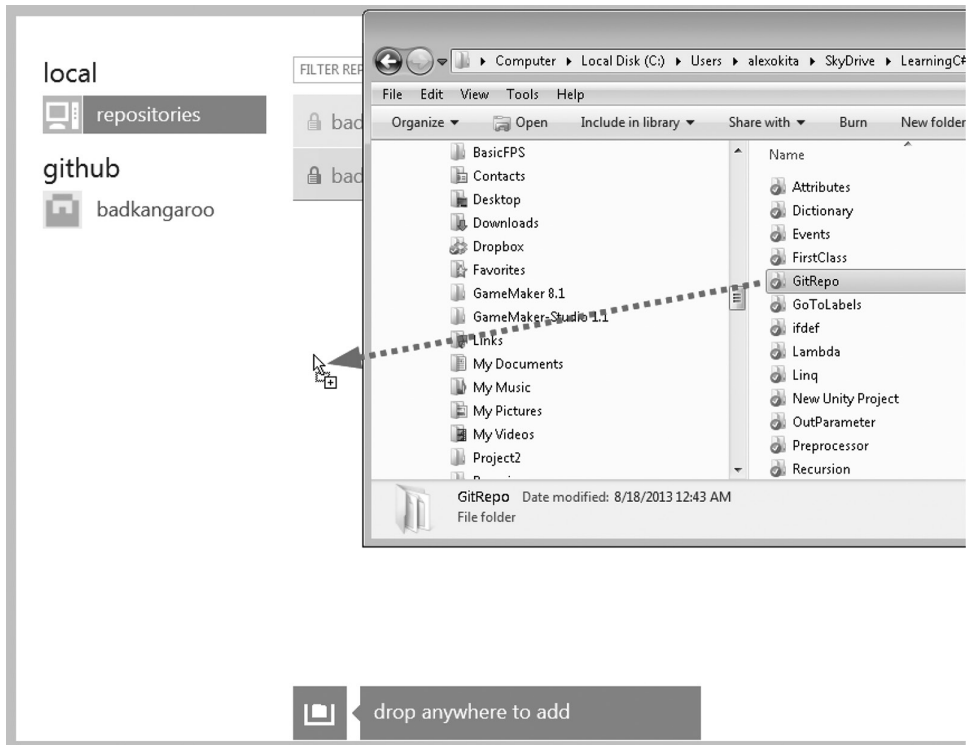
Once set up, it's time to pick a directory for your local repository. This can be changed later on, so any place will do. It's best to start with a new Unity 3D project, or a project that's already under way in Unity 3D. Unity 3D needs to be informed that the files it's generating are to be controlled by some sort of source control. To make this change, select **Edit** → **Project Settings** → **Editor**. The Inspector panel will now display the options for Version Control; select **Meta Files** to ensure that your Unity 3D project can be managed by Git.



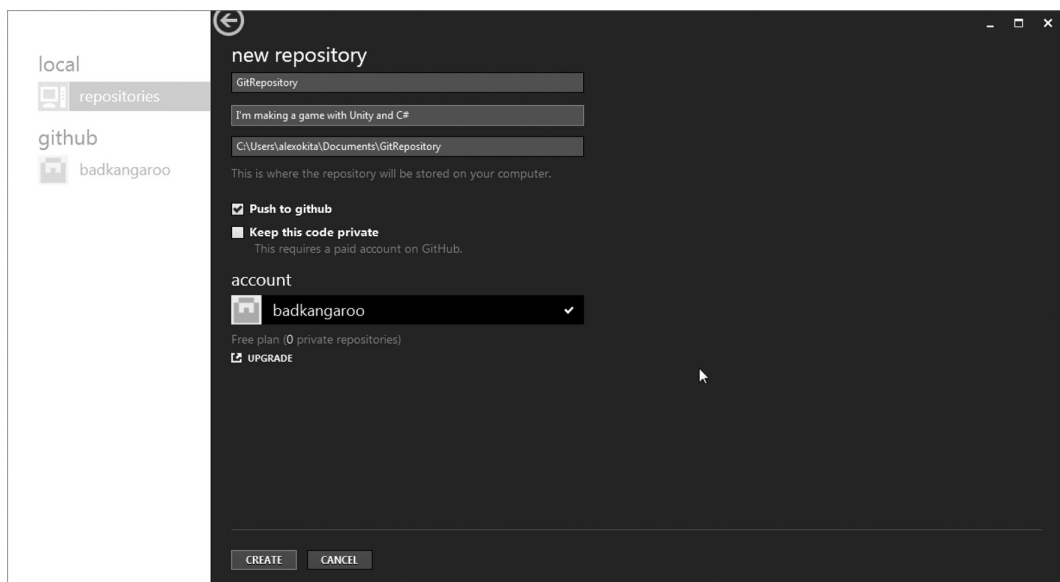
For this example, we'll start with a fresh Unity 3D project. To have something in the repository, I've also created a new C# file. Once the project has been created and Unity 3D has opened, it's simply a matter of dragging the directory on the desktop to the GitHub app.



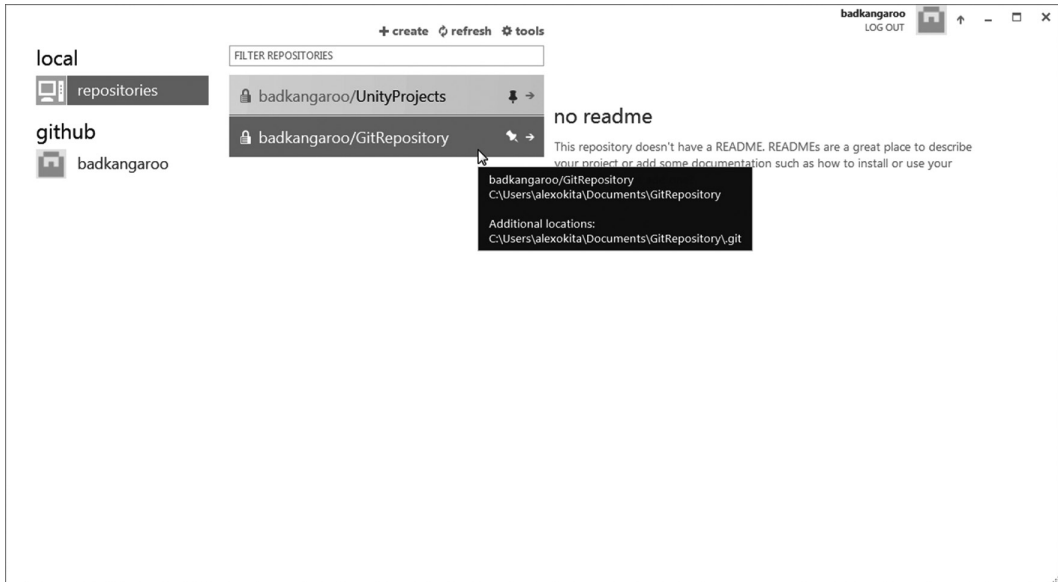
To find the directory, you can right click on the Assets Directory in the Project panel. This opens the directory with the Assets folder in it.



Dropping the file in the panel will begin the repository creation process.

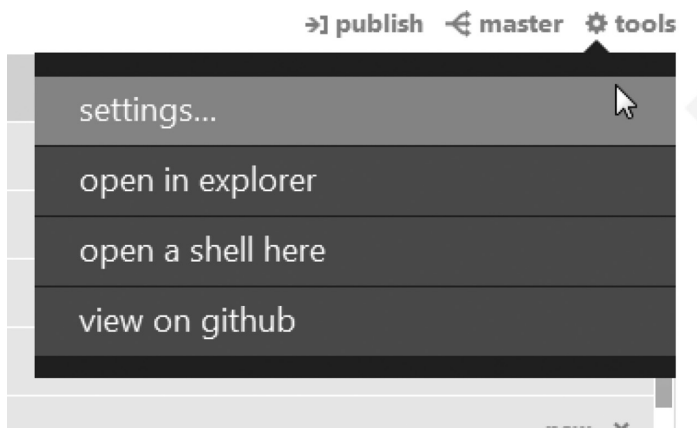


You'll be prompted with a dialog to describe your project; this can be changed later so it's not important to make this meaningful if you're just learning how to use Git. Click on create to push the project to the server.

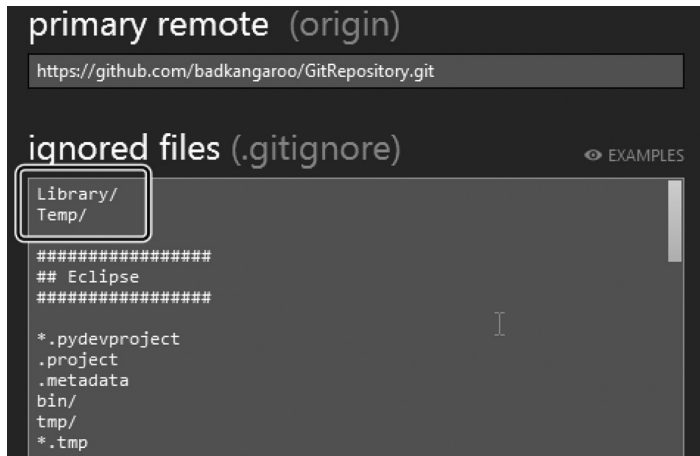


This action adds the repo to the list of repositories that GitHub is keeping track of. Double click on the new repository entry and you'll be taken to the push dialog. We'll want to make a few modifications to the project before pushing it.

By default, the project will have all of its subdirectories pushed to the server. This can be problematic if we leave all of the default settings. The Temp directory and the Library directories should be excluded from the push. These two directories are managed by Unity 3D and should not be checked in. Each Unity 3D user will have unique versions of these two directories.



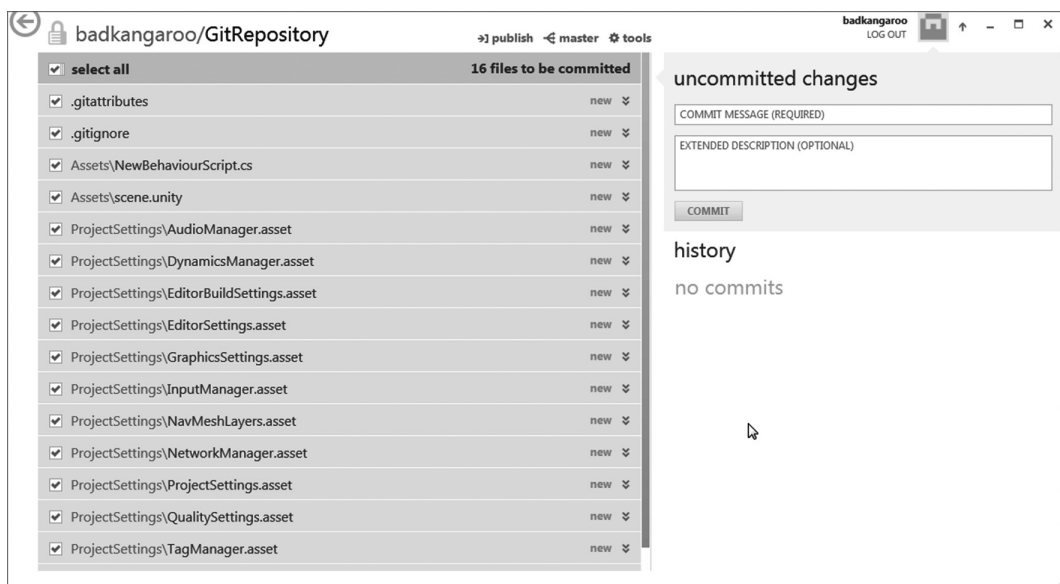
Under the tools button, select settings; this opens a dialog box that shows what appears to be some text files.



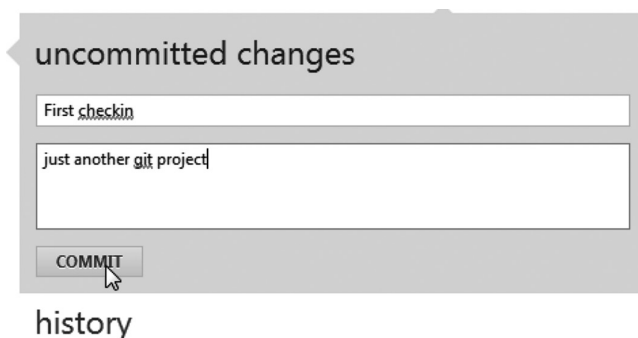
The left panel shows the ignored files dialog. To make sure GitHub ignores the Library and Temp directories, we add the following two lines:

```
Library/  
Temp/
```

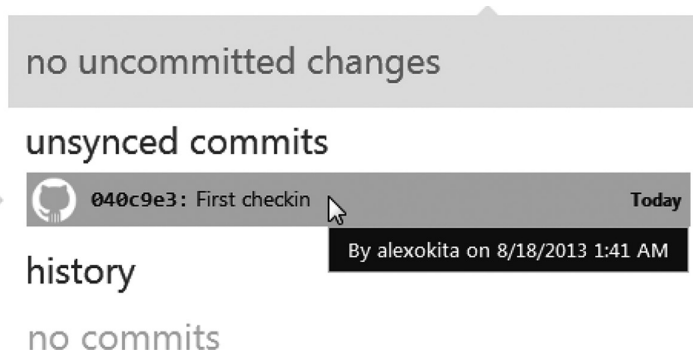
This tells GitHub to ignore those directories and your submission will look like the following:



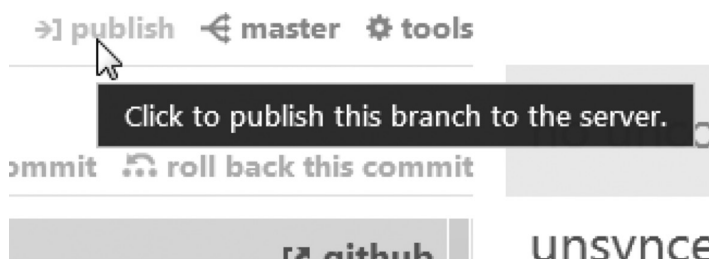
Add some comments; comments are always required.



This gives us some clue as to what changes were being made when the check-in was done. After entering a comment, press the COMMIT button. This sounds final, but no code has been submitted to GitHub yet.



The above image shows unsynced commits; this means that GitHub and our local version are not in sync. To make these changes appear on the trunk, we need to click on the publish button at the top of the panel.



This will upload our changes to the main line source control and make our changes available to the public.



We're almost there!

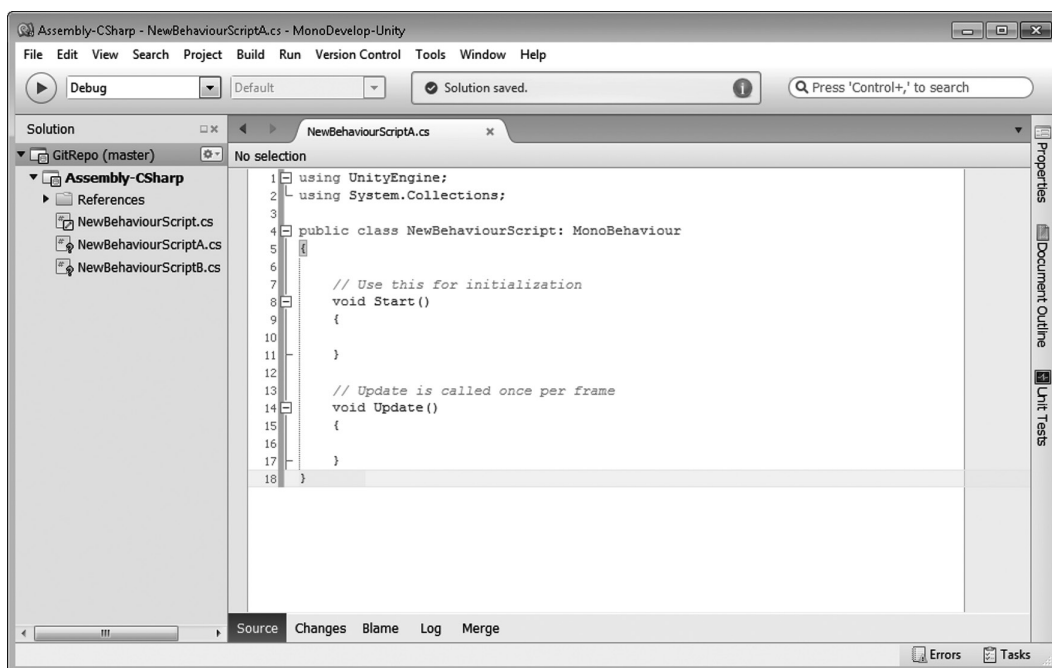
Only after the in sync button has appeared have we actually submitted our project to GitHub. Once the project is on the GitHub server, it can be shared among any number of other programmers.



For personal use, GitHub is a great way to keep your project's history intact. However, it is a public repository and anything you do will be open to public scrutiny. Again, there are more private ways in which source control can be managed, but this requires a bit more computer software setup on the server, which goes beyond the scope of this book.

5.16.1 Push

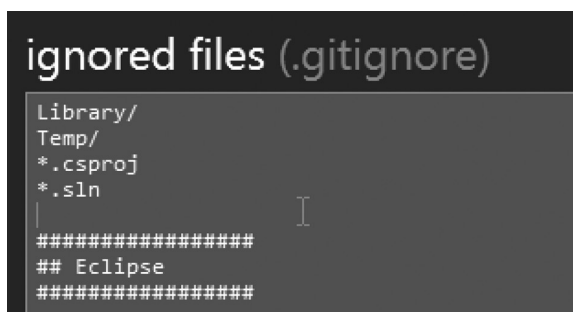
When we make changes to our local version of the project, we'll want to keep them in sync with the version on the server.



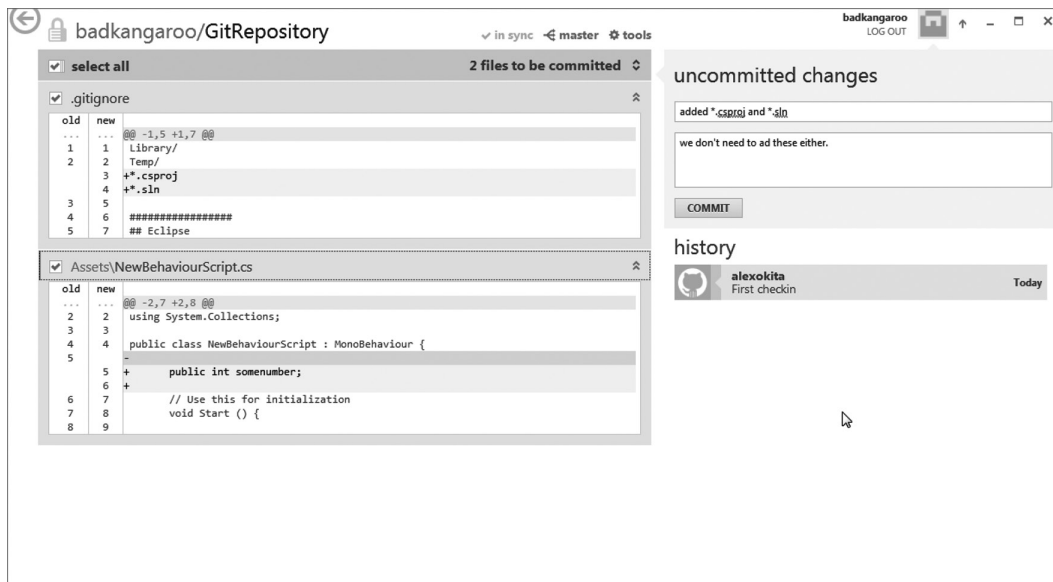
If we add a line to our C# file, we'll also be creating some additional files with the check-in.

5.16.2 Gitignore

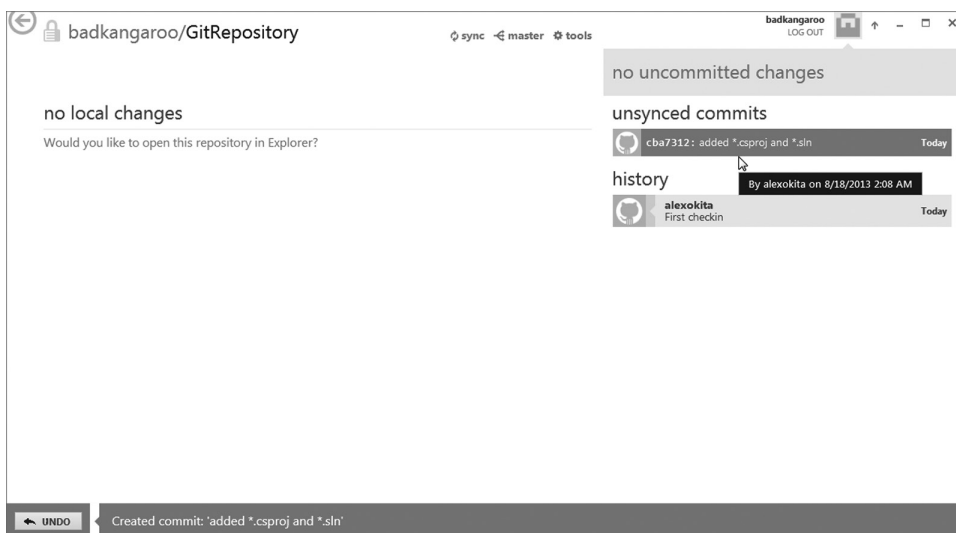
Often, there are files which we do not want to check-in. These files are usually created locally and will be unique to each user. Local preferences and settings affect only one person at a time and can cause trouble if another person gets the changes. These files should be left out of a check-in and not pushed up to the repo.



Add the *.csproj and *.sln to the ignored files in the tools dialog box in the GitHub client. This will require a new comment and sync. To check-out our changes, we can expand the changed files in the client.



These show what lines were changed or added. Like before, we have new uncommitted local changes, which we need to push to the server.

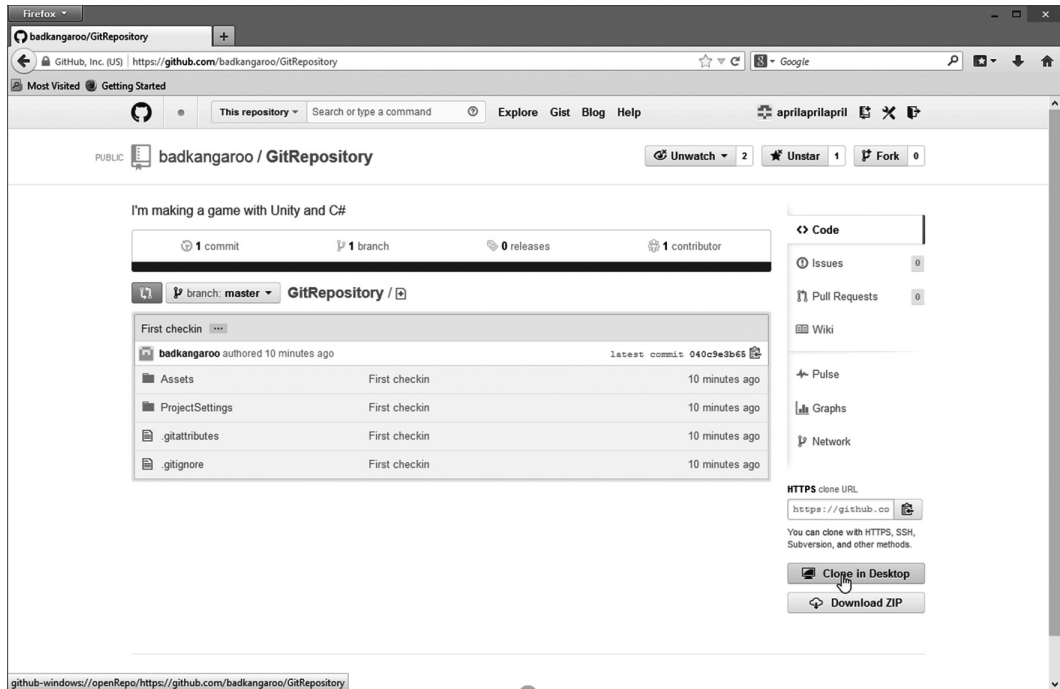


Press the sync button at the top to push the updates to the server. From this point on, any programmer who wants to check out your project can download the zip or open the project with the GitHub client and test your game. Everything appears on the GitHub website.

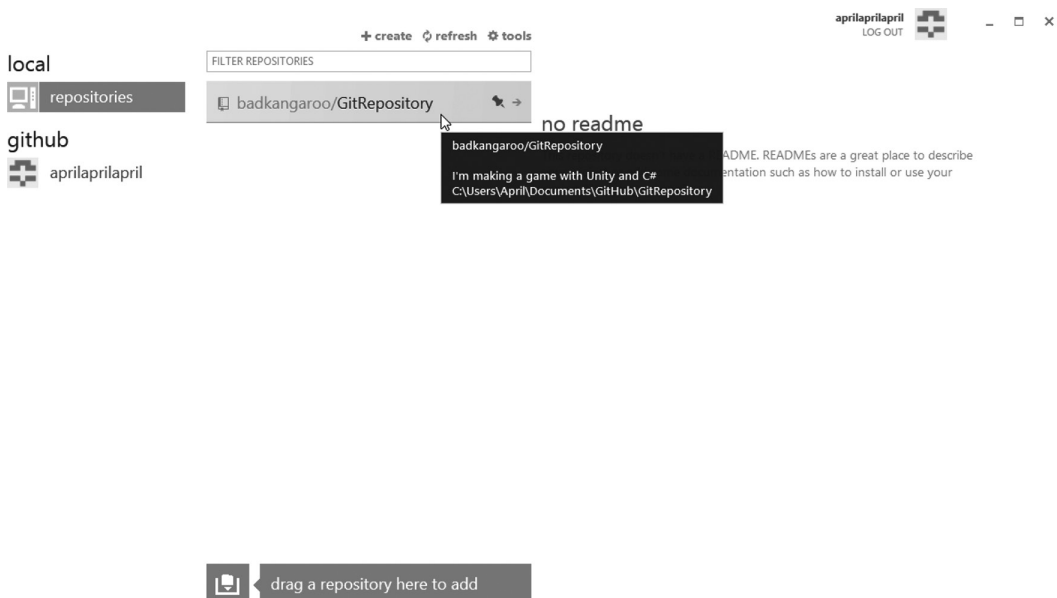
As you work, you may find other files that will need to be added to the ignore list. These include built files, when you start making APK files for iOS and ADK files for Android. These files shouldn't be checked-in, unless you're working to share a built binary file. These can be quite large in some cases, so it's better to leave them out of the repo.

5.16.3 Pull

When you created your GitHub account, your user name becomes a directory under the github.com site. For this book, you'll use <https://github.com/badkangaroo/>. To find this example repository, you'll add `GitRepository/` after the `badkangaroo/` user name. On the lower right, there are two options.



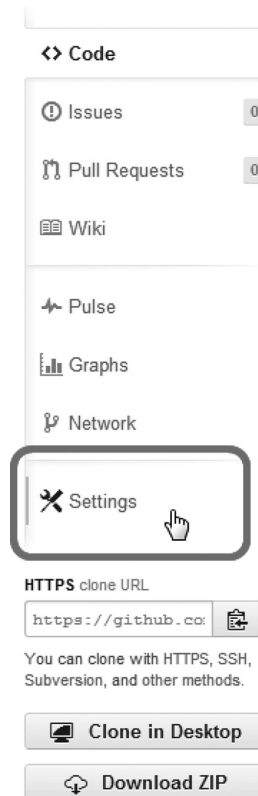
The first is **Clone in Desktop**; this will open the GitHub software. After a quick sync, the project will be cloned to your computer.



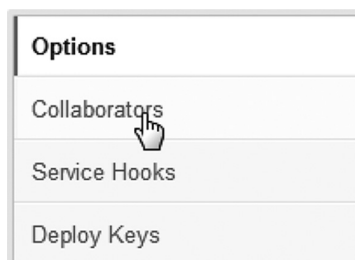
To get this resource another client needs to have the GitHub software installed. Cloning the project makes an exact copy of the latest revision on GitHub to the client's computer. On windows the clone is created in `C:\Users\Username\Documents\GitHub\` followed by the name of the project, so in this case it's `C:\Users\Username\Documents\GitHub\GitRepository` where this can be used as the project directory for Unity 3D. The GitHub website is somewhat similar to this setup <https://github.com/Username/GitRepository> you'll need to go here to manage the projects settings.

5.16.4 Contributors

As the creator of a project, you're the only one who can make modifications to the files in the project. To allow others to make modifications, you'll need to add them as collaborators.

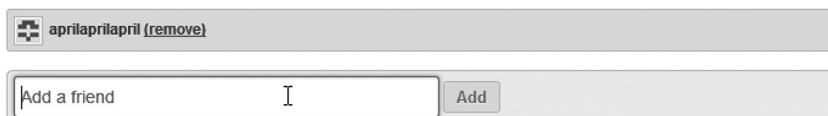


On your GitHub website for the project, select Settings, to the right of your repository. Here, you'll be able to change the name of the directory and pick several other options. To add people to the list contributors, select the Collaborators option.



Add in the name they used to sign up for GitHub and now they'll be able to pull your code, make changes, and then push their modifications to the trunk. Users not on this list will not be able to make any contributions to the trunk of the code.

Manage Collaborators



The screenshot shows the 'Manage Collaborators' section of a GitHub repository. At the top, there is a list of collaborators. One collaborator, 'aprilaprilapril', is listed with a small GitHub logo icon to the left and a '(remove)' link to the right. Below this list is a form to add a new collaborator. It consists of a text input field with the placeholder text 'Add a friend' and a cursor inside, followed by an 'Add' button.

To make a submission, you push your work onto the server. This becomes the root of your source tree. From here on out, the source control server becomes the real home or main line of your project. Any team member should be making his or her contributions to the server, not your computer directly.

Someone needing your latest work would pull it from the server. *Push* and *pull* are commonly used terms when dealing with source control. Other words like *get* and *check-out* refer to pulling source files from the server. *Check-in* or *submit* often refers to pushing data to the server. After their contributions have been checked-in, you'll need to pull them from the server to update your local version to reflect the changes made by other programmers.

If you're working on your own, then you'll be the only one pushing changes. To get the latest projects for this book, I needed to pull only one file from GitHub; this was mentioned in Section 2.5. As files are added and changes are made, GitHub turns into a great location to keep track of your file changes.

Once you've written code, tested it, and are sure that everything is working, it's time to push your changes to the server. Only after the code has been synced can other users pull it from the server and get your changes.

5.16.5 What We've Learned

There was a lot to cover in this chapter on how to manage your source code. There's still a lot remaining to learn, when it comes to dealing with complex source control problems. Unless you've started working in a team, it's unlikely you'll have to deal with too many of these issues right away.

You'll be constantly updating the `gitignore` file; after a few iterations, the number of additions to the `gitignore` should come less often. Maintaining a clean git repository will help keep things running smoothly.

```
Library/  
Temp/  
*.csproj  
*.sln  
*.userprefs
```

The `*` notation indicates that any file that ends with `.userprefs` or any file that ends with `.sln` should be ignored. The `*` is an old carryover from the DOS command shell; further back, Unix command shells also used the same notation for searching for files in directories.

The GitHub client app is new. It's far from perfect and requires a great deal of updating to make it more user friendly. At every turn, you might have an error prompting you to use a shell. For the uninitiated, this can be frustrating. However, GitHub is popular, and there are humans around who are able to help.

Most likely, you'll be able to search for a problem which many others have encountered and have already asked for help. There are also GitHub alternatives that can interface with the GitHub service.

Even better, there are plug-ins for as little as \$5 that integrate with Unity 3D and can directly interface with GitHub from inside of the Unity 3D editor. It's worth some effort to investigate various source control software options as there is no clear winner. Many professional studios prefer using PerForce, a very complete package that includes tools to help merging multiple files. However, at the price its offered at per seat (the term used to describe software licensing), they exclude many independent developers.

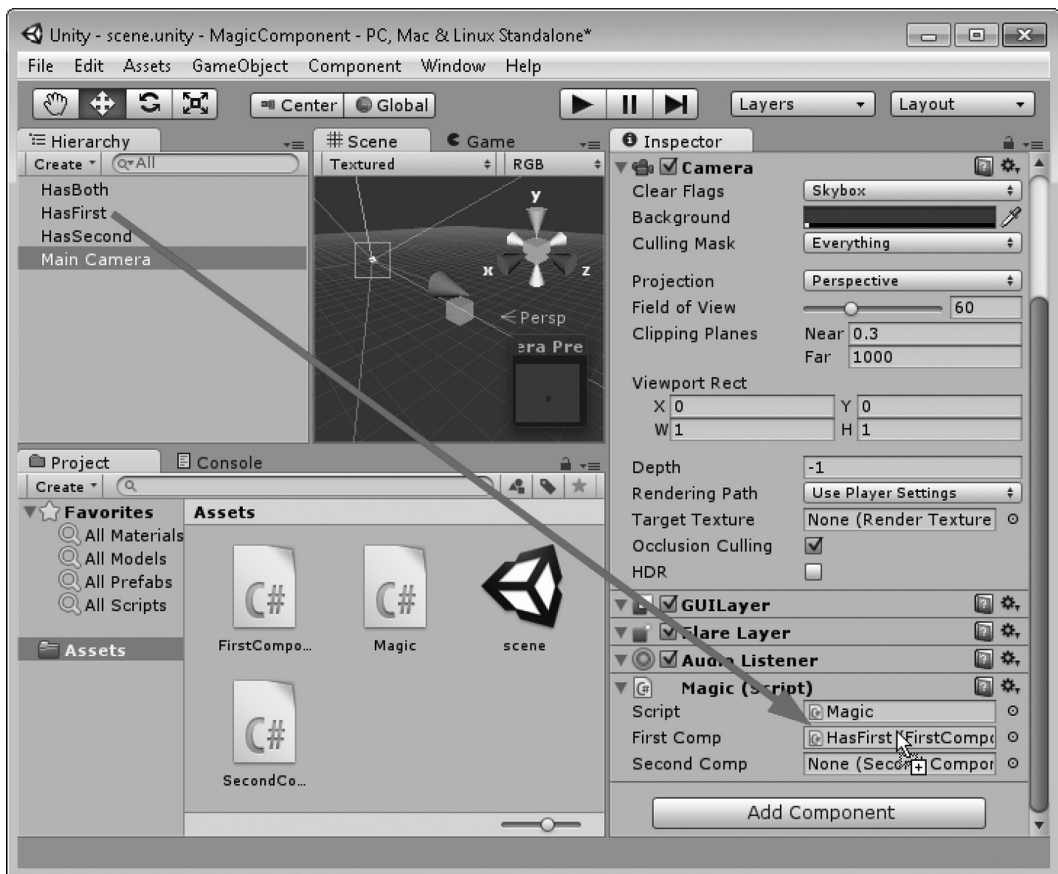
5.17 Leveling Up: On to the Cool Stuff

We've come further along than most people who set out to learn how to program. By now, you should be able to read through some code and get some idea of how it's working. We've gotten through some of the tough beginnings of learning how to write in C#, but things are just about to get more interesting.

The clever tricks that programmers use to save time are about to start to come into play. If you take the time, you should review Chapters 1 through 4 to strengthen your grasp of the core concepts we've been studying.

There's an interesting topic that hasn't found a place in any chapters yet. If you open the MagicComponent project in Unity 3D, you'll be able to see some interesting behavior. Say we have an object in the scene with a component attached. We'll look at something called `FirstComponent.cs` attached to a cube.

If the Main Camera has a component attached that has a couple of variables that are set for `FirstComponent` and `SecondComponent`, how do we add the `GameObject`'s component to that slot in the Inspector panel? We drag it, of course!



Therefore, even though the variable isn't `GameObject`, you can still drag the object from the scene into the Inspector panel. Unity 3D is clever enough to find a component of the same type as the slot in the Inspector.

This feature also means that any variable changed on the object in the scene will have a direct connection to the `Magic.cs` class that is holding onto the variable. This means a couple of strange things. First, the objects in the scene are actually instances, but they are acting in a rather weird limbo. Second, the connection between variables is stored in some magic data stored in the scene file.

The weird limbo infers that when you drag a class onto an object in the editor, it is immediately instantiated. This behavior is unusual in that this only happens in the Unity Editor, your player and the finished game never has any behavior similar to this. The class instances an object and attaches it to the `GameObject` in the scene, as though you used `GameObject.Instance()` or even `new FirstComponent();`, or perhaps `Component.Add();`, and added it to the `GameObject` in the scene.

The fact that the `Start ()` and `Update ()` functions aren't called means that the game isn't running. However, the class is still an instanced object created by the `cs` file in the Assets Directory. There are editor scripts that do run while the game might not be playing. Editor scripts are often used to help game designers.

Editor scripts can be used to create tools and other fun interfaces specifically for the Unity 3D Editor. These are often not running while the game is playing. We shouldn't venture too far into Unity 3D-specific tricks as this distracts us from learning more about C#, but it's fun to know what to look for if we want to build our own C# tools for Unity 3D.

The strange fact that the variable in `Magic.cs` has a handle on an object in the scene without having to run `GameObject.Find();` or `Transform.Find();` means that there's a lot going on in the scene file. All of this is actually stored in the meta files that seem to be appearing as you work with others on Unity 3D projects.

These meta files are required for these magic data connections between objects in a Unity 3D Scene file. Of course, other file settings and project settings are also stored in these meta files. If you open one, you'll find many things called GUIDs, which stands for globally unique identifiers, and they're used in meta files to find a specific object in the scene and apply settings to them.

It wasn't long ago when game engines referenced everything in the scene by their GUID alone. Today, they're at least hidden from view, but they still affect our lives as game programmers. We can't reference a specific object by name alone. If you've got several objects named `GameObject`, then you're going to have a rough time remembering which one actually has the data you're looking for. Because of this, everything is assigned a GUID and that's just the way it's gotta be.

