

6

Intermediate

Chapters 7 and 8 are big and dense. In many ways, they're exciting as well. Getting into a programming language means discovering many cool tricks. The basics of any programming language have already been covered. To be honest, much of what you need has already been covered.

From here on out, you'll be learning some of the more refined systems that make the C# language elegant and concise. Rather than having long chains of `if-else` statements, you can use a `switch` statement. Rather than having a large uncontrolled nest of different variables, you can group them together in a struct.

Don't worry about not making rapid progress; learning is more about the journey than it is about the goal. Of course, if you do have a goal of making a featured mobile game, then by all means speed forth toward your goal. Just try not to skip around too much; there's much stuff that is explained for a reason.

6.1 What Will Be Covered in This Chapter

The long journey begins. We're going to cover some important concepts here, including the following:

- Pseudocode
- Class constructors
- Arrays
- Enums
- Switch statement
- Structures
- Namespaces
- More on functions
- More on inheritance
- More on type casting
- Some work on vectors
- Goto
- Out parameter
- Ref parameter
- Operator overloading

This chapter discusses a few concepts more than once, or rather, we look at different aspects of type casting, for instance, but we divide the concept over more than one chapter. In between the chapters, we cover a few other concepts that do relate back to the topic. Learning a single concept can mean taking in a few separate concepts before coming back to the original topic.

6.2 Review

By now you've been equipped with quite a vocabulary of terms and a useful set of tools to take care of nearly any basic task. Moving forward, you'll only extend your ability and come to grips that you've still got a long way to go. Even after years of engineering software, there's always more to learn. Even once

you think you've covered a programming language, another version is released and you need to learn more features.

Aside from learning one language, most companies will eventually toss in a task that involves learning another programming language. In a modern setting, it's likely you'll need to learn some of the programming languages while on the job. This might involve knowing something like Scala or PHP, yet another programming language you'll have to learn.

Learning additional programming languages isn't as daunting as you might think. Once you know one language, it's far easier to learn another. The keyword `var` is used in many other programming languages in the exact same context. It's a keyword used to store a variable. In C#, we see `var v = "things";`, which is similar to JavaScript where `var v = "things";`.

So far the principles that have been covered in this book are general enough such that almost every other programming language has similar concepts. Nearly all languages have conditions and operators. Most languages use tokens and white space in the same way as well.

6.3 Pseudocode

In a practical sense, you can do anything you set out to do.

It's just about time now to start trying to think like a programmer. Armed with functions, logic, and loops, we can start to do some more interesting things with Unity 3D. To do this, we're going to learn more about what Unity 3D has in the way of functions which we can use.

There's hardly any feature implemented in a game you've played that you too can't do, given the time. Everything that is being done using Unity 3D is something you too can do. To create a game object, they use the same function we used in Chapter 5.

Programmers start with a task. For this chapter, we're going to move the cube around with the keyboard. Therefore, we're going to do two things: First, we need to figure out how to read the keyboard and then we need to change the cube's position.

6.3.1 Thinking It Through

A common exercise is to write pseudocode or at least think about what the code will look like. We need to have a section for reading keyboard inputs, acting on the keyboard inputs, and then moving the cube. This is going to happen on every frame, so we need to put this in the `Update ()` function.

We need at least four inputs: forward, backward, left, and right. Unity 3D is a 3D engine, so we're going to use `x` for left and right, and `z` for forward and backward. This might look like the following.

```
bool moveForward
bool moveBackward
bool moveLeft
bool moveRight
```

However, we need to set those somehow with an input command from the keyboard, which might look like the following:

```
moveForward = keyboard input w
moveBackward = keyboard input s
moveLeft = keyboard input a
moveRight = keyboard input d
```

So far everything makes sense. We need to move the cube so that it involves a `Vector3`, since everything that has a position is using `transform.position` to keep track of where it is. Therefore, we should set the `transform.position` to a `Vector3`. We should have an updated position to start with.

```
Vector3 updatedPosition = starting position
```

We should have a starting position, so the `Vector3` isn't initialized with 0, 0, 0 when we start. We might want to start in the level somewhere else other than the scene of origin. However, we need to move to the `Vector3` that we're modifying.

```
Transform.position = updatedPosition
```

Now we should think about modifying the `updatedPosition` when a key is pressed. Therefore, something like the following should work:

```
If moveForward updatedPosition.z = updatedPosition.z + 0.1
If moveBackward updatedPosition.z = updatedPosition.z - 0.1
If moveLeft updatedPosition.x = updatedPosition.x - 0.1
If moveRight updatedPosition.x = updatedPosition.x + 0.1
```

We've set positions before with variables, which was pretty easy. Therefore, the only thing we need to figure out is the keyboard input.

6.3.2 Class Members

Classes provided by Unity 3D become accessible when you add `using UnityEngine` at the top of your C# code. All of the classes found inside of that library have names which you can use directly in your own code. This includes `Input`, and all of the members of `Input` are imported into your code.

6.3.2.1 A Basic Example

Let's start off with creating a new class that will have some functions in it. We've done this before by creating a new C# class in the Project panel. This can be found in the `PseudoCode` project from the repository.

```
using UnityEngine;
using System.Collections;
public class Members
{
    public void FirstFunction ()
    {
        print("First function");
    }
}
```

Create a new class called `Members.cs` because we're going to use members of the `Members` class. Then we'll add `public void FirstFunction()` with a basic `print("First Function");` to print to the Console panel when it's used.

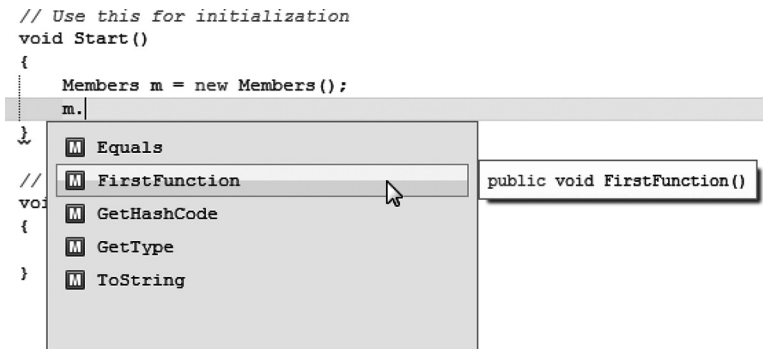
There are two classifications for the stuff that classes are made of: data members and function members. As you might imagine, function members are the lines of code that execute and run logic on data. Data members are the variables that are accessible within the class.

In our Example class we've been abusing a lot, we'll add in the following line to the `Start ()` function:

```
//Use this for initialization
void Start ()
{
    Members m = new Members();
}
```

This creates what's called an instance. We're assigning `m` to being a new copy or instance of the `Members` class. The keyword `new` is used to create new instances. `Members();` is used to tell the new keyword what class we're using. It seems like we're using the class as a function, and to a limited extent we are, but we'll have to leave those details for Section 6.3.3.1.

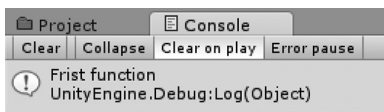
The `m` variable that is an instance of the `Members` class is now a thing we can use in the function. When we use `m`, we can access its members.



Type “`m.`” and a new pop-up will appear in MonoDevelop showing a list of things that `m` can do. `FirstFunction`, as well as a few other items, is in the list. These other items were things that `Object` was able to do. Accessing the different functions within a class is a simple matter of using the dot operator.

```
void Start ()
{
    Members m = new Members();
    m.FirstFunction();
}
```

Finish off the `m.` by adding `FirstFunction()`; . Running the script when it’s attached to a cube or something in the scene will produce the following Console output. Objects and its members have many uses, though; this might not be clear from this one example.



6.3.2.2 Thinking like a Programmer

A peek into a programmer’s thinking process reveals a step-by-step process of evaluating a situation and finding a solution for each step. Logic takes the front seat in this case as the programmer needs to not only investigate each function he or she must use but also clean up the thinking process as he or she writes code.

Intuition and deduction is a huge part of programming. The programmer needs to guess what sort of functions he or she should be looking for. With each new development environment, the programmer needs to learn the software’s application programming interface, the interface between you (the programmer) and the application you’re working with, in this case, Unity 3D.

At this point, we have some basic tools to do something interesting with our game scene. However, to make a game, we’re going to read the keyboard mouse and any other input device that the player might want to use. A appropriate place to start would be looking up the word “input” in the Solution Explorer in MonoDevelop.

Dig into the `UnityEngine.dll` and expand the `UnityEngine` library. You should find an `Input` class that looks very promising. From here, we can start hunting for a function inside of `input` that might be useful for us. At this point, I should mention that programmers all seem to use a unique vocabulary.

The words “Get” and “Set” are often used by programmers for getting and setting values. For getting a keyboard command, we should look for a function that gets things for us. We can find “`GetKey`” in the numerous functions found in `Input`.


```

+ public static bool GetKey (string name) ...
+ public static bool GetKey (KeyCode key) ...
+ public static bool GetKeyDown (string name) ...
+ public static bool GetKeyDown (KeyCode key) ...
+ public static bool GetKeyUp (string name) ...
+ public static bool GetKeyUp (KeyCode key) ...

```

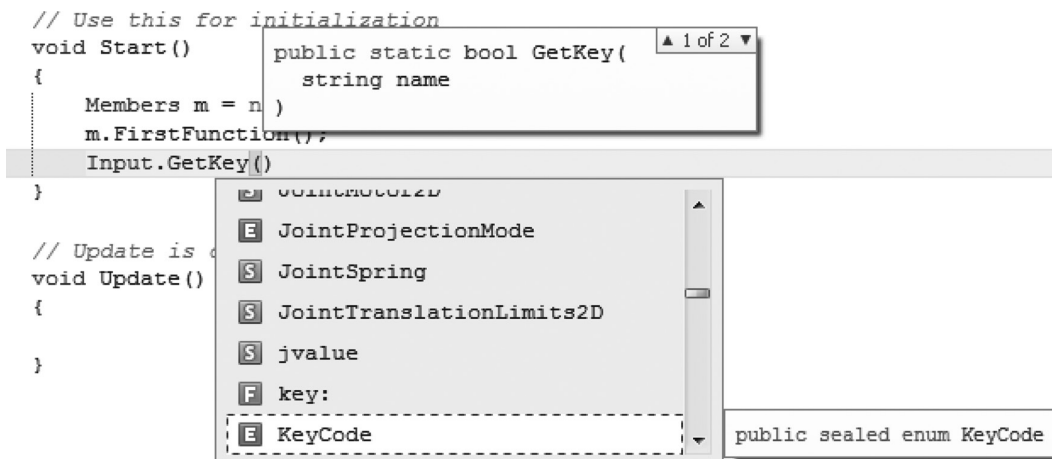
There seems to be `string name` and `KeyCode key` for each one of the functions. We did mention that reusing names for things means erasing a previously used name. However, this doesn't always hold true. In the case of functions, when you duplicate a name, you're allowed to share the name, as long as you do something different with the arguments found in parentheses. This is called overriding, and we'll learn more about that in Section 6.13.1. It's just good to know what you're looking at so you don't get too lost.

So now that we've found something that looks useful, how do we use it? The "GetKey" function looks like its public function, which is good. Functions with the keyword `public` are functions that we're allowed to use.

The context when using the dot operator in numbers changes an `int` to a `double` or `float` if you add in an `f` at the end of the number. When you add the dot operator after the name of a class found in `UnityEngine`, you're asking to gain access to a class member. In this case, we found `GetKey` inside of `Input`, so to talk to that function we use the dot operator after `Input` to get to that function.

There were two `GetKey` functions: The first had "`KeyCode key`" and the second had "`string name`" written in the arguments for the function. This means we have two options we can use.

When you add the first parenthesis, `MonoDevelop` pops up some helpers.



These help you fill in the blanks. There are many `KeyCode` options to choose from. You can see them all by scrolling through the pop-up. I picked the `KeyCode.A` to test this out. I'm guessing that pressing the A key on the keyboard is going to change something. In the `Example.cs` file, add in the following:

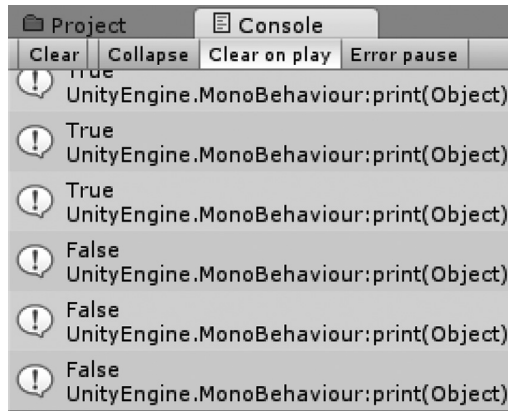
```

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

```

We're setting the `bool AKey` to this function; why a `bool` and why even do this? Remember that the function was designated as a `public static bool`. The last word is the reason why we're using a `bool AKey`. The variable type we're setting matches the return type of the function.

Finally, we're printing out the value of `AKey`. Run the game and press the A key on your keyboard and read the Console output from Unity 3D.



When the A key is down, we get True; when it's not down, we get False. It looks like our hunch was correct. Our AKey bool is now being controlled by the Input class' member function GetKey. We know the return type of GetKey because of the bool that was written just before the name of the function. We also know how to access the function inside of Input through the dot operator.

Let's keep this going. Make a bool for WKey, SKey, and DKey. This will allow us to use the classic WASD keyboard input found in many different games. Then we'll make them set to the different GetKeys that we're going to use from Input.

Now we're going to make our cube move around, so we're going to look at the Transform class. To move our cube around, we're going to keep track of our current position. To do this, we're going to make a Vector3 called pos as a class scoped variable.

```

5      public Vector3 pos;
6      // Use this for initialization
7      void Start () {
8          pos = transform.position;
9      }

```

The first thing we want to do is set the pos to the object's current position when the game starts. This means we can place the cube anywhere in the scene at the beginning of the game and the pos will know where we're starting.

```

12     void Update () {
13         bool AKey = Input.GetKey(KeyCode.A);
14         bool SKey = Input.GetKey(KeyCode.S);
15         bool DKey = Input.GetKey(KeyCode.D);
16         bool WKey = Input.GetKey(KeyCode.W);
17         transform.position = pos;
18     }

```

In the Update () function which is called, we're going to set the transform.position of the cube to pos in each frame. This means that if we change the x, y, or z of the pos Vector3 variable, the cube will go to that Vector3. Now all we need to do is change the pos.x and pos.z when we press one of the keys.

```

Vector3 pos = Vector3.zero;
void Update ()
{
    bool AKey = Input.GetKey(KeyCode.A);
    bool SKey = Input.GetKey(KeyCode.S);
    bool DKey = Input.GetKey(KeyCode.D);
    bool WKey = Input.GetKey(KeyCode.W);

```

```

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

```

I've added an `if` statement controlled by each `bool` we created at the beginning of the `Update ()` function. Then we changed the `pos.x` and `pos.z` according to the direction I wanted the cube to move in by adding or subtracting a small value. Try this out and experiment with some different values.

This is not the only solution, nor is it the best. This is a simple solution and rather restricted. The speed is constant, and the rotation of the cube is also fixed. If we want to improve on this solution, we're going to need a better way to deal with many variables.

A big part of programming is starting with something basic and then refining it later on. You start with the things you know and add to it stuff you figure out. Once you learn more, you go back and make changes. It's a process that never ends. As programmers learn more and figure out more clever tricks, their new code gets written with more clever tricks.

6.3.3 Return

`return` is a powerful keyword; it's a very clever trick. It's used in a couple of different ways, but certainly the most useful is turning a function into data. If a function doesn't give out data, it's given the return type `void`. For instance, we've been using `void Start ()` and `void Update ()`.

`void` indicates that the function doesn't need to return any data; using the keyword `return` in a function like this returns a `void` or a nothing. Any keywords that precede the `return` type modify how the function operates; for instance, if we wanted to make a function available to other classes, we need to precede the `return` type with the keyword `public`.

6.3.3.1 A Basic Example

Using the `return` keyword gives a function a value. You're allowed to use the function name as though it were the function's return value.

```

void Start ()
{
    print(ImANumber());
}
int ImANumber()
{
    return 3;
}

```

When you run this basic example, you'll see a 3 printed out in the Console panel. To prove that `ImANumber()` is just a 3, you can even perform math with the function. In the following example, you'll get many 6s being printed out to the Console panel.

```

void Start ()
{
    print (ImANumber() + ImANumber());
}
int ImANumber()
{
    return 3;
}

```

Most often when we want to reduce complexity within a single function, we need to separate our code into other smaller functions. Doing so makes our work less cluttered and easier to manage. Once the code has been separated into different smaller functions, they become individual commands that can contain their own complexity.

We're going to reduce the number of lines required in the `Update` loop. To do this, we'll write our own function with a return type `Vector3`. Reducing the number of lines of code you have to muddle through is sometimes the goal of clean code.

`ImANumber()` isn't a variable; it's a function. In other words, you will not be able to assign something to `ImANumber()` as in the following example:

```
ImANumber() = 7;
```

There are ways to do something similar to this. We'll need to use the contents in parentheses to assign `ImANumber()` a value for it to return.

6.3.4 Arguments aka Args (Not Related to Pirates)

We've seen the use of arguments (also known as args) earlier when we initialized a "`Vector3(x,y,z);`" with three parameters. To start, we'll write a function that's very simple and takes one arg.

6.3.4.1 The Basic Example

```

void Start ()
{
    print(INeedANumber(1));
}
int INeedANumber(int number) {
    return number;
}

```

We start with an `int INeedANumber (int number)` as our function. The contents of this function are filled with `int number`, indicating two things. First is the type that we expect to be in our function's argument list as well as a name or an identifier for the `int` argument. The identifier `number` from the argument list exists throughout the scope of the function.

```

void Start () {
    int val = INeedANumber(3) + INeedANumber(7);
    print (val);
}
int INeedANumber(int number) {
    return number;
}

```

In this second example, we use the `INeedANumber()` function as a number. It just so happens to be the same number we're using in its argument list. When we print out `val` from this, we get 10 printed to the Console panel. However, this doesn't have to be the case.

```
int INeedANumber(int number) {
    return number + 1;
}
```

If we were to modify the return value to `number + 1` and run the previous example, we'd have 12 printed out to the Console panel. This would be the same as adding 4 and 8, or what is happening inside of the function `3 + 1` and `7 + 1`.

6.3.4.2 Multiple Args

When you see functions without anything in parentheses, programmers say that the function takes no args. Or rather, the function doesn't need any arguments to do its work. We can expand upon this by adding another argument to our function. To tell C# what your two arguments are, you separate them with a comma and follow the same convention of type followed by identifier.

```
void Start ()
{
    int val = INeedTwoNumbers(3, 7);
    print (val);
}
int INeedTwoNumbers (int a, int b)
{
    return a + b;
}
```

MonoDevelop automatically knows what your argument list looks like and pops up a helper to let you know what you add into the function for it to work. This function then takes the two arguments, adds them together, and then prints out the result to the Unity 3D's Console panel.

```
int val = INeedTwoNumbers();
print(val);
: INeedTwoNumbers(int a, int b)
return a + b;
```

Just for the sake of clarity, you're allowed to use any variety of types for your args. The only condition is that the final result needs to match the same type as the return value. In more simple terms, when the function is declared, its return type is set by the keyword used when it's declared.

This also includes data types which you've written. We'll take a look at that in a bit, but for now we'll use some data types we've already seen.

```
int INeedTwoNumbers (int a, float b)
{
    return a * (int)b;
}
```

Mixing different types together can create some interesting effects, some of which might not be expected. We'll study the consequences of mixing types later on as we start to learn about type casting, but for now just observe how this behaves on your own and take some notes. To convert a `float` value to an `int` value, you precede the `float` with the number type you want to convert it to with `(ToType)` `DifferentType`, but we'll get into type conversions again later.

So far we've been returning the same data types going into the function; this doesn't have to be the case. A function that returns a boolean value doesn't need to accept booleans in its argument list.

```
bool NumbersAreTheSame (int a, int b)
{
    bool ret;
    if (a == b) {
        ret = true;
    } else {
        ret = false;
    }
    return ret;
}
```

In this case, if both `a` and `b` are the same number, then the function returns `true`. If the two numbers are different, then the function returns `false`. This works well, but we can also shorten this code by a couple of lines if we use more than one `return`.

```
bool NumbersAreTheSame (int a, int b)
{
    if (a == b) {
        return true;
    } else {
        return false;
    }
}
```

The `return` keyword can appear in more than one place. However, this can cause some problems. If we return a value based on only a limited case, then the compiler will catch this problem.

```
bool NumbersAreTheSame (int a, int b)
{
    if (a == b)
    {
        return true;
    }
}
```

This example will cause an error stating the following:

```
Assets/Example.cs(16,6): error CS0161: 'Example.NumbersAreTheSame(int, int)':
not all code paths return a value
```

The rest of the possibilities need to have a return value. In terms of what Unity 3D expects, all paths of the code need to return a valid `bool`. The return value always needs to be fulfilled, otherwise the code will not compile.

6.3.4.3 Using Args

Doing all of these little changes repeatedly becomes troublesome, so they leave some things up to other people to change till they like the results. Each function in the `Input` class returns a unique value. We can observe these by looking at the results of `Input.GetKey()`.

```
void Update () {
    print (Input.GetKey(KeyCode.A));
}
```

Once you type in `Input`, a pop-up with the members of the `Input` class is shown. Among them is `GetKey`. Once you enter `GetKey` and add in the first parenthesis, another pop-up with a list of various inputs is shown. Choose the `KeyCode.A`. Many other input options are available, so feel free to experiment with them on your own.

To test this code out, run the game and watch the output in the Console panel. Hold down the “a” key on the keyboard and watch the `false` printout replaced with `true` when the key is down. Using the `print()` function to test things out one at a time is a simple way to check out what various functions do. Test out the various other keys found in the `Input.GetKey()` function.

```
Vector3 Movement(float dist)
{
    Vector3 vec = Vector3.zero;
    if (Input.GetKey(KeyCode.A))
    {
        vec.x -= dist;
    }
    if (Input.GetKey(KeyCode.D))
    {
        vec.x += dist;
    }
    if (Input.GetKey(KeyCode.W))
    {
        vec.z += dist;
    }
    if (Input.GetKey(KeyCode.S))
    {
        vec.z -= dist;
    }
    return vec;
}
```

We should add an argument to the `Movement()` function. With this we’ll add in a simple way to change a variable inside of the `Movement()` function and maintain the function’s portability. Replace the `0.1f` value with the name of the argument. This means that anything put into the function’s argument list will be duplicated across each statement that uses it.

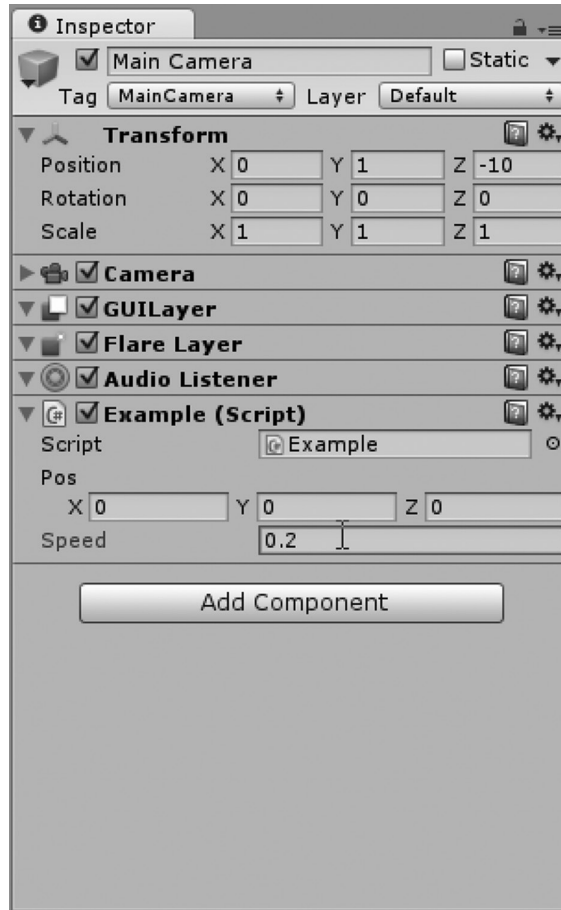
This means we need to pass in a parameter to the argument list in the `Movement()` function. We can test by entering a simple float, which is what the `Movement()` argument list expects.

```
void Update ()
{
    transform.position += Movement(0.2f);
}
```

This means we’re just reducing the number of places a number is being typed. We want to make this number easier to edit and something that can be modified in the editor.

```
using UnityEngine;
using System.Collections;
public class Example : MonoBehaviour
{
    public Vector3 pos = Vector3.zero;
    public float speed;
    //the rest of your code below...
```

Add in a public float so that the Inspector panel can see it. Then add the variable to the `Movement()`’s parameter list.



Change the value for `Delta` in the Inspector panel and run the game. Now the WASD keys will move the cube around at a different speed, thanks to the use of a public variable. There are a few different ways to do the same thing. For instance, we could have ignored using the arguments and used `Delta` in place of `Dist`. However, this means that the function would rely on a line of code outside of the function to work. Everywhere you want to use the function, you'd have to write in a public `float Delta` statement at the class level.

6.3.5 Assignment Operators

Operators that fill variables with data are called assignment operators. We've been using `=` to assign a variable a value in the previous chapters. There are different types of assignment operators that have added functionality.

```
void Update ()
{
    transform.position += new Vector3(0.1f, 0.0f, 0.0f);
}
```

Introduce the `+=` operator to the `Vector3`. When operators are used in pairs there are no spaces between them. The `+` operator adds two values together. The `=` operator assigns a value to a variable. If there was white space between the `+` and the `=`, this would raise a syntax error. The `+=` operator allows you to add two vectors together without having to use the original variable name again. This also works with single numbers. An `int MyInt += 1;` works just as well. Rather than having to

use `pos.z = pos.z + 0.1f`, you can use `pos.z += 0.1f`, which is less typing. If you run the code up above, then you'll see your cube scooting off in the positive x direction.

6.3.5.1 A Basic Example

As we have seen with the `++` and the `-` unary operators, the `+=` operator and its negative version, the `-=`, work in a similar way. One important and easy-to-forget difference is the fact that the `++/--` works only on integer data types. The `+=` and `-=` work on both integer and floating point data types.

```
float f = 0;
void Update ()
{
    f += 0.25f;
    print(f);
}
```

In this example, the `f` variable will be incremented 0.25 with each update. In the code fragment above, this increase is something that cannot be done with an integer. This is similar to using `f = f + 0.25f`; though the `+=` is a bit cleaner looking. The change is primarily aesthetic, and programmers are a fussy bunch, so the `+=` is the preferred method to increment numbers.

A part of learning how to program is by decoding the meaning behind cryptic operators, and this is just one example. We're sure to come across more mysterious operators, but if you take them in slowly and practice using them, you'll learn them quickly.

```
void Update ()
{
    transform.position += Movement();
}
Vector3 Movement()
{
    return new Vector3(0.1f, 0.2f, 0.3f);
}
```

Rather than using a new `Vector3` to add to the `transform.position`, we want to use a function. To make this work, the function has to have a `Vector3` return type. For a function to return the type, we need to include the keyword `return` in the function.

This function is now a `Vector3` value. Based on any external changes, the value returned can also change. This makes our function very flexible and much more practical. Again, the cube will scoot off to the x if you run this code. This and the previous examples are doing the exact same thing. Don't forget that C# is case sensitive, so make sure the `vec` is named the same throughout the function.

```
void Update ()
{
    transform.position += Movement(0.2f);
}
Vector3 Movement(float dist)
{
    Vector3 vec = Vector3.zero;
    if (Input.GetKey(KeyCode.A))
    {
        vec.x -= dist;
    }
    if (Input.GetKey(KeyCode.D))
    {
        vec.x += dist;
    }
}
```

```

        if (Input.GetKey(KeyCode.W))
        {
            vec.z += dist;
        }
        if (Input.GetKey(KeyCode.S))
        {
            vec.z -= dist;
        }
        return vec;
    }
}

```

If `+=` works to add a number to a value, then `-=` subtracts a value from the number. There are some more variations on this, but they're going to have to wait for Section 8.9.

The `Movement()` function is portable. If you copy the function into another class, you have made reusable code. As any programmer will tell you, you should always write reusable code. The function operates mostly on its own. It relies on very few lines of code outside of the function. There are no class scoped variables it's depending on, so you need to only copy the lines of code existing in the function.

This function is not completely without any external dependencies. The `Input.GetKey` function does rely on the `UnityEngine` library, but there are ways to reduce this even further. To figure out other ways to reduce complexity and dependencies, we need to learn more tricks.

Wherever you need to read input and return a `Vector3` to move something, you can use this function. Copy and paste the function in any class, and then add the `transform.position += Movement();` into the `Update ()` function of that object, and it will move when you press the WASD keys.

6.3.6 What We've Learned

The earlier discussion was an introduction to arguments and return values. We started to learn a bit about how to use members of classes and how to find them. There's still a lot more to learn, but I think we have a pretty good start and enough of a foundation to build on by playing with things in the game engine.

It's time to start having functions talk to other functions, and to do this, we're going to start writing some more classes that can talk to one another. Accessing other classes is going to require some more interesting tricks, and with everything we learn, we expand the tools that we can build with.

6.4 Class Constructors

A class is first written as in the following example:

```

class Zombie
{
}

```

We start off with very little information. As we assign member fields and member functions, we create places for the class to store data and provide the class with capability.

```

class Zombie
{
    public string Name;
    public int brainsEaten;
    public int hitPoints;
}

```

The use of these parameters is inferred by their name and use. When we create a new `Zombie()` in the game, we could spend the next few lines initializing his parameters. Of course, we can add as many parameters as needed, but for this example, we're going to limit ourselves to just a few different objects.

```

void Start ()
{
    Zombie rob = new Zombie();
    rob.Name = "Zombie";
    rob.hitPoints = 10;
    rob.brainsEaten = 0;
}

```

One simple method provided in C# is to add the parameters of the classes in a set of curly braces. Each field in `Zombie` is accessible through this method. Each one is separated by a comma, not a semicolon.

```

void Start () {
    Zombie rob = new Zombie(){
        Name = "Zombie",
        brainsEaten = 0,
        hitPoints = 10
    };
}

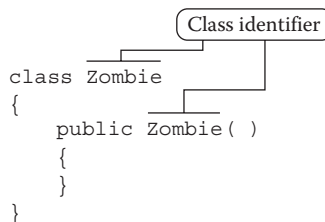
```

Note the trailing `;` after the closing curly brace. This system doesn't do much to shorten the amount of work involved. With each public data field provided in the `Zombie` class, we need to use the name of the field and assign it. Doing this for every new zombie might seem like a bit of extra work.

While building prototypes and quickly jamming code together, shortcuts like these can inform how we intend to use the class object. Rather than coming up with all possibilities and attempting to predict how a class is going to be used, it's usually better to use a class, try things out, and then make changes later.

6.4.1 A Basic Example

A class constructor would save a bit of extra work. Open the `ZombieConstructor` project to follow along. You might have noticed that the statement `Zombie rob = new Zombie();` has a pair of parentheses after the class it's instanting. When a class `Zombie()` is instantiated, we could provide additional information to this line. To enable this, we need to add in a constructor to the `Zombie()`'s class. This looks like the following:



```

class Zombie
{
    public Zombie( )
    {
    }
}

```

To give this example more meaning, we'll use the following code:

```

class Zombie
{
    public string Name;
    public int brainsEaten;
    public int hitPoints;
    public Zombie()
    {
        Name = "Zombie";
        brainsEaten = 0;
        hitPoints = 10;
    }
}

```

After the data fields, we create a new function named after the class. This function is called a class constructor. The function `public Zombie()` contains assignment statements that do the same thing as the previous class instantiation code we were using.

```
Zombie rob = new Zombie();
```

This statement `Zombie rob = new Zombie();` invokes the `Zombie()` constructor function in the `Zombie` class. When the constructor function is called, `Name`, `brainsEaten`, and `hitPoints` are all assigned at the same time. However, this will assume that every zombie is named “Zombie,” has eaten no brains, and has 10 hitpoints. This is not likely the case with all zombies. Therefore, we’d want to provide some parameters to the class constructor function.

```
public Zombie(string n, int hp)
{
    Name = n;
    brainsEaten = 0;
    hitPoints = hp;
}
```

By adding in a few parameters to the interface, we’re allowed to take in some data as the class is instantiated.

```
void Start ()
{
    Zombie rob = new Zombie("Rob", 10);
}
```

So now when we create a new zombie, we’re allowed to name it and assign its hitpoints all at the same time without needing to remember the names of the data fields of the classes. When the constructor is invoked, the first field corresponds to a `string n`, which is then assigned to `Name` with the `Name = n;` statement. Next we will assume that a new zombie has not had a chance to eat any brains just yet, so we can assign it to 0 when it’s instantiated. Finally, we can use the second argument `int hp` and use that to assign to the `hitPoints` with the `hitPoints = hp;` statement.

6.4.2 What We’ve Learned

Class constructors allow us to instantiate classes with unique data every time a new class is instantiated. Putting this into use involves a few extra steps.

```
class Zombie
{
    public string Name;
    public int brainsEaten;
    public int hitPoints;
    GameObject ZombieMesh;
    public Zombie(string n, int hp)
    {
        Name = n;
        brainsEaten = 0;
        hitPoints = hp;
        ZombieMesh = GameObject.CreatePrimitive(PrimitiveType.Capsule);
        Vector3 pos = new Vector3();
        pos.x = Random.Range(-10, 10);
        pos.y = 0f; //optional
        pos.z = Random.Range(-10, 10);
        ZombieMesh.transform.position = pos;
    }
}
```

By adding a mesh to each zombie, we can more directly observe the instantiation of a new zombie in a scene. To make this more clear, we create a vector with a random x and a random z position so that they can appear in different places.

```
void Start ()
{
    string[] names = new string[]{"stubbs", "rob", "white"};
    for(int i = 0; i < names.Length; i++)
    {
        Zombie z = new Zombie(names[i], Random.Range(10, 15));
        Debug.Log(z.Name);
    }
}
```

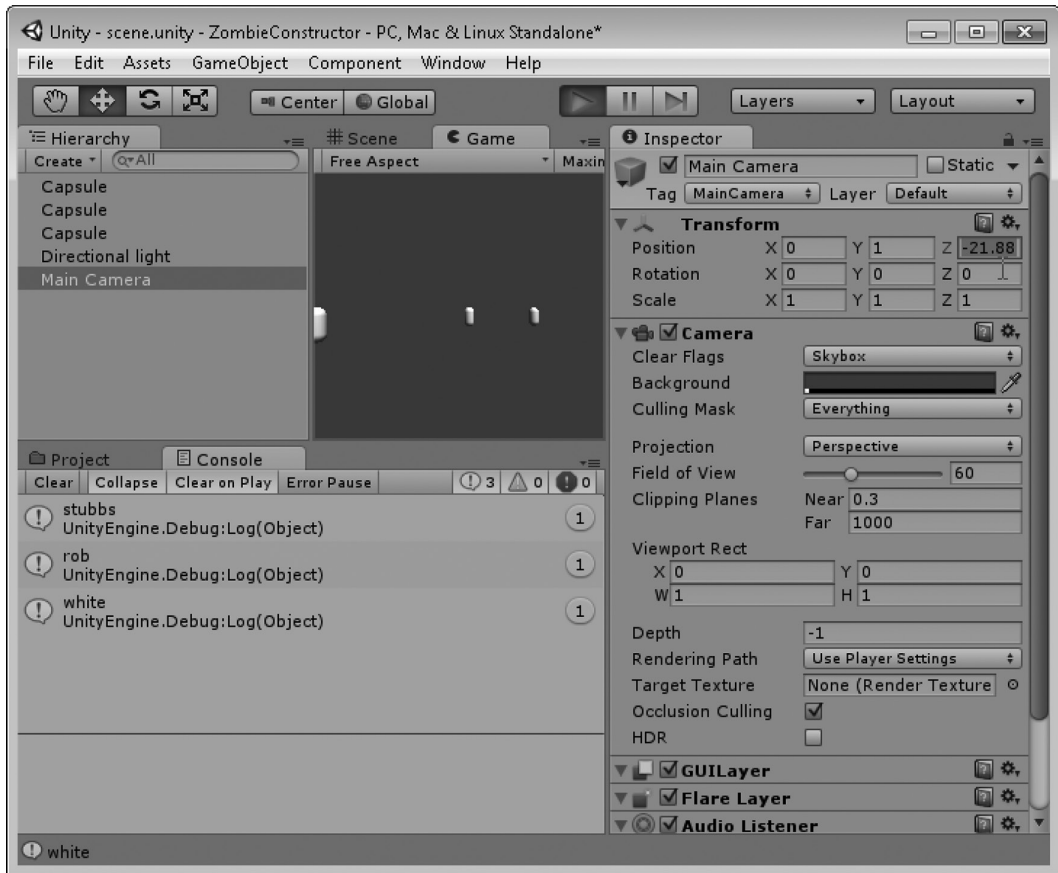
To make use of the different parameters, we create a new list of zombie names. Then in a `for` loop, we create a new zombie for each name in the list. For good measure, we assign a random number of hitpoints for each one with `Random.Range(10, 15)` which assigns a random number to each zombie between 10 and 15.

```
using UnityEngine;
using System.Collections;
public class Example : MonoBehaviour
{
    //Use this for initialization
    void Start () {
        string[] names = new string[]{"stubbs", "rob", "white"};
        for(int i = 0; i < names.Length; i++) {
            Zombie z = new Zombie(names[i], Random.Range(10, 15));
            Debug.Log(z.Name);
        }
    }
}
```

The full code of `Zombie.cs` should look like the following sample:

```
class Zombie
{
    public string Name;
    public int brainsEaten;
    public int hitPoints;
    GameObject ZombieMesh;
    public Zombie(string n, int hp)
    {
        Name = n;
        brainsEaten = 0;
        hitPoints = hp;
        ZombieMesh = GameObject.CreatePrimitive(PrimitiveType.Capsule);
        Vector3 pos = new Vector3(
            Random.Range(-10, 10), 0,
            Random.Range(-10, 10));
        ZombieMesh.transform.position = pos;
    }
}
```

The only new thing we've added here is the `string[]` object, which we'll get to next. Just so we know what's going on, we show a log of each zombie's name after it's created. We'll go further into arrays in Chapter 7; if the `string[]` is a bit confusing, we'll clear that up next.



6.4.3 What We've Learned

A class constructor is very useful and should almost always be created when you create a new class. We are also allowed to create multiple systems to use a constructor. We'll get around to covering that in Section 6.13.1 on function overrides, but we'll have to leave off here for now.

When building a class, it's important to think in terms of what might change between each object. This turns into options that can be built into the constructor. Giving each object the ability to be created with different options allows for more variations in game play and appearance. Setting initial colors, behaviors, and starting variables is easier with a constructor. The alternative would be to create the new object and then change its values after the object is already in the scene.

6.5 Arrays Revisited

By now, we're familiar with the bits of knowledge that we'll need to start writing code. In Chapter 7, we'll become more familiar with the integrated development environment known as MonoDevelop, and we'll go deeper into variables and functions.

Let's start off with a task. Programmers usually need something specific to do, so to stretch our knowledge and to force ourselves to learn more, we're going to do something simple that requires some new tricks to accomplish. If we're going to make a game with a bunch of characters, we're going to make and keep track of many different bits of information such as location, size, and type.

6.5.1 Using Arrays in Unity 3D

So far we've dealt with variables that hold a single value. For instance, `int i = 0;` in which the variable `i` holds only a single value. This works out fine when dealing with one thing at a time. However, if we want a whole number of objects to work together, we're going to have them grouped together in memory.

If we needed to, we could have a single variable for each box `GameObject` that would look like the following:

```
public class Example : MonoBehaviour
{
    public GameObject box1;
    public GameObject box2;
    public GameObject box3;
    public GameObject box4;
    public GameObject box5;
    public GameObject box6;
    public GameObject box7;
    public GameObject box8;
    public GameObject box9;
    public GameObject box10;
```

While this does work, this will give a programmer a headache in about 3 seconds, not to mention if you want to do something to one box, you'd have to repeat your code for every other box. This is horrible, and programming is supposed to make things easier, not harder on you. This is where arrays come to the rescue.

```
public class Example : MonoBehaviour
{
    public GameObject[] boxes;
```

Ah yes, much better, 10 lines turn into one. There's one very important difference here. There's a pair of square brackets after the `GameObject` to indicate that we're making an array of game objects. Square brackets are used to tell a variable that we're making the singular form of a variable into a plural form of the variable. This works the same for any other type of data.

```
//a single int
int MyInt;
//an array of ints
int[] MyInts;
//a single object
object MyObject;
//an array of objects
object[] MyObjects;
```

To tell the `boxes` variable how many it's going to be holding, we need to initialize the array with its size before we start stuffing it full of data. This will look a bit like the `Vector3` we initialized in Section 3.10.2. We have a couple of options. We can right out tell the `boxes` how many it's going to be holding.

```
public class Example : MonoBehaviour
{
    public GameObject[] boxes = new GameObject[10];
```

Or we can initialize the number of boxes using the `Start()` function and a public `int` variable.

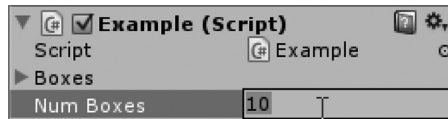
```
public class Example : MonoBehaviour
{
    public int numBoxes = 10;
```

```

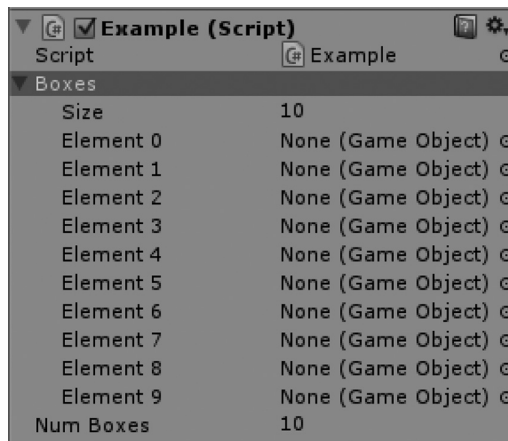
public GameObject[] boxes;
//Use this for initialization
void Start ()
{
    boxes = new GameObject[numBoxes];
}

```

In the above code, we're going to add a `numBoxes` variable and then move the initialization of the `boxes` variable to the `Start ()` function using the `numBoxes` to satisfy the array size. In the Unity inspector panel you'll see a field "Num Boxes" appear rather than `numBoxes`. Unity automatically changes variable names to be more human readable, but it's the same variable.



In the editor, we can pick any number of boxes we need without needing to change any code. Game designers like this sort of stuff. Once you run the game, the `boxes` array is initialized. To see what is contained in the array, you can expand the `boxes` variable in the Inspector panel and get the following:



We've got an array filled with a great deal of nothing, sure. However, it's the right number of nothing. Since we haven't put anything into the array of `GameObjects` we shouldn't be surprised. So far everything is going as planned. Testing as we write is important. With nearly every statement we write, we should confirm that it's doing what we think it should be doing. Next we should put a new cube primitive into each one of the parts of this array.

6.5.1.1 Starting with 0

Zeroth. Some older programming languages start with 1. This is a carryover from FORTRAN that was created in 1957. Other programming languages mimic this behavior. Lua, for example, is a more modern programming language that starts with 1. C# is not like these older languages. Here we start with 0 and then count to 1; we do not start at 1. Thus, an array of 10 items is numbered 0 through 9.

Now we have an empty array, and we know how many things need to be in it. This is perfect for using a `for` loop. Numbering in C# may seem a bit strange, but a part of numbering in programming is the fact that 0 is usually the first number when counting.

We get accustomed to counting starting at 1, but in many programming paradigms, counting starts with the first number, 0. You might just consider the fact that 0 was there before you even started

counting. Therefore, the first item in the `boxes` array is the 0th or zeroth item, not the first. It's important to notice that when dealing with arrays, you use square brackets.

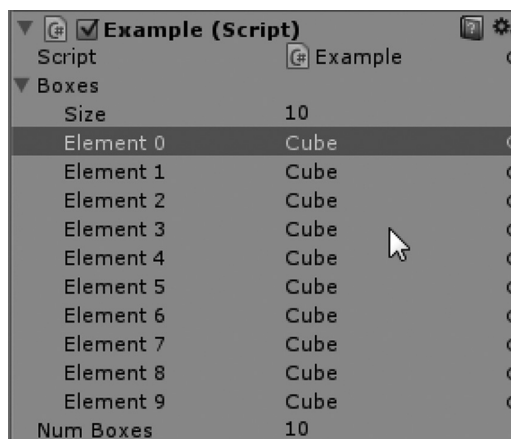
```
void Start ()
{
    boxes = new GameObject[numBoxes];
    for (int i = 0; i < numBoxes; i++)
    {
    }
}
```

Right after we initialize the `boxes` array, we should write the `for` loop. Then we'll create a new box game object and assign it to the `boxes`.

```
void Start ()
{
    boxes = new GameObject[numBoxes];
    for (int i = 0; i < numBoxes; i++)
    {
        GameObject box =
            GameObject.CreatePrimitive(PrimitiveType.Cube);
        boxes[i] = box;
    }
}
```

Notice the notation being used for `boxes`. The `boxes[i]` indicates the slot in the array we're assigning the box we just made. When the `i = 0`, we're putting a box into `boxes[0]`; when `i = 1`, we're assigning the box to `boxes[1]` and so on.

Items in the array are accessed by using a number in the square brackets. To check that we're doing everything right, let's run the code and check if the array is populated with a bunch of cube primitives just as we asked. Therefore, if you want to get some information on the fourth cube, you should use `boxes[3]`; again the 0th, pronounced "zeroth," item makes it easy to forget the index in the array we're referring to.



So far this is promising. We created an array called `boxes` with 10 items. Then we wrote a `for` loop that creates a box on every iteration and then adds that box into a numbered slot in the `boxes` array. This is working well. Arrays lend themselves very well for iterating through.

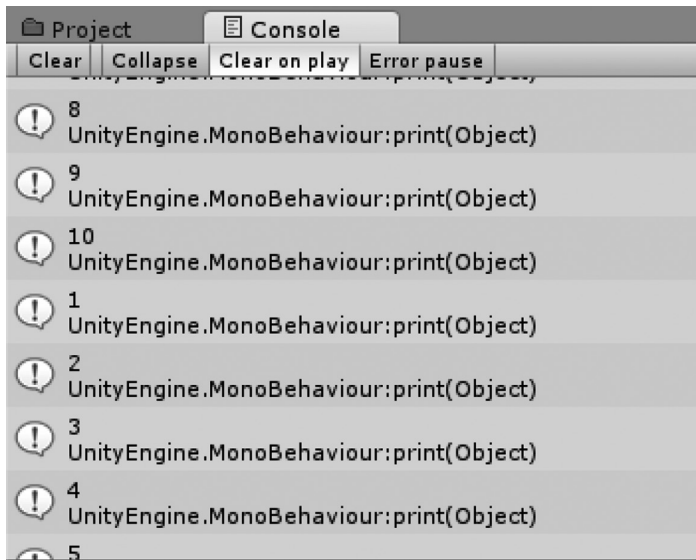
We need to match the variable type with the type that's inside of the array. Because the array `boxes[]` is filled with type `GameObject`, we need to use `GameObject` on the left of the `in` keyword. To the right of the `in` keyword is the array we're going to iterate through.

We're accessing all of the array, not just a single item in it, so we don't need to use something like `foreach(GameObject g in boxes[0])` which would infer that at index 0 of `boxes`, there's an array for us to iterate through. Though arrays of arrays are possible, it's not our goal here.

Iterating through arrays with the `foreach` doesn't give us the benefit of a counter like in the `for` loop, so we need to do the same thing as if we were using a `while` loop or a heavily rearranged `for` loop by setting up a counter ahead of time.

```
//Update is called once per frame
void Update ()
{
    int i = 0;
    foreach (GameObject go in boxes)
    {
        go.transform.position = new Vector3(1.0f, 0, 0);
        i++;
        print(i);
    }
}
```

Let's check our Console panel in the editor to make sure that this is working. It looks like the loop is repeating like one would expect. At the beginning of the `Update ()` function, `int i` is set to 0; then while the `foreach` loop is iterating, it's incrementing the value up by 1 until each item in `boxes` has been iterated through.



Put the `i` to some use within the `foreach` loop statement by multiplying the `x` of the vector by `i * 1.0f` to move each cube to a different `x` location. Note that multiplying any number against 1 is all that interesting, but this is just a simple part of this demonstration. Again, check in the editor to make sure that each cube is getting put in a different `x` location by running the game.

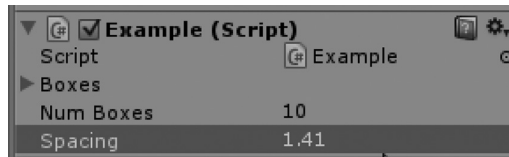
```
//Update is called once per frame
void Update ()
{
    int i = 0;
    foreach (GameObject go in boxes)
    {
        go.transform.position = new Vector3(i * 1.0f, 0, 0);
    }
}
```

```

        i++;
        print(i);
    }
}

```

So far everything should be working pretty well. You can change the offset by changing the value that the `i` is being multiplied by. Or, better yet, we should create a public variable to multiply the `i` variable by.



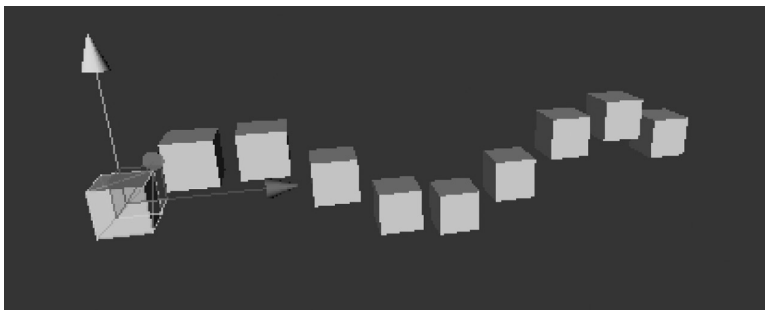
If you create a new public float `spacing` in the class scope, you can use it in your `foreach` loop.

```
go.transform.position = new Vector3(i * spacing, 0, 0);
```

By adding this here, you can now edit `spacing` and watch the cubes spread apart in real time. This is starting to get more interesting! Next let's play with some math, or more specifically `Mathf`.

6.5.1.2 *Mathf*

`Mathf` is a class filled with specific math function, which we'll need to use fairly often. `Mathf` contains many functions such as `abs` for absolute value, and `Sin` and `Cos` for sine and cosine. To start with, we're going to create a `float` called `wave` and we'll assign that to `Mathf.Sin(i)`; which produces a sine wave when we put `bob` in place of `y` in the `Vector3`.



To make this animated, we can use another useful trick from the `Time` class. Let's take a moment to thank all the busy programmers who know how to implement the math functions we're using here. There's more to these functions than just knowing the mathematics behind the simple concept of something like `Sin`. There's also a great deal of optimization that went into making the function work. This sort of optimization comes only from years of computer science experience and lots of know-how and practice. All of the mechanics behind this stuff is far beyond the scope of this book, so we'll just take advantage of their knowledge by using `Mathf`.

```
float wave = Mathf.Sin(Time.fixedTime + i);
go.transform.position = new Vector3(i * spacing, wave, 0);
```

6.5.1.3 *Time*

`Time.fixedTime` is a clock that starts running at the beginning of the game. All it's doing is counting up. Check what it's doing by printing it out using `print(Time.fixedTime)`; and you'll just see a

timer counting seconds. The seconds will be the same for all of the cubes, but by adding `int i` to each cube's version of `wave`, each one will have a different value for the wave.

We could have had a `public double 1.0` and incremented this with a small value, `0.01`, for instance. This will have a similar effect to `Time.fixedTime`. The difference here is that `fixedTime` is tied to the computer's clock. Any slowdowns or frame rate issues will result in your `Sin` function slowing down as well.

What you should have is a slowly undulating row of cubes. This is also a pretty cool example of some basic math principles in action. We could continue to add more and more behaviors to each cube in the array using this method of extending the `foreach` statement, but this will only get us so far.

Once we want each cube to have its own individual sets of logic to control itself, it's time to start a new class. With the new class, it's important that we can communicate with it properly. Eventually, these other cubes will end up being zombies attacking the player, so we're on the right track.

6.5.2 Instanting with `AddComponent()`;

Create a new C# script in the Project panel and name it `Monster`. We're going to create a basic movement behavior using this new script. We're adding this new script to the cube primitive we were making before, but this time the script will be instantiated along with the cube.

It's important to understand that `AddComponent()` is a function specific to Unity 3D's `GameObject` class. The `GameObject` is a class written to handle everything related to how Unity 3D manages characters, items, and environments in a scene. Should you leave this comfortable game development tool and move on to some other application development, you will have to most likely use a different function to perform a similar task.

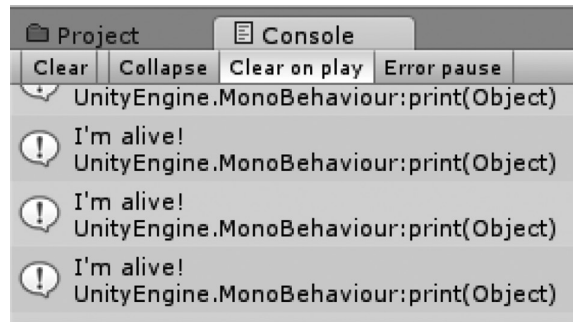
When you have a `GameObject` selected in the game editor, you have an option to add component near the bottom of the Inspector panel. We used this to add the `Example` script to our first cube. This should lead you to thinking that you can also add component with code. Looking into `GameObject`, you can find an `AddComponent()` function with a few different options.

```
//Example.cs
void Start ()
{
    boxes = new GameObject[numBoxes];
    for (int i = 0; i < numBoxes; i++)
    {
        GameObject box =
            GameObject.CreatePrimitive(PrimitiveType.Cube);
        box.AddComponent("Monster");//add component here!
        boxes [i] = box;
    }
}
```

In `MonoDevelop`, you can enter `box.AddComponent("Monster");` to tell the box to add in the script of the same name. To prove that the script is behaving correctly, add in a `print` function to the `Start ()` function of the `Monster.cs` file.

```
using UnityEngine;
using System.Collections;
public class Monster : MonoBehaviour
{
    //Use this for initialization
    void Start ()
    {
        print("im alive!");
    }
}
```

When the game starts, the `Example.cs` will create a bunch of new instances of the `Monster.cs` attached to each cube. When the script is instantiated, it executes the `Start ()` function. The Console should reflect the fact that the `Start ()` function was called when the script was instantiated.



This reduces the amount of work a single script needs to do. Each object should manage its own movement. To do this, each object will need its own script. In the `Monster.cs` file, we need to replicate some of the variables that we created in the `Example.cs` file.

```
using UnityEngine;
using System.Collections;
public class Monster : MonoBehaviour
{
    public int ID;
```

Remember that we need to make these `public`, otherwise another script cannot find them. MonoDevelop is already aware of these variables.

After the box is created, we need to add in the “Monster” class to it. However, that’s just half of the task necessary to get our script up and running. We added an `ID` and a spacing value to `Monster.cs`, but we have yet to initialize them. We need to get a connection to the component we just added to the box object. To do this, we need to use `GetComponent()`, but there’s a bit more than just that.

6.5.3 Type Casting Unity 3D Objects

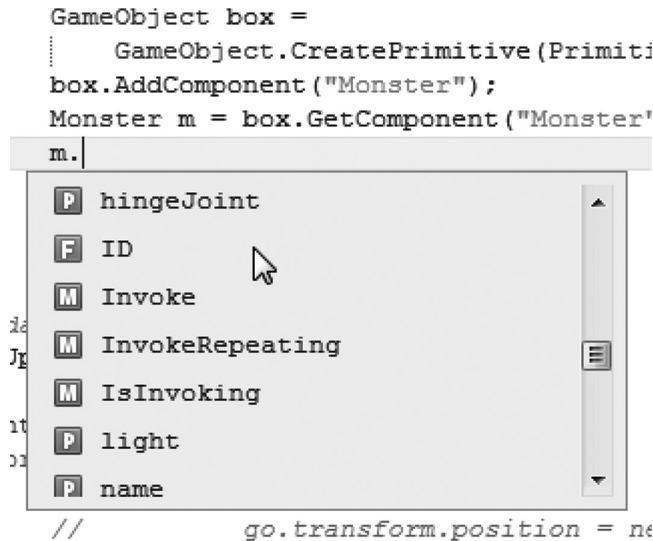
`GetComponent()` returns a component type, but it’s unaware of the specific type we’re looking for. We use some type casting to convert one type to another. Remember from before, we can turn an `int 1` to a `float 1.0f` through type casting.

When type casting between objects or reference types, things get more tricky. A component is about as general as Unity 3D will get. However, we want a type `Monster` when we get that component back. Therefore, we need to say “I want this component from the box as a type monster.” To do this, C# has the keyword `as` for some basic type casting.

```
void Start ()
{
    boxes = new GameObject[numBoxes];
    for (int i = 0; i < numBoxes; i++)
    {
        GameObject box =
            GameObject.CreatePrimitive(PrimitiveType.Cube);
        box.AddComponent("Monster");
        Monster m = box.GetComponent("Monster") as Monster;
        boxes[i] = box;
    }
}
```

After adding the component, we need to get access to it. Therefore, we create a `Monster` variable named `m`. Then we use the `GameObject.GetComponent();` function to get an object called “Monster,” and we ask that we get it as a type `Monster` by adding `as Monster` after we ask for it.

This cast is necessary because `GetComponent()` doesn't necessarily know what it's getting; other than it's some `Component` type, you have to tell it what it's getting. `Monster` and "Monster" appear in different places. This is the difference between the word "Monster" and the actual class object called `Monster`. The object is what is used after the `as` keyword because we're referring to the type, not what it's called. This might be a little bit confusing, but `GetComponent()` is expecting a string and not a type for an argument.



When you enter the `m.`, a dialog box pops up in MonoDevelop. This is a handy helper that shows you all of the things that the object can do. This also shows you any of the public variables you may have added to the class. Now that we have a connection to the `Monster` script that's attached to the box, we can set a couple of parameters.

```

13   GameObject box = GameObject.CreatePrimitive(PrimitiveType.Cube);
14   box.AddComponent("Monster");
15   Monster m = box.GetComponent("Monster") as Monster;
16   m.ID = i;
17   m.spacing = spacing;
18   boxes[i] = box;

```

We use the dot notation to access the members of the `Monster` class found in `m`. Therefore, `m.ID` is `i` that increments with each new box made. Then the spacing will be the spacing we set in the `Example.cs` file.

```

13   // Update is called once per frame
14   void Update () {
15       float wave = Mathf.Sin( Time.fixedTime + ID);
16       transform.position = new Vector3(ID * spacing, wave, 0.0f);
17   }

```

Add a very similar line of code to the `Update ()` in the `Monster.cs` file and then remove it from the `Update ()` in the `Example.cs` file. The spacing was only set once when the object was created, which you can't update it by sliding on the `Example.cs` file. However, each object is acting on its own, running its own script.

There are a few different ways to use `GetComponent()`, but we'll look at those in Section 6.14 when we need to do more type casting. Not all casting operations work the same. We're going to change a few

other things we're doing to make the movement behavior more interesting as well, but we will need to learn a few more tricks before we get to that.

Alternatively, we can assign and get the `Monster m` variable at the same time with the following notation:

```
Monster m = box.AddComponent("Monster") as Monster;
```

This automatically assigns `m` as it's assigned to the `box` component. The component assignment and the variable assignment save a step.

6.5.4 What We've Learned

This was a pretty heavy chapter. The new array type was used to store a bunch of game objects, but it could have easily been used to store a bunch of numbers, or anything for that matter. We learned about a `foreach` loop that is handy for dealing with arrays.

In this chapter, we also made use of some math and found a nice timer function in the `Time` class. After that, we figured out how to attach script components through code, and then we were able to gain access to that component by `GetComponent()` and a type cast to make sure it wasn't just a generic component.

There are still a few different data types we need to study. Now that we're dealing with so many types, we're going to learn more about type casting as well.

6.6 Enums

If we've got two classes, an `Example.cs` and a `Monster.cs`, we've got the `Example.cs` creating and assigning objects, effectively spawning a `Monster`. We could take the older version of the `Example.cs` we wrote and turn it into `Player.cs` which would give us a total of three objects.

```
using UnityEngine;
using System.Collections;
public class Player : MonoBehaviour
{
    public float Speed = 0.1f;
    //Update is called once per frame
    void Update ()
    {
        gameObject.transform.position += Movement(Speed);
    }
    Vector3 Movement(float dist)
    {
        Vector3 vec = Vector3.zero;
        if (Input.GetKey(KeyCode.A))
        {
            vec.x -= dist;
        }
        if (Input.GetKey(KeyCode.D))
        {
            vec.x += dist;
        }
        if (Input.GetKey(KeyCode.W))
        {
            vec.z += dist;
        }
        if (Input.GetKey(KeyCode.S))
```

```

        {
            vec.z -= dist;
        }
        return vec;
    }
}

```

Therefore, here's the `Player.cs` I'm using for this chapter; we're going to use this to move the little box around in the scene. We'll pretend that the box is a pretty cool-looking monster hunter armed with a sword, shotgun, or double-barreled shotgun.

We may want to rename the `Example.cs` to something like `MonsterSpawner.cs` or `MonsterGenerator.cs`, but I'll leave that up to you. Just remember to rename the class declaration to match the file name. This will spill out monsters to chase the player around. Then the `Monster.cs` attached to each of the objects that the `MonsterSpawner` creates will then seek out the player and chase him around. At least that's the plan.

NOTE: Prototyping game play using primitive shapes is a regular part of game development. This is somewhat related to what has become to be known as “programmer art.” Changes to code often mean changes to art. It's difficult to build a game keeping art and code parallel. Often a simple change in code means days of changes to art. It's quite often easier to prototype a game making no requests of an artist. Most of the time, the programmer doesn't even know himself what to ask of an artist.

6.6.1 Using Enums

The keyword “enum” is short for enumeration. A programmer might say that an `enum` is a list of named constants. The meaning may seem obtuse. Translated, an enumeration is a list of words that you pick. For our `Monster` to have a better set of functions for its behavior, we're going to create a new `enum` type called `MonsterState`.

We've already interacted with the `PrimitiveType` `enum`.

```

public enum PrimitiveType
{
    Sphere,
    Capsule,
    Cylinder,
    Cube,
    Plane
}

```

We've also seen that enums can store many different names. For instance, the `InputType` `enum` had a different enumeration for every key on the keyboard and each input for your mouse or trackpad and controller. Enumerating through a long list of “things” is to help name every possibility with something more useful than a simple numeric index. It's important to remember that enumerations don't need to follow any particular pattern. It's up to you to decide on any organization to keep your enums organized.

In the `MoreLogic` project, open the `Example.cs` found in the `Assets` list in the `Project` panel.

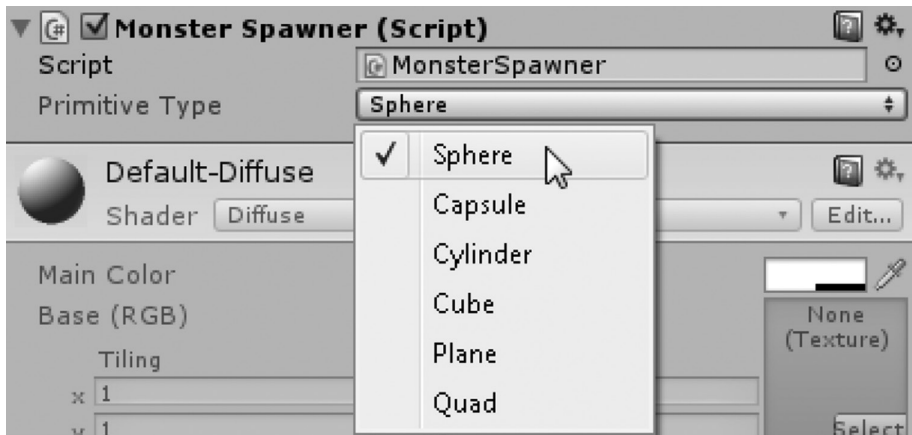
```

public PrimitiveType primitiveType;
GameObject obj;
//Use this for initialization
void Start () {
    obj = GameObject.CreatePrimitive(primitiveType);
}

```


At the beginning of the `Example.cs` class, we have a `public PrimitiveType` called `primitiveType`. Note the difference in case; the second use of the word is lowercase and is not using a previously defined version of the word `PrimitiveType`.

In the editor, you'll see the availability of a new pop-up of the different `PrimitiveTypes`. In the `Start ()` function, we use the `CreatePrimitive` member function in `GameObject` to create a new object of the selected type.



We can extend this more to get a better feel of how enums work for us by creating our own enums. A new enum is a new type of data. New data types are easy to create, especially for enums.

```
public enum colorType
{
    red,
    blue,
    green
}
```

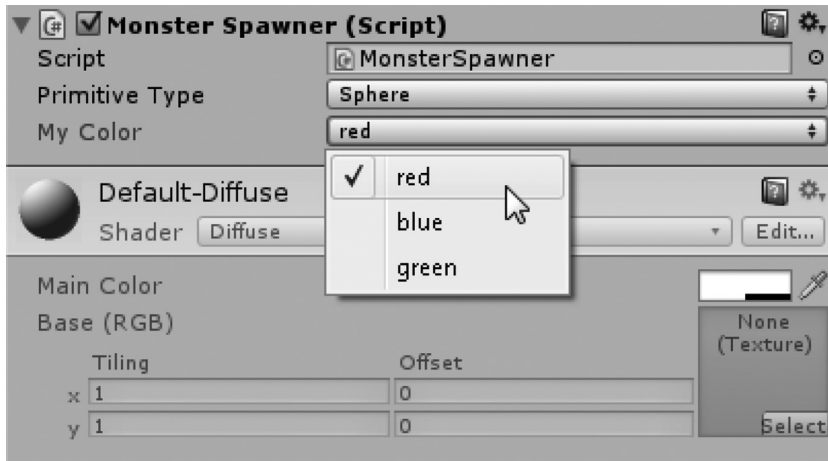
In this case, we use the `public` keyword followed by `enum` to tell the compiler we're creating a new enum which can be made accessible to other classes. We follow the declaration with a name. In this case, `colorType` is the name of our new enum. Following the name of the enum, we need to start a new code block with any number of words separated by commas. Just don't add a comma after the last word in the list.

```
public enum colorType{red,blue,green}
```

To be clear, declaring an enum doesn't require the line breaks between each word. White space has no effect on how an enum, or practically any variable for that matter, is declared. After the new data type is created, we need to create a variable that uses that data type.

```
public enum colorType{red,blue,green}
public colorType myColor;
```

Create a new `public colorType` with the name `myColor` following the declaration of the enum. In the editor, we'll be able to pick the enum we want from our new list of words we added to the enum `colorType`.



To make use of the enum, we can use several different methods. The system we might already be familiar with is using a bunch of different if statements.

```
obj = GameObject.CreatePrimitive(primitiveType);
if (myColor == colorType.red) {
    obj.renderer.material.color = Color.red;
}
if (myColor == colorType.blue) {
    obj.renderer.material.color = Color.blue;
}
if (myColor == colorType.green) {
    obj.renderer.material.color = Color.green;
}
```

This setup is clumsy; after the obj is created, we check for what word myColor is set to. Compare it against the colorType's options and act when you've got a match. A slightly cleaner solution is to use a switch statement.

Just as a note, if you look at the scene and all you see is a black sphere, you might need to add in a directional light. Without any lights, every object in the scene will appear dark since there's no light to illuminate them. You can add in a light by selecting GameObject → Create Other → Directional Light. This will drop in a light that will lighten up any objects in your scene.

6.6.2 Combining What We've Learned

We've been using PrimitiveType.Cube to generate our example monsters. We could just as easily change that to a sphere or anything else in the PrimitiveType enum. As we've seen, the PrimitiveType has some words that reflect what type of Primitive we can choose from. Likewise, we're going to make a list of states for the monster to pick.

```
Public class Monster : MonoBehaviour {
    public enum MonsterState {
        standing,
        wandering,
        chasing,
        attacking
    }
    public MonsterState mState;
```

Enums are declared at the class scope level of visibility. `MonsterState` is now another new data type. When setting up enum names, it's really important to come up with a convention that's easy to remember.

If you make both the enum `MonsterState` and `mState` `public`, you'll be able to pick the state using a menu in the Unity's Inspector panel. You should consider using enums for setting various things such as weapon pickup types or selecting types of traps.



Then, just as we define any other variable, we declare a variable type and give type a name. This is done with the line `MonsterState mState;` which gives us `mState` to use throughout the rest of the `Monster` class. `mState` is a variable with type `MonsterState`. In some instances, we may need to ignore what was set in the editor. To set an enum in code rather than a pop-up, we can use the following code. To use the `mState`, we need to set it when the `Monster.cs` is started.

```
void Start ()
{
    mState = MonsterState.standing;
}
```

`mState` is set to `MonsterState.standing`; this allows us to use `mState` in the `Update ()` function to determine what actions we should take. Like before, we could use a series of `if` statements to pick the actions we could take. For instance, we could execute the following code:

```
//Update is called once per frame
void Update ()
{
    if (mState == MonsterState.standing)
    {
        print("standing monster is standing.");
    }
    if (mState == MonsterState.wandering)
    {
        print("wandering monster is wandering.");
    }
    if (mState == MonsterState.chasing)
    {
        print("chasing monster is chasing.");
    }
    if (mState == MonsterState.attacking)
    {
        print("attacking monster is attacking.");
    }
}
```

This will work just fine, but it's rather messy. If we add more enums to the `MonsterState`, we will need more `if` statements. However, there's an easier way to deal with an enum, so it's time to learn a new trick.

6.6.3 What We've Learned

So far enums have been made to make a list of options. Normally, enumerations are limited to a single option, but this doesn't always have to be the case. Enums are capable of multiple options, which has been discussed throughout this section.

Using an enum means to take action based on a selection. An enum is actually based on a number. You can cast from an enum to an `int` and get a usable value. When this is done, the first name in an enum is 0 and anything following is an increment up in value based on its position in the list of enumerations.

Therefore, based on the following code, the logged value is 3 since the first state is 0 and the fourth state is 3.

```
public enum MonsterState
{
    standing,
    wandering,
    chasing,
    attacking
}
public MonsterState mState;
void Start ()
{
    mState = MonsterState.attacking;
    int number = (int)mState;
    Debug.Log(number);
}
```

There are other ways in which the enum can be manipulated, but to understand how and why this works, we'll want to cover a few other topics before getting to that.

6.7 Switch

The `switch` comes into play fairly often once we have reached more than one condition at a time. For instance, we could come across a situation where we are looking at a lengthy ladder of `if-else` statements.

```
public int i;
void Start ()
{
    if(i == 0)
    {
        Debug.Log ("i is zero");
    }
    else if(i == 1)
    {
        Debug.Log ("i is one");
    }
    else if(i == 2)
    {
        Debug.Log ("i is two");
    }
    else if(i == 3)
```

```
    {
        Debug.Log ("i is three");
    }
    else if (i == 4)
    {
        Debug.Log ("i is four");
    }
    else
    {
        Debug.Log("i is greater than 4");
    }
}
```

There should be something awkward feeling about this block of code. There is in fact a system in place called a `switch` statement that was made to alleviate the awkwardness of this long chain of `if-else` statements. The `switch` starts with the keyword `switch` followed by a parameter `()` that controls a block of code encapsulated by a pair of curly braces `{}`.

```
void Update ()
{
    switch (someVariable)
    {
    }
}
```

The contents of the `switch` statement use the keyword `case`. Each `case` is fulfilled by an expected option that matches the argument in the `switch` parameter.

6.7.1 A Basic Example

A `switch` can be used with any number of types. A simple `switch` statement can look like the following code using an `int` like the above `if-else` chain. This can be found in the `SwitchStatement` project in the `Example.cs` component attached to the `Main Camera`.

```
using UnityEngine;
using System.Collections;
public class Example : MonoBehaviour {
    public int i = 1;
    void Start ()
    {
        switch (i)
        {
            case 0:
                Debug.Log("i is zero");
                break;
            case 1:
                Debug.Log("i is one");
                break;
            case 2:
                Debug.Log("i is two");
                break;
        }
    }
}
```

This is a basic `switch` with a `case`. The `case` is followed by 1 or 2 ending with a colon. Before another `case`, there's a couple of statements, with the last statement being `break`; that ends the `case`: statement.

The first case 1: is executed because `i == 1`; or in more plain English, `i` is 1 that fulfills the condition to execute the code following the case. The `break;` following the statements jumps the computer out of the `switch` statement and stops any further execution from happening inside of the `switch` statement.

```
int i = 1;
switch (i)
{
    case 1: print("got one"); break;
    case 2: print ("got two"); break;
}
```

When you deal with short case statements, it's sometimes easier to remove the extra white space and use something a bit more compact. Each case is called a *label*; we can use labels outside of a `switch` statement, but we will have to find out more about that in Section 6.7.5.

The `switch` statement is a much better way to manage a set of different situations. Upon looking at the `switch` statement for the first time, it might be a bit confusing. There are a few new things going on here. The general case of the `switch` is basically taking in a variable, pretty much of any kind. The code then picks the case statement to start. The main condition is that all of the cases must use the type that is used in the `switch()` argument. For instance, if the `switch` uses the "mState" enum, you can't use a case where an "int" is expected.

The advantage of `switch` may not be obvious when looking at the block of code that was written here. The important reason why `switch` is useful is speed. To step through many different `if` statements, the contents of each argument needs to be computed.

This means that having a dozen "if" statements means that each one needs to be tested even though the contents of the statement are to be skipped. When using a `switch`, the statement needs to have only one test before evaluating the contents of the case.

The `switch` statement can use a few different data types, specifically integral types. Integral types, or data that can be converted into an integer, include booleans or bools, chars, strings, and enums. In a simple boolean example, we can use the following:

```
bool b = true;
switch (b)
{
    case true: print("got true"); break;
    case false: print ("got false"); break;
}
```

With integers where `int i = 1; switch(i){}` is used to pick the case which will be used. Using integers allows for a long list of cases when using a `switch` statement.

```
bool b = true;
switch (b)
{
    case true:
        print("got true");
        break;
    case false:
        print("got false");
        break;
}
```

Here `i = 1`, so case 1: is evaluated and "case 1" is printed. Each appearance of the keyword `case` is called a case label. Cases are built by using the keyword `case` followed by the value we are looking for from the variable used in the `switch`. After the case label is declared, any number of statements can be added. The code following the colon after the case statement is evaluated until the `break;` statement that stops the `switch`. The `break` statement must appear before another case label is added.

6.7.2 Default:

What if there's a case that's not included in the `switch` statement? It would be difficult to cover every case for an integer. You'd have to write a case for every and any number. That's where the `default:` case comes in. When dealing with any of the `switch` statements, a default condition can be added when a case appears that isn't handled.

```
int i = 3;
switch (i)
{
    case 0:
        Debug.Log("i is zero");
        break;
    case 1:
        Debug.Log("i is one");
        break;
    case 2:
        Debug.Log("i is two");
        break;
    default:
        Debug.Log("Every other number");
        break;
}
```

With the default added, any conditions that aren't taken care of can be handled. If the default isn't added, then any unhandled case will be ignored and the `switch` will skip any code from any of the cases. The default case is an optional condition when working with the `switch` statement.

Of course, a `switch` statement seems to be most at home when in use with an enum. A slightly different variation on the appearance of the enum itself is that we need to use the dot operator to compare the incoming value against one of the enum values.

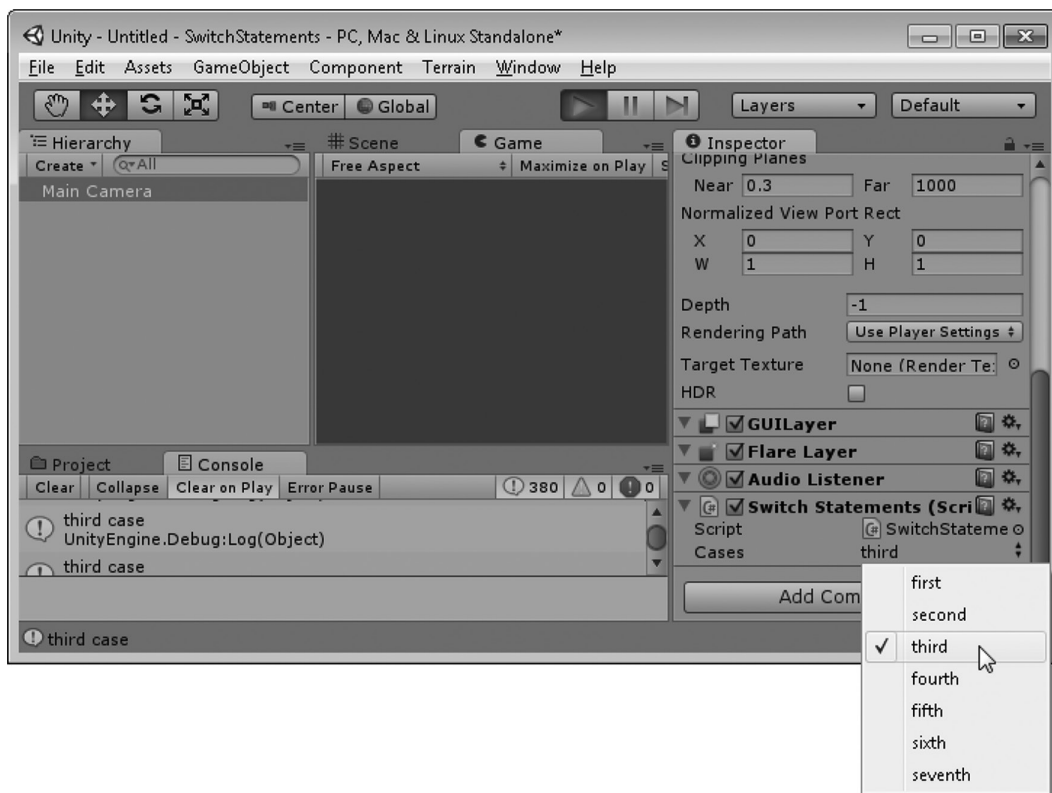
```
public enum MyCases
{
    first,
    second,
    third,
    fourth,
    fifth,
    sixth,
    seventh
}
public MyCases cases;
void Update ()
{
    switch(cases)
    {
        case MyCases.first:
            Debug.Log("first case");
            break;
        case MyCases.second:
            Debug.Log("second case");
            break;
        case MyCases.third:
            Debug.Log("third case");
            break;
        case MyCases.fourth:
            Debug.Log("fourth case");
            break;
    }
}
```

```

    case MyCases.fifth:
        Debug.Log("fifth case");
        break;
    case MyCases.sixth:
        Debug.Log("sixth case");
        break;
    case MyCases.seventh:
        Debug.Log("seventh case");
        break;
    default:
        Debug.Log("other case");
        break;
}
}

```

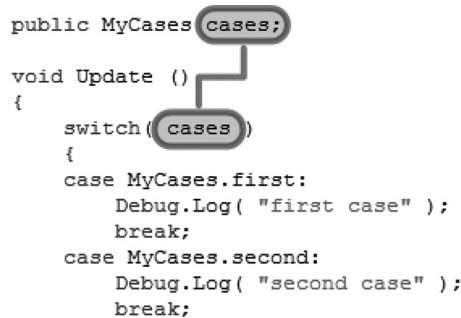
Here we have an elaborate array of cases, each one falling in order based on the enum values for `MyCases`. Inside of the Unity 3D editor, the C# class provides a useful roll-out in the Inspector panel when assigned to a `GameObject`.



This conveniently gives us the opportunity to pick an enum and watch the `Debug.Log()` send different strings to the Console panel.

```
public MyCases cases;

void Update ()
{
    switch (cases)
    {
        case MyCases.first:
            Debug.Log( "first case" );
            break;
        case MyCases.second:
            Debug.Log( "second case" );
            break;
    }
}
```



It's important that you note what variable is being used in the switch statement's argument. It's often mistaken to use `switch(MyCases)` with the type in the argument. This of course doesn't work as expected. Here we get the following error:

```
Assets/SwitchStatements.cs(21,25): error CS0119: Expression denotes a 'type',
where a 'variable', 'value' or 'method group' was expected
```

Each case statement must have a `break;` statement before the next case appears. You can have any number of statements between case and break, but you must break at the end. This prevents each case from flowing into one another. If you forget to add break after the first case statement, then you get the following warning:

```
Assets/SwitchStatements.cs(21,17): error CS0163: Control cannot fall through
from one case label to another
```

It's easy to forget to add these breaks, but at least we are informed that we're missing one. However, we're directed to the beginning of the switch statement. It's up to you to make sure that there's a break statement at the end of each case. It's also required after the default case.

Normally, the default label is the last case at the end of the series of case labels. However, this isn't a requirement. You can write the default label anywhere in the switch statement. Style guides usually require that you put the default at the very end of the switch. Not doing so might invoke some harsh words from a fellow programmer if he or she finds a misplaced default label.

6.7.3 What We've Learned

Switch statements are a common combination. Each label is clear and the parameter for the switch is also quite clear. We know what we're expecting and it's obvious what conditions each case needs to be written for.

```
enum cases {
    firstCase,
    secondCase,
    thirdCase
}
cases MyCases;
```

The above enum might be quite contrived, but we know how many labels we need if we use this case in a switch. It should be obvious that our first case label should look like `case cases.firstCase:` and we should fill in the rest of the case statements in order.

switch statements are limited to a select different types of data.

```
float f = 2.0f;
switch (f)
{
    case f < 1.0f:
        Debug.Log("less than 1.0f");
        break;
    case f > 3.0f:
        Debug.Log("more than 3.0f");
        break;
    default:
        Debug.Log("neither case");
        break;
}
```

The above code might look valid, but we'll get the following error:

```
Assets/Test.cs(15,1): error CS0151: A switch expression of type 'float'
cannot be converted to an integral type, bool, char, string, enum or nullable
type
```

A switch is allowed to use only “integral, bool, char, string, enum, or nullable” types of data. We have yet to cover some of these different data types, but at least we know what a data type is. We do know that we can use ints and enums. In most cases, this should be sufficient.

switch statements should be used when one and only one thing needs to happen based on a single parameter.

```
int a = 0;
int b = 1;
switch (a)
{
    case 0:
        switch (b)
        {
            case 1:
                Debug.Log("might not be worth it");
                break;
        }
        break;
}
```

Having a switch statement nested in one of the cases of another switch statement isn't common. There's nothing really stopping you from being able to do this, but it's probably better to look for a more elegant solution.

```
void Update ()
{
    int a = 0;
    switch (a)
    {
        case 0:
            FirstFunction();
            break;
        case 1:
```

```

                SecondFunction();
                break;
            }
        }
    }
    void FirstFunction()
    {
        Debug.Log("first case");
    }
    void SecondFunction()
    {
        Debug.Log("second case");
    }
}

```

Using a function in each case is a simple way to keep things tidy, though it's not always clear what's going on. Anyone reading the code will be forced to jump back and forth between the `switch` and the different functions. However, this does mean that you might be able to make more switch cases within each function in the `switch` statement.

```

void Update ()
{
    int a = 0;
    switch (a)
    {
        case 0:
            FirstFunction(a);
            break;
        case 1:
            SecondFunction();
            break;
    }
}
void FirstFunction(int i)
{
    switch (i)
    {
        case 0:
            Debug.Log("first case");
            break;
    }
}
void SecondFunction()
{
    Debug.Log("second case");
}
}

```

The parameter in the `switch` statement is now in the body of the `switch` statement. This also means that the parameter can be manipulated before it's used in the case.

```

void Update () {
    int a = 0;
    switch (a)
    {
        case 0:
            a = 1;
            FirstFunction(a);
            break;
        case 1:
            SecondFunction();
    }
}

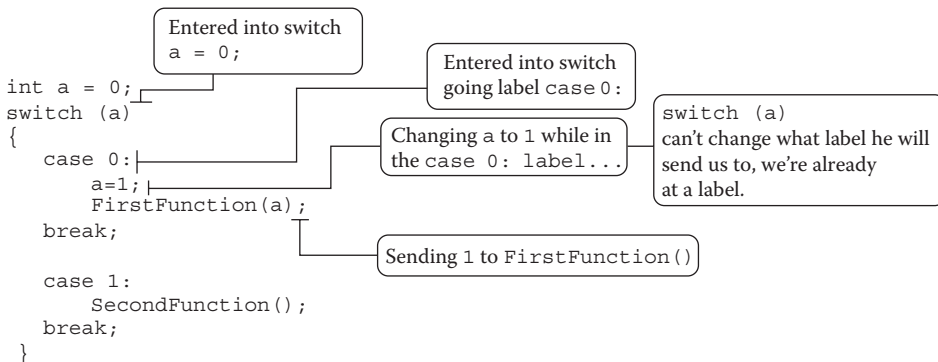
```

```

        break;
    }
}
void FirstFunction(int i)
{
    switch (i)
    {
        case 0:
            Debug.Log("first case");
            break;
        case 1:
            Debug.Log("i was incremented!");
            break;
    }
}
void SecondFunction() {
    Debug.Log("second case");
}

```

Because `a = 1;` appears after we've entered the case and is followed by a `break;`, case 1: is not triggered, and we don't skip to the second case. The logic might seem cloudy, but it's important that we really understand what's going on here.



Once inside of the switch statement, the switch cannot change our destination once we've arrived at a label. After we've got to the label, we can change the data that got us there. Once we get to the `break;` at the end of the case, we leave the switch and move to any statements that follow the switch statement code block.

```

switch (a)
{
    case 0:
        a = 1;
        FirstFunction(a);
        continue;//nope!
}

```

A switch is not a loop, so we can't go back to the beginning of the switch by using `continue;` actually this is just an error telling you that there's no loop to continue to. A switch statement isn't meant to be looped; there are systems that allow this, but in general, these sorts of statements are one-way logic controls. A switch can work only with a single argument.

Something like `switch(a, b)` could make sense, but this isn't the case. Keeping things simple is not only your goal, but you're also limited by what is allowed. In some cases, the language can force simplicity. In practice, it's best to keep things as simple as possible if only for the sake of anyone else having to understand your work.

6.7.4 Fall Through

Using the `break;` statement jumps us out of the `switch`. Therefore, something like the following will execute one or the other statements after the condition and jump out of the `switch`.

```
switch(condition)
{
    case first_condition:
        //do things
        break;
    case second_condition:
        //do something else
        break;
}
```

If we want to, a case can be left empty, and without a `break;` included, anything in the case will “fall through” to the next case until the code is executed and a `break;` statement is found.

```
switch(condition)
{
    case first_condition:
    case second_condition:
        //do something else
        break;
}
```

In the above example, if we get to `case first_condition:`, we’ll simply find the next case afterward and run the code there until we hit the next `break;` statement. It’s rare to use this in practice as the conditions included act the same; in this case, both the first and second conditions do the same thing. However, things get a bit more awkward if we need to do something like the following:

```
switch(condition)
{
    case first_condition:
    case second_condition:
        //do something else
        break;
    case third_condition:
    case fourth_condition:
    case fifth_condition:
        //do another thing
        break;
}
```

Here we have two different behaviors based on five different conditions. The first two and the next three cases can result in two different behaviors. Although you might not use this behavior immediately, it’s important to know what you’re looking at if you see it. There are some catches to how this can be used.

```
switch(condition)
{
    case first_condition:
        //code that is out of place.
    case second_condition:
        //do something else
        break;
    case third_condition:
        //do another thing
        break;
}
```

In the above example, our first condition might have some added code before the second condition. This might visually make sense; go to the first condition, execute some instructions, fall through to the next case, do more instructions, and break. However, this behavior isn't allowed in C#.

In order to accomplish what was described, we need to be more explicit.

6.7.5 goto Case

```
switch(condition)
{
    case first_condition:
        //code that is out of place.
        goto case second_condition;
    case second_condition:
        //do something else
        break;
    case third_condition:
        //do another thing
        break;
}
```

Using the goto keyword allows us to hop from one case in the switch statement to another. This allows for an expected fall through-like behavior, but also gives us an added functionality of going to any other case once inside of the switch statement.

```
switch(condition)
{
    case first_condition:
        //doing something first
        goto case third_condition;
    case second_condition:
        //do something else
        break;
    case third_condition:
        //do another thing
        goto case second_condition;
}
```

The utility of the above statement is questionable. Although the statement is valid, it's hardly something that any programmer would want to read. Bad habits aside, should you see this in someone else's code, it's most likely written as a work-around by someone unfamiliar with the original case.

Aside from being awkward and strange looking, it's important that the switch case finally lead to a break, otherwise we're caught in the switch statement indefinitely. Debugging a situation like this is difficult, so it's best to avoid getting caught in a switch statement longer than necessary.

Strange and unusual code practices when working inside of a switch statement aren't common, as the regular use of a switch statement is quite simple. If you need to do something more clever, it's better to have a switch statement that points toward a function with additional logic to handle a more specific set of conditions.

6.7.6 Limitations

The switch statement is limited to integral types, that is to say, we can only use types that can be clearly defined. These are called integral types and a float or double is not an integral type.

```
float myFloat = 1f;
switch(myFloat)
{
    case 1.0f:
```

```
        //do things
        break;
    case 20.0f:
        //do something else
        break;
}
```

The above code will produce the following error:

```
Assets/Switch.cs(20,1): error CS0151: A switch expression of type 'float'
cannot be converted to an integral type, bool, char, string, enum or nullable
type
```

Without deferring to a chapter on types, integral types often refer to values that act like whole numbers. Letters and words are included in this set of types. Excluded from the `switch` are floats and doubles, to name some non-integral types. Because a `float 1` can also be seen as `1.0` or even `1.00000f`, it's difficult for a `switch` to decide how to interpret the above example's `myFloat` as being a `1` or a `1.0f`.

6.7.7 What We've Learned

The `switch` statement is powerful and flexible. Using special case conditions where we use either the fall-through behavior or `goto case` statement, we're able to find additional utility within the `switch` statement.

Just because we can doesn't mean we should use fall-through or `goto case`. Often, this leads to confusing and difficult-to-read code. The bizarre and incomprehensible fall-through or `goto case` should be reserved for only the most bizarre and incomprehensible conditions. To avoid bad habits from forming, it might be better to forget that you even saw the `goto case` statement.

Now that warnings have been issued, experimenting and playing with code is still fun. Finding and solving problems with unique parameters is a part of the skill of programming. Using weird and strange-looking structures helps in many ways. If we're able to comprehend a strange set of code, we're better equipped to deal with more regular syntax.

`switch` statements can be used with any number of different parameters. Strings, ints, and other types are easily used in a `switch` statement.

```
string s = "some condition";
switch (s)
{
    case "some condition":
        //do things for some condition
        break;
    case "other condition":
        //do something else
        break;
}
```

6.8 Structs

We are now able to build and manipulate data. Enums, arrays, and variables make sense. Basic logic flow control systems are no longer confusing. What comes next is the ability to create our own forms of data. The built-in types may not be able to describe what we're trying to do.

Data structures allow you to be specific and concise. By themselves `floats`, `ints`, and `Vector3` are useful. However, copying and moving around each variable individually can look rather ugly. This lack of organization leaves more opportunity for errors.

When having to deal with a complex character or monster type, collections of data should be organized together. The best way to do this is by using a struct, or structure. A `Vector3` is a basic collection of three similar data types: a `float` for `x`, `y`, and `z`. This is by no means the limitation of what a structure can do.

6.8.1 Structs

Structures for a player character should include the location, health points, ammunition, weapon in hand, and weapons in inventory and armor, and other related things should be included in the `player.cs`, not excluded. To build a struct, you use the keyword `struct` followed by a type. This is very similar to how an enum is declared. What makes a structure different is that we can declare each variable in the struct to different data types.

After a structure is built to access any of the `public` components of the structure, you use dot notation. For example, `playerData.hitPoints` allows you to access the `int` value stored in the structure for hitpoints. `PlayerData` now contains all of the vital values that relate to the player.

```
public struct PlayerData {
    public Vector3 Pos;
    public int hitPoints;
    public int Ammunition;
    public float RunSpeed;
    public float WalkSpeed;
}
PlayerData playerData;
```

When you first look at declaring and using a struct, there might seem a few redundant items. First, there's a `public struct PlayerData`, then declare a `PlayerData` to be called `playerData`, and notice the lowercase lettering on the identifier versus the type. The first time `PlayerData` appears, you're creating a new type of data and writing its description.

6.8.2 Struct versus Class

At the same time, the struct may look a lot like a class, in many ways it is. The principal differences between the struct and the class is where in the computer's memory they live and how they are created. We'll discuss more on this in a moment.

You fill in some variables inside of the new data type and name the data inside of the new type. This new data type needs to have a name assigned to it so we know how to recognize the new form of packaged data. Therefore, `PlayerData` is the name we've given to this new form of data. This is the same as with a class.

The second appearance of the word `PlayerData` is required if we want to use our newly written form of data. In order to use data, we write a statement for a variable like any other variable statement. Start with the data type; in this case, `PlayerData` is our type; then give a name to call a variable of the stated type. In this case, we're using `playerData` with a lowercase `p` to hold a variable of type `PlayerData`.

A struct is a data type with various types of data within it. It's convenient to packaging up all kinds of information into a single object. This means that when using the struct, you're able to pass along and get a breadth of information with a single parameter. This matters most when having to pass this information around.

This system of using a struct as data means that a struct is a value type, whereas a class is a reference type. You might ask what the difference between a value type and a reference type is. In short, when you create a struct and assign a value to it, a new copy is made of that struct. For instance, consider the following:

```
PlayerData pd = playerData;
```


In this case, `pd` is a new struct, where we make another version of `playerData` and copy the contents into `pd`. Therefore, if we make `PlayerData pd2` and assign `playerData` to it, or even `playerData`, we'll have yet another copy of that same data in a new variable. Each variable `pd`, `pd2`, and `playerData` now all have unique duplicates of the data.

However, should we change struct to class, we'd have a different arrangement.

```
public class PlayerData {
    public Vector3 Pos;
    public int hitPoints;
    public int Ammunition;
    public float RunSpeed;
    public float WalkSpeed;
}
```

The only change to the above code is changing struct to class. This is still a perfectly valid C# class and appears to be the same as a struct, but the similarities cease once a variable is assigned. The differences can be seen in the following simple example. Start by looking at the `ClassVStruct` project and open the `ClassVStruct.cs` component attached to the Main Camera.

```
using UnityEngine;
using System.Collections;
public class StructVClass : MonoBehaviour {
    struct MyStruct{
        public int a;
    }
    class MyClass{
        public int a;
    }
    //Use this for initialization
    void Start () {
        MyClass mClass = new MyClass();
        MyStruct mStruct = new MyStruct();
        mClass.a = 1;
        mStruct.a = 1;
        MyStruct ms = mStruct;
        ms.a = 3;
        Debug.Log(ms.a + " and " + mStruct.a);
        MyClass mc = mClass;
        mc.a = 3;
        Debug.Log(mc.a + " and " + mClass.a);
    }
}
```

In the above code, we have a struct called `MyStruct` and a class called `MyClass`. Both have a public field called `int a`; once in the `Start ()` function a new `MyClass()` and a new `MyStruct()` are created. As soon as these classes are created, we assign 1 to the `.a` fields in both the struct and the class.

After this, we have an interesting change. Using the `MyStruct ms = mStruct`, we create a copy of the `MyStruct` from `mStruct` and assign that copy to `ms`. Doing the same thing to `MyClass mc`, we assign `mClass` to `mc`. If we check the value of `ms.a` and `mStruct.a`, we get 3 and 1, respectively.

Doing the same thing to `mc.a` and `mClass.a`, we get 3 and 3, respectively. How did this happen and why is `mc.a` changing the value of `mClass.a`? When a class is assigned to a variable, only a *reference* to the class is assigned. A new copy is not made and assigned to the variable `mc`.

In the statement `MyClass mc = mClass`, we assign `mc` a reference to `mClass`. This contrasts to `MyStruct ms = mStruct`, where `ms` gets a copy of the value stored in `mStruct`. After a struct is assigned, it becomes an independent copy of the struct that was assigned to it. In order to break the reference, we cannot use `mc = mClass`, otherwise a reference will be created.

```

MyClass mc2 = new MyClass();
mc2.a = mClass.a;
mc2.a = 7;
Debug.Log(mc.a + " and " + mClass.a + " and " + mc2.a);

```

By making a new instance of `MyClass` called `mc2`, we can assign the value from `mClass.a` to `mc2.a`, which will avoid a reference from being created between `mc2` and `mClass`. The above code prints out the following output:

```
3 and 3 and 7
```

Even though we assigned the `.a` field of `mc2` to `mClass.a`, we lose the reference between `mc2` and `mClass`. When we change `mc2.a` to 7, it has no effect on `mc.a` and `mClass.a`. The differences between the struct and the class is subtle but distinct.

The values of structs are copied when assigned in the `ms = mStruct` fashion. To accomplish the same behavior with a class, we need to assign each field separately with `mc2.a = mClass.a;`. Only in this fashion can we actually make a copy of the value to `mc2.a` from `mClass.a`, not a reference.

6.8.3 Without Structs

The alternative to the struct is to use another form of data containment like the array. Each object in the array can hold a different data type, but it must be addressed by its index number in the array. It would be up to you to remember if the walk speed was at index 4 or 3 in the array. Organizational issues make this far too easy to mess up and forget which index holds what data.

Once you start adding new types of data to this method of record keeping, chances are you'll forget something and your code will break. There are languages out there that don't have structs, and this is the only way to manage a collection of data types. Be thankful for structs.

```

public object[] PlayerDataArray;
//Use this for initialization
void Start () {
    PlayerDataArray[0] = new Vector3();//position
    PlayerDataArray[1] = 10;//hit points
    PlayerDataArray[2] = 13;//ammo
    PlayerDataArray[3] = 6.5f;//run speed
    PlayerDataArray[4] = 1.2f;//walk speed
}

```

When another creature comes into contact with the player, it should be given a copy of the entire `PlayerData` stored in `playerData`. This confines all of the data into a single object and reduces the need for making separate operations to carry over different data. To do the same to a class, we'd have to use the same idea of copying each parameter one at a time because of the earlier-mentioned behavior between a class and struct assignment.

6.8.4 Handling Structs

We'll start with the Structs project in Unity 3D. In this project, we have a scene with a box and an attached `Struct.cs` component. Structs are best used to contain a collection of data for a particular object. If we wanted to make a simple set of parameters for a box, we might use the following struct:

```

struct BoxParameters
{
    public float width;

```

```

        public float height;
        public float depth;
        public Color color;
    }
    //put the new struct to use and name it myParameters
    BoxParameters myParameters;

```

With this added to a new class attached to a box in the scene in Unity 3D, we have a system of storing and moving a collection of data using a single object. Within this object, we've assigned public variables for width, height, depth, and color.

To make use of this, we can adjust each one of the parameters individually within the structure using the dot operator.

```

//Use this for initialization
void Start ()
{
    myParameters.width = 2;
    myParameters.height = 3;
    myParameters.depth = 4;
    myParameters.color = new Color(1,0,0,1);
}

```

In the `Start ()` function, we can access the `myParameters` variables and assign them values. Once there are usable values assigned to the `BoxParameters myParameters`, we can use them as a variable passed to a function.

```

void UpdateCube(BoxParameters box)
{
    Vector3 size = new Vector3(box.width, box.height, box.depth);
    gameObject.transform.localScale = size;
    gameObject.renderer.material.color = box.color;
}

```

The new function can access the `BoxParameters` passed to it using the same dot accessor and expose values that were assigned to it; the dot is an operator, and not exclusive to just accessors, but properties, functions, and all struct and class members. The first statement creates a new `Vector3 size`. The size is then assigned a new `Vector3()`, where we extract the width, height, and depth and assign those values to the `Vector3`'s `x`, `y`, and `z` parameters.

Once the `Vector3` is created, we assign the values to the `gameObject.transform.localScale`. After this, we're also able to change the `gameObject` material's color by assigning the `renderer.material.color` to the `box.color` from the `BoxParameters` values.

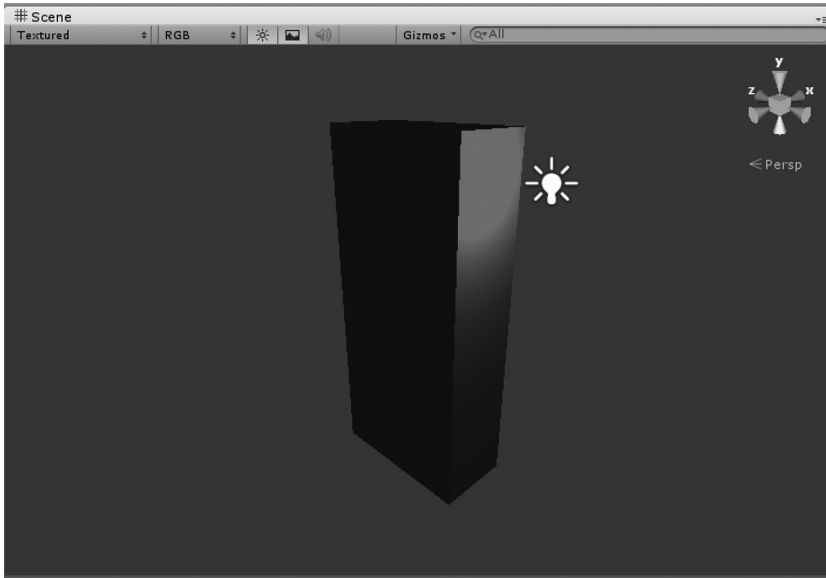
To put these two things together, we can make some interesting manipulations in the `gameObject`'s `Update ()` function.

```

void Update ()
{
    float h = (100 * Mathf.Sin(Time.fixedTime))/10;
    myParameters.height = h;
    UpdateCube(myParameters);
}

```

Here we use some simple math `(100 * Mathf.Sin(Time.fixedTime))/10`; and assign that value to `float h`. Then we need to update only one variable in the struct to update the entire `gameObject`.



Now we've got a stretching cube. By grouping variables together, we've reduced the number of parameters we need to pass to a function or even the number of functions we would have to use to accomplish the same task.

6.8.5 Accessing Structs

If a structure is limited to the class where it was created, then it becomes less useful to other classes that might need to see it. By creating another class to control the camera, we can better understand why other classes need access to the cube's `BoxParameters`.

By moving the struct outside of the class, we break out of the encapsulation of the classes.

```
using UnityEngine;
using System.Collections;
//living out in the open
public struct BoxParameters
{
    public float width;
    public float height;
    public float depth;
    public Color color;
}
public class Struct : MonoBehaviour
{
    public BoxParameters myParameters;
    //Use this for initialization
    void Start () {
        myParameters.width = 2;
        myParameters.height = 3;
        myParameters.depth = 4;
        myParameters.color = new Color(1,0,0,1);
    }
    void UpdateCube(BoxParameters box)
    {
        Vector3 size = new Vector3(box.width, box.height, box.depth);
        gameObject.transform.localScale = size;
        gameObject.renderer.material.color = box.color;
    }
}
```

```

    }
    //Update is called once per frame
    void Update ()
    {
        float h = (100 * Mathf.Sin(Time.fixedTime))/10;
        myParameters.height = h;
        UpdateCube(myParameters);
    }
}

```

From the above completed code of the Struct class, we can see that the struct has been moved outside of the class declaration. This changes the BoxParameters into a globally accessible struct.

```

//living out in the open
public struct BoxParameters
{
    public float width;
    public float height;
    public float depth;
    public Color color;
}

```

When a class is made, its contents are encapsulated within curly braces. The variables or functions created inside of the class contents are members of that class. Should any new enum or struct be moved outside of that class, it becomes a globally declared object.

Create a new class called UseStruct.cs and add it to the Main Camera in the scene.

```

using UnityEngine;
using System.Collections;
public class UseStruct : MonoBehaviour
{
    BoxParameters ThatBox;
    //uses globally accessible BoxParameters struct!
    //Use this for initialization
    void Start ()
    {
    }
    //Update is called once per frame
    void Update ()
    {
        ThatBox =
        GameObject.Find("Cube").GetComponent<Struct>().myParameters;
        gameObject.transform.position =
        new Vector3(0,ThatBox.height*0.5f, -10);
    }
}

```

The new class can see the BoxParameters struct because it's not confined within the Struct.cs class. The myParameters should be made public so it can be accessed outside of the Struct class. We can then assign UseStruct's ThatBox struct the Cube's Struct.myParameters. Through some clever use of ThatBox.height, we can make the camera follow the resizing of the Cube in the scene with the following statement:

```

gameObject.transform.position = new Vector3(0, ThatBox.height*0.5f, -10);

```

6.8.6 Global Access

In practice, it's a good idea to have a more central location for your globally accessible structs and enums. A third class in the project called Globals.cs can contain the following code:

```
using UnityEngine;
using System.Collections;
public struct BoxParameters
{
    public float width;
    public float height;
    public float depth;
    public Color color;
}
```

With something as simple as this, we're able to create a central location for all globally accessible information. Starting with simple things such as `public struct BoxParameters`, we can then continually add more and more useful information to our globally accessible data. Each C# script file need not declare a class. Although it's common, it's not restrictively enforced that a C# script file contains a class. Utility files like a `Globals.cs` are handy and can provide a clean system in which each programmer on a team can find handy structures that can be shared between different objects in the scene.

6.8.7 What We've Learned

In Section 6.8.6, we talked about global access. This concept is not that new. When you write a class, in this case `Structs`, it's accessible by any other class. Therefore, when we assigned `UseStruct` to the Main Camera in the scene, it's able to use `Structs`.

6.9 Class Data

Using a struct is preferred over a class due to how the computer manages memory. Without going into detail, one type of memory is called the heap and the other is the stack. In general, the stack is a smaller, more organized, faster part of the memory allocated to your game. The heap is usually larger and can take longer to access than the stack.

Values can be allocated to the stack, and a struct is a value type; thus, a stack can be accessed faster than a class. Though often structs and other value types are used in ways that disallow their allocation to the stack, they go into the heap instead. Classes and other reference data types end up in the heap regardless.

In a simple function call, the variables that appear in it are all pushed into the stack. As soon as the function is complete, any of the temporary values created and used inside of the function are immediately destroyed once the function is done executing.

```
void DoThings()
{
    int[] arrayOfInts = new int[100]
    for (int i = 0; i < 100 ; i++)
    {
        arrayOfInts[i] = i;
    }
}
```

The `DoThings()` function does pretty much nothing, but makes an array of ints that are added to the stack. As soon as this function is done, the array of ints is cleared out. In general, the stack grows and shrinks very fast. The second object that's added to the stack is the `int i`, which is being used inside of the `for` loop.

The primary advantage a class has over a struct is the addition of a constructor and class inheritance. Aside from that, you're able to add assignments to each variable in a class as it's created.

```

struct MyStruct
{
    public int a = 0;
    public MyStruct()
    {
    }
}
class MyClass
{
    public int a = 0;
    public MyClass()
    {
    }
}

```

The code above produces the following errors:

```

Assets/ClassVStruct.cs(7,28): error CS0573: 'ClassVStruct.MyStruct.a':
Structs cannot have instance field initializers
Assets/ClassVStruct.cs(8,24): error CS0568: Structs cannot contain explicit
parameterless constructors

```

In a struct, you're not allowed to assign values to the fields. Therefore, `public a = 0;` is allowed only in a class. Default values are allowed only in classes. Likewise, the constructor `public MyStruct()` isn't allowed, but it is in a class.

We could easily declare a class that has nothing more than data in it; this will look much like any struct.

```

class PlayerData
{
    public Vector3 position;
    public int hitpoints;
    public int ammo;
    public float runSpeed;
    public float walkSpeed;
}

```

This would do the same things as a struct of the same complexity is concerned. However, a struct has the chance to be a bit faster than a class to access. A struct is also a step up from an enum. The declaration might look somewhat similar, but an enum doesn't allow any values to be assigned to its constituent declarations.

6.9.1 Character Base Class

For all of the monsters and players to have an equal understanding of one another, they need to share a common base class. This means that simply they all need to share the same parent class that holds data structures, which all of them are aware of. This concept is called inheritance, and it's something that only a class can do.

```

class Monster
{
}
class Zombie : Monster
{
}

```

The `Zombie` uses the `: Monster` statement to tell C# that the `Zombie` is inheriting from `Monster`. We will go into this again in 6.13 and 6.23, but it's important that we're used to seeing this notation and what it's for.

Once each type of character shares the same parent class, it can use the same data. This is important so that zombies can understand what data makes up a human, and vampires can know to avoid zombie blood. To do this, the zombies need to know how to deal with the player's data structure and the player needs to know how to deal with zombies and vampires.

When we start our game, we're going to have a starting point for the player and for each monster. Classes are a clean way to store data when a game is started. Starting data can include basic chunks of information that don't change, such as minimum and maximum values.

Structs don't allow you to declare values for variables when they are created. Classes, on the other hand, do allow you to assign values. More important, we can make them unchangeable.

6.9.2 Const

The `const` keyword is used to tell anyone reading the class that the value assigned cannot change during the game. The keyword is short for constant, and it means that you shouldn't try to change it either. There are a few different ways which you can assign a value to behave the same as a `const`, but just by looking at the code, you should know that the following code has a value that shouldn't be changed while playing.

```
class Monster
{
    const int MaxHitPoints = 10;
}
```

If a value is declared to be `const`, then you should keep in mind how it's used. These are good for comparisons, if a value is greater than `MaxHitPoints`, then you have a very clear reason for why this value is set.

When you've declared a class and set several `const` values, this gives the class purposeful location to store all kinds of "set in stone" numbers. One problem with giving a class `const` values is that they can not be changed after they have been set. So make sure that these values shouldn't need to change once the game is running.

This allows you to create a constant value for the class. To make the value accessible, we need to change the declaration.

```
class Monster
{
    public const int MaxHitPoints = 10;
}
```

The accessor must come before the readability of the variable. Likewise, this can also be made `private const int MaxHitPoints`. This allows you to set the value when writing the code for the classes. However, if you want to make a variable have a limited availability to set, then we can use a different declaration.

6.9.3 Readonly

The `readonly` declaration allows you to change the variable only when a class is declared or as it's written when it was initialized.

```
public class Monster
{
    public readonly int MaxHitPoints = 10;
    public void SetMaxHP(int hp)
    {
        this.MaxHitPoints = hp;
    }
}
```


The code at the bottom of previous page will produce the following error:

A readonly field 'Monster.MaxHitPoints' cannot be assigned to (except in a constructor or a variable initializer)

This means that a readonly variable can only be set in a variable, an initializer, or a constructor. This means that the only way to set `MaxHitPoints` can be with the following code:

```
public class Monster
{
    public readonly int MaxHitPoints = 10;
    public Monster(int hp)
    {
        this.MaxHitPoints = hp;
    }
}
```

By using a constructor, we can set the `MaxHitPoints` once, or we can use the value when the variable is declared. Therefore, the first way `MaxHitPoints` can be set is when we use the initialization line `public readonly int MaxHitPoints = 10;` where we set `MaxHitPoints` to 10, or in the constructor where we set the value to the `(int hp)` in the argument list. This differs from the `const` keyword, where we can use the initializer only to set the value of the variable `MaxHitPoints`.

These are only simple uses of the `const` and `readonly` variable declarators. Once you begin reading other programmer's code you've downloaded online or observed from team members, you'll get a better idea of the various reasons for using these keywords. It's impossible to cover every context in a single chapter.

6.9.4 What We've Learned

Using these keywords is quite simple, and it makes quite clear what you're allowed to do with them. By understanding what the keywords mean to the accessibility of the variable, you'll gain a bit of insight into how the value is used.

`const` and `readonly` mean that the variable cannot be changed either outside of an initial declaration or as the code is written. This means that the value needs to remain unchanged for a reason. Of course, knowing that reason requires a great deal of context, so it's impossible to lay out every reason here. However, it's safe to assume someone added the declaratory keyword for a reason.

6.10 Namespaces

Classes are organized by their namespace. A namespace is a system used to sort classes and their contained functions into named groups. It will be problematic if programmers in the world come up with unique names for their classes and functions. Namespaces are used to control scope. By encapsulating functions and variables within a namespace, you're allowed to use names within the namespace that might already be in use in other namespaces.

6.10.1 A Basic Example

A namespace is basically an identifier for a group of classes and everything contained in the classes. Within that namespace, we can declare any number of classes. Starting in the Namespaces project, we'll look at the `MyNamespace.cs` class and within the new file add in the following code:

```
namespace MyNamespace
{
}
```

A namespace is declared like a class. However, we replace the keyword `class` with `namespace`. This assigns the identifier following the keyword to being a new namespace. A new namespace gives us a way to categorize our new classes under a group identified in this case as `MyNameSpace`.

Inside of the namespace, we can add in a new class with its own functions. Adding a class to the namespace should be pretty obvious.

```
namespace MyNameSpace
{
    public class MyClass
    {
    }
}
```

However, to make the class available to anyone who wants to use the class in the namespace, we need to make it public. Within the class in the namespace, we'll add in a function to make this example complete.

```
namespace MyNameSpace
{
    public class MyClass
    {
        public void MyFunction()
        {
            print("hello from MyNamespce");//oops?
        }
    }
}
```

If we try to use `print` inside of the new `MyFunction` code block, we'll get the following error:

```
Assets/MyNameSpace.cs(7,25): error CS0103: The name 'print' does not exist in the current context
```

Of course, this requires some external libraries to pull from, so we'll need to add in a directive to `System`. Our finished example looks like the following:

```
using UnityEngine;
namespace MyNameSpace
{
    public class MyClass
    {
        public void MyFunction() {
            print("hello from MyNamespce");
        }
    }
}
```

6.10.2 Directives in Namespaces

Now we need to test out our new namespace and its contained class and function. To do this, we'll need to start a new `Example.cs` and assign it to the camera in a new Unity 3D Scene.

```
using UnityEngine;
using System.Collections;
using MyNameSpace;//adding in a directive to our namespace
public class Example : MonoBehaviour {
```

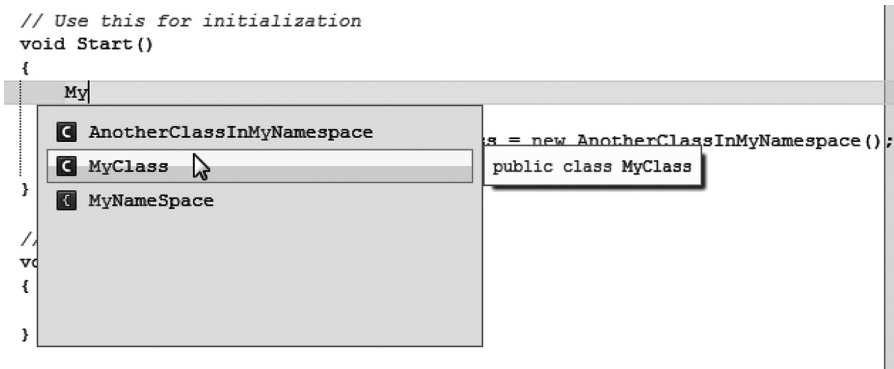
```

//Use this for initialization
void Start () {
}
//Update is called once per frame
void Update () {
}
}

```

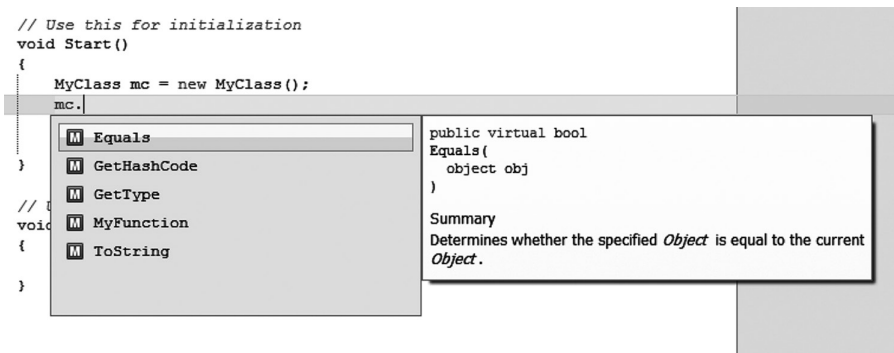
At the end of the directives, add in a new line using `MyNameSpace;` to incorporate the work we just finished. This will give us access to the classes and functions within the new namespace. This should also be a clue as to what a directive is doing.

If you examine the first line using `UnityEngine;`, we can assume there's a namespace called `UnityEngine`, and within that namespace, there are classes and functions we've been using. Now we should have a way to find the classes within our own `MyNameSpace`.



If we go into the `Start ()` function in the `Example.cs` file and add in `My...`, we'll be prompted with an automatic text completion that includes `MyClass` which is a class inside of `MyNameSpace`. This is reinforced by the highlighted info `class MyNameSpace.MyClass` following the pop-up.

After making a new instance of `MyClass()`, we can use the functions found inside of the class.



This shows how the functions and classes are found inside of a namespace. Using a new namespace should usually be done to prevent your class and variable names from getting mixed up with identifiers that might already be in use by another library.

```

void Start ()
{
    MyClass mc = new MyClass();
    mc.MyFunction();
}

```

When this code is run, you'll be able to get the following output from the Console in Unity 3D:

```
hello from MyNameSpace
UnityEngine.Debug:Log(Object)
MyNameSpace.MyClass:MyFunction() (at Assets/MyNameSpace.cs:6)
Example:Start () (at Assets/Example.cs:10)
```

This shows where the function was called and what namespace the function originated in. In most cases, you'll create a new namespace for the classes in your new game. This becomes particularly important when you need to use third-party libraries. You might have a function called `findClosest()`, but another library might also have a function with the same name.

If we create C# file named `AnotherNameSpace` and add the following code, we'd be able to observe how to deal with multiple namespaces and colliding function and class names.

```
namespace AnotherNameSpace
{
    using UnityEngine;
    public class MyClass
    {
        public void MyFunction ()
        {
            Debug.Log("hello from AnotherNameSpace");
        }
    }
}
```

Therefore, now if we compare `MyNameSpace` and `AnotherNameSpace`, we're going to have duplicated class and function names. Thus, how do we know which `MyClass` is being used? In the `Example.cs` file, we need to add a new directive to include the `AnotherNameSpace` into the `Example`.

6.10.3 Ambiguous References

Coming up with unique names for every variable, class, or function can be difficult. Another class might already use commonly used words like speed, radius, or vector. Namespaces offer a system to help separate your class member names from another class that might already be using the same names.

```
using UnityEngine;
using System.Collections;
using MyNameSpace;
using AnotherNameSpace;
```

This will involve duplicates for `MyClass`, which is cause for some problems. Then Unity 3D will show you the following error:

```
Assets/Example.cs(10,17): error CS0104: 'MyClass' is an ambiguous reference
between 'MyNameSpace.MyClass' and 'AnotherNameSpace.MyClass'
```

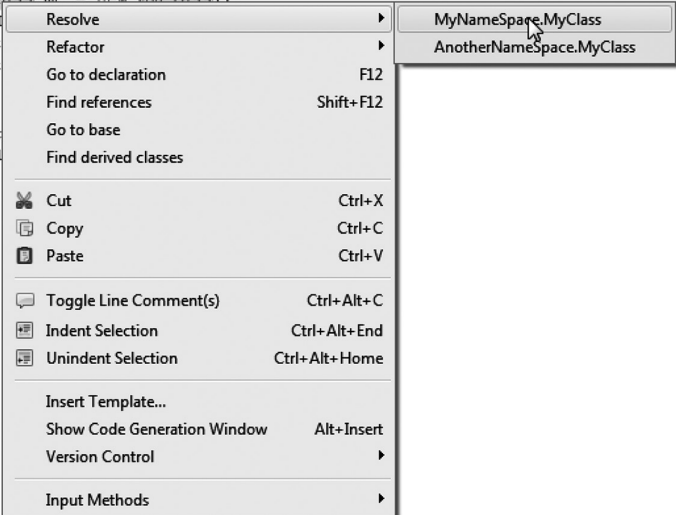
Ambiguous references become a common problem if we start to include too many different namespaces. We could possibly fix this by changing the name of the classes in one of the functions, but that wouldn't

be possible if the reference came from a dynamic linking library (DLL) or another author's code. We could also change namespaces which we're using, but again, that would be a work around and not a solution.

```
// Use this for initialization
void Start()
{
    MyClass mc = new MyClass();
    // Unknown resolve error.
    MyNamespace.AnotherClass.MyFunction();
}
```

If we hover over the class with the cursor in MonoDevelop, we'll get a pop-up telling us which version is currently being used. Of course, this is just a guess, and a bad one at that. This can be resolved by expanding the declaration. The right-click pop-up has a selection called Resolve.

```
// Use this for initialization
void Start()
{
    MyClass mc = new MyClass();
    mc.MyFunction();
    // Update
    void Update()
    {
    }
}
```



This will allow us to pick which `MyClass()` we want to use. The resolution is to add on the full dot notation for the namespace we want to use.

```
void Start ()
{
    MyNamespace.MyClass mc = new MyNamespace.MyClass();
    mc.MyFunction();
}
```

Of course, it's not the most pretty, but this is the most correct method to fix the namespace we're requesting a `MyClass()` object from. Things can get too long if we need to nest more namespaces within a namespace.

```
using UnityEngine;
namespace MyNameSpace
{
    namespace InsideMyNameSpace
    {
        public class MyClass
        {
            public void MyFunction ()
            {
                Debug.Log("hello from InsideMyNameSpace!");
            }
        }
    }
    public class MyClass
    {
        public void MyFunction()
        {
            Debug.Log("hello from MyNameSpace");
        }
    }
}
```

This example shows a new namespace within `MyNameSpace` called `InsideMyNameSpace`. This namespace within a namespace is called a nested namespace. The following code will work as you might expect:

```
void Start ()
{
    MyNameSpace.InsideMyNameSpace.MyClass imc =
    new MyNameSpace.InsideMyNameSpace.MyClass();
    imc.MyFunction();//hello from InsideMyNameSpace!
}
```

This is long and messy: Is there a way we can fix this? As a matter of fact, there is a way to make this easier to deal with.

6.10.4 Alias Directives

Sometimes namespaces can get quite long. Often when you need to refer to a namespace directly you may want to shorten the name into something easier to type. Namespace aliases make this easier.

```
using UnityEngine;
using System.Collections;
using imns = MyNameSpace.InsideMyNameSpace;//shortens a long directive
using AnotherNameSpace;
```

Like a variable `int x = 0;` we can assign a directive using a similar syntax `using x = UnityEngine;` for instance. When we have a namespace nested in another namespace, we can assign that to an identifier as well. To shorten `MyNameSpace.InsideMyNameSpace` to `imns`, we use a statement that looks like `using imns = MyNameSpace.InsideMyNameSpace;` to reduce how much typing we need to use to express which namespace we're using.

```
void Start ()
{
    imns.MyClass imc = new imns.MyClass();
    imc.MyFunction();
}
```

Now we can reduce the declaration of `imc` to a nice short statement. Of course, unless we have an ambiguous name to worry about, there's no need for being so explicit with the function we're specifically trying to call. With descriptive function and variable names, we can avoid directive aliases.

6.10.5 Putting Namespaces to Work

Namespaces allow you to create a new sort of data that can be shared between classes. For instance, in your game you'll most likely have a fairly detailed set of rules and objects that the rest of your game classes will want to share. The best way to keep all of these classes organized is for any sort of shared data to originate from the same namespace.

```
namespace Zombie
{
    using UnityEngine;
    using System.Collections;
    public class MonsterInfo
    {
        public int health;
        public int armor;
        public int attack;
        public MonsterInfo()
        {
            health = 10;
            armor = 1;
            attack = 3;
        }
    }
}
```

Here, we have a new namespace called `Zombie` that is saved in `Zombie.cs` in the `Assets` directory of our Unity 3D project. In this new namespace, we have a class called `MonsterInfo` that holds three different numbers. An `int` is used for `health`, `armor`, and `attack`. Once this namespace is created and named, we're able to use any class within it from any other class we create in our game project.

Now if we want to create another monster that uses `MonsterInfo` to store its game-related information, all we need to do is add `using Zombie;` in the C# file before the class declaration. Therefore, in a new `Monsters.cs` file, we add the following code sample. This also goes for any other class that might need to know what a monster's information looks like.

```
using UnityEngine;
using System.Collections;
using Zombie;
public class Monsters : MonoBehaviour
{
    MonsterInfo monster = new MonsterInfo();
    //Use this for initialization
    void Start ()
    {
        MonsterInfo m = monster;
        Debug.Log(m.health);
        Debug.Log(m.armor);
        Debug.Log(m.attack);
    }
    //Update is called once per frame
    void Update ()
    {
    }
}
```

This file now has a class called `Monsters`, and when it's instantiated, it will create a new `MonsterInfo`. When the `Monsters' Start ()` function is called, we'll get a list of the health, armor, and attack printed out to the Console panel. This should all be fairly straightforward, but here's where the real use comes into play. To see why namespaces are used, we'll create another C# file called `Player.cs`, and in the file, we'll also add the `using Zombie;` statement to import all of the same classes that are used in `Monster.cs`. With this addition, the player can also have access to the same `MonsterInfo` data that is being stored in the `Monsters.cs`.

```
using UnityEngine;
using System.Collections;
using Zombie;
public class Player : MonoBehaviour
{
    public Monsters monster;
    public int attackPower;
    void AttackMonster()
    {
        if (monster != null)
        {
            MonsterInfo mi = monster.monsterInfo;
            Debug.Log(mi.armor);
            if (attackPower > = mi.armor && mi.health > 0)
            {
                monster.TakeDamage(attackPower - mi.armor);
            }
        }
    }
}
```

The access to the `monsterInfo` is important for the player if he or she needs to know any values from the monster before making an attack. In this case, we create a new function called `AttackMonster()` where we check for a monster; if we have one, we obtain his `monsterInfo` by using `MonsterInfo mi = monster.monsterInfo;` to create a local version of the `monsterInfo` data.

Then we check on our `attackPower`, and if we can strike harder than the monster's armor and the monster is still alive, we'll apply some damage to the monster with a function in the monster called `TakeDamage();`, which means that `Monster` has a function we need to fill in.

```
//added into Monster.cs
public void TakeDamage(int damage)
{
    monsterInfo.health -= damage;
}
```

We'll add in a simple function called `TakeDamage()` that takes in an integer value. On the player side, the damage applied is the player's `attackPower` after we subtract the monster's armor value. This then reduces the `monsterInfo.health` attribute.

6.10.6 Extending Namespaces

Namespaces can be extended quite easily. In another new C# file named `ExtendingMyNamespace.cs`, we have the following code:

```
namespace MyNamespace
{
    using UnityEngine;
    public class AnotherClassInMyNamespace
```



```

    {
        public void MyFunction()
        {
            Debug.Log("hello from MyNamespace");
        }
    }
}

```

This means that there's a new class called `AnotherClassInMyNamespace()` in addition to `MyClass()`. These two namespace files work together, and when the `using MyNamespace;` directive is used, it allows for both `MyClass()` and `AnotherClassInMyNamespace()` to exist side by side. No additional code is needed aside from `using MyNamespace;`, which is required for `Example.cs` to see both versions of `MyNamespace`.

However, there are things to be careful about. If you add class `MyClass` inside of any other namespace `MyNamespace` files, you'll get the following error:

```
Assets/MyNamespace.cs(4,22): error CS0101: The namespace 'MyNameSpace'
already contains a definition for 'MyClass'
```

The collection of namespaces acts as one. They aggregate together into a single form when compiled, so even though they are separated into two different files, they act like one.

6.10.7 What We've Learned

This is a simple way to allow multiple classes work together by sharing a class through a namespace. Both the player and the monster are aware of `monsterInfo`, thanks to using the `monsterInfo` found in the `Zombie` namespace. We should get used to the idea of adding multiple classes to the `Zombie` namespace to make it more useful. Things such as ammunition types and weapons should all be added as different classes under the `Zombie` namespace.

In many cases, it's useful to build a namespace for your game where you'll be creating classes and functions that need to be used throughout your project. Something like `CreateZombie()` seems like a function that will be called often. If that's the case, then it's often useful to have a namespace called `ZombieGame;`. By adding `using ZombieGame;` to all of your classes, they'll all have access to the `CreateZombie()` function.

The namespace should also contain common structs that every class will need to make use of. Specifics such as a `ZombieInfo{}` struct could contain position, aggression level, defenses, and weapons. Character spell abilities and effects can be held in the namespace. Basically, anything you need to do often and anywhere should be kept in the namespace.

6.11 Functions Again

So far we've been working in either the `Start ()` function or the `Update ()` function because they are entry points. Because of this, they are automatically called by the engine. In a non-Unity 3D Engine program, the default entry point would be something like `public static void Main()`.

6.11.1 Parameter Lists

When a function is called, its identifier followed by its parameter list is added to code. Parameters are like little mail slots on the front door of a house. There might be a slot for integers, floats, and arrays. Each slot on the door, or argument in the parameter list, is a type followed by an identifier.

6.11.1.1 A Basic Example

If we look at a basic example, we'll see how all this works.

```
using UnityEngine;
using System.Collections;
public class Example : MonoBehaviour {
    int a = 0;
    void Start ()
    {
    }
    void SetA (int i)
    {
        a = i;
    }
}
```

`void SetA (int i)` takes in one parameter of type `int`. This is a simple example of a value parameter; there are other types of parameters which we'll go into next. The value parameter declares the type and an identifier whose scope is limited to the contents of the function it's declared with.

```
void Start ()
{
    print(a);
    SetA(3);
    print(a);
}
```

To use our function, add in the preceding lines of code to the `Start ()` function in Unity 3D. When you start the game, you'll see a 0 followed by a 3. We can do this any number of times, and each time we're allowed to change what value we set A to. Of course, it's easier to use `a = 3` in the `Start ()` function, but then we wouldn't be learning anything.

6.11.2 Side Effects

When a function directly sets a variable in the class the function lives in, programmers like to call this a side effect. Side effects are usually things that programmers try to avoid, but it's not always practical. Writing functions with side effects tend to prevent the function from being useful in other classes. We'll find other systems that allow us to avoid using side effects, but in some cases, it's necessary.

```
Class MySideEffect
{
    int a = 0;
    void SetA ()
    {
        a = 5;
    }
}
```

The above `SetA()` function sets `int a` to 5. The `SetA()` function does have a clear purpose, setting A, but there's nothing in the function that indicates where the variable it's setting lives. If the variable lived in another class which `MySideEffect` inherited from, then you wouldn't even see the variable in this class.

Once the complexity in a class grows, where and when a variable gets changed or read becomes more obscure. If a function remains self-contained, testing and fixing that function becomes far easier.

Reading what a function does should not involve jumping around to find what variable it's making changes to. This brings us back to the topic of scope. Limiting the scope or reach of a function helps limit the number of places where things can go wrong.

```
class MySideEffect {
    int a = 0;
    void SetA ()
    {
        a = 5;
    }
    void SetAgain ()
    {
        a = new int();
    }
}
```

The more functions that have a side effect, the more strange behaviors might occur. If an `Update ()` function is expecting one value but gets something else entirely, you can start to get unexpected behaviors. Worse yet, if `SetAgain()` is called from another class and you were expecting `a` to be 5, you're going to run into more strange behaviors.

6.11.3 Multiple Arguments

There aren't any limits to the number of parameters that a function can accept. Some languages limit you to no more than 16 arguments, which seems acceptable. If your logic requires more than 16 parameters, it's probably going to be easier to separate your function into different parts. However, if we're going to be doing something simple, we might want to use more than one parameter anyway.

```
int a = 0;
void Start ()
{
}
void SetA (int i, int j)
{
    a = i * j;
}
```

We can add a second parameter to `SetA`. A comma token tells the function to accept another variable in the parameter list. For this example, we're using two `ints` to multiply against one another and assign a value to `a`. There isn't anything limiting the types we're allowed to use in the parameter list.

```
int a = 0;
float b = 0;
void Start ()
{
}
void SetA (int i, float j)
{
    a = i;
    b = j;
}
```

The function with multiple parameters can do more. Value parameters are helpful ways to get data into a function to accomplish some sort of simple task. We'll elaborate more on this by creating something useful in Unity 3D.

6.11.4 Useful Parameters

Often, we need to test things out before using them. Let's say we want to create a new primitive cube in the scene, give it a useful name, and then set its position. This is a simple task, and our code might look like the following:

```
void Start ()
{
    GameObject g = GameObject.CreatePrimitive(PrimitiveType.Cube);
    g.name = "MrCube";
    g.transform.position = new Vector3(0,1,0);
}
```

Our cube is given a name, and it's assigned a place in the world. Next, let's say we want to make a bunch of different cubes in the same way. We could do something like the following:

```
void Start ()
{
    GameObject g = GameObject.CreatePrimitive(PrimitiveType.Cube);
    g.name = "MrCube";
    g.transform.position = new Vector3(0,1,0);
    GameObject h = GameObject.CreatePrimitive(PrimitiveType.Cube);
    h.name = "MrsCube";
    h.transform.position = new Vector3(0,2,0);
    GameObject i = GameObject.CreatePrimitive(PrimitiveType.Cube);
    i.name = "MissCube";
    i.transform.position = new Vector3(0,3,0);
    GameObject j = GameObject.CreatePrimitive(PrimitiveType.Cube);
    j.name = "CubeJr";
    j.transform.position = new Vector3(0,4,0);
}
```

If this code looks horrible, then you're learning. It's accomplishing a task properly, but in terms of programming, it's horrible. When you intend to do a simple task more than a few times, it's a good idea to turn it into a function. This is sometimes referred to as the "Rule of Three," a term coined in an early programming book on refactoring code.

6.11.4.1 The Rule of Three

The Rule of Three is a simple idea that any time you need to do any given task more than three times it would be better served by creating a new single procedure to accomplish the task that has been done manually. This reduces the chance of error in writing the same code more than three times. This also means that changing the procedure can be done in one place.

```
void CreateANamedObject (PrimitiveType pt, string n, Vector3 p)
{
    GameObject g = GameObject.CreatePrimitive(pt);
    g.name = n;
    g.transform.position = p;
}
```

We take the code that's repeated and move it into a function. We then take the values that change between each iteration and change it into a parameter. In general, it's best to put the parameters in the same order in which they're used. This isn't required, but it looks more acceptable to a programmer's eyes.

```
void Start ()
{
    CreateANamedObject(PrimitiveType.Cube, "MrCube", new Vector3(0,1,0));
}
```

We then test the function at least once before writing any more code than we need to. If this works out, then we're free to duplicate the line of code to accomplish what we started off doing.

```
void Start ()
{
    CreateANamedObject(PrimitiveType.Cube, "MrCube", new Vector3(0,1,0));
    CreateANamedObject(PrimitiveType.Cube, "MrsCube", new Vector3(0,2,0));
    CreateANamedObject(PrimitiveType.Cube, "MissCube", new Vector3(0,3,0));
    CreateANamedObject(PrimitiveType.Cube, "CubeJr", new Vector3(0,4,0));
}
```

Again, though, we're seeing a great deal of duplicated work. We've used arrays earlier, and this is as good a time as any to use them. By observation, the main thing that is changing here is the name. Therefore, we'll need to add each name to an array.

```
string[] names = new string[] { "MrCube", "MrsCube", "MissCube", "CubeJr" };
```

The variable declaration needs to change a little bit for an array. First of all, rather than using `type identifier`, a pair of square brackets are used after the type. The statement starts with `string[]` rather than `string`. This is followed by the usual identifier we're going to use to store the array of strings.

Unlike an integer type, there's no default value that can be added in for this array of strings. To complete the statement, we need to add in the data before the end of the statement. The new keyword is used to indicate that a new array is going to be used. The array is a special class that is built into C#. Therefore, it has some special abilities that we'll get into later.

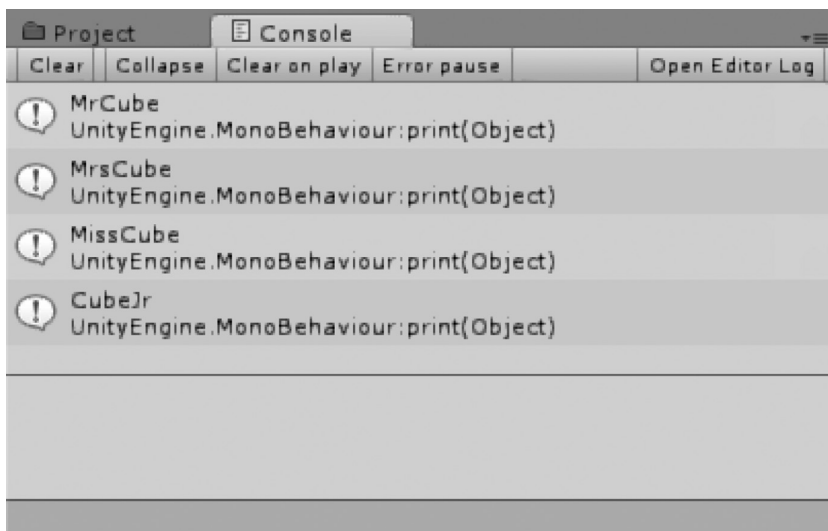
Now that we've declared an array of names, we'll need to use them.

6.11.5 Foreach versus For

The `foreach` loop is often our `goto` loop for iterating through any number of objects.

```
string[] names = new string[] { "MrCube", "MrsCube", "MissCube", "CubeJr" };
void Start ()
{
    foreach(string s in names)
    {
        Debug.Log(s);
    }
}
```

The parameters for the `foreach` also introduce the keyword `in` that tells the `foreach` iterator what array to look into. The first parameter before the `in` keyword indicates what we're expecting to find inside of the array. Therefore, for this use of the `foreach` iterator, we get the output in the following page.



As expected, we get a list of the names found in the array we added. As we add in new functions and variables, it's important to test them out one at a time. This helps make sure that you're headed in the right direction one step at a time. Now we can switch out the `print` function for the clever function we wrote now.

```
foreach(string s in names)
{
    CreateANamedObject(PrimitiveType.Cube, s, new Vector3(0, 1, 0));
}
```

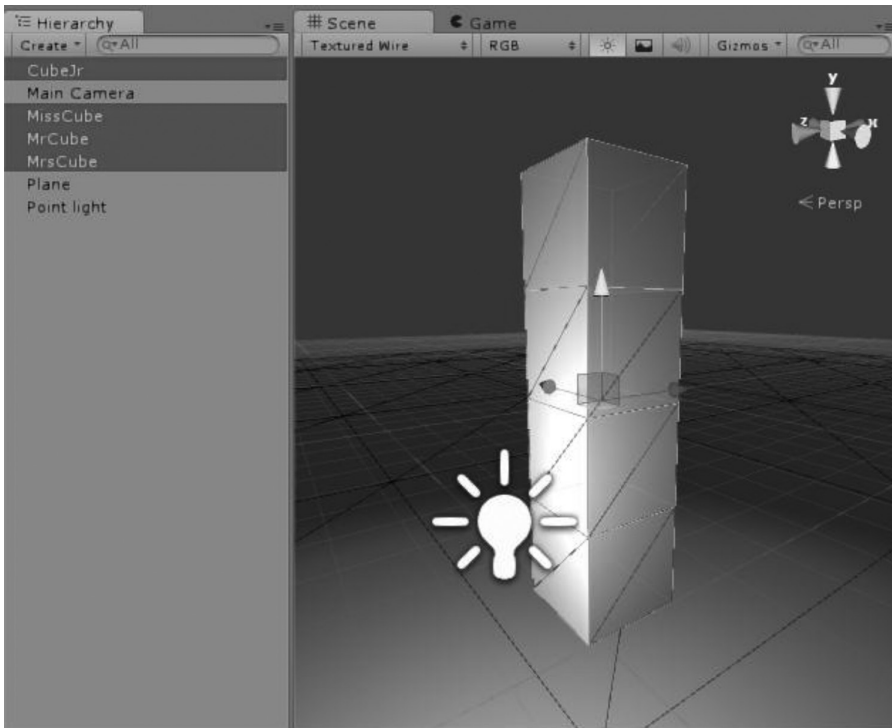
The `foreach` loop in some respects operates in the same way as a `while` loop.

```
float y = 1.0f;
foreach(string s in names)
{
    CreateANamedObject(PrimitiveType.Cube, s, new Vector3(0, y, 0));
    y += 1.0f;
}
```

It's simple to add in a variable outside of the loop that can be incremented within the loop.

```
string[] names = new string[] { "MrCube", "MrsCube", "MissCube", "CubeJr" };
void Start ()
{
    float y = 1.0f;
    foreach(string s in names)
    {
        CreateANamedObject(PrimitiveType.Cube, s, new Vector3(0, y, 0));
        y += 1.0f;
    }
}
void CreateANamedObject (PrimitiveType pt, string n, Vector3 p)
{
    GameObject g = GameObject.CreatePrimitive(pt);
    g.name = n;
    g.transform.position = p;
}
```

It's not time to admire our clever function. The `foreach` means we can add in any number of names to the array and the tower of cubes will get taller with named boxes.



6.11.6 What We've Learned

The `foreach` loop is useful in many ways, but it's also somewhat limited. If we had more than one array to iterate through at the same time, we'd have some difficulties. In some cases, it's easier to use a regular `for` loop. Arrays are lists of objects, but they're also numbered.

Arrays are used everywhere. On your own, experiment by adding more parameters to the function and more names to the list. We'll take a look at how to use other loops with multidimensional arrays and even jagged arrays in Chapter 7.

6.12 Unity 3D Execution Order

C# is an imperative language, which means that operations are executed in order, first one thing and then another. When Unity 3D makes an instance of a new `gameObject` with `MonoBehaviour` components, each component will have at least five functions called on it before the end of the frame where it was created. Additional rendering calls can be called as well.

Specifically, the `Start ()` and `Update ()` functions in Unity 3D are called in each `MonoBehaviour` in a specific order. Several other functions are also used by Unity 3D, and they each have specific moments when they are called.

A total of eight functions which Unity 3D calls are as follows:

```
Awake ()
OnEnable ()
Start ()
FixedUpdate ()
```

```
Update ()  
LateUpdate ()  
OnDisable()  
OnDestroy()
```

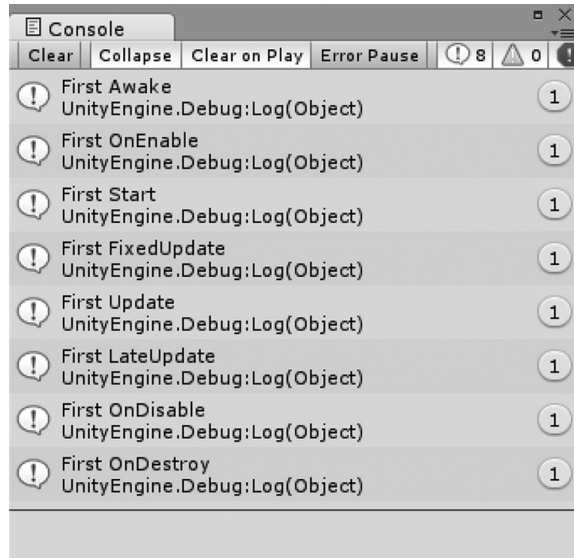
The order of this should make sense, considering that C# will execute statements in order. However, what happens when another class is instantiated in the `Awake()` function and it too will have its own `Awake()`, `OnEnable()`, and `Start ()` functions?

6.12.1 A Basic Example

Begin by examining the `ExecutionOrder` project; the `First.cs` class will look a bit like the following:

```
using UnityEngine;  
using System.Collections;  
public class First : MonoBehaviour  
{  
    void Awake()  
    {  
        Debug.Log("First Awake");  
    }  
    void OnEnable()  
    {  
        Debug.Log("First OnEnable");  
    }  
    void Start ()  
    {  
        Debug.Log("First Start");  
    }  
    void FixedUpdate ()  
    {  
        Debug.Log("First FixedUpdate");  
    }  
    void Update ()  
    {  
        Debug.Log("First Update");  
    }  
    void LateUpdate ()  
    {  
        Debug.Log("First LateUpdate");  
        //Destroy(this);  
    }  
    void OnDisable()  
    {  
        Debug.Log("First OnDisable");  
    }  
    void OnDestroy()  
    {  
        Debug.Log("First OnDestroy");  
    }  
}
```


This prints out the following:



The Console shows that the code is executed in the same order in which we laid them out in the class. When the `LateUpdate ()` function is called, we use `Destroy(this);` to begin deleting the class from the scene. When the class begins to delete itself, it's first disabled and then deleted.

When a new instance of a `MonoBehaviour` is created, it will also call its `Awake()`, `OnEnabled()`, and `Start ()` functions. Intuitively, it would make sense if you create an object in the `Awake()` function; the new object might wait till the end of `Awake()` before it begins its `Awake()` function. However, this isn't always the case.

```
void Awake ()
{
    Debug.Log("Awake Start");
    this.gameObject.AddComponent(typeof(Second));
    Debug.Log("Awake Done");
}
```

Say we create a new class called `Second.cs` and have it log its `Awake()` and `Start ()` functions as well. Then, in the `Second.cs` class we have the following code:

```
public class Second : MonoBehaviour
{
    void Awake ()
    {
        Debug.Log("Second Awake Start");
    }
    void OnEnable ()
    {
        Debug.Log("Second OnEnable Start");
    }
}
```

This produces the following output:

```
First Awake Start
Second Awake Start
Second OnEnable Start
First Awake Done
First OnEnable
```

Before the First class was created, the Second was able to get its Awake() done and move on to OnEnable() without interruption. First waited for Second to finish its Awake() and OnEnable() functions before First finished its own Awake() function. Afterward, OnEnable() of the First was finally called, even if Second took its own time to finish its Awake() function.

```
void Awake ()
{
    Debug.Log("Second Awake Start");
    for (int i = 0; i < 1000; i++)
    {
        Debug.Log("wait!");
    }
    Debug.Log("Other Awake Done");
}
```

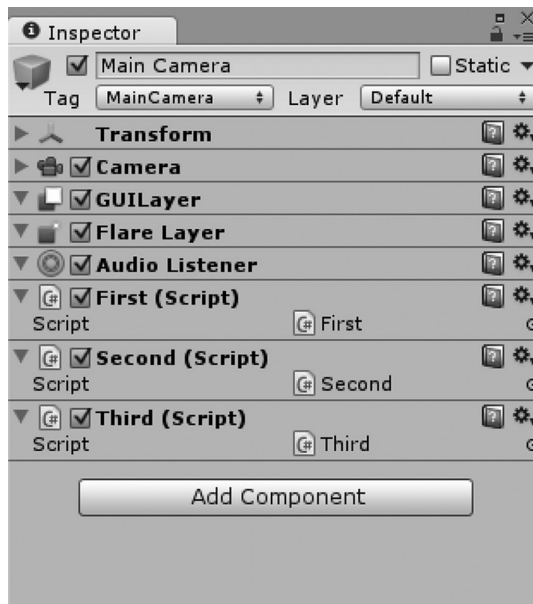


The First.cs class finishes its Awake() function, then the Second.cs class finishes its Awake() function and starts its OnEnable(). Then the First.cs class finishes its Awake() and starts its OnEnable() function. The First and Second being their Start () function. Notice that “First Awake Done” doesn’t appear until after “Second OnEnable.” The “Second OnEnable” is only printed once the “Second Awake Done” is printed, indicating that the for loop was able to finish.

When more objects are being created, the order in which these functions are called becomes more difficult to sort out. If you created several different objects, some of them might take even longer or shorter to finish their Awake(), OnEnable(), and Start () functions. Figuring all of this out takes a bit of thinking things through.

6.12.2 Component Execution Order

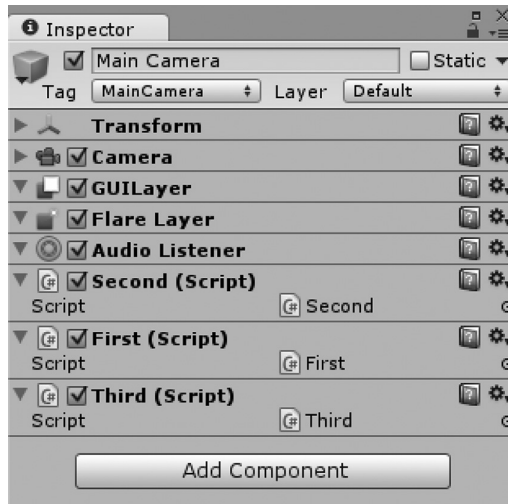
The execution of classes becomes a bit more confusing when they're attached to a `gameObject` as a prefab. In the scene, adding `Second` and `Third` classes to the `Main Camera` will show us some unexpected results.



Once all three `MonoBehaviours` have been attached to the `Main Camera`, we can observe the execution order of the functions found in each class.

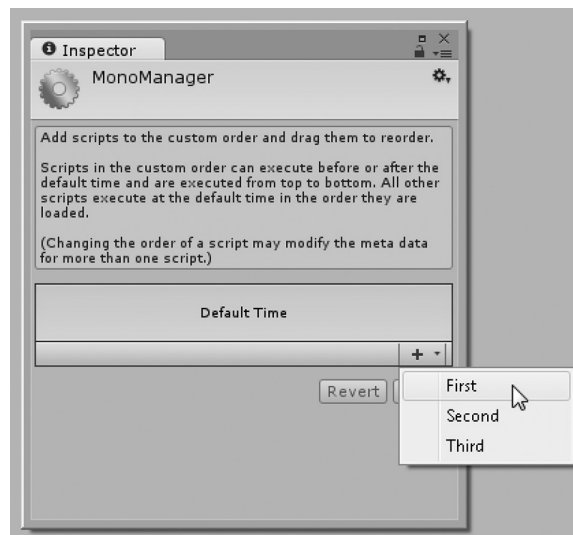


The `Third Awake()` and `OnEnable()` are called before the `Second` and the `First` even though the components look like they've been reordered in the opposite order on the `Main Camera`. However, this isn't always the case.



In the above screenshot, we swapped the `First` component with the `Second`. Even after changing the order in which they appear in the components list, we get no change in the Console's log output. This tells us that there's no specific order in which the objects are created and how they are ordered in the `gameObject`'s components list. However, there is a simple system in which we can change the execution order manually.

In Unity 3D under the `Edit` → `Preferences` → `Script Execution Order`, we can open a panel that tells Unity 3D how to prioritize when a class is called. By clicking on the `+` icon on the lower right of the panel, we can add our classes to an ordered list.



After your classes have been added to the list, you can make any changes by altering the number in the box to the right. Clicking on the `-` icon removes them from the list. Press `Apply` once you've added all of your classes to the list.



Once the scripts have been added to this list, they are guaranteed to be called in this order, ignoring how they are ordered in the components list on the `gameObject` they've been added to. Playing the scene with the components attached to the Main Camera in any order will always result in the following output:



After the scripts order is set, the Console panel shows us a more expected result. Of course, there are other ways to ensure that a class behaves as you might expect, but we will have to hold off on those

systems for Section 6.23. For now it's important that we observe how the functions operate and how to best use these behaviors to our advantage.

From observation, we know that we can maintain when each function is called.

6.12.3 What We've Learned

In this chapter, we looked at how to manage execution order between multiple classes. For simple projects, this execution ordering system should suffice. However, when you're not inheriting from `MonoBehaviour`, this management system will no longer work.

Once we get into more complex systems that do not inherit from `MonoBehaviour`, we'll build our own system for managing the execution order. There are systems that allow you to better control what functions are called and when, but we'll have to learn a few other things before getting to that.

Unity 3D expects many different things of your classes when you inherit from `MonoBehaviour`. If you include an `Update ()` function, it will automatically be called on every frame. However, when there are too many `Update ()` functions on too many classes running in a scene, Unity's frame rate can suffer.

Once a game gets moving along and creatures and characters are spawning and dying, it's impossible to know when and in what order each function is executed. Each object in the scene should be able to operate on its own. If an object depends on a variable, the value of which needs to be updated before it's used, then you may run into problems reading stale data.

6.13 Inheritance Again

Inheriting members of a parent class is only one facet of what inheritance does for your code. The behaviors of the functions also carry on in the child classes. If the parent has functions that it uses to find objects in the world, then so do the children. If the parent has the ability to change its behavior based on its proximity to other objects, then so do the children.

The key difference is that the children can decide what happens based on those behaviors. The data that is collected can be used in any way. A child class inherits the functions of its parent class. When the child's function is executed, it operates the same way as its parent. It can be more interesting to have the child's function behave differently from the inherited version of the function.

6.13.1 Function Overrides

The `override` keyword following the `public` keyword in a function declaration tells the child version of the function to clear out the old behavior and implement a new version. We've seen how member functions work when they're used by another class. Now we're going to see how to effect the operation of inherited functions.

6.13.1.1 A Basic Example

Revisiting the `Parent.cs` and `Child.cs` classes again in the `ParentChild` project, we should add the following code to the `Parent` class. Start by writing three new functions to the `Parent` class. Start with `ParentFunction ()` and then add in a print to say hello.

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

```

    public void ParentFunction()
    {
        print("parent says hello");
        FunctionA();
        FunctionB();
    }
    public void FunctionA()
    {
        print("function A says hello");
    }
    public void FunctionB()
    {
        print("function B says hello");
    }
}

```

When the `ParentFunction()` is added to the `Start ()` function, this will produce the expected "parent says hello" followed by "function A says hello" and then "function B says hello" in the Console panel in the Unity 3D editor. So far nothing surprising has happened.

The child class is based on the parent class, and we have an opportunity to make modifications by adding layers of code. To make see this modification, we need to reuse the code from the `Parent` class in the `Child` class. Take the `ParentFunction()` and add it to the `Start ()` function of the `Child` class.

```

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

```

Running this will produce the same output to the Console panel as the `Parent` class. Right now, there is no function overriding going on. If we intend to override a function, we should make the `Parent` class allow for functions to be overridden by adding the `virtual` keyword to the function we plan to override.

```

public virtual void FunctionA () {
    print("function A says hello");
}

```

Adding the `virtual` keyword after the `public` declaration and before the return data type, we can tell the function it's allowed to be overridden by any class inheriting its functions. Back in the `Child` class, we have the option to override the virtual function.

```

using UnityEngine;
using System.Collections;
public class Child : Parent
{
    void Start ()
    {
        ParentFunction();
    }
    public override void FunctionA()
    {
        print("Im a new version of function A");
    }
}

```

This has a new instruction for `FunctionA()`. As you might expect, the output to the Console panel is changed to something new. The Child class will send new data to the Console that should read

```
parent says hello
Im a new version of function A
function B says hello
```

Without the keywords `override` and `virtual`, we get the following warning:

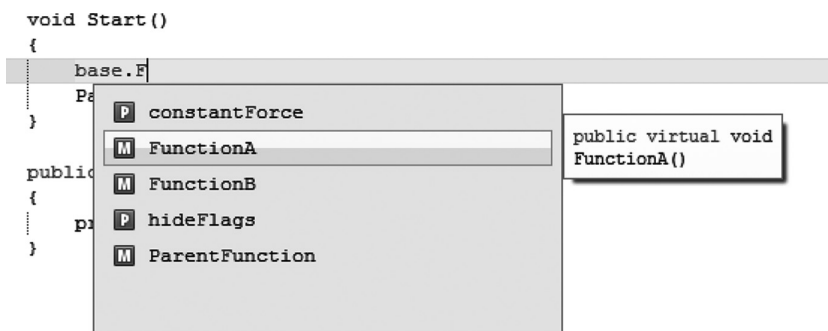
```
Assets/Child.cs(11,21): warning CS0108: 'Child.FunctionA()' hides inherited member 'Parent.FunctionA()'. Use the new keyword if hiding was intended
```

We can try this and watch the resulting output. In the Child class, we'll add in the new keyword. However, the output will remain the same as though the Parent class' version of `FunctionA()` was being used. This is because the `FunctionA()` living in the Child class is a new one, and not the one being used by the `ParentFunction()`. As you can imagine, this can get confusing.

```
public virtual void FunctionA()
{
    print("function A says hello");
}
```

6.13.1.1.1 Base

The Child class' version of the function hides the inherited member from its Parent class. Hiding means that the inherited version of the function is no longer valid. When called, only the new version will run. However, we have written code in the Parent class' version of the `FunctionA()`, which we may or may not want to override. To use the original version, we can add the keyword `base` to the beginning of the `FunctionA()`.



This produces the expected 3 to be printed out in the Console panel. For the Child class to take over the `FunctionA()` and create its own version of the function, use the `override` keyword.

```
public override void FunctionA()
{
    print("Im a new version of function A");
}
```

Therefore, when the `Start ()` function calls on the `FirstFunction()`, the resident version is called resulting in "new version of `FirstFunction()`" printed in the Console panel in Unity 3D when the game is run. Normally, overriding functions come into play when we have more than one child class that needs to use the same function.

6.13.2 Class Inheritance

We've seen a bit about how a function can inherit some attributes from a previous version of a class. To see how this works, create a couple of nested classes in a new C# file named `Monsters`.

```
using UnityEngine;
using System.Collections;
public class Monsters : MonoBehaviour
{
    //Use this for initialization
    void Start ()
    {
    }
    //Update is called once per frame
    void Update ()
    {
    }
    class Monster
    {
        public int HitPoints;
    }
    class Zombie : Monster
    {
        public int BrainsEaten = 0;
    }
    class Vampire : Monster
    {
        public int BloodSucked = 0;
    }
}
```

In the public class `Monsters`, we've added at the bottom a class `Monster` and a class `Zombie` as well as a class `Vampire`. Both `Zombie` and `Vampire` are followed with `: Monster` to indicate that they are both inheriting functions and fields from `Monster`. Note all this is happening within one C# file called `Monsters` (plural).

6.13.2.1 Sharing Common Attributes

The whole point of inheritance is the ability to create a general class of object from which child objects can inherit the same properties. If all of the characters in a game are going to share a system for taking damage, then they should all use the same system. Games use “mechanics” as a generalized term for game rules. Often used like “this game’s mechanics are fun.” Recovering health, taking damage, and interacting with objects should all be added to what is called a *base class*.

Consider building base classes for things such as destructible objects and monsters. Base classes allow for the multitude of objects in your game to have hitpoints and share the ability of the player to interact with them. Breaking down a door and basting away a zombie shouldn't require a different set of code to accomplish the same thing.

By adding `public int HitPoints` to `Monster`, we've given both `Zombie` and `Vampire` a `HitPoints` variable. This is a main strength of why we need to use inheritance. Objects that share common attributes should be able to share that data. We need to make this `public` so that when the class is used, you can access the `HitPoints` variable from outside of the class. To demonstrate, we'll need to make some instances of both the zombie and the vampire.

```
//Use this for initialization
void Start () {
    Zombie Z = new Zombie();
    Vampire V = new Vampire();
    Z.HitPoints = 10;
    V.HitPoints = 10;
}
```

Now Vampires and Zombies have HitPoints. In the Start () function, we'll create a new Zombie named Z and a new Vampire named V. Then we'll give them some starting hitpoints by using the identifier and adding `a.HitPoints = 10;` to make the assignment. However, if we add in a constructor, we can put the `HitPoints = 10;` into the constructor for the monster instead! While we're at it, extend the Monster by adding a capsule to its presence. We'll also make him announce himself when he's created.

```
class Monster
{
    public int HitPoints;
    public GameObject gameObject;
    //class constructor
    public Monster()
    {
        HitPoints = 10;
        gameObject =
            GameObject.CreatePrimitive(PrimitiveType.Capsule);
        Debug.Log("A new monster rises!");
    }
}
```

To make use of the HitPoints variable, we will create a function that will deal with damage done to the monster. Adding in a new public function that returns an int called TakeDamage will be a good start.

```
class Monster
{
    public int HitPoints;
    public GameObject gameObject;
    public Monster()
    {
        HitPoints = 10;
        gameObject =
            GameObject.CreatePrimitive(PrimitiveType.Capsule);
        Debug.Log("A new monster rises!");
    }
    public virtual int TakeDamage(int damage)
    {
        return HitPoints - damage;
    }
}
```

In the TakeDamage() argument list, we'll add in (int damage) that will return the HitPoints - damage to let us know how many HitPoints the monster has after taking damage. Now both the Zombie and the Vampire can take damage. To allow the Zombie and Vampire to reuse the function and override its behavior, we add in virtual after the public keyword.

```

class Vampire : Monster
{
    public int BloodSucked = 0;
    public override int TakeDamage(int damage)
    {
        return HitPoints - (damage/2);
    }
}

```

Because vampires are usually more durable than zombies, we'll make the vampire take half the damage that is dealt to him. Use `public override` to tell the function to take over the original use of the `Monster.TakeDamage()` function. This allows us to reuse the same function call on both zombies and vampires. For the `Zombie`, we'll return the default result of the `Monster.TakeDamage()` function by using `return base.TakeDamage(damage);`. The keyword `base` allows us to refer the original implementation of the code.

```

void Start ()
{
    Zombie Z = new Zombie();
    Vampire V = new Vampire();
    Debug.Log (Z.TakeDamage(5));
    Debug.Log (V.TakeDamage(5));
}

```

In the preceding `Start ()` function of `Monsters.cs`, we'll make the following uses of `Z` and `V`. The results of our code in `Start ()` should look like the following:

```

5
UnityEngine.Debug:Log(Object)
Monsters:Start () (at Assets/Monsters.cs:10)
8
UnityEngine.Debug:Log(Object)
Monsters:Start () (at Assets/Monsters.cs:11)

```

This looks useful, though remember that we're dealing with `int` values and not `float`. When we divide `int 5` by `int 2`, we get 3, not 2.5. When building new objects, the goal is to reuse as much code as possible. Likewise, we can leave the `TakeDamage()` out of the zombie altogether if we don't need to make any changes.

```

class Zombie : Monster
{
    public int BrainsEaten = 0;
}
class Vampire : Monster
{
    public int BloodSucked = 0;
    public override int TakeDamage(int damage)
    {
        return HitPoints - (damage/2);
    }
}

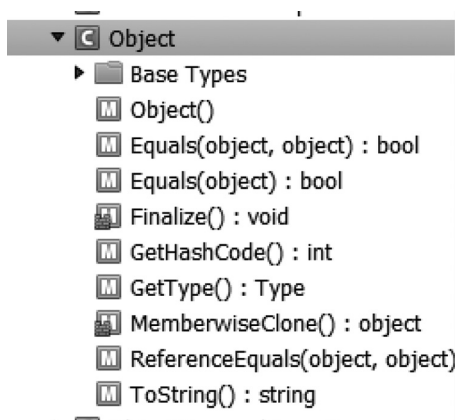
```

However, it's sometimes unclear what's going on and might end up being less obvious when damage is dealt to the two monsters. In the end though, it's up to you to choose how to set up any functions that are inherited from your base monster class.

This should serve as a fair example as to what objects and classes are created in the way they are. When functions and variables are separated in classes, they allow you to compartmentalize for very specific purposes. When you design your classes, it's important to keep object-oriented concepts in mind.

6.13.3 Object

Changing behaviors of previously implemented functions is a powerful tool in object oriented programming (OOP) languages. The base class from which all classes in Unity 3D inherit is object. The object class is the base class to which unity can reference. For Unity 3D to be able to interact with a class, it must be based on object first. The .Net Framework provides object as a foundation for any other class being created. This allows every new object a common ground to communicate between one another.



Referring to MonoDevelop and expanding References → UnityEngine.dll, you can find the object class. Inside that you'll find several functions and some variables. If we were to override ToString(), we could add some useful parameters to our monsters to provide a more customized return value.

```
class Zombie : Monster
{
    public int BrainsEaten = 0;
    public Zombie()
    {
        Debug.Log("zombie constructor");
        gameObject.transform.position = new Vector3(1, 0, 0);
    }
    public override string ToString()
    {
        return string.Format("[Zombie]");
    }
}
```

As soon as we enter `public override`, MonoDevelop will pop up a list of functions that are available to override. Selecting `ToString()` automatically fills in the rest of the function. If we add `Debug.Log(Z.ToString());` to the `Start ()` function, we'll get the following printed to the Unity's Console panel:

```
[Zombie]
UnityEngine.Debug:Log(Object)
Monsters:Start () (at Assets/Monsters.cs:12)
```

The printout `[Zombie]` might not be as useful as something more detailed. This does inform us what we're working with but not much more. Then again, if we print out what the vampire to string is, we're not going to get anything more useful since we have yet to provide any specific `ToString()` functionality.

```

Monsters+Vampire
UnityEngine.Debug:Log(Object)
Monsters:Start () (at Assets/Monsters.cs:13)

```

Therefore, it's often useful to find functions that are already in use and update them to fit our current class. The object class was created with `ToString()`, which provides every object in C# to have some sort of `ToString()` behavior. This also means that we can override `ToString()` with pretty much any class we create.

6.13.4 What We've Learned

At this point, we've got a great deal of the basics down. We can write classes and we know how the basics of inheritance work. This is important, as we proceed to use inheriting properties and methods more and more. We've studied a bit of flow control systems using `if` and `switch`, which are indispensable statements.

We've created several classes already, but we've yet to really use classes as objects. In Chapters 7 and 8, we're going to learn the core of what OOP is all about.

Often when you start writing classes for a game, you can get started faster by creating a class for every object. This is sometimes required if you're working with a group of people. Everyone begins by creating classes for his or her own set of objects. Only after a few different classes have been written will it become clear that a base class might exist.

```

//written by programmer A
class Zombie
{
    int afterLife = 10;
    int brainsEaten = 0;
    float stumbleSpeed = 2.1f;
    public void TakeDamage(int damage, bool isFire)
    {
        if(!isFire)
        {
            afterLife -= damage;
        } else {
            afterLife -= damage * 2;
        }
    }
}

//written by programmer B
class TreasureChest
{
    bool Broken = false;
    int damageToOpen = 5;
    int goldCoins = 100;
    public bool BreakChest(int smash)
    {
        if(smash > damageToOpen)
        {
            Broken = true;
        } else {
            Broken = false;
        }
        Return Broken;
    }
}

```

If we look at the above two classes, each written by a different programmer, we might notice that one has a `return bool` if the damage has reached a threshold. The `BreakChest()` function looks for a number greater than its `damageToOpen` value. If it is, then it returns `true`, otherwise the chest remains

not Broken. The `Zombie` class, on the other hand, has an `afterLife` value that is decremented every time the `TakeDamage()` function is called.

Both work in similar ways, but the `Zombie` decrements the `afterLife` pool, while the `TreasureChest` only requires enough damage to be done in one hit to break. When the two programmers come together to collaborate, they might want to agree on some similar behaviors, and then agree on more generic names for the variables to build a base class.

6.14 Type Casting Again

When programmers use the word *integral type*, they mean things that can be numbered. For instance, a boolean can be converted from true and false to 1 and 0; note that this can be done by simply using `int i = 0;` and `bool t = i;` but it's close. Try the following code in the Integrals project:

```
using UnityEngine;
using System.Collections;
public class CastingAgain : MonoBehaviour
{
    enum simpleEnum
    {
        a,
        b,
        c
    }
    //Use this for initialization
    void Start ()
    {
        simpleEnum MySimpleEnum = simpleEnum.b;
        int MyInt = MySimpleEnum as int;
        Debug.Log(MyInt);
    }
}
```

Create an enum type with `enum simpleEnum{a,b,c};` and then create a variable for the `simpleEnum` called `MySimpleEnum` to store the `simpleEnum.b` value. In this case, `simpleEnum.b;` is the second value of the enumerator.

As we try to cast `MySimpleEnum as int;` to `MyInt`, we get an error!

"Assets/CastingAgain.cs(13,42): error CS0077: The 'as' operator cannot be used with a non-nullable value type 'int'"

The `as` operator can't be used for this because an `int` is not actually an enum. Therefore, we get a conflict here. We're trying to use the enum as though it were already an `int`, which it's not. There is a way to convert different types from one to another.

```
void Start ()
{
    simpleEnum MySimpleEnum = simpleEnum.b;
    int MyInt = (int)MySimpleEnum;
    Debug.Log(MyInt);
}
```

The line `int MyInt = (int)MySimpleEnum;` is using the explicit cast `(int)` to convert the `MySimpleEnum` into the `int` value. This isn't always possible, but in some cases, this will work when an explicit cast has been defined.

Here we get "1" in the Console when we run the game. Remember that numbering lists in C# starts at 0, so the second item in a list will be indexed at 1. Of course, not all explicit conversions work. For instance, you can't turn a `Monster GameObject` into a `float` value. Zombies aren't just numbers!

6.14.1 (<Type>) versus "as"

Here we've started to see a bit of a difference in type casting methods. This is the difference between (<Type>) and `as` problems that can actually come up during an interview process if you're trying to get a job as a programmer.

We should also note that some casting is also going on when we use `float f = 1;` where `1` is an `int` value, but it's accepted into a `float f` without any question. This is an implicit cast, or rather this is a cast that is accepted without any explicit cast operation. This also works for a `double d = 1.0f;` where `1.0f` is a `float` value and `d` is a `double`.

The two different methods we can use are called *prefix-casting* and *as-casting*. Prefix casting is what the (`int`) is called; this is also referred to as an explicit cast operator. The `as` operator works a bit differently than the explicit cast operator.

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

Add in three classes: `Humanoid`, `Zombie`, and `Person`. Make the `Zombie` and `Person` inherit from the `Humanoid` class, and we'll be able to get started with our tutorial. Discovering the differences with `as` and (<Type>) is significant only once we start getting into the nitty-gritty of doing something with the results of the type casting.

```
//Use this for initialization
void Start ()
{
    Humanoid h = new Humanoid();
    Zombie z = h as Zombie;
    Debug.Log(z);
}
```

In our `Start ()` function, we'll create a new `Humanoid h`, which will instantiate a new `Humanoid` type object identified by `h`. Then we'll create another object called `z` and assign `h as Zombie` to `z`. This assumes that the `Humanoid` is a `Zombie` type object. When we use `Debug.Log(z)`, we get `Null`.

```
Null
UnityEngine.Debug:Log(Object)
TypeCasting:Start () (at Assets/TypeCasting.cs:16)
```

Compare this result to the following explicit cast operator. If we use the prefix system as in the following example code,

```
void Start () {
    Humanoid h = new Humanoid();
    Zombie x = (Zombie) h;
    Debug.Log(x);
}
```

we get a different result, this time an error:

```
InvalidCastException: Cannot cast from source type to destination type.
TypeCasting.Start () (at Assets/TypeCasting.cs:15)
```

This tells us that a type `Humanoid` cannot be converted to a type `Zombie`. This doesn't change when converting between a `Person` and a `Zombie`. To create an explicit cast, we need to implement a function to allow the cast to work.

```
void Start () {
    Person p = new Person();
    Zombie z = p as Zombie;
    Debug.Log(z);
}
```

As before, we get a similar error.

```
Assets/TypeCasting.cs(23,30): error CS0039: Cannot convert type 'TypeCasting.
Person' to 'TypeCasting.Zombie' via a built-in conversion
```

We need to add some lines of code to allow us to do this conversion between `Zombies` and `People`, referring to `Zombies` as the `zombie monster` and `People` as in a `human person`.

You can imagine that this would happen quite a lot when playing a zombie game. C# is trying to make changes to the data to conform it so that it matches the type we're asking for it to be converted to. In some cases, the `as` operator is more appropriate than the prefix operator. A `Null` is much easier to deal with than an error. This conversion makes sense only when the types are incompatible if there is no conversion possible.

We could do this between `ints` and `floats`, no problem aside from losing numbers after a decimal. There are conversions available through C#, but none has been made to convert a `Humanoid` to a `Zombie`. There should be some method made available to do so. The conversion work has been written for the built-in data types. The work has not yet been done to convert between a `Person` and a `Zombie`, so we shall do this.

6.14.2 User-Defined Type Conversion

First, let's modify the `Humanoid`, which has some `hitpoints`.

```
class Humanoid
{
    public int hitpoints;
}
```

This way we can have some significant meaning for converting between a `Person` and a `Zombie`. We'll assume zombies use negative `hitpoints` and a person has a positive number for `hitpoints`. Thanks to inheritance, we can safely assume that both the `Person` and the `Zombie` will have a `hitpoints` property when they derive from the `Humanoid` class.

Now we'll create a new function in the `Person`, so we can convert him into a `Zombie`.

```
class Person : Humanoid
{
    static public implicit operator Zombie(Person p)
```



```

    {
        Zombie z = new Zombie();
        z.hitpoints = p.hitpoints * -1;
        return z;
    }
}

```

With this we can see that we'll need to add a few new keywords. The keywords `implicit` and `operator` work together to allow us to use the `Zombie` cast when working with a `Person` type object. Now we can check what the `hitpoints` of a `Person` is if it was to be treated as a `Zombie` type object.

```

void Start ()
{
    Person p = new Person();
    p.hitpoints = 10;
    Zombie z = p as Zombie;
    Debug.Log(z.hitpoints);
}

```

Through the `implicit` keyword, we can quite nicely assign the `Zombie z` to the `Person` type. We're also allowed to do something very simple like the following code sample:

```

void Start ()
{
    Zombie z = new Person();
    Debug.Log(z);
}

```

This automatically assigns `z` a new `Person()` that is then implicitly converted to a `Zombie` on the assignment. Likewise, we can use a prefix conversion and get the same result by using an `as` operator.

```

void Start ()
{
    Person p = new Person();
    Zombie z = (Zombie)p;
    Debug.Log(z);
}

```

This seems like the most useful conversion; however, there's a simple problem. What if we expect only a very specific return type from the conversion?

6.14.3 Implicit versus Explicit Type Conversion

Explicit casts are used if there is a conversion that might end with some loss of data. Implicit casts assume that no data is lost while converting from one type to another. This can be seen when casting an `int` to a `float`. The `int` value 1 is the same as a `float` value 1.0, so an implicit cast infers that nothing is lost. In some edge cases, there can be data lost, so implicit casts are not perfect, though they are generally reliable.

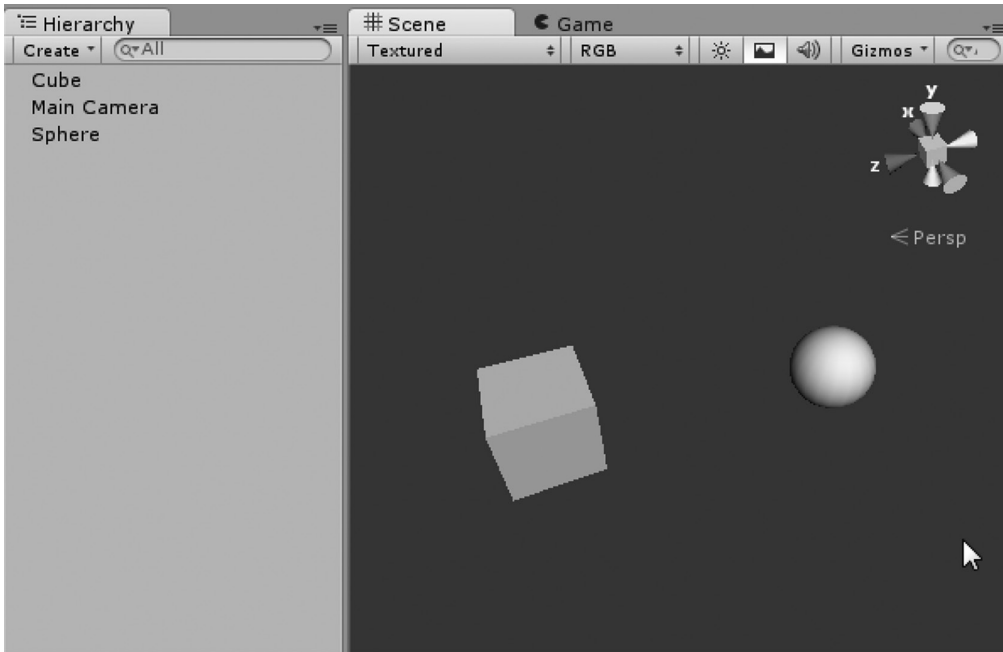
6.14.4 Break

If `break` is not present, then the evaluation continues through the rest of the cases. This might be a bit confusing, but basically the `switch` jumps to the line where the corresponding case appears, and `break` stops the code evaluation and ends the `switch` statement. There are other uses for `break`, but we don't need to go into them right now.

Copy the `Monster.cs` from the `enums` project into the `Assets` directory for the current `Integrals` project. For using an enum in the `switch`, we need to use the full enum value for the case value.

This means `MonsterState.standing` should be used to indicate the enum we are looking for. The `mState` stores the `MonsterState`'s enum values that are being used.

To determine the state our monster should be in, we should collect some data in the scene. We'll need to do a bit of setup with a game scene to collect data from.



I've added a cube and a sphere to the scene. Then I added the `Player.cs` component to the cube and the `MonsterGenerator.cs` to the sphere. To save some time, you will be able to grab the Unity 3D scene from the website for the book. It's still a good practice for you to do this work on your own.

My `MonsterGenerator.cs` code looks a bit like the following:

```
using UnityEngine;
using System.Collections;
public class MonsterGenerator : MonoBehaviour
{
    public int numMonsters;
    //Use this for initialization
    void Start ()
    {
        for (int i = 0; i < numMonsters; i++)
        {
            GameObject sphere =
                GameObject.CreatePrimitive(PrimitiveType.Sphere);
            sphere.AddComponent("Monster");
        }
    }
    //Update is called once per frame
    void Update ()
    {
    }
}
```

This code is similar to before where we created 10 objects and attached the `Monster.cs` component to a primitive. Running the game will produce 10 spheres, with `Monster.cs` attached to them.

6.14.5 What We've Learned

Using the `explicit` and `implicit` type casts requires a bit of thought. When dealing with a large number of different types, we need to keep in mind what we're trying to accomplish by casting between types. In many cases, we should avoid unnecessary cast operations.

If data needs to be shared between different classes, it's better to create a parent class from which both classes inherit. Rather than requiring specific casts between people and zombies, a new version should be instantiated between the two without using a cast.

The zombie-person cast is not the best use case, though it does illustrate how a cast is performed. Most casts should only be between specific data types. It should also be noted that casting is allowed between structs.

```
struct a
{
    static public explicit operator b(a A)
    {
        return new b();
    }
}
struct b
{
    static public explicit operator a(b B)
    {
        return new a();
    }
}
```

Here we can make an `explicit` cast from `a` to `b` and back again. Though there's nothing really going on here, a struct is somewhat different from a class, but offers many of the same functionality. Defining `explicit` cast operators is one of the shared abilities between a class and a struct.

6.15 Working with Vectors

Vector math is something that was taught in some high school math classes. Don't worry; you're not expected to remember any of that. Vector math is easily calculated in C# using the Unity 3D libraries, as vector math is often used in video games. Adding, subtracting, and multiplying vectors are pretty simple in Unity 3D. We'll take a look at how simple vector math is in Unity 3D later on in this section.

There are also many tools added into the `MonoBehaviour` class we've been inheriting for most of the classes we've been writing. Only `Start()` and `Update()` have been added to our prebuilt class we've been starting with, but there are many others which we haven't seen yet.

6.15.1 Vectors Are Objects

Let's add behaviors to the `MonsterGenerator.cs` we were working with from Chapter 5.

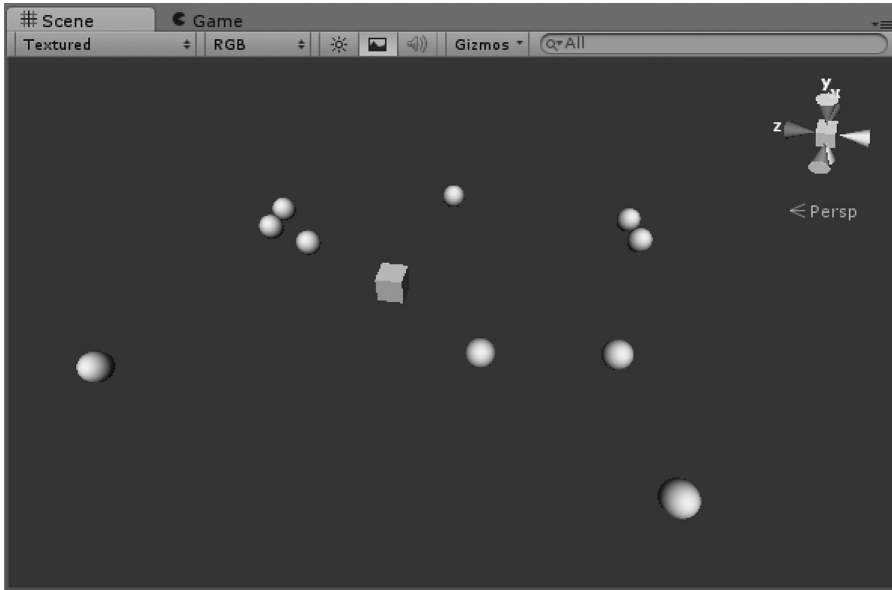
```
public int numMonsters;
//Use this for initialization
void Start ()
{
    for (int i = 0; i < numMonsters; i++)
    {
        GameObject sphere =
            GameObject.CreatePrimitive(PrimitiveType.Sphere);
        sphere.AddComponent("Monster");
        Vector3 pos = new Vector3();
        pos.x = Random.Range(-10, 10);
```

```

        pos.z = Random.Range(-10, 10); //not y
        sphere.transform.position = pos;
    }
}

```

`Random.Range();` is a new function. `Random` is an object that's located in the `MonoBehaviour` parent class. We use this to give us a number between -10 and 10. The two arguments are separated by a comma. The first argument is the minimum number we want and the second number is the maximum value we would want. Then `Random` sets the `x` and `z` to a number between these two values.



Running the `MonsterGenerator` creates a field of sphere monsters placed randomly in the `x` and `z` directions. As an alternative, we could have put the positioning information inside of the monster himself using the “`Start()`” function.

6.15.2 Stepping through `MonsterGenerator`

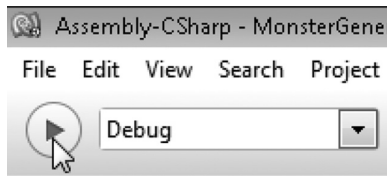
We introduced breakpoints in Section 4.12.9, but this is a good point to come back to how breakpoints are used to see how your code is working. Why does each monster appear at a different place? Let's add in a breakpoint near the first line within the “`for`” statement.

```

8      void Start()
9      {
10         for (int i = 0; i < numMonsters; i++)
11         {
12             GameObject sphere =
13             GameObject.CreatePrimitive(PrimitiveType.Sphere);
14             sphere.AddComponent("Monster");
15             Vector3 pos = new Vector3();
16             pos.x = Random.Range(-10, 10);
17             pos.z = Random.Range(-10, 10);
18             sphere.transform.position = pos;
19         }
20     }

```

Here, I have the breakpoint added on line where the `Monster` component is added. Press the Run button and attach MonoDevelop to the Unity 3D thread.



In MonoDevelop, select `Run → Start Debugging` and then select `Unity 3D` in the following pop-up panel. Go back to the Unity 3D editor and press the Play button. You may notice not much happening in Unity 3D, which means that MonoDevelop grabbed Unity 3D's thread and is holding it at the breakpoint.

Start stepping through the code. Pressing the `F10` key on the keyboard will push the arrow line at a time through the “for” statement. After pressing it once, you'll notice that the highlighted line will begin to move over your code. Also notice the data populating the Callstack and the Locals windows that open below the code.

Click on the word “sphere.” Each instance of the word will highlight and a new dialog box will open up with specifics on the new sphere object. Hover the cursor over “sphere” to invoke a pop-up with information on the sphere object.

```
for (int i = 0; i < numMonsters; i++)
{
    GameObject sphere =
        GameObject.CreatePrimitive(PrimitiveType.Sphere);
    sphere.AddComponent("Monster");
    Vector3 pos = new Vector3();
    pos.x = Random.Range(-10, 10);
    pos.z = Random.Range(-10, 10);
    sphere.transform.position = pos;
}
```

Step again and we'll get details on the new `Vector3` called `pos`, which I'm using as an abbreviation of “position.” Press `F10` to *step over* once to move to the next line.

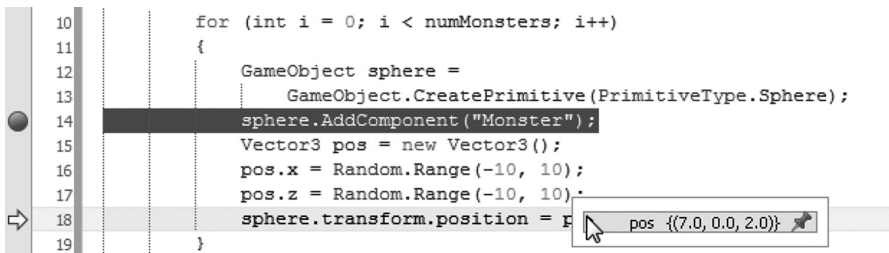
```
12 |         GameObject sphere =
13 |             GameObject.CreatePrimitive(PrimitiveType.Sphere);
14 |         sphere.AddComponent("Monster");
15 |         Vector3 pos = new Vector3();
16 |         pos.x = Random.Range(-10, 10);
17 |         pos.z = Random.Range(-10, 10);
18 |         sphere.transform.position = pos;
19 |     }
```

It's important to understand that data isn't fulfilled on the line it's created on. Only after stepping one more line with `F10` does the data become fulfilled.

```
12 |         GameObject sphere =
13 |             GameObject.CreatePrimitive(PrimitiveType.Sphere);
14 |         sphere.AddComponent("Monster");
15 |         Vector3 pos = new Vector3();
16 |         pos.x = Random.Range(-10, 10);
17 |         pos.z = Random.Range(-10, 10);
18 |         sphere.transform.position = pos;
19 |     }
```

In the following line, you'll read `pos.x = Random.Range(-10f, 10f);` which means that the `x` value in the `Vector3` named `pos` will be set to a random number between `-10` and `10`. If we hover `pos` in the line where the `x` is being set, the first value is still `0.0`; this is to change once we press `F11` again to move on.

Now that we're another line down, you'll see that the first value of `pos` is now an interesting number, in this case `-4.4`. The `x` value is now readable after it has been set to the `Random.Range` created the line before. Pressing `F10` again, we'll be able to see the `sphere.transform.position` being set to `pos`, which we should check.



And I get a value for `x` and `z` as `7.0` and `2.0`, respectively; you'll get a different value for each since we're using a random number. Keep on pressing `F10` and you'll be able to follow the flow of the code as each line highlights again. Each time the line is executed, the values coming from the `Random.Range()` function will be different. This is why the code will scatter the monsters around the `MonsterGenerator`.

As an alternative, we can move the code into the `Monster` itself. If the code was taken out of the `MonsterGenerator` and added into the `Start()` function of `Monster`, we can remove the "sphere." from the "transform.position" and set the position of the monster directly when it's created.

```
sphere.AddComponent("Monster");
Vector3 pos = new Vector3();
pos.x = Random.Range(-10, 10);
pos.z = Random.Range(-10, 10);
sphere.transform.position = pos;
```

When the object was created in the `MonsterGenerator`, the `transform.position` was a part of a different object. Therefore, we had to tell the object the `transform.position` we wanted to set, and not the position of the `MonsterGenerator`.

The option of where to set the initial position of an object is up to the programmer. It's up to you to decide who and where the position of a new object is set. This flexibility allows you to decide how and when code is executed.

6.15.3 Gizmos

While stepping through code helps a great deal with understanding how our code is being executed, visualizing things as they move can be more helpful. That's where Unity 3D has provided us with a great deal of tools for doing just that.

Unity 3D calls its helper objects "Gizmos." Visualizing data is carried out by using `Gizmos` to draw lines and shapes in the scene. In other game engines, this may be called `DrawDebugLine` or `DrawDebugBox`. In Unity 3D, the engineers added the `Gizmos` class. The members of this class include `DrawLine` and `DrawBox`.

Colored lines, boxes, spheres, and other things can be constructed from the code. If we want to draw a line from the monster to a target, we might use a line between the two so we know that the monster can "see" its next meal. It's difficult to mentally visualize what the vectors are doing. Having Unity 3D display them in the scene makes the vector math easier to understand.

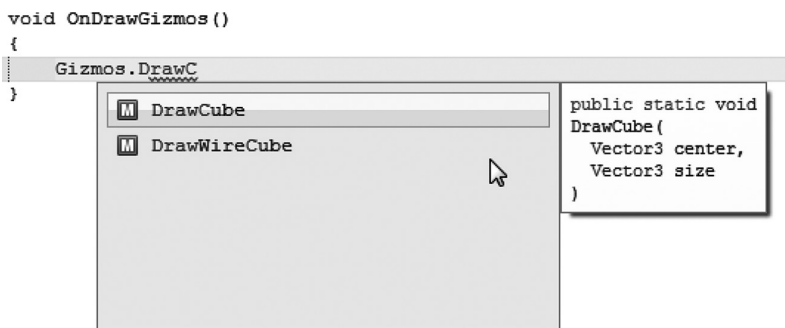
6.15.3.1 A Basic Example

The first example in the `Gizmos.cs` in the Integrals project is the `DrawLine()`; function that requires two arguments or parameters. The first argument is a start vector followed by an end vector. Start in a fresh `MyGizmos.cs` class and add in a function called `OnDrawGizmos()` after the `Start ()` and `Update ()` functions. The `OnDrawGizmos` can be placed before or after the `Start ()` or `Update ()` function, but to keep organized we'll add it after the preconstructed functions.

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

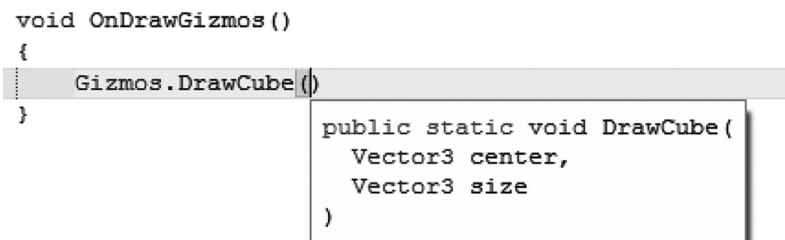
This is what your `MyGizmos.cs` should look like. To see what's going on, create a new Sphere GameObject Object using the GameObject → Create Other → Sphere menu, and then add the `Gizmos.cs` component in the Inspector panel.

To draw a Gizmo, we'll need to pick the Gizmo we want to use.



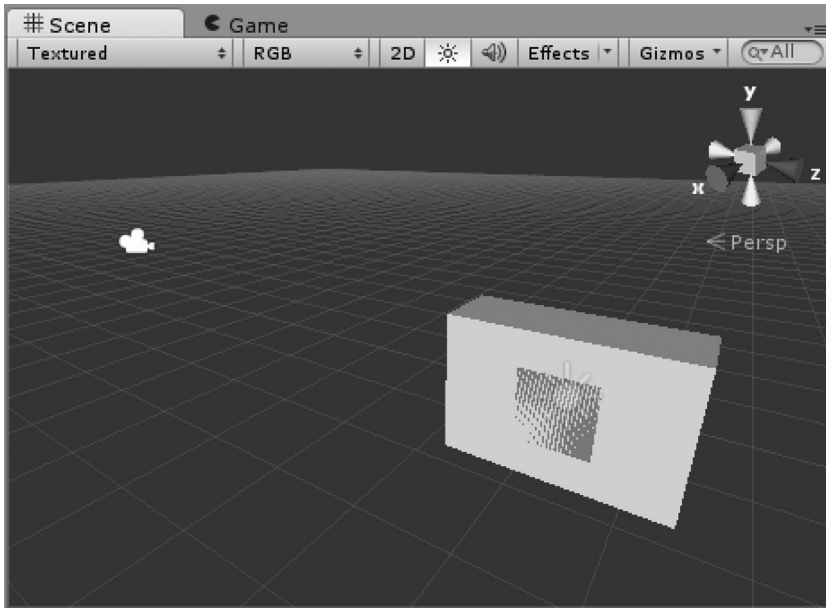
When we start with “Gizmos.” we are presented with quite a list of things we can try out. Thanks to MonoDevelop, it's quite easy to figure out how these are used. Each function has some pretty clear clues as to what data types are used and what they are for.

6.15.3.1.1 DrawCube



Starting with “DrawCube(” we’ll be prompted with a `Vector3` for the cube’s center and another `Vector3` for the cube’s size. To fulfill the `DrawCube` member function in `Gizmos`, enter the following arguments:

```
void OnDrawGizmos()
{
    Gizmos.DrawCube(new Vector3(0, 0, 0), new Vector3(1, 2, 3));
}
```



The Gizmo will be drawn with the center point at `Vector3(0,0,0)`, which is the scene’s origin, and the size will be `Vector3(1, 2, 3)`, making the object 1 unit wide, 2 units tall, and 3 units deep. Gizmos are pretty interesting as they represent some very useful debugging tools.

6.15.3.1.2 DrawRay

Some of the other Gizmo functions require data types we haven’t encountered yet, such as rectangles and rays. With a little bit of detection, we can figure out how these work. Let’s start off with a `DrawRay` we’re presented with (`Ray r`) as its argument. We can start with a new `Ray` (and we’ll be given a (`Vector3 origin`, `Vector3 direction`)).

We can fulfill this with `Gizmos.DrawRay(new Ray(new Vector3(0,0,0), new Vector3(0,1,0)))`. This might seem like a bit of a convoluted line to read, so we can make this more easy to read by breaking out one of the arguments.

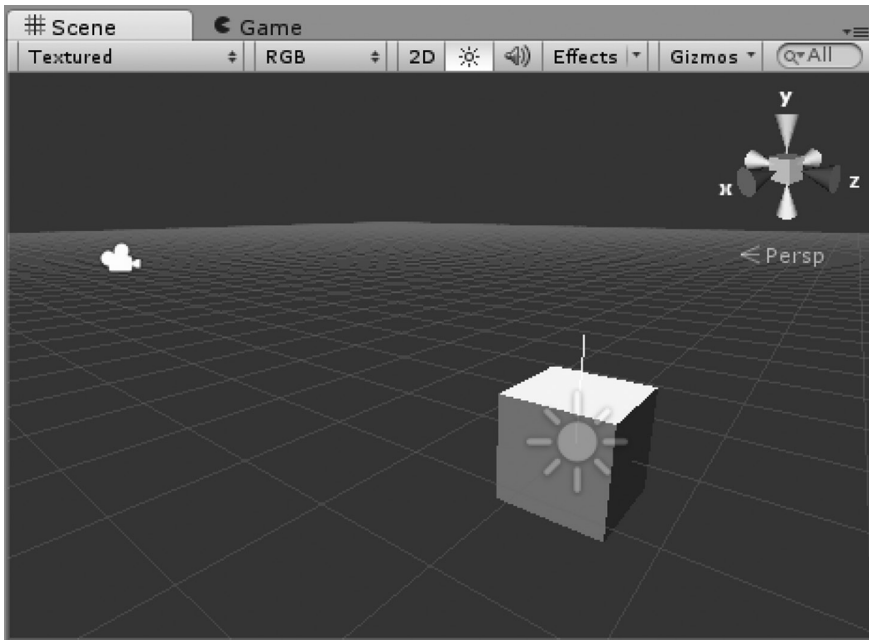
```
void OnDrawGizmos() {
    Gizmos.DrawRay(new Ray(

```

Ray (Vector3 origin, Vector3 direction)

```
void OnDrawGizmos()
{
    Ray r = new Ray();
    r.origin = new Vector3(0, 0, 0);
    r.direction = new Vector3(0, 1, 0);
    Gizmos.DrawRay(r);
}
```


This code will produce a simple white line that starts at the origin and points up in the y-axis 1 unit. Rays are useful tools that show an object's direction or angle of movement. We'll take a closer look once we get more used to how Gizmos are used. Assuming the object with the script applied is a cube at the origin of the scene, you'll have a similar effect to the following figure:

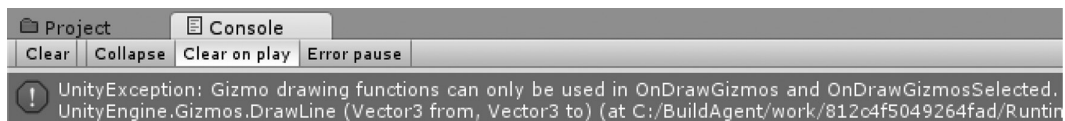


6.15.4 Using Gizmos

How to use Gizmos may not be obvious. Let's start with the following code located in the `Update ()` function of the `Monster.cs` to draw a line from where the monster is to the origin of the world, or `Vector3(0,0,0);`.

```
//Update is called once per frame
void Update ()
{
    Gizmos.DrawLine(transform.position, new Vector3(0, 0, 0));
}
```

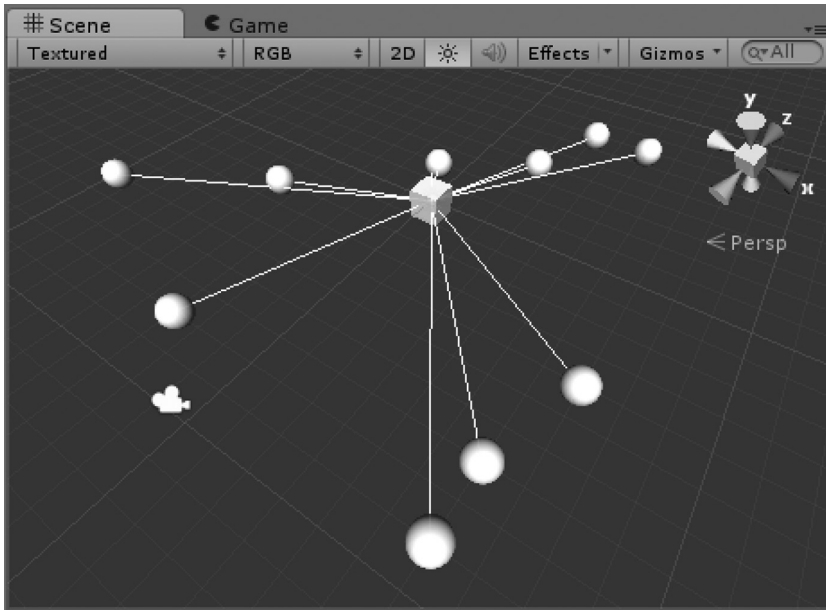
Unfortunately, this produces the following error!



This error does tell us what is wrong. It's expecting the Gizmos drawing function to be used in an `OnDrawGizmos` or `OnDrawGizmosSelected` function. Therefore, let's move that line of code into that function. The `OnDrawGizmos` function can be placed anywhere in the class scope.

```
void OnDrawGizmos()
{
    Gizmos.DrawLine(transform.position, new Vector3(0, 0, 0));
}
```

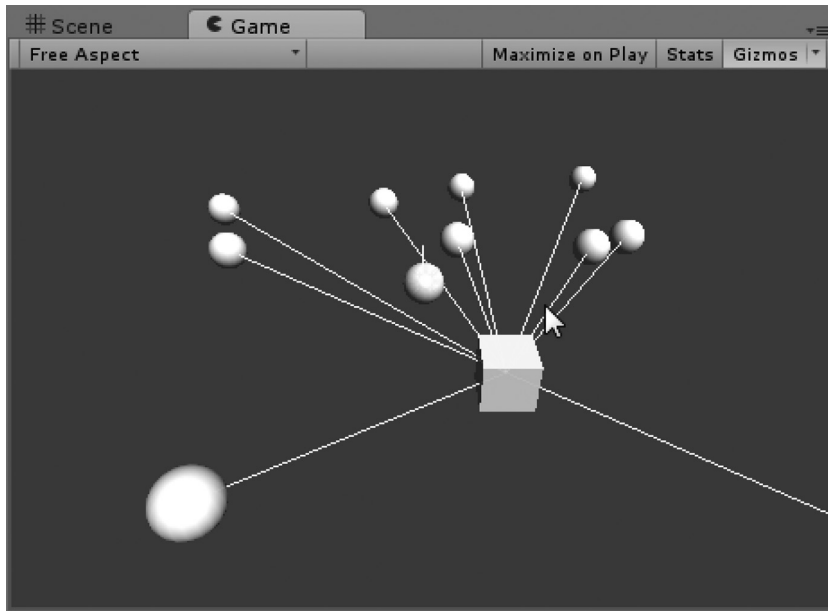
Remember how a child can inherit a function from its parent. There's more to inheritance than simply inheriting functions. We can add to and overwrite the functions given to a class from its parent. This draws a line from `transform.position` to `Vector3(0,0,0)`, which is the origin of the scene.



Now we're drawing a bunch of white lines from the center of the monster, or the monster's `transform.position`, to the center of the world or `Vector3(0,0,0)`. If you still have this code around for finding the player, we can make use of the player `GameObject`.

```
public GameObject PlayerObject;
void Start ()
{
    int number = (int)mState;
    Debug.Log(number);
    Vector3 pos = new Vector3();
    pos.x = Random.Range(-10f, 10f);
    pos.z = Random.Range(-10f, 10f);
    transform.position = pos;
    GameObject[] AllGameObjects = GameObject.FindObjectsOfType(typeof(
    GameObject)) as GameObject[];
    foreach (GameObject aGameObject in AllGameObjects)
    {
        Component aComponent = aGameObject.GetComponent(typeof(Player));
        if (aComponent != null)
        {
            PlayerObject = aGameObject;
        }
    }
}
```

`PlayerObject` contains a `transform.position` we can use. As long as an object has a `Player.cs` assigned to it, the `Monster` script will be able to find it. To view the Gizmos in the Game panel, you can turn them on by clicking on the `Gizmos` button located to the top right of the Game panel.



Like before, using the WASD keys will move the cube around in the world and the monsters will be able to draw a line to you. Now the monsters know where you are, but now they have to move toward you. To do this, we need to add in a `Vector3` `Direction` to tell the monster which way to go to get to the player.

A `Vector3` `Direction` needs to be declared at the class level of scope. Remember that it's a variable that's visible to all other functions in the class. This means we can use it again in the `OnDrawGizmos()` function. The easiest method to calculate `Direction` is to use `Vector3.Normalize`. This method found in `Vector3` will take a vector and reduce the values to something much easier to deal with.

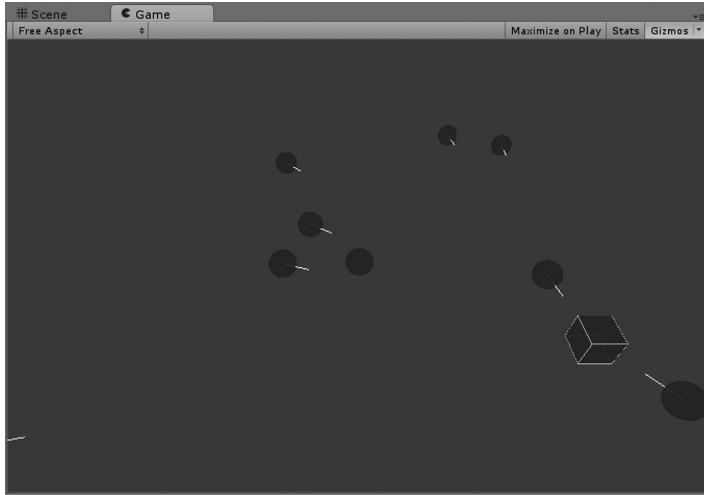
```
Vector3 Direction;//class scoped variable
void Update ()
{
    Direction = Vector3.Normalize(PlayerObject.transform.position -
    transform.position);
}
```

`Direction` needs to live at the class scope. Variables that are scoped to the entire class can be used by any function in the class. Therefore, when `Direction` is updated in the `Update ()` function, it will be available to the `OnDrawGizmos()` function. Now that we know what direction the vector is going in, we need to use it in the `OnDrawGizmos()` function. To make this clear, we'll consider this in terms of pseudocode.

Using `OnDrawGizmos()` draw a line in the direction to move toward the center of the object. The center of the object is `transform.position`, and the direction to move toward is the normalized `Vector3` we called `Direction`. In C#, this translates into the following code:

```
void OnDrawGizmos()
{
    Gizmos.DrawLine(transform.position, transform.position + Direction);
}
```

The first argument is the current `transform.position` of the monster. This is the starting point of the line. Then we need to set the end point for the line, which starts at the same place, but then it is drawn with the addition of the direction.



The end result is a bunch of small lines drawn from the center of each monster toward the player. By adding the `Direction` to the `Update ()` function, the line will be recalculated. Now we have a good starting point for adding a `Vector3` in the direction of the player.

The `Direction` we're using ranges from 0 to 1, and traveling a full unit per update is quite a lot. This means we may move over a hundred units per frame. Consider that we may get over a hundred updates per second. Therefore, we need to shrink the `Direction Vector3` to a smaller value. A `Vector3` can be multiplied by a single float value that multiplies the x, y, and z equally.

```
Vector3 Direction;//class scoped variable
void Update ()
{
    Direction =
    Vector3.Normalize(
        PlayerObject.transform.position - transform.position
    );
    transform.position += Direction * 0.1f;
}
```

In this example, we're multiplying the `Direction` by `0.01f` that should slow the monster down so we have a chance of running away.



Run away!!! The little sphere monsters will be chasing after the `GameObject` that has a `Player.cs` component attached. Before we go further, we should put some limits by using a bit of logic. If we get too close to the player, we should stop moving toward him. If we don't stop, then we'll eventually sit inside of the player. To do this, we'll need to know how far away from the player we are.

```
Void Update () {
    Vector3 MyVector = PlayerObject.transform.position - transform.position;
    float DistanceToPlayer = MyVector.magnitude;
```

Start with a new `Vector3` made from subtracting our current position from the position we're headed toward. Then to get the magnitude of that vector, we create a `float` called `DistanceToPlayer` and set that to the `MyVector.magnitude`. If you remember anything from math class, some of this should sound familiar.

Now we have data to make use of with some logic. We'll surround the code that moves the monster with an `if` statement controlled by some arbitrary number. If we are more than 3 units away from the player, then move toward him.

```
Vector3 MyVector =
PlayerObject.transform.position - transform.position;
float DistanceToPlayer = MyVector.magnitude;
if (DistanceToPlayer > 3.0f)
{
    Direction =
    Vector3.Normalize(PlayerObject.transform.position - transform.
    position);
    transform.position += Direction * 0.1f;
}
```

6.15.4.1 Building Up Parameters

Now they should all stop once they get close enough to the player. Based on game design you may want to change this arbitrary distance away from the player, your monster will stop. It's time to start breaking out our hard numbers and making them editable.

```
Vector3 Direction;//class scoped variable
public float AttackRange = 3.0f;
void Update ()
{
    Vector3 MyVector =
    PlayerObject.transform.position - transform.position;
    float DistanceToPlayer = MyVector.magnitude;
    if (DistanceToPlayer > AttackRange)
    {
        Direction =
        Vector3.Normalize(PlayerObject.transform.position -
        transform.position);
        transform.position += Direction * 0.1f;
    }
}
```

Replace the `3f` with a `public float` so it can easily be modified from an outside class. We may change the speed with a `public float` as well. It's always a good practice to change hard numbers into something that can be changed later on by a design change. Naming the variables also gives more

meaning to how the variable is used. You may notice that there aren't any numbers inside of the `Update ()` loop left to parameterize. Now we're thinking like a programmer.

```
Vector3 Direction;//class scoped variable
public float AttackRange = 3.0f;
public float SpeedMultiplier = 0.01f;
void Update ()
{
    if (mState == MonsterState.standing)
    {
        print("standing monster is standing.");
    }
    if (mState == MonsterState.wandering)
    {
        print("wandering monster is wandering.");
    }
    if (mState == MonsterState.chasing)
    {
        print("chasing monster is chasing.");
    }
    if (mState == MonsterState.attacking)
    {
        print("attacking monster is attacking.");
    }
    //Gizmos.DrawLine(transform.position, new Vector3(0, 0, 0));
    Vector3 MyVector =
        PlayerObject.transform.position - transform.position;
    float DistanceToPlayer = MyVector.magnitude;
    if (DistanceToPlayer > AttackRange)
    {
        Direction =
            Vector3.Normalize(PlayerObject.transform.position -
                transform.position);
        transform.position += Direction * SpeedMultiplier;
    }
}
```

6.15.5 Optimizing

Now we have some optimization we could do. The math we do to `MyVector` is somewhat redundant. This can be encapsulated and can more directly set the `DistanceToPlayer`. We can delete the entire line for `MyVector` by moving the code directly into the line that sets the distance value.

```
float DistanceToPlayer = (PlayerObject.transform.position - transform.
position).magnitude;
```

Likewise, we don't even need the `DistanceToPlayer` to be calculated before it's used. This means we can do the calculation inside of the `if` statement's parameters.

```
if (PlayerObject.transform.position - transform.position).magnitude >
AttackRange)
```

Again we can reduce this even more. Calculating the `Direction` and then using its value may also be done in the line of code where it's eventually used.

```
if ((PlayerObject.transform.position - transform.position).magnitude >
AttackRange)
```

```
{
    transform.position +=
        Vector3.Normalize (PlayerObject.transform.position -
            transform.position) * SpeedMultiplier;
}
```

Unfortunately, this means that `Direction` is never calculated and it can't be used for drawing the Gizmo. When we reduce our code this much, we start to run into problems. Usually, it's a good practice to create data before it's used. This shortens the length of each line of code and makes your code easier to read.

The line where you stop cutting down lines of code is up to you. It's difficult to say how much you should smash things down. In many cases, when you reduce code to just a few lines, your code's readability is also reduced.

6.15.6 What We've Learned

Unity 3D has many tools built in for doing vector math functions. It's a good idea to figure out how they're used. We'll explore as many different uses of these tools as we can, but we're going to be hard pressed to see all of them.

A Gizmo is a handy tool to help see what your code is trying to do. Gizmos draw lines, boxes, and other shapes. You can even have text printed into your 3D scene for debugging purposes. Again, we'll try to use as many of the different Gizmos as we can.

The sort of optimization we did here is in the reduction of lines of code only. There was no actual change to the speed at which the code here was executed. There are some cases in which a line or two should be added to speed things up.

For now we're going to focus on keeping our code clear and readable. Later on, we'll learn some basic programming idioms. Programming idioms are little tricks that programmers often use to reduce complex tasks to just a line or two. They're sometimes rather cryptic, but once you understand them, they're quite useful.

6.16 goto Labels

We've gone through several of the usually employed logical controls in Section 4.11.3. This time we have the `goto` keyword. It is a commonly misused control statement, and it's about time we learned about how it's used.

The `goto` keyword allows you to skip around in a statement. This includes sending the computer back up in the code to run statements that it has already gone past. When using many `goto` statements, you end up making your code difficult to read as you need to jump around to follow the flow.

A `goto` statement is the keyword `goto` followed by an identifier, which is referred to as a *label*. This looks like `goto MyLabel;` which tells C# which label to go to, in this case as `MyLabel;` which is the identifier followed by a colon.

A `goto` statement should be encapsulated by some sort of logical statement. It's very easy to get caught in a loop with a `goto` statement. This has a similar behavior to `for(;;)` and `while(true)` which will freeze Unity 3D in an endless loop forcing you to kill the process from the Task Manager.

```
void Start ()
{
    StartOver:
        goto StartOver;
}
```

Using a label and a `goto` in this way will create an infinite loop that will require you to kill Unity 3D from the Task Manager. You can assume `StartOver:`, which tells where the computer will jump

to in the `Start ()` function when the `goto` tells the function where to jump to. Anything between the identified label and the `goto` command is executed as normal. There's nothing wrong with this statement syntactically, and it's just not very useful. This can be controlled by any code between the label and the `goto` statement.

```
void Start ()
{
    int num = 0;
StartOver:
    num++;
    if(num > 10) {
        goto Stop;
    }
    print ("stuck in a loop");
    goto StartOver;
Stop:
    print ("im free!");
}
```

This is the same as `for(int i = 0; i < 10; i++)`; only some might find it a bit harder to read. Of course, it's up to you to choose when to use a `goto` statement, but there are some useful cases. Finding a good reason to use the `goto` statement comes with experience in spotting when it might be useful. Because they're tricky to use and to read, most programmers shy away from using them at all. We're learning as much as we can in this book, so we may also learn how it's used.

6.16.1 A Basic Example

In the `GoToLabels` project, we'll start with the `Example.cs` attached to the Main Camera.

```
using UnityEngine;
using System.Collections;
public class Example : MonoBehaviour
{
    public int num = 1;
    void Start ()
    {
        if (num > 0)
        {
            goto MyLabel;//goes to MyLabel
        }
        print("number was less than 0");//gets skipped if num isn't > 0
        MyLabel://goes here when num is greater than 0
        print("I jumped to MyLabel");
    }
    //Update is called once per frame
    void Update ()
    {
    }
}
```

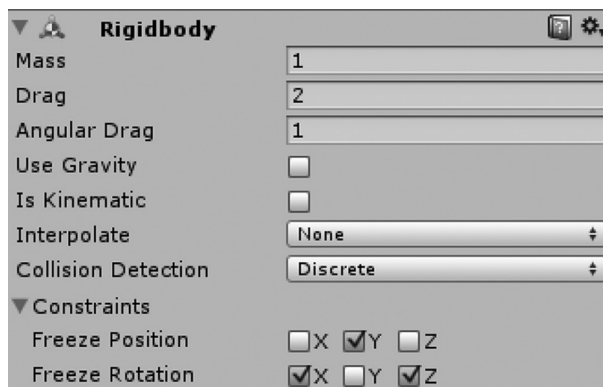
By adding in a `public int num` to the class, we're able to pick the `goto` label we're using. If we set `num` to 0, then both `print` statements will be executed. If the number is greater than 0, then only the second `"I jumped to MyLabel"` `print` function will be executed.

When we started off with the `switch` statement, we looked at the `case` keyword. The `case` keyword set up conditions which the statement would skip through till a case fits the statement. It then executed the code that followed the case.

`switch` and `goto` statements share the `:` colon notation for labels. However, `goto` lets you set up labels anywhere in a code block, whereas the `switch` statement allows labels only within the `switch` statement. When `goto` is used, it sends the software to the line following the label it was told to jump to.

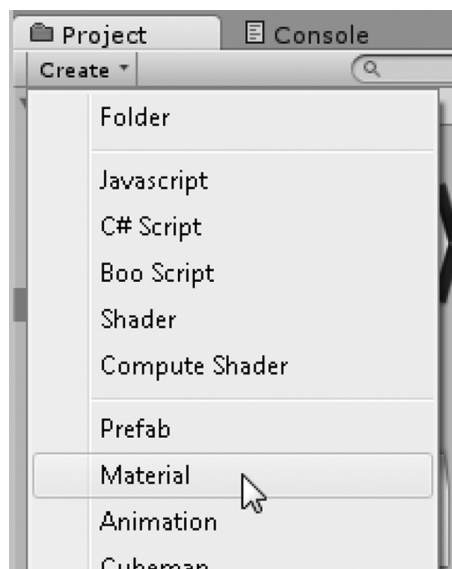
6.16.2 Zombie State Machine

Let's start by building a simple zombie character in Unity 3D. I created a zombie character using the `GameObject` → `Create Other` → `Capsule` menu. To this I needed to add in a `Rigidbody` component by selecting `Component` → `Physics` → `Rigidbody` and then change the parameters to the following settings:

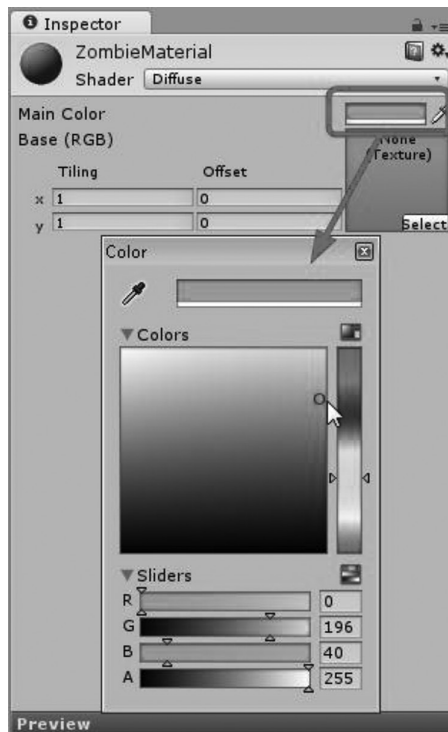


By adding some Drag, we keep the `Rigidbody` from sliding around on a surface like it were on ice. Then since we're not in need of any Gravity, we turn that off as well. To keep the zombie from tipping over, we need to freeze the X and Z rotations. He still needs to look around, so we'll leave Y unfrozen. Then to keep him on the ground, we'll freeze the Y position.

For fun I've added small spheres for eyes, but made sure to remove the collision component from the sphere objects parented to the `Zombie Capsule`. You can also create a `Material` using the `Project` menu `Create` → `Material`.



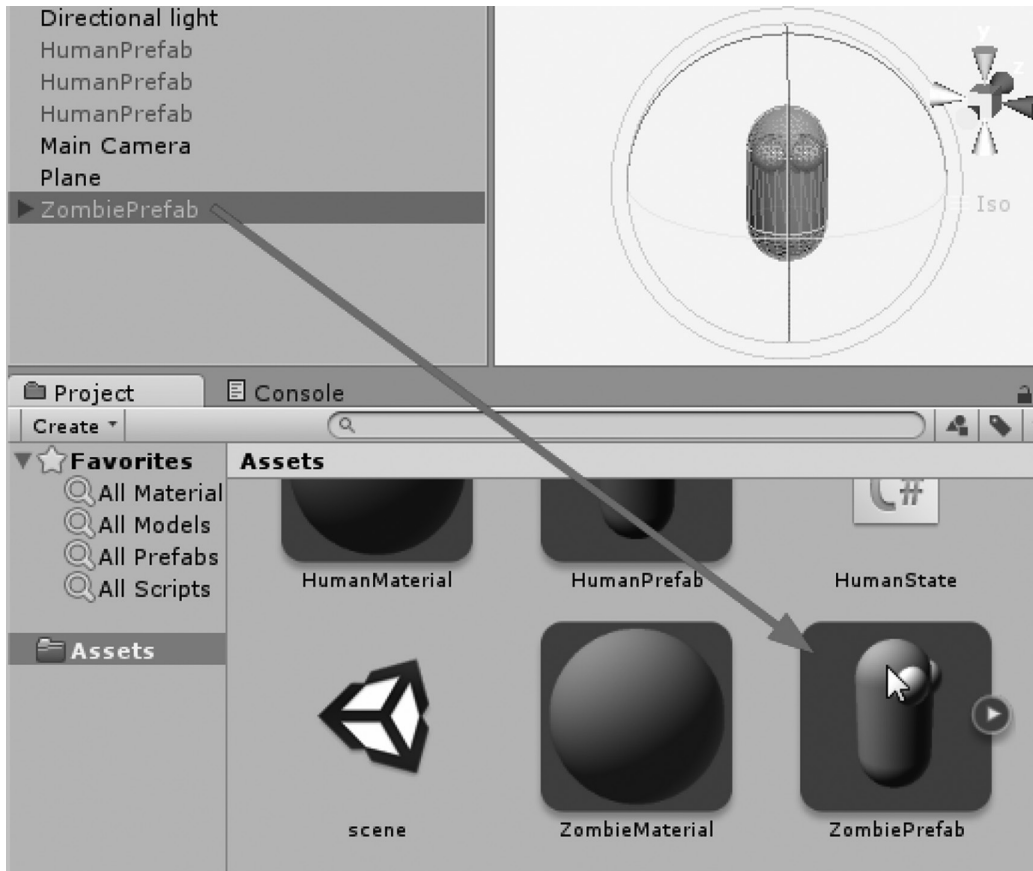
This can allow you to make the little guy green.



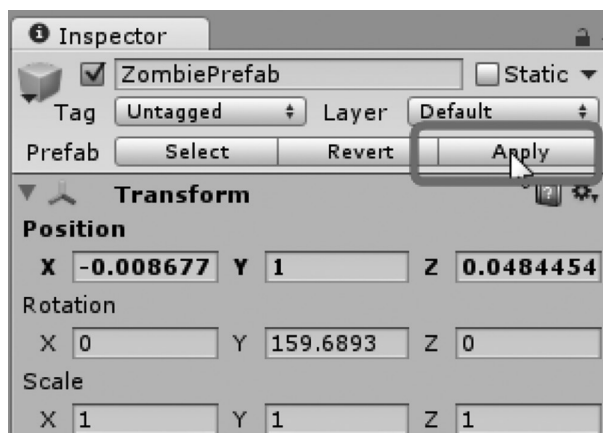
All you need to do once you create the Material and set its color is to drag it onto the Capsule's Mesh in the Scene editor.



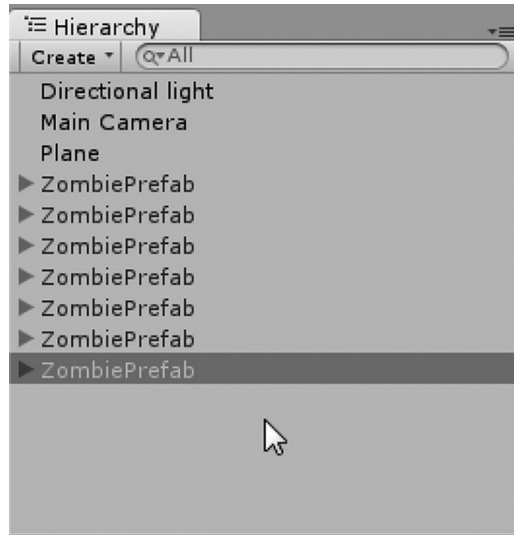
Once a `ZombieState.cs` is attached to the object, the object is dragged from the Hierarchy to the Project panel.



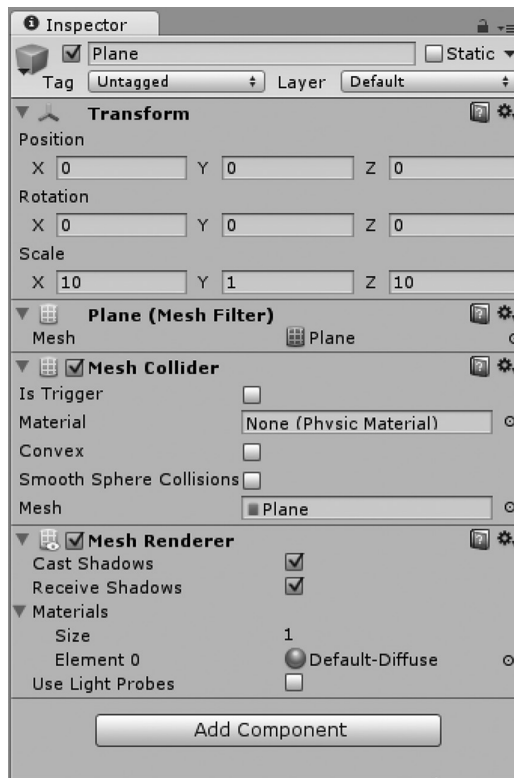
This created a Unity 3D prefab object. A prefab is a nicely packaged object that is an aggregation of different objects. This saves us some time if we need to make changes to multiple instances of the guy in the scene. We just make a modification to one; then we can press the Apply Changes to Prefab button in the Inspector panel, and the changes propagate to the rest of the objects that are from the same prefab.



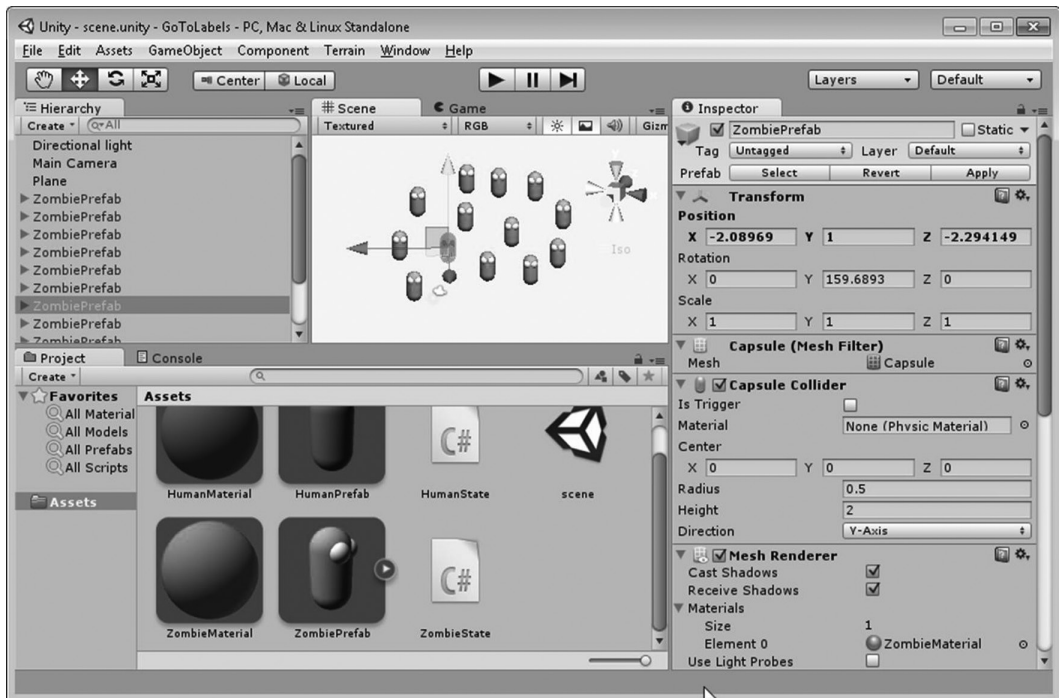
This will update all copies of the prefab in the scene.



Make multiple duplicates of the `ZombiePrefab`. Add in a `Plane` using the `GameObject` → `Create Other` → `Plane` menu so that the zombies have somewhere to walk on. Then set its size to the following settings:



Then for good measure, I've created a `Directional light` using the `GameObject` → `Create Other` → `Directional Light` menu so that nothing shows up as black. This will give us something to see from the `Main Camera`.



My scene has some extra stuff that we'll get to in a moment. If the zombies don't look up to the latest standards in video game graphics, you're right. This is what is often referred to as "programmer art." If you are an artist, then you might want to spend some time here to spruce things up. However, I'll let you do that on your own.

Labels are traditionally seen as messy. However, in some cases, they can be used quite well, in case of a simple artificial intelligence (AI) behavior. The zombie AI can exist in a limited number of states; in many cases, the states are often stored in an enum; we'll start with just two values in the enum to begin with. Creating a `ZombieState.cs` is where we will begin. In the new C# file, we'll add in a simple enum with the following information and then make a variable to hold the enum as follows:

```
enum ZState{
    idleing,
    wandering,
}
ZState MyState;
```

Then to store some additional information, we'll want to have a timer to check how long we'll be in each state. For some basic interesting behaviors, we'll want to have a variable for the closest object and the furthest object away from the zombie. This is considered a fairly standard practice as any specific monster should exist only in one state at a time. The code then turns into a large `switch` statement, with code specific to each state embedded in each segment of the `switch` statement. However, we can use the `switch` statement to simply control which `goto` label we jump to instead.

```
float stateTimer;
float closestDistance;
float furthestDistance;
GameObject closestGameObject;
GameObject furthestGameObject;
```

With these values, we'll be able to build up some interesting zombie's wandering and idleing behaviors. In the `Start ()` function, we need to initialize these values so they can be used later.

```
void Start ()
{
    stateTimer = 0.1f;
    MyState = ZState.idleing;
    closestDistance = Mathf.Infinity;
}
```

The zombie now has a short timer to start off with, and then we'll have an initial state to start in. Then since the `closestDistance` variable is going to start off as 0, we need this to be a large number instead, otherwise nothing will be closer than the default value; this needs to be fixed with a `Mathf.Infinity`, quite a large number indeed. We then add in a `switch` statement at the beginning of the `Update ()` function to different labels of `goto` inside of the `Update ()` function.

```
void Update ()
{
    switch(MyState)
    {
        case ZState.idleing:
            goto Ideling;
        case ZState.wandering:
            goto Wandering;
        default:
            break;
    }
    Ideling:
        return;
    Wandering:
        return;
}
```

This creates our basic starting place for building up some more interesting behaviors. If we look at the `Ideling:`, we should add in a system to hold in the loop there for a moment before moving on.

```
Ideling:
    stateTimer -= Time.deltaTime;
    if(stateTimer < 0.0f)
    {
        MyState = ZState.wandering;
        stateTimer = 3.0f;
    }
    return;
```

If we start with a `stateTimer -= Time.deltaTime;`, we'll count down the `stateTimer` by the time passed between each frame. Once this is less than 0.0f, we'll set the `MyState` value to another state and add time to the `stateTimer`.

```
Wandering:
    stateTimer -= Time.deltaTime;
    if(stateTimer < 0.0f)
    {
        MyState = ZState.idleing;
        stateTimer = 3.0f;
    }
    return;
```

Now we have a system to toggle between each state after 3.0f seconds or so. The `return;` after the label tells the `Update ()` function to stop evaluating code at the return and start over from the top. To add to the Idling behavior, we'll create a new function called `LookAround();`.

```
void LookAround()
{
    GameObject[] Zombies = (GameObject[])
    GameObject.FindObjectsOfType(typeof(GameObject));
    foreach (GameObject go in Zombies)
    {
        ZombieState z = go.GetComponent<ZombieState>();
        if(z == null || z == this)
        {
            continue;
        }
        Vector3 v = go.transform.position - transform.position;
        float distanceToGo = v.magnitude;
        if (distanceToGo < closestDistance)
        {
            closestDistance = distanceToGo;
            closestGameObject = go;
        }
        if (distanceToGo > furthestDistance)
        {
            furthestDistance = distanceToGo;
            furthestGameObject = go;
        }
    }
}
```

This function has a few things going on here. First, we have the line at the top:

```
GameObject[] Zombies = (GameObject[])
GameObject.FindObjectsOfType(typeof(GameObject));
```

This looks through all of the `GameObjects` in the scene and prepares an array populated with every game object in the scene called `Zombies`. Now that we have an array, we can iterate through each one looking for another zombie. To do this, we start with a `foreach` loop.

```
foreach (GameObject go in Zombies)
{
    ZombieState z = go.GetComponent<ZombieState>();
```

Then we use `ZombieState z = go.GetComponent<ZombieState>();` to check if the `GameObject` `go` has a `ZombieState` component attached to it. This ensures that we don't iterate through things such as the ground plane, light, or main camera. To make this check, we use the following lines:

```
if(z == null || z == this)
{
    continue;
}
```

6.16.3 This as a Reference to Yourself

Here we're also seeing `z == this`, which is important to point out. The array that is being returned includes all objects in the scene, with a `ZombieState` component attached. This includes the `ZombieState` that is doing the checking.

In the case where you want to check if you are not going to complete any computations based on where you are and you are included in your own data, then it's a good idea to note that you can exclude yourself from being calculated by using `this` as a reference to the script doing the evaluation. If you were making a distance check to the closest zombie and you are a zombie, then you'd be pretty close to yourself. Of course, that's not what we want to include in our calculations when we're looking for another nearby zombie.

This `if` statement checks if the `z` is null, or rather the `GameObject go` has no `ZombieState` attached. We also don't want to include the instance of the object itself as a potential zombie to interact with. The `continue` keyword tells the `foreach` loop to move on to the next index in the array. Then we need to do some distance checking.

```
Vector3 v = go.transform.position - transform.position;
float distanceToGo = v.magnitude;
```

With `Vector3 v = go.transform.position - transform.position;`, we get a `Vector3`, which is the difference between the `transform.position` of the zombie who is checking the distances to the `GameObject go` in the array of `Zombies`. We then convert this `Vector3 v` into a `float` that represents the distance to the zombie we're looking at. Once we have a distance, we compare that value to the closest distance and set the `closestGameObject` to the `GameObject` if it's closer than the last `closestDistance`, and likewise for the `furthestDistance`.

```
if (distanceToGo < closestDistance)
{
    closestDistance = distanceToGo;
    closestGameObject = go;
}
if (distanceToGo > furthestDistance)
{
    furthestDistance = distanceToGo;
    furthestGameObject = go;
}
```

Now we have populated the `closestGameObject` and the `furthestGameObject` values.

```
Ideling:
    stateTimer -= Time.deltaTime;
    if(stateTimer < 0.0f)
    {
        MyState = ZState.wandering;
        stateTimer = 3.0f;
        closestDistance = Mathf.Infinity;
        furthestDistance = 0f;
        LookAround();
    }
    return;
```

Now we can add the `LookAround();` function to our `Ideling:` label in the `Update ()` function. We also want to reset the `closestDistance` and `furthestDistance` values just before running the function, otherwise we won't be able to pick new objects each time the function is run. After this, we want to use the `closestGameObject` and `furthestGameObject` to move our little zombie around.


```

void MoveAround()
{
    Vector3 MoveAway =
        (transform.position -
         closestGameObject.transform.position).normalized;
    Vector3 MoveTo =
        (transform.position -
         furthestGameObject.transform.position).normalized;
    Vector3 directionToMove = MoveAway - MoveTo;
    transform.forward = directionToMove;
    gameObject.rigidbody.velocity =
        directionToMove * Random.Range(10, 30) * 0.1f;
    Debug.DrawRay(transform.position, directionToMove, Color.blue);
    Debug.DrawLine(transform.position,
        closestGameObject.transform.position, Color.red);
    Debug.DrawLine(transform.position,
        furthestGameObject.transform.position, Color.green);
}

```

We want two vectors to create a third vector. One will move us away from the `closestGameObject` and the other will point at the `furthestGameObject`. This is done with the lines at the beginning. We then use the normalized value of the `Vector3` to limit the values that these will give us to a magnitude of 1.

```

Vector3 MoveAway = (transform.position -
    closestGameObject.transform.position).normalized;
Vector3 MoveTo = (transform.position -
    furthestGameObject.transform.position).normalized;

```

A third `Vector3` is generated to give us a direction to go in, which is `Vector3 directionToMove = MoveAway - MoveTo`; this gives us a push away from the closest object and moves the zombie toward the furthest object. We then turn the zombie in that direction by using `transform.forward = directionToMove`; and then we give the zombie a push in that direction as well:

```

gameObject.rigidbody.velocity = directionToMove * 0.3f;

```

The `directionToMove * 0.3f` lowers the speed from 1.0 to a slower value. Zombies don't move so fast, so we want to reduce the speed a bit. To see what's going on, I've added in the three lines:

```

Debug.DrawRay(transform.position, directionToMove, Color.blue);
Debug.DrawLine(transform.position,
    closestGameObject.transform.position, Color.red);
Debug.DrawLine(transform.position,
    furthestGameObject.transform.position, Color.green);

```

These lines are drawn closest, furthest, and toward the direction of movement and the zombie. This can now be added to the `Update ()` function's Wandering label.

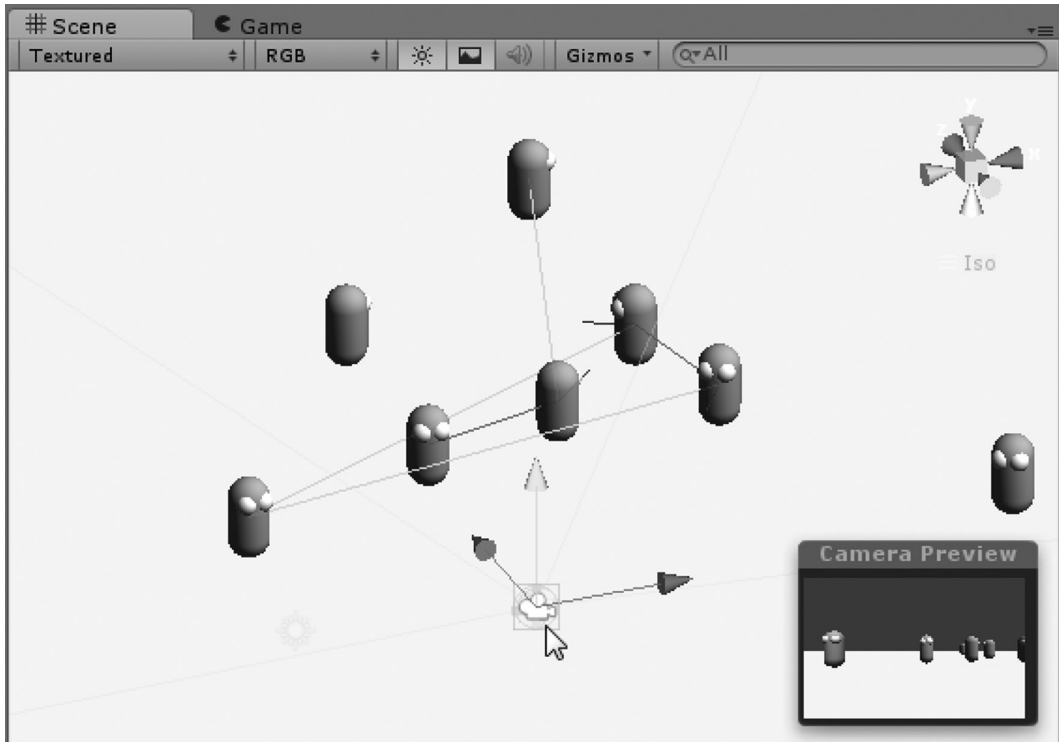
```

Wandering:
    stateTimer -= Time.deltaTime;
    MoveAround();
    if(stateTimer < 0.0f)
    {
        MyState = ZState.idleing;
        stateTimer = 3.0f;
    }
    return;

```

With this in place above the `stateTimer` that switches functions, we have a continuous execution of the function while the `Update ()` function is being called. This updates the `MoveAround()` code as though it were inside of the `Update ()` function; by doing this we're able to keep the code clean and independent in its own function.

One note before going on: Make sure that all of the little guys have the same `transform.position.y`, otherwise you'll have individuals looking up or down, which will throw off the rotation values when aiming the zombies at one another.



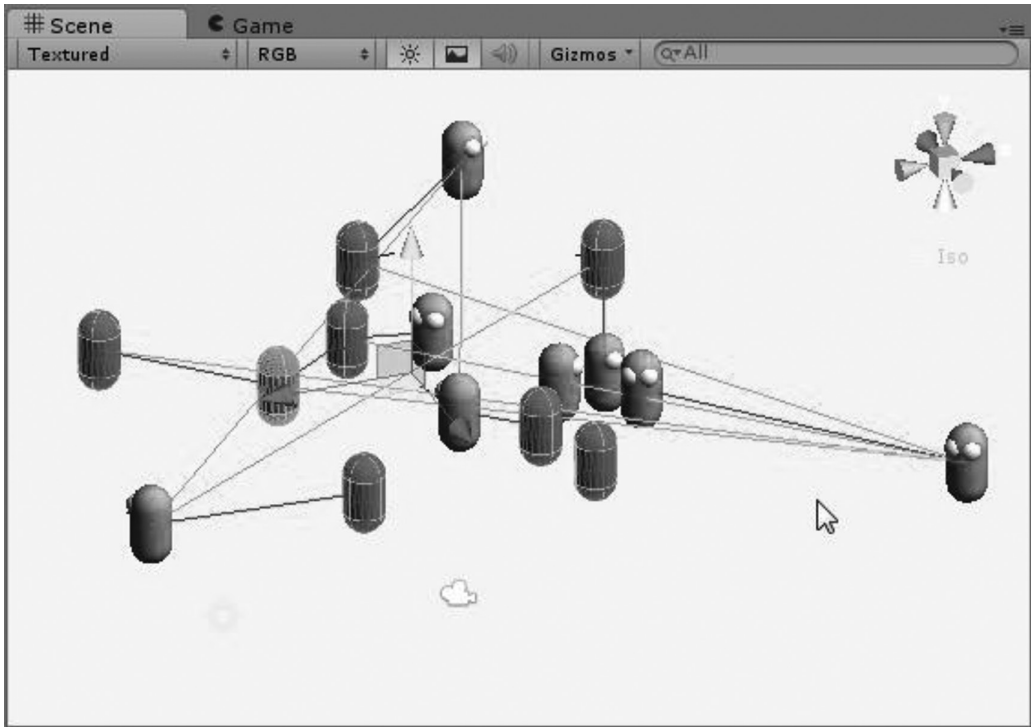
Running this will begin the process of multiple zombies wandering around one another, scooting toward the zombie who is furthest away from him, and pushing away from the one who is closest. This behavior tends to keep them in one general area so long as they don't go too fast in any one direction for too long.

6.16.4 HumanState Based on ZombieState

We can easily create a new behavior based on the zombie with the following code in a new `HumanState.cs`:

```
using UnityEngine;
using System.Collections;
public class HumanState : ZombieState
{
}
```

By removing the `Start ()` and `Update ()`, we can create a `HumanPrefab`, with the `HumanState.cs` file attached rather than the `ZombieState.cs` component.



Adding a bunch of humans right now, noted here with the differently colored capsule, we'll get the same behavior as the `ZombieState`. However, we can make a simple change to the `LookAround()` function and not touch anything else. First, in the `ZombieState.cs`, we need to change the `LookAround()` function so we can enable overriding the function.

```
virtual public void LookAround()
```

Add in `virtual` and `public` before the `void`. This will allow the `HumanState.cs` to override the function call and allow us to change the behavior. In addition, we'll be accessing some variables in `ZombieState`, so they too need to be made `public`.

```
public float closestDistance;
public float furthestDistance;
public GameObject closestGameObject;
public GameObject furthestGameObject;
```

6.16.5 The `is` Keyword

With these variables made `public`, we'll be able to use these from the new `LookAround();` function in `HumanState`. In Chapter 5, we covered the `as` keyword when we were doing implicit casting. The handy use for implicit casting comes into play when we want to check an object's type as a condition statement.

For instance, `Zombie z`, as we know before, has a `ZombieState` component. To do a check, we can use the following statement: `if (z is ZombieState)`. We can check if the `z` is of type `ZombieState`. If the `z` is indeed a `ZombieState`, then we get `true`, otherwise the `if` statement is not executed.

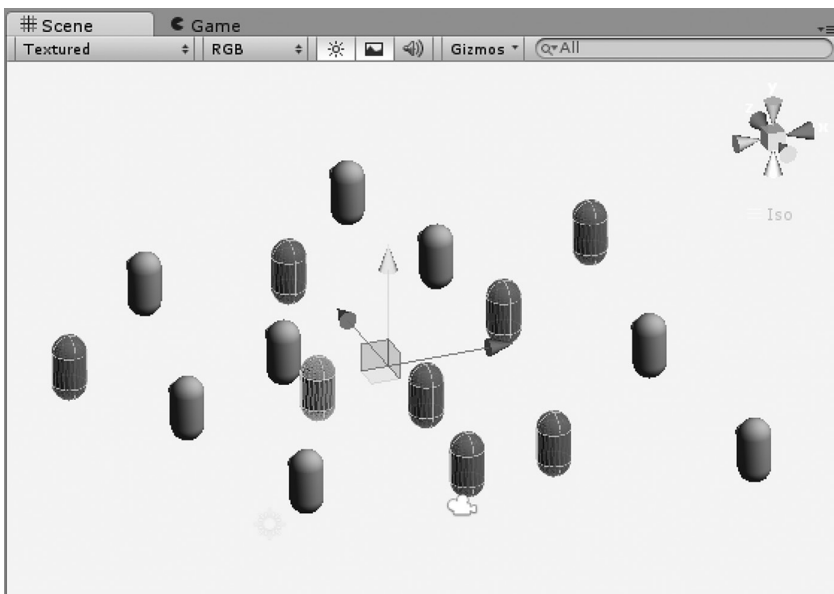
```

override public void LookAround()
{
    GameObject[] Zombies = (GameObject[])
    GameObject.FindObjectsOfType(typeof(GameObject));
    foreach (GameObject go in Zombies)
    {
        ZombieState z = go.GetComponent<ZombieState>();
        if (z == null || z == this)
        {
            continue;
        }
        Vector3 v = go.transform.position - transform.position;
        float distanceToGo = v.magnitude;
        if (distanceToGo < closestDistance)
        {
            if (z is ZombieState)
            {
                closestDistance = distanceToGo;
                closestGameObject = go;
            }
        }
        if (distanceToGo > furthestDistance)
        {
            if (z is HumanState)
            {
                furthestDistance = distanceToGo;
                furthestGameObject = go;
            }
        }
    }
}

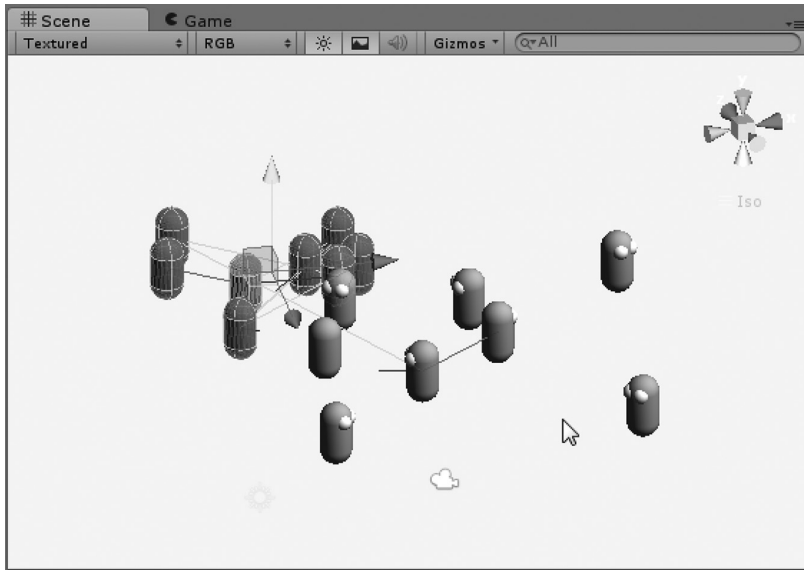
```

We're adding only two new if statements. Inside of the `closestDistance`, we check again to see if the `z` is a `ZombieState`; if it is, then we set the `closestGameObject` and distance to that object. If the furthest object is a `HumanState`, then we set the `furthestGameObject` to the furthest object.

What starts off as an evenly distributed mass of humans and zombies like this:



sorts out into a clump of humans wandering away from the zombies.



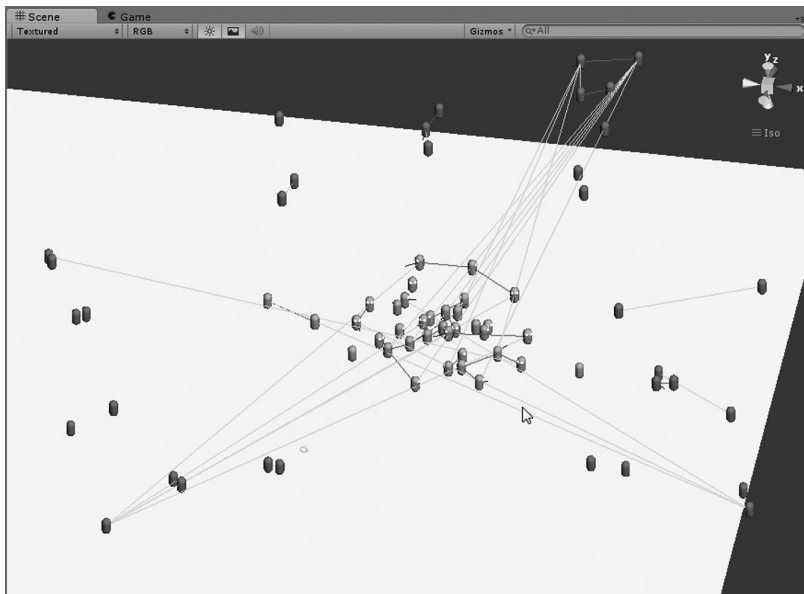
Once enough humans and zombies have been added, we get a more interesting behavior by limiting how far a human is willing to go to join another human.

```

if (distanceToGo > furthestDistance && distanceToGo < 10)
{
    if(z is HumanState) {
        furthestDistance = distanceToGo;
        furthestGameObject = go;
    }
}

```

This turns into the following grouping habit:



Notice that the humans tend to cluster into smaller groups and pairs. In any case, I encourage you to play with the code to find other emergent behaviors.

6.16.6 What We've Learned

`Goto` labels don't have to be cumbersome to be useful. As a simple divider to a long `Update ()` function, they can work fairly easily and still remain clear so long as the mechanism that switches between labels is clear and contained in one place such as a `switch` statement.

You can easily make an argument to leave all of the logic within the `switch` statement and avoid the labels altogether, but this can easily make the `switch` statement larger and bulkier than necessary as well. In the end, the only difference is readability. If by using labels you make your code easier to follow, then using labels is fine. If your `switch` statement begins to overflow with different cases, then you might want to try another system outside of the `switch`, and labels might be a nice option.

On your own it's a good idea to see what this code would look like by adding different states. Perhaps when a zombie gets close enough to a human, that human could turn into a zombie. When a pair of humans gets close to a zombie, then the zombie might turn into a human. Simple mechanics like this can quickly add a very unpredictable level of behavior and emerge into a game mechanic simply through experimentation with the code itself.

6.17 More on Arrays

Arrays have a few different descriptions. Your standard array is called a single-dimensional array, which we have dealt with earlier. Other array types include multidimensional and jagged. Arrays can store any type of data; this includes other arrays.

6.17.1 Length and Count

Declaring an array is simple and uses a different operator than a class or function to contain its data. The square brackets `[]` are used to tell the compiler that your identifier is going to be used for an array. The following statement declares an array of integers:

```
int[] ints;
```

We can dynamically declare an array of `ints` using the following notation. Use curly braces `{}` to contain the contents of the array assigned to `primes`.

```
int[] primes = {1, 3, 5, 7, 11, 13, 17, 23, 27, 31};
```

Arrays have several functions associated with them, which we can use with different loops. To get the number of items in an array, we use the `Length` property of the array.

6.17.1.1 A Basic Example

We'll go with the `MoreArrays` project and start with the `Example.cs` component attached to the `Main Camera` in the scene provided.

```
int[] primes = {1, 3, 5, 7, 11, 13, 17, 23, 27, 31};
int items = primes.Length;
```

In this example, we assign an `int items` to the `Length` of the array. This is a common practice when using an array. We'll see why this is more useful than creating an array of a set length ahead of time in

a moment. To use this in a `for` loop, we use the following syntax shown in the following code fragment added to the `Start ()` function:

```
//Use this for initialization
void Start ()
{
    int[] primes = {1, 3, 5, 7, 11, 13, 17, 23, 27, 31};
    int items = primes.Length;
    for (int i = 0; i < items; i++)
    {
        print(primes [i]);
    }
}
```

This code prints out a new line for each number stored in the array. Try this out by assigning the script to an object in a new game scene; I usually pick the Main Camera. Each time the `for` loop iterates, the `int i` moves to the next item in the array starting with the zeroth index. Once the end of the list is reached, the `for` loop exits. We could also use a `while` loop to do the same thing.

```
int j = 0;
while (j < items)
{
    print(primes [j]);
    j++;
}
```

This produces the same output with a few less characters, but the counter `j` required for the `while` loop needs to live outside of the `while` statement's code block. Yet another option, and one that's slightly tailored for an array, is `foreach`.

6.17.2 Foreach: A Reminder

With an array of `ints` in our `primes` variable, we can use the following syntax to print out each item in the array:

```
foreach (int i in primes)
{
    print(i);
}
```

This syntax is quite short, simple, and easy to use. The keyword `in` assigns the `int i` variable to each element in the `primes` array it's given.

Which form you use is up to you, but it's important to know that you have more than one option. Of course, you can use any identifier for counting indexes; `int j` could just as easily be `int index` or anything else that makes sense. Though the common convention is to use the `for (int i = 0; i < items; i++)` form for iterating through an array, the `int i` variable initialized in the parameter list of the `for` loop can be used for various things related to the element we're stepping through in the array. We'll be able to use this more often than not when we need to know where we are in the array. If this information isn't necessary, then it's easier to stick to the `foreach` iterator when using an array.

We can declare arrays of different lengths by using different bits of data. Each one is of a different length. Using the `Length` property of each array, it's easier for us to print out each array without knowing how many items are stored in each one.

```
int[] primes = {1, 3, 5, 7, 11, 13, 17, 23, 27, 31};
int[] fibonacci = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144};
int[] powersOfTwo = {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024};
```

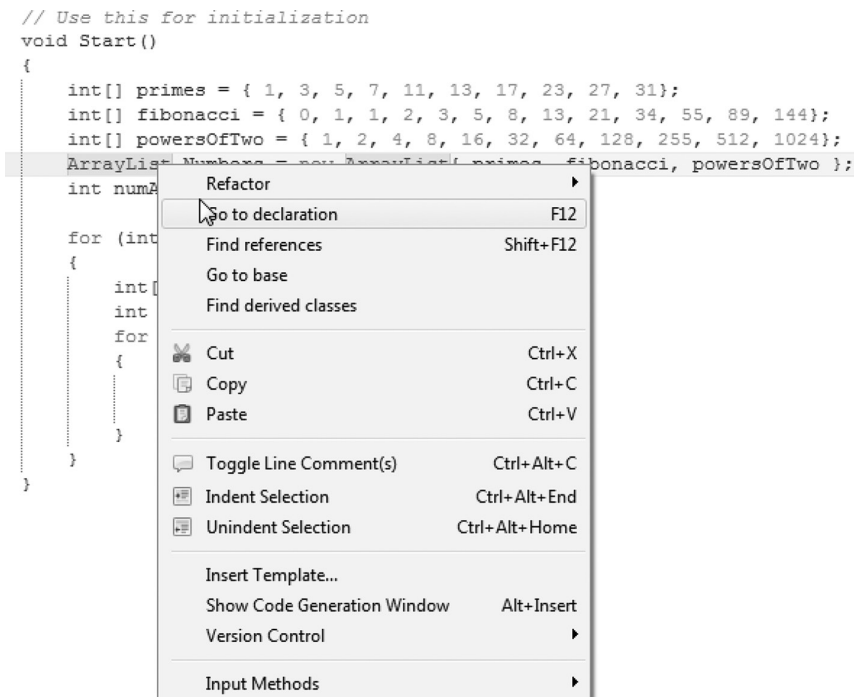
Of course, these aren't complete sets of numbers, but that's not the point. They are different lengths, and we don't need to keep track of how many items are stored in each array to print out their contents.

```
void Start ()
{
    int[] primes = {1, 3, 5, 7, 11, 13, 17, 23, 27, 31};
    int[] fibonacci = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144};
    int[] powersOfTwo = {1, 2, 4, 8, 16, 32, 64, 128, 255, 512, 1024};
    ArrayList Numbers = new ArrayList{primes, fibonacci, powersOfTwo};
    int numArrays = Numbers.Count;
    for (int i = 0; i < numArrays; i++)
    {
        int[] Nums = Numbers [i] as int[];
        int items = Nums.Length;
        for (int j = 0; j < items; j++)
        {
            int Num = Nums [j];
            print(Num);
        }
    }
}
```

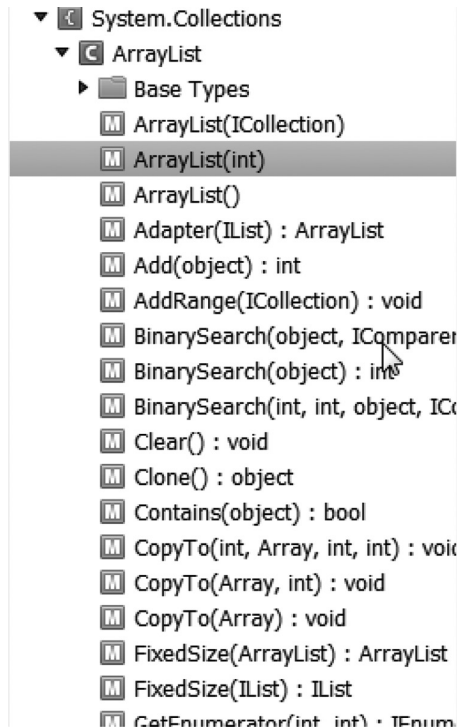
In the above code fragment, we've added in three different arrays of integers using the `Start ()` function. After that we declare a new `ArrayList` called `Numbers` fulfilled with a new `ArrayList` `{primes, fibonacci, powersOfTwo}`. This fills the new `ArrayList` with the three different `int[]` arrays, where each `int[]` has a different `Length` property.

`ArrayLists` differ from a regular array in that its `Length` property is called `Count`. Therefore, to get the number of items stored in the `ArrayList`, we use `Numbers.Count` rather than `Length`. This is assigned to an `int numArrays` for use in a `for` loop.

To learn this, we would have to ask on a forum, "How do I get the size of an `ArrayList`?" Alternatively, we could right click on the `ArrayList` type and select `Go to declaration` from the following pop-up:



This brings us to the declaration of classes and all of the things that the `ArrayList` allows us to do. Scroll through the options till we find something we can use.



In the Assembly Browser, we'll find a `Count` property that sounds reasonably usable. It's possible to do many different things with the other options found in the `ArrayList` class, so we'll experiment with them next. You'll need to remember the `Go to definition` function in `MonoDevelop` in order to be a self-sufficient programmer. If it's late at night and you have a final due in the morning, you may not have enough time to wait for an answer on a forum.

Unfortunately, there aren't so many options with the `int[]` as this brings you to only what an `int32` is defined as. Luckily, most of the questions have been asked about how to deal with an array in C#. There is, however, a class definition for `Array`. This can be found by using the `Search` field in the Assembly Browser.

From here, you can find all of the functions and properties available to use with the `Array` type. The `ArrayList`'s `count` can be used as a condition to stop the `for` loop. Once we have the `ArrayList`'s `count`, it's used in the following `for` loop as the condition to stop the loop.

```
int numArrays = Numbers.Count;
for (int i = 0; i < numArrays; i++)
{
}
```

This code will allow us to iterate through each array stored in the `Numbers` array list. Next, we need to get the `int[]` array stored at the current index of the array list. This is done by using the `for` loop's initializer's `int i` variable and a cast. `ArrayList` stores each item as a generic `Object` type. This means that the item at `Numbers[i]` is an `Object` type. In this case, the `Object` stored at the index is an `int[]` type.

```
int[] Nums = Numbers[i] as int[];
```

Casting the Object to an `int[]` is as simple as `Numbers[i] as int[]`; . Without this cast, we get the following error:

```
Assets/Arrays.cs(15,31): error CS0266: Cannot implicitly convert type
'object' to 'int[]'. An explicit conversion exists (are you missing a cast?)
```

This assumes that an Object is trying to be used as an `int[]`, as indicated by the declaration `int[] Nums`, and to do so, we need to use the `as int[]` cast. Once `Nums` is an `int[]` array, we can use it as a regular array. This means we can get the `Nums.Length` to start another for loop.

```
int[] Nums = Numbers[i] as int[];
int items = Nums.Length;
for(int j = 0; j < items; j++)
{
```

This loop will begin the next loop printing out each item stored in the `Nums` integer array. As we did earlier, we use the following fragment:

```
for(int j = 0; j < items; j++)
{
    int Num = Nums[j];
    print (Num);
}
```

This fragment is inside of the first for loop. Using this, we'll get each item of each array stored at each index of the `Numbers` array list. This seems like a great deal of work, but in truth it's all necessary and this will become second nature once you're used to the steps involved in dealing with arrays.

`ArrayLists` allow you to store any variety of object. To store an array of arrays, you need to use an `ArrayList`. What might seem correct to start off with might be something like the following:

```
Array Numbers = {primes, fibonacci, powersOfTwo};
int numArrays = Numbers.Length;
```

In fact, this is syntactically correct, and intuition might tell you that there's no problem with this. However, the `Array` type isn't defined as it is without any type assigned to it. However, we can use the following:

```
object[] Numbers = {primes, fibonacci, powersOfTwo};
```

This turns each `int[]` into an object, and `Numbers` is now an array of objects. To an effect we've created the same thing as before with the `ArrayList`; only we've defined the length of the `Numbers` array by entering three elements between the curly braces.

For clarification, we can use the `foreach` iterator with both the `ArrayList` and the `int[]` array.

```
ArrayList Numbers = new ArrayList{primes, fibonacci, powersOfTwo};
foreach (int[] Nums in Numbers)
{
    foreach(int n in Nums)
    {
        print (n);
    }
}
```

This produces the same output to the Console panel in Unity 3D as the previous version; only it's a lot shorter. It's important to recognize the first `foreach` statement contains `int[] Nums` in `Numbers` as each element in the `Numbers` array is an `int[]` array type.

6.17.3 Discovery

Learning what a class has to offer involves a great deal testing. For instance, we saw many other functions in `ArrayList` which we didn't use. We saw `Count` that returned an integer value. This was indicated when the listing showed us `Count: int` in the Assembly Browser.

However, it's good to learn about the other functions found in `ArrayList`. Let's pick the `ToArray() : object[]` function; this sounds useful, but what does it do? More important, how do we find out? The steps here begin with testing various features one at a time.

The return type indicates that it's giving us an object, so let's start there.

```
ArrayList Numbers = new ArrayList{primes, fibonacci, powersOfTwo};  
object thing = Numbers.ToArray();  
print (thing);
```

This sends the following output to the Console panel:

```
System.Object []  
UnityEngine.MonoBehaviour:print (Object)  
Arrays:Start () (at Assets/Arrays.cs:14)
```

The thing is apparently a `System.object[]` that was sent to the Console output panel. From this, we learned that `thing` is, indeed, an `object[]`, or rather it's an array of objects. What can we do with this? Let's change this to match what we've observed.

```
object[] thing = Numbers.ToArray() as object[];  
print (thing.Length);
```

Now that we know it's an array, let's cast it to being a proper array by adding in a cast. Now the `thing` returns a length. We get 3 sent to the Console panel of Unity 3D when we press the Play game button. But what's going on here? Remember that earlier we had to use `numbers.Count` to get the size of the `ArrayList`. Now we're dealing with a regular `Array`, not an `ArrayList`. Therefore, should we need to use an `Array` and not an `ArrayList`, this function could come in handy! Good to know.

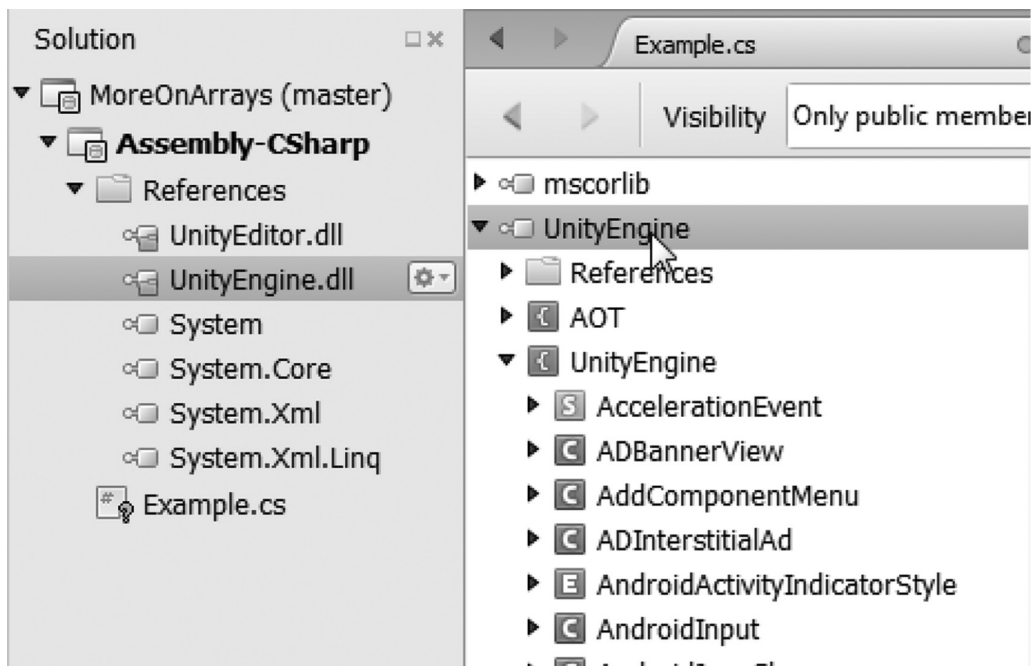
This little exercise is important; a part of learning any new language is testing the code that you discover when browsing through a class' functions. Anything listed in the Assembly Browser is there for a reason. At some point, somebody thought the function was necessary, so he or she added it.

6.17.4 Putting It Together

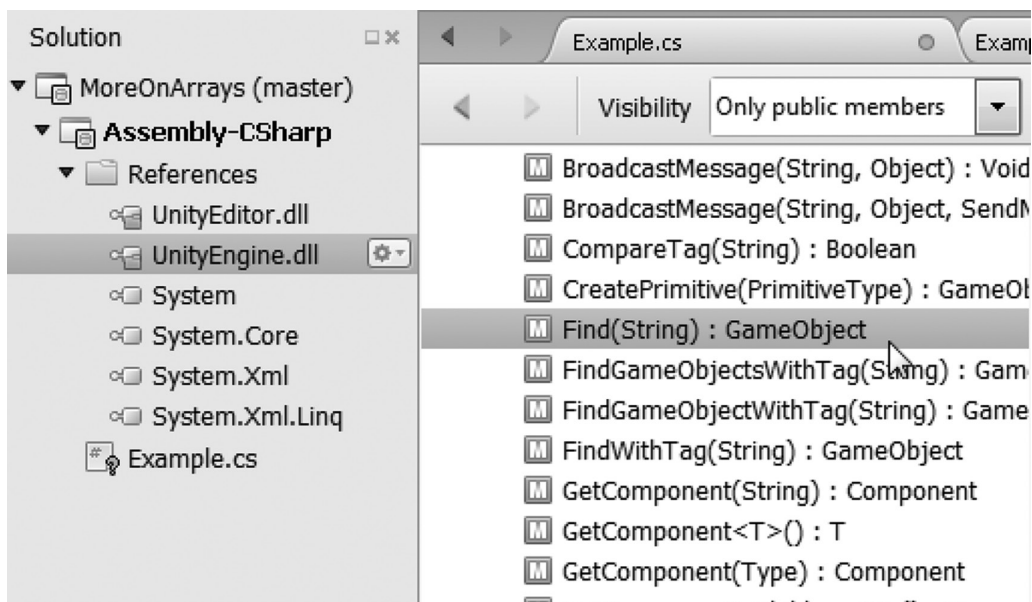
When we're building a game, one important use of an array is storing many different objects and sending them commands in a coordinated manner. We did this earlier by making a line of undulating cubes. However, what if we want to keep track of a number of monsters approaching the player?

The first thing we'd need to do is figure out how to find the monsters. We can do this by keeping track of them as they're spawned, but this would mean there's a sort of master controller both spawning them and keeping track of each one.

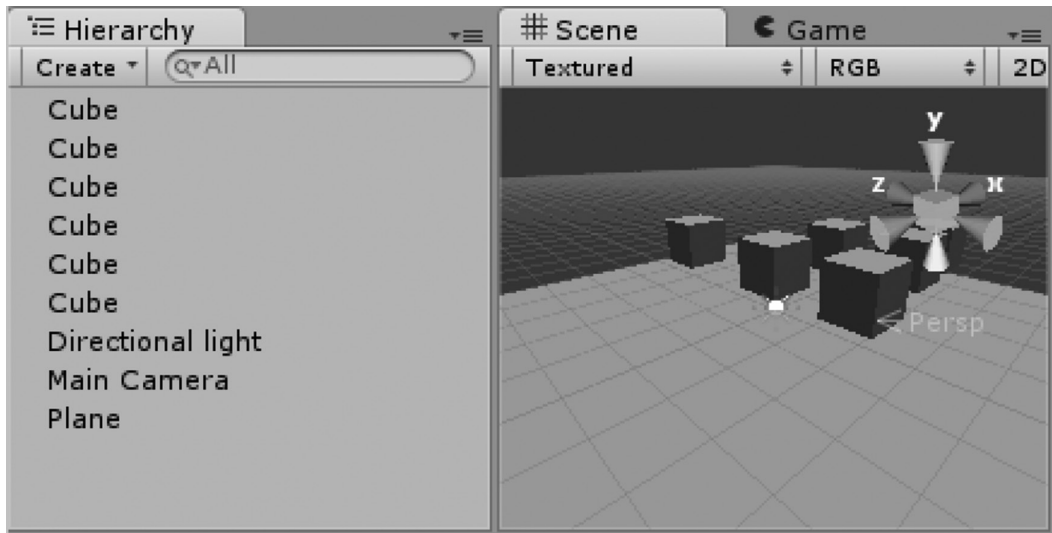
Unity's `GameObject` class will come to the rescue. Expand the References folder found in the Solution panel in MonoDevelop.



This will open the Assembly Browser, where we'll look for the `GameObject` class we've worked with in the past. If you scroll through the various functions found within the `GameObject` class, we'll come across a few different functions with the word "find" in them.



Perhaps we can discover how these are used.



To start our test, I've built a simple scene that includes some cubes. I then created a new C# class called `Monster` and assigned it to each of the cubes. I added in a light to make things a bit easier to see. Finally, I added in a ground cube so that there's something to look at other than the vastness of nothing that exists in an empty level.

To the Main Camera, I've added a new `Example.cs` script so we can begin testing out some new scripting ideas.

To get started with `GameObject.Find()` : `GameObject`, the Assembly Browser tells us that it returns a `GameObject`. Therefore, in the `Start ()` function, we'll add in the following code:

```
GameObject go = GameObject.Find("Cube");
print (go);
```

Once we add in the first parenthesis (after `Find`, we're reminded that it's looking for a string that is the name of the `GameObject` we're expecting to return. Let's test out "Cube," since there seem to be many in the scene we're working with. Remember to use an uppercase C since that's how it's named in the scene.

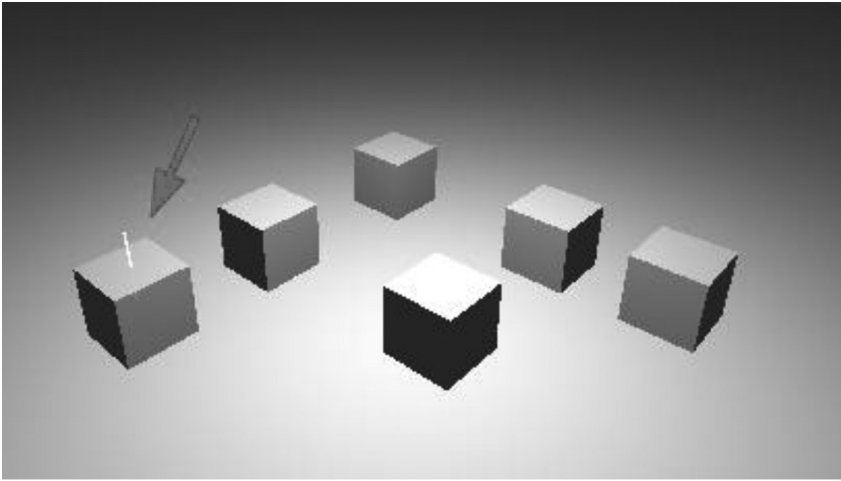
The above code fragment added to the `Start ()` function produces the following output in the Console panel.

```
Cube (UnityEngine.GameObject)
UnityEngine.MonoBehaviour:print(Object)
Example:Start () (at Assets/Example.cs:9)
```

This output looks promising. Therefore, what can we do with the `Cube` we just found? Our camera knows about something else in the scene, which is pretty important. If a monster is going to find a player, he'll have to find it in code. However, which `Cube` did the camera find?

```
//Update is called once per frame
void Update () {
    GameObject go = GameObject.Find("Cube");
    Vector3 CubePosition = go.transform.position;
    Vector3 Up = new Vector3(0, 10, 0);
    Debug.DrawRay(CubePosition, Up);
}
```

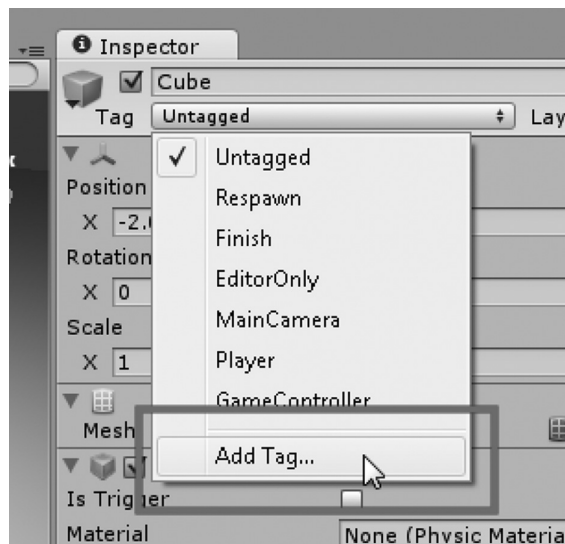
We're going to use the `Update ()` function for this. I've moved my code from the `Start ()` function to the `Update ()` function so we can use the `Debug.DrawRay()` function we used earlier.



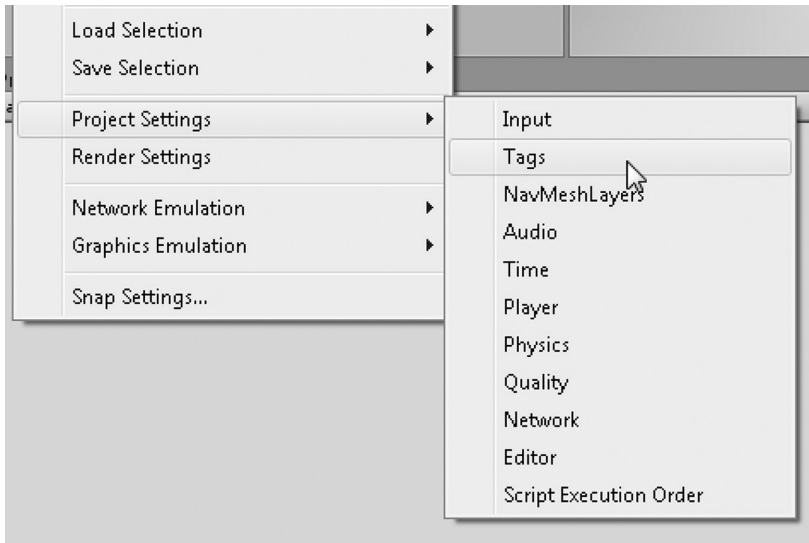
When we run the game, we can take a look at the Scene panel that will draw Gizmos used by the Debug functions. It looks like it's just picking the first Cube at the top of the list of cubes in the Hierarchy panel. This is most likely the case, but that doesn't necessarily mean it's the closest Cube in the scene.

How would we go about finding the nearest Cube to the camera? We'd want a list of all of the cubes, compare their distances, and pick the cube with the smallest distance value. This means we want an array of all of the `GameObjects` of the Cube in the scene.

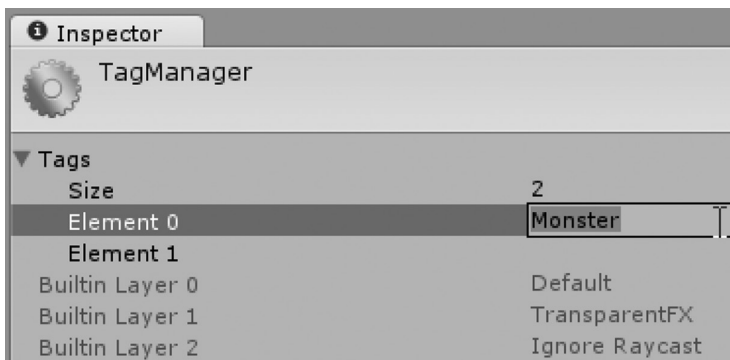
The closest function we can find is `GameObject.FindGameObjectsWithTag()` that returns a `GameObject[]` array. Therefore, I guess this means each `Monster` will need a `Monster` tag assigned. In Unity 3D, to create a new Tag, you select any object in the scene. In the Inspector panel, go under the object's name and open the Tag pull-down menu. Select `Add Tag...`



The following Tag Manager panel will open. The Tag Manager is also accessible from the Edit menu: Select `Edit → Project Settings → Tags`.



This will take you to the same place. Once there, expand the Tags submenu and add in **Monster** to the zeroth element of the Tag array. You may notice that an additional slot will automatically be added when you start typing.



Once this is done, open the `Monster.cs` we created and assigned to each of the cubes in the scene.

```
//Use this for initialization
void Start ()
{
    tag = "Monster";
}
```

In the `Start ()` function in the `Monster` class, I've added in `tag = "Monster";` which sets the tag of any object that the script is assigned to "Monster." This is a bit easier than going to each cube in the scene and changing the tag manually. Remember that programmers are lazy.

Moving back to the `Example.cs`, we'll want to test out the `FindGameObjectsWithTag()` function.

```
//Update is called once per frame
void Update ()
{
    GameObject[] gos = GameObject.FindGameObjectsWithTag("Monster");
    print (gos.Length);
}
```

We'll want to have an array of `GameObject[]` to fill in with whatever data is provided by the `GameObject`'s function. To double check that there's data in the identifier `gos`, we'll see if the array has a `Length` greater than 0 by printing the `Length` property of the array. Now we get the following output from the `Example.cs` script.

```
6
UnityEngine.MonoBehaviour:print(Object)
Example:Update () (at Assets/Example.cs:14)
```

Awesome, now we're getting somewhere. Now we're going to get each `GameObject`'s transform. position.

```
//Update is called once per frame
void Update ()
{
    GameObject[] gos = GameObject.FindGameObjectsWithTag("Monster");
    Debug.Log(gos.Length);
    foreach (GameObject g in gos)
    {
        print(g.transform.position);
    }
}
```

This code will print out a bunch of different `Vector3` values. Each position isn't that useful, so we're going to do something about that. A distance in math terms is called a magnitude. To get a magnitude, we need to provide a `Vector3()` and use the `.magnitude` property of the `Vector3()` class. This is better explained with some sample code.

```
foreach(GameObject g in gos)
{
    Vector3 vec = g.transform.position - transform.position;
    float distance = vec.magnitude;
    print(distance);
}
```

To get a single vector that informs us of the magnitude of any vector, we subtract one `Vector3` from another. We use `Vector3 vec = g.transform.position - transform.position;` to get the difference between two different vectors. To find `vec`'s magnitude, we use `float distance = vec.magnitude;` to store that value. Running the above code fragment in the `Update ()` function prints out a bunch of numbers; each one should be a game object's distance from the camera, or whatever you assigned the `Example.cs` script to in the scene.

Now we collect the different distances and sort through them to find the smallest one. Here's an interesting point. What if there are more or less cubes in the scene? This array may be of any size. Therefore, we'll make sure that we have an array that can adapt to any size; that's where the `ArrayList` really comes in.

```
void Update ()
{
    GameObject[] gos = GameObject.FindGameObjectsWithTag("Monster");
    ArrayList distances = new ArrayList();
    foreach(GameObject g in gos)
    {
        Vector3 vec = g.transform.position - transform.position;
        float distance = vec.magnitude;
        distances.Add(distance);
    }
    print (distances.Count);
}
```


We're at an appropriate point to iterate the order of operation when it comes to loops. When the `Update ()` function begins, we start with a new array `gos` that is fulfilled with any number of `GameObjects` with the tag "Monster." The number of monsters in the scene can change, but since the `Update ()` function starts fresh every time, the array will be updated to account for any changes from the last time the `Update ()` function was called.

Right after that, we declare a new `ArrayList` of distances. This `ArrayList` is created anew each time the `Update ()` function is called. Then we hit the `foreach()` loop. This runs its course, and using the `Add` function in the `ArrayList` class, we add in a distance. Once the loop exits, the `distances.Count` matches the `gos.Length`. It's important to note that the numbers match up. This means that the first element in `distances` matches the first element in `distances`.

This is where it's important to know when to use a `for` loop versus a `foreach` loop. We need the index number from the `for` loop's initialization. We're working with the `int i` as a number to relate one array to another. To start, we'll use any of the objects from one array and compare that value against the rest of the values.

```
float closestValue = (float)distances[0];
GameObject closestObject = gos[0];
```

If there's at least one value in the array, the rest of the function will work as expected. If there are 0 objects with the tag "Monster," then the rest of the function will produce a great deal of errors. We'll find out how to deal with exceptions in Section 7.12, but for now, keep at least one monster in the scene. To continue, we'll add in a simple `for` loop using the number of game objects we started with.

```
GameObject[] gos = GameObject.FindGameObjectsWithTag("Monster");
Debug.Log(gos.Length);
float closestValue = (float)distances [0];
GameObject closestObject = gos [0];
for (int i = 0; i < gos.Length; i++)
{
    float d = (float)distances [i];
    if (d < closestValue)
    {
        closestObject = gos [i];
        closestValue = (float)distances [i];
    }
}
```

Here we iterate through each index of the `distances[]` and `gos[]` array. We use these to compare a stored value, in this case `closestValue`, against a value stored in the `distances[]` array. We prepare the `closestValue` float variable before the `for` loop. To store this as a float, we need to convert it from an object type stored in the `distances[]` array to a float using the `(float)` cast syntax.

We need to do this once for the first `closestValue` and once for each element in the `distances[]` array before we compare the two float values. The second conversion needs to happen within the `for` loop. You can see the second conversion on the fourth line down in the above fragment.

Once we have two float values, we can compare them. If we come across a value that's smaller than the one we already had, then we assign our `closestObject` and our `closestValue` to the values found at the index we're comparing against. If we find no other value smaller than the ones we're currently comparing all of the other objects to, then we've discovered the closest object.

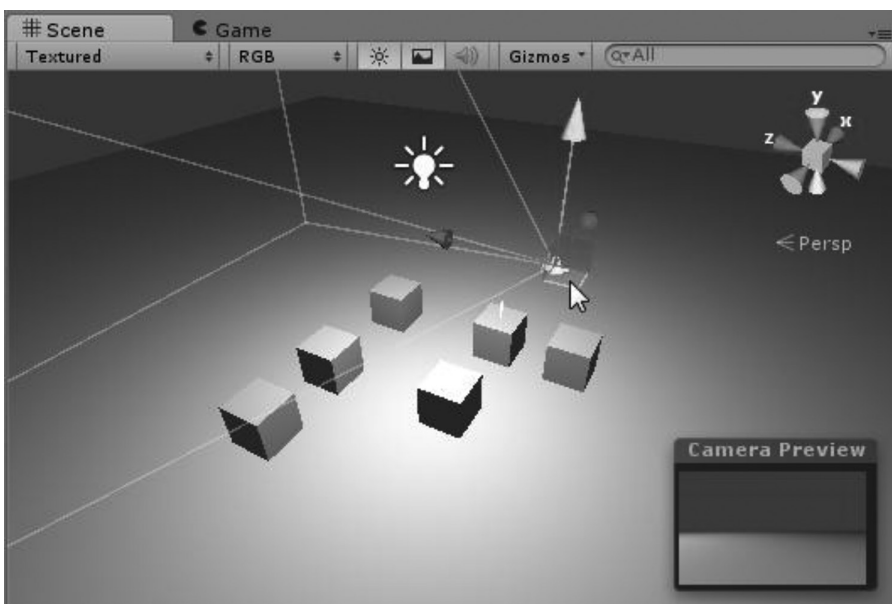
To visualize this, we can add in a handy little `Debug.Ray()` starting from the closest game object.

```
Vector3 up = new Vector3(0,1,0);
Vector3 start = closestObject.transform.position;
Debug.DrawRay(start, up);
```

We'll start the ray from the transform.position of the closestObject and shoot a small ray up from it. The completed code should look something like the following:

```
using UnityEngine;
using System.Collections;
public class Example : MonoBehaviour {
    //Use this for initialization
    void Start () {
    }
    //Update is called once per frame
    void Update () {
        GameObject[] gos = GameObject.FindGameObjectsWithTag("Monster");
        ArrayList distances = new ArrayList();
        foreach(GameObject g in gos) {
            Vector3 vec = g.transform.position - transform.position;
            float distance = vec.magnitude;
            distances.Add(distance);
        }
        float closestValue = (float)distances[0];
        GameObject closestObject = gos[0];
        for(int i = 0; i < gos.Length; i++) {
            float d = (float)distances[i];
            if (d <= closestValue) {
                closestObject = gos[i];
                closestValue = d;
            }
        }
        Vector3 up = new Vector3(0,1,0);
        Vector3 start = closestObject.transform.position;
        Debug.DrawRay(start, up);
    }
}
```

This is a handy bit of a fun code to know. As you move the camera around in the Scene editor, you'll notice that the little white ray will jump to the closest cube in the scene.



6.17.5 What We've Learned

You might be done with arrays, but there's still a few tricks left. We'll leave off here for now and move on to a different topic next. Arrays are a fundamental part of programming in general. A database is not much more than a complex array of different types of data. The principal difference is that a database has some relational connections between one element and another. Arrays without this relational data mean you'll have to keep track of the different relations.

Sorting through numbers is such an important part of general computing that many algorithms do nothing more than sort and organize numbers. Computer scientists spend years trying to speed up these sorting processes. We can't necessarily spend too much time on comparing these different methods of sorting in this book, as the field of study has filled countless volumes of text. If it were not for this importance, companies wouldn't base their entire business on storing, sorting, and organizing data.

You could imagine dealing with a few dozen monster types; each one with different sets of data, stats, and members can quickly become cumbersome to deal with. There are ways to maintain some control; that's where data structures come into play.

6.18 Out Parameter

We have been thinking that functions return a single value. Another method to get more than one value from a function is to use the `out` keyword in the parameters. The `out` keyword is usable in most situations even if you need only one return value though its use is different from how `return` is used.

```
int getSeven()
{
    return 7;
}
```

The above is a function that basically returns a 7. In use, you might see something similar to the following code fragment:

```
int seven = getSeven();
```

What limits us is the ability to return multiple values. What we'd like to be able to do but can't is something like Lua that looks like the following:

```
function returns7and11()
    return 7, 11
end
local seven, eleven = returns7and11()
```

While Lua has some clever tricks like returning multiple values, we'd have to do some extra work in C# to do the same thing. This is not elegant to say the least, and this does not mention the performance issues that will creep up if something like this is used too often.

```
ArrayList sevenEleven()
{
    ArrayList al = new ArrayList();
    al.Add(7);
    al.Add(11);
    return al;
}
void Start ()
{
    ArrayList se = sevenEleven();
    int seven = (int)se[0];
    int eleven = (int)se[1];
}
```

From the above example, we created a function called `seven()` that returned a 7. `return` which is used to provide a system to turn the function into some sort of value, which can be an array, but then we start to lose what sort of data is inside of the array. Because of this, we will run into type safety issues. For instance, we could modify the statement in `sevenEleven()` to look like the following:

```
ArrayList sevenEleven()
{
    ArrayList se = new ArrayList();
    se.Add(7);
    se.Add("eleven");//oops, not an int!
    return se;
}
```

The parser will not catch the error, until the game starts running and the `sevenEleven()` function is called and the `int eleven` tries to get data from the array. You will get this printed out to the Console panel.

```
InvalidCastException: Cannot cast from source type to destination type.
```

This type of error is not something we'd like to track down when you're using an array for use with numbers. Vague errors are rarely fun to figure out. It's best to avoid using anything like the code mentioned above.

6.18.1 A Basic Example

In the `OutParameter` project, we'll start with the `Example` component attached to the `Main Camera` as usual. The identifier defined in the argument list in the first function is named `s` who is assigned the return value from the `seven Out ()` function. So far this is a very simple use of how the `return` keyword works, which we've seen earlier. However, you might be inclined to get more than one value from a function.

```
void sevenOut(out int s)
{
    s = 7;
}
void Start ()
{
    int i;//new int
    sevenOut (out i);//i is now 7
    print(i);//prints 7 to the console
}
```

You need to do a couple of things before the `out` keyword is used. First, you need to prepare a variable to assign to the function. Then to use the function, you need to put that variable into the parameters list preceded by the same `out` keyword. When the function is executed, it looks at the variable's value inside of it and sends it back up to the parameters list with its new value. Of course, we can add more than one `out` value.

```
void goingOut(out int first, out int second, out int third)
{
    first = 1;
    second = 2;
    third = 3;
}
void Start ()
{
    int i;
    int j;
```

```

    int k;
    goingOut(out i, out j, out k);
    print(i + " " + j + " " + k);
}

```

The above code produces 1, 2, 3 in the Console. Using the function in this way allows you to produce many useful operations on incoming values as well. The additional benefit is the fact that the function can act as a pure function. This means that as long as none of the variables inside of the function rely on anything at the class level, the code can be copied and pasted or moved to any other class quite easily.

```

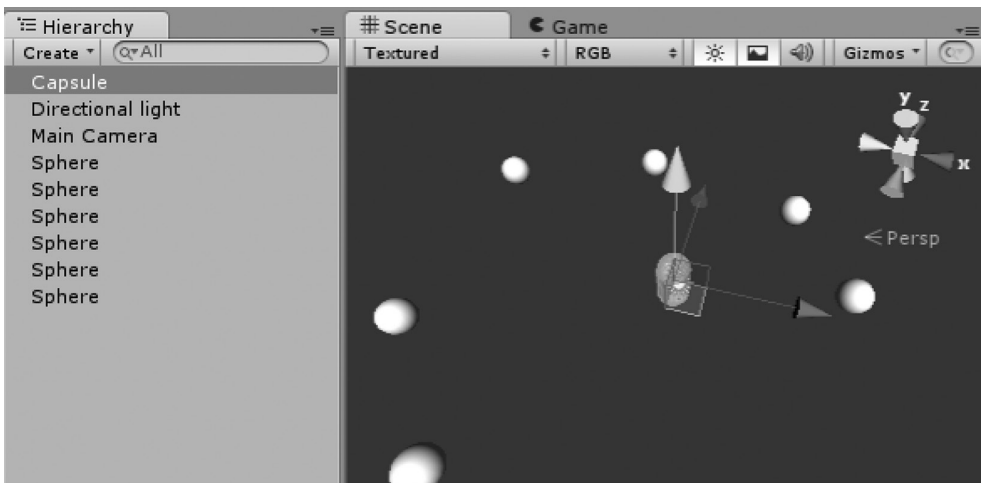
void inAndOut(int inComing, out int outGoing)
{
    outGoing = inComing * 2;
}
void Start ()
{
    int outValue = 0;
    print(outValue); //writes 0 to the console
    inAndOut(6, out outValue);
    print(outValue); //writes 12 to the console
}

```

This example shows an `inComing` value that is multiplied by 2 before it's assigned to the `outGoing` value. The variable `outValue` is initialized before the function is used and then it's assigned a new value when the function is executed. The `out` keyword is handy and allows for more tidy functions.

6.18.2 Simple Sort (Bubble Sort)

Arrays are an integral part of any game. Lists of monsters and values make for an important part of any list of data when making decisions on what to do in an environment. We'll start with a simple scene.



To the Capsule in the scene, attach a new script component named `OutParameter`; we'll add our new behavior using the `out` parameter here. We're going to create a simple `ArrayList` that will have all of the objects in the scene added to it. Then we're going to create a new `GameObject[]` array that will be sorted based on the distance to the capsule.

Sorting algorithms have a long history in computer programming. Many thesis papers have been written on the topic. However, we're going to make use of a simple sort algorithm to order an array of game objects from the shortest distance to the longest distance.

We'll start with a simple system to get an `ArrayList` of objects in the scene.

```
public GameObject[] GameObjectArray;
//Use this for initialization
void Start ()
{
    ArrayList aList = new ArrayList();
    GameObject[] gameObjects =
        (GameObject[])GameObject.FindObjectsOfType(typeof(GameObject));
    foreach(GameObject go in gameObjects)
    {
        if(go.name == "Sphere")
        {
            aList.Add(go);
        }
    }
    GameObjectArray =
        aList.ToArray(typeof(GameObject)) as GameObject[];
}
```

This creates a new `ArrayList` called `aList` using `ArrayList aList = new ArrayList();` at the beginning of the `Start ()` function. To populate this list, we need to get all of the `GameObjects` in the scene with `GameObject[] gameObjects = (GameObject[])GameObject.FindObjectsOfType(typeof(GameObject));`, which does a few different tasks in one statement.

First, it creates a new `GameObject[]` array called `gameObjects`. Then we use `GameObject.FindObjectsOfType()` and assign the function in `GameObject` the `typeof(GameObject)`; this returns a new array of every `GameObject` in the scene.

After we get all of our data, we use a `foreach` loop `foreach(GameObject go in gameObjects)` to check through all of the different objects. In the parameters of the `foreach`, we create a `GameObject go` that stores the current iteration as we go through the `gameObjects` array. Then we filter the objects and take only the ones named `Sphere` using the `if` statement `if(go.name == "Sphere")` which adds the `go` to the `aList` array if the names match.

Finally, the `aList` is assigned to `GameObjectArray` which was created at the class scope. The statement is fulfilled by `aList.ToArray()` that takes the argument `typeof(GameObject)`. We then convert the returned data to `GameObject[]` by a cast as `GameObject[]`; at the end of the statement.

Now we're ready to sort through the `GameObjectArray` based on distance.

```
void sortObjects(GameObject[] objects, out GameObject[] sortedObjects)
{
    for(int i = 0; i < objects.Length-1; i++)
    {
        Vector3 PositionA = objects[i].transform.position;
        Vector3 PositionB = objects[i+1].transform.position;
        Vector3 VectorToA = PositionA - transform.position;
        Vector3 VectorToB = PositionB - transform.position;
        float DistanceToA = VectorToA.magnitude;
        float DistanceToB = VectorToB.magnitude;
        if(DistanceToA > DistanceToB)
        {
            GameObject temp = objects[i];
            objects[i] = objects[i+1];
            objects[i+1] = temp;
        }
    }
    sortedObjects = objects;
}
```

Here we have a new function added to our class. What we do here is simple and works only if it's iterated a few times. The idea is that we need to discover the distances between two objects in the array. Compare the distances, and if one distance is greater than another, then swap the two objects in the array. As we repeat this, we rearrange the objects in the array with each pass. As this can repeat any number of times, we're doing things in a very inefficient manner, but it's getting the job done. We'll look at optimizing this loop again in this section, in which we go further into sorting algorithms.

The function starts with `GameObject[] objects`, which accepts an array of `GameObjects` called `objects`. Then we start up a `for` loop to check through each object. This begins with `for(int i = 0; i < objects.Length-1; i++)`; what is important here is that we don't want to iterate through to the last object in the array. We want to stop one short of the end. We'll see why as soon as we start going through the rest of the loop.

Next we want to get the location of the current object and the next object. This is done with `Vector3 PositionA = objects[i].transform.position;`. Then we use `Vector3 PositionB = objects[i+1].transform.position;` to get the next object in the list. You'll notice `objects[i+1]` adds 1 to `i` that will reach the end of the array. If we used `i < objects.Length`, and not `i < objects.Length-1`, we'd be looking at a place in the array that doesn't exist. If there were 10 objects in the scene, the array is `Length 10`, which means that there's no `objects[11]`. Looking for `objects[11]` results in an index out of range error.

Then we get some vectors representing the object's position minus the script's position. This is done with `Vector3 VectorToA = PositionA - transform.position;` and `Vector3 VectorToB = PositionB - transform.position;`. Then these are converted to magnitudes, which is a math function that can be done to a `Vector3`. This gives us two float values to compare `float DistanceToA = VectorToA.magnitude;` and `float DistanceToB = VectorToB.magnitude;`.

Finally, we need to compare distance and do a swap if A is greater than B.

```
if (DistanceToA > DistanceToB)
{
    GameObject temp = objects[i];
    objects[i] = objects[i+1];
    objects[i+1] = temp;
}
```

We use a `GameObject temp` to store the `object[i]` that is going to be written over with the swap. The swap begins with `objects[i] = objects[i+1];` where we will for a single statement have two copies of the same object in the array. This is because both `objects[i]` and `objects[i+1]` are holding on to the same `GameObject` in the scene. Then to finish the swap, we replace the `objects[i+1]` with `temp` that holds onto `objects[i]`.

To send the sorted array back, we use `sortedObjects = objects;` and the `out` parameter pushes the array back up out of the function into where it was called on. To check on the sorting, we'll use the following code in the `Update ()` function:

```
void Update ()
{
    sortObjects(GameObjectArray, out GameObjectArray);
    for(int i = 0; i < GameObjectArray.Length; i++)
    {
        Vector3 PositionA =
        GameObjectArray[i].transform.position;
        Debug.DrawRay(PositionA, new Vector3(0, i * 2f, 0),
        Color.red);
    }
}
```

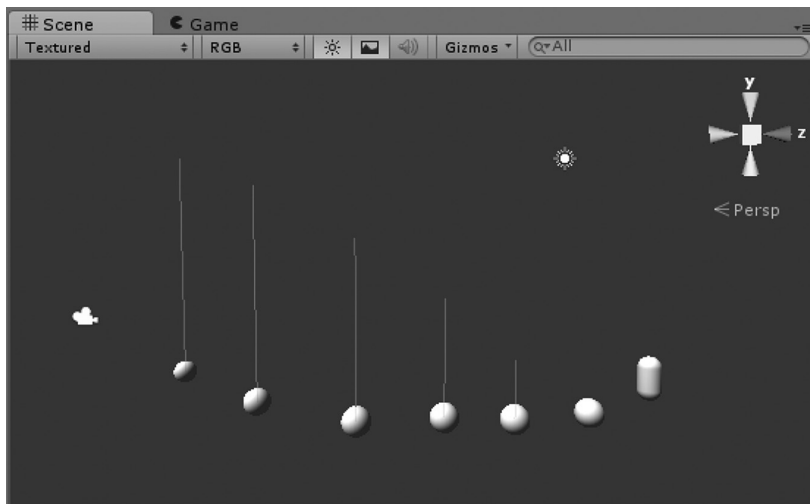
Call on the new function, `sortObjects(GameObjectArray, out GameObjectArray)`. Woah! You can do that? Yes you can! You can have both in and out parameters reference the same array variable. What this will do is that it will take in the array, sort it, and send it back out sorted! Once the array has been sorted, it can be sorted again and again.

6.18.3 Simple Sort Proof

To test our sort, we'll use a `for` loop. Each higher value in the array will hold onto a different model. The furthest model will be the last in the array and thus have the highest index in the array. The closest object will be at `GameObjectArray[0]`, and the furthest one will be at the end of the array. Therefore, if there are 10 objects in the scene, the furthest object will be at `GameObjectArray[9]`. Remember that arrays start at 0, not 1.

So we'll get the position of the current object with `Vector3 PositionA = GameObjectArray[i].transform.position;` We then use this value in a `Debug.DrawRay` function. This looks like `Debug.DrawRay(PositionA, new Vector3(0, i * 2f, 0), Color.red);` where we take the starting position `PositionA` and draw a ray up from it with `new Vector3(0, i * 2f, 0)` that tells it to draw a straight line up. The length of the line is `i * 2f`, or twice the `int i`.

In the Scene panel with the game running, you'll see the following:



As you move the Capsule with the script applied around in the scene, you'll notice the length of the lines changing based on the distance away from the capsule. This means that you can use this sort algorithm to place a priority on which object is closest and which is the furthest away based on its index in the array.

6.18.4 What We've Learned

We should leave off here for now; sorting is a very deep topic that leads into the very messy guts of computer science. The algorithm isn't complete and lacks any efficiency. It's effective, but only superficially. The only reason why this works at all is the fact that `Update()` is called many times per second.

The first time through the `sortObjects()` function, the sort is not complete. Only neighboring differences will be swapped. If the first and last objects need to be swapped, all of the objects between them must be swapped several times before the full array can be arranged properly. Proper sorting algorithms usually require several iterations. The speed at which they can do their job depends greatly on how it was implemented. Different implementations have different names.

Most of the commonly used sorting algorithms have names such as bubble sort, heap sort, or quick sort. What we did here is a simple implementation of a bubble sort. This means we looked at the first two elements in the array, and if one had some greater value than the other, we swap them.

For games such as tower defense, or anything where a character needs to shoot at the closest target and then move on to the next closest, we can use sorting to help provide the character with some more interesting behaviors. Perhaps he can target more than one object with different weapons at the same time. Maybe the character can simply shoot at the closest three objects. The great thing about writing your own game is that you can explore behaviors like this freely, but only once you understand how to use the code you're writing.

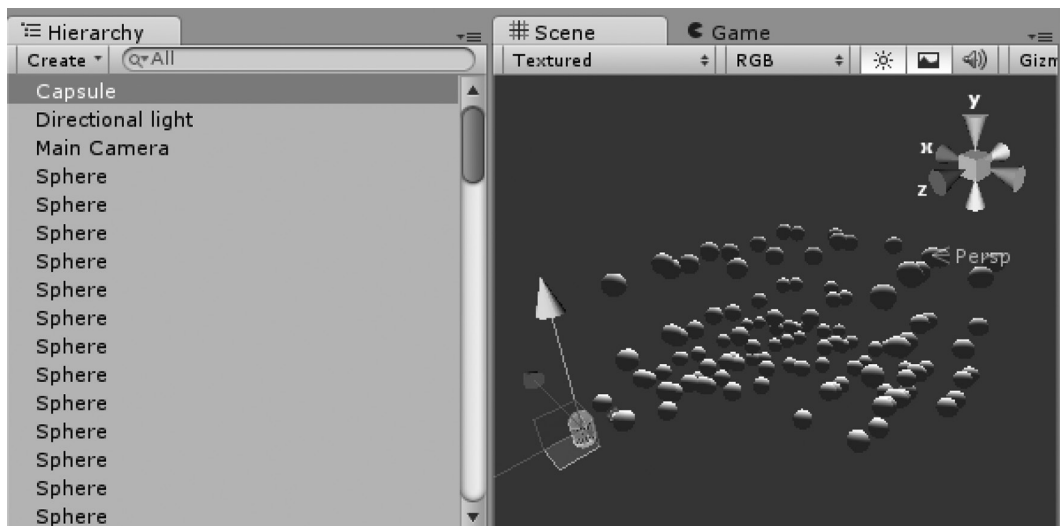
There are many different ways to accomplish the same task, and you're not limited to the `out` parameter. We could have done the following instead:

```
GameObject[] sortObjects(GameObject[] objects)
{
    for(int i = 0; i < objects.Length-1; i++)
    {
        Vector3 PositionA = objects[i].transform.position;
        Vector3 PositionB = objects[i+1].transform.position;
        Vector3 VectorToA = PositionA - transform.position;
        Vector3 VectorToB = PositionB - transform.position;
        float DistanceToA = VectorToA.magnitude;
        float DistanceToB = VectorToB.magnitude;
        if (DistanceToA > DistanceToB) {
            GameObject temp = objects[i];
            objects[i] = objects[i+1];
            objects[i+1] = temp;
        }
    }
    return objects;
}
```

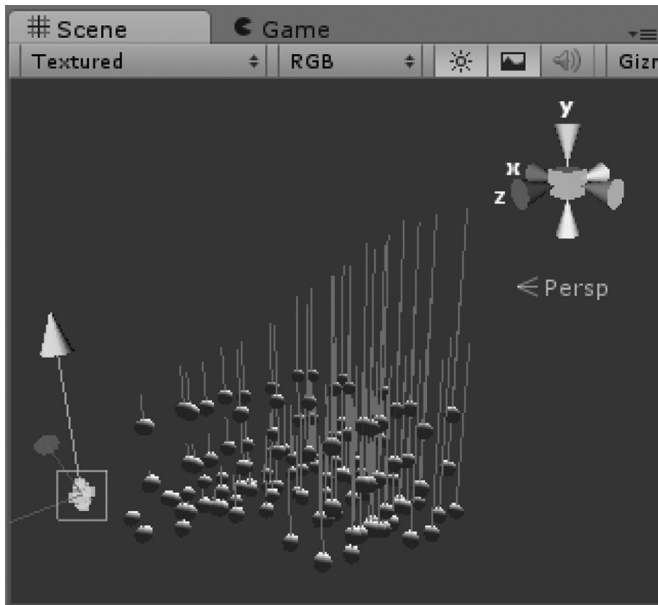
Here we simply return the objects and make sure that we use `GameObject[] sortObjects(GameObject[] objects)`; however, this isn't fun. We wouldn't have been able to see the `out` keyword in action using an interesting behavior. We're not limited by how our functions operate with C#. We're able to use its flexibility to test out various ideas, not only with game play but with the language itself.

Of course in the end, we're required to keep our code as simple as possible. The `out` keyword is best used when we can't return a single value. In the case with a single return value, the later version of `sortObjects` where we simply return the sorted array is more readable.

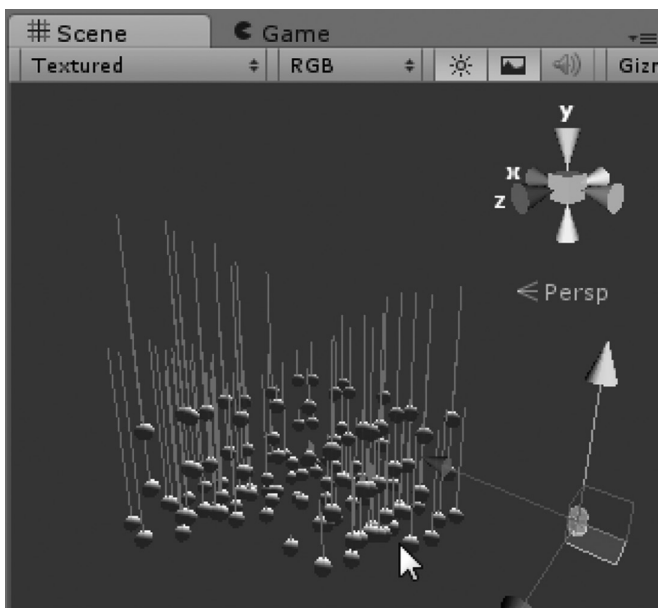
One last thing before moving on is to remember that sorting is usually quite expensive. In this example, we don't notice how long this is taking since there aren't many objects to sort through. Try the example again with several hundred objects in the scene.



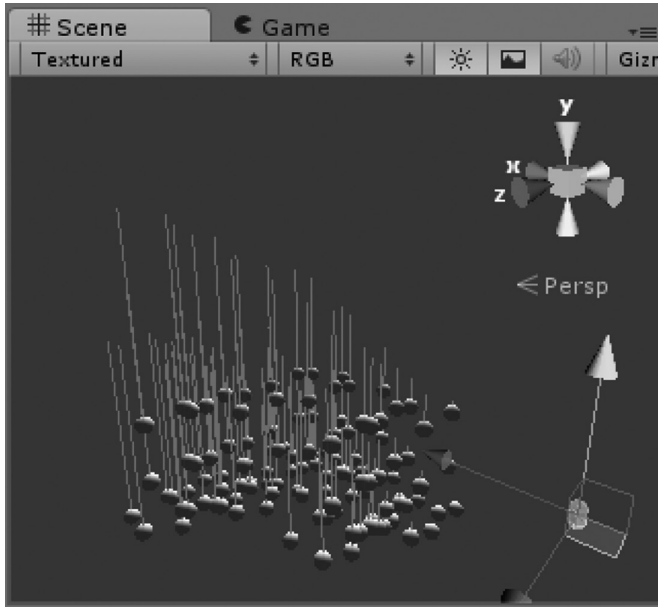
You'll be able to actually watch the red lines rearrange themselves in the array as you move the capsule around!



Here we start on one side of the group of spheres.



Quickly move the capsule with the script attached to the other side of the spheres. You'll be able to watch the red lines change as the array is sorted.



After a few moments, the array is sorted and the lines stop changing in length, indicating that the sorting is done. Since we don't have a condition to stop the sorting, the function is still being called, so this process takes up much of central processing unit (CPU) time, but if you're a nerd, it's fun to watch!

Just in case use the following adjustment to the DrawRay call:

```
Debug.DrawRay(PositionA, new Vector3(0, i * 0.1f, 0), Color.red);
```

This makes the lines shorter and easier to see.

6.19 Ref Parameter

The `ref` keyword, which is short for *reference*, works with variables in a slightly less intuitive way than we have experienced to this point. This is related to how we've been thinking about variables. In the usual case, we use the statement `x = 1;`, and now when we use `x`, we can assume we're dealing with the value it's storing.

The difference with `ref` enters when we start to manipulate the data stored under an identifier. If we use a statement like `x = 1; y = x;`, we aren't actually taking `x` and putting it into `y`. What's really going on is that `y` is looking at the value that `x` is holding onto and changes its own value to match what `x` is storing.

In memory, `y` is storing a 1 and `x` is storing a different 1. The statement `x = 1; ref y = x;` would actually mean that `y` is now looking at the identifier, not the value being stored at the location of `x`. If `ref y` is looking at the identifier and not the value, were we to modify `y`, the action is going to take place in the contents of `x`.

This may seem a bit hard to understand, and to be honest, it is. This concept is one that takes a while to get used to and that is why some programming languages are more difficult to learn than others. However, the concept becomes more apparent when we see the code in action.

6.19.1 A Basic Example

The `ref` keyword acts somewhat like the `out` keyword. In practice we'll see how they differ with the following example code fragment.

```
public int x;
void RefIncrementValue(ref int in)
{
    in += 1;
}
void Start ()
{
    print(x);//before executing the ref function
    RefIncrementValue(ref x);
    print (x);//after executing the ref function
}
```

The code above takes in the `ref` of `x` which is assigned in by the parameters of the function. The `in` argument now has a direct connection to the contents of `x`. When `in += 1` is executed, the value that `x` was holding is incremented. After the function is completed, the value of the original `x` has been modified.

Here's a bad way to do the same thing using side effects.

```
public int x;
void IncrementX()
{
    x += 1;
}
void Start ()
{
    print(x);//before calling the increment function
    IncrementX();
    print(x);//after calling the increment function
}
```

The biggest problem with this is the fact that the increment function works only on the `x` variable. What this means is that you can now reduce the number of functions that affect any particular variable. Otherwise, we'd need a function for every variable we want to modify.

```
public int x;
public int y;
void RefIncrementValue(ref int inValue)
{
    inValue += 1;
}
void Start ()
{
    print(x);
    print(y);
    RefIncrementValue(ref x);
    print(x);
    RefIncrementValue(ref y);
    print(y);
}
```

With the above code, we can easily manipulate any value sent to the `RefIncrementValue()` function. Hopefully, you're thinking to yourself, "wow, I wonder how I can use this?" and you'd be on the right track. The `ref` keyword is in particular an interesting keyword. In many cases, a function is best when there are no side effects.

6.19.2 Code Portability Side Effects

When you've started writing code, it's better to reuse as much as you can to help speed along your own development. Often you'll find simple tricks that help with various tasks. When programmers talk about reusing code, we need to think about how a function works.

If a function relies on the class it was written in, then it becomes less portable. When a function can operate on its own, it becomes more utilitarian and more portable.

```
using UnityEngine;
using System.Collections;
public class Example : MonoBehaviour {
    int a = 1;
    int b = 3;
    int ReliesOnSideEffects()
    {
        return a + b;
    }
    void Start ()
    {
        print (ReliesOnSideEffects());
    }
}
```

In this Example class, we're adding `a + b` and returning the result. For this function to operate properly if used by another class, we'd also have to copy the two variables in the class used to complete its task. Portable code shouldn't have to do this to work. We have plenty of options that work.

```
using UnityEngine;
using System.Collections;
public class Example : MonoBehaviour {
    int a = 1;
    int b = 3;
    int AddsNumbers(int x, int y) {
        return x + y;
    }
    void Start () {
        print (AddsNumbers(a, b));
    }
}
```

This version accomplishes the same task, adding `a` to `b`, but the function we're using requires no outside variables. With something like this, we're more able to copy and paste the function in a class where it's needed. We're allowed to try out many different iterations of this simple task. Once the task gets more complex, we're going to explore more options.

6.19.3 What We've Learned

In Section 6.18, we covered the `out` keyword, and here we learned the `ref` keyword. A decision must be made between which keyword to use for each situation. In most cases, the `out` keyword is going to be more useful. In the case where we create a variable with a limited scope, it's better to use `out`.

```
void someFunction(out i)
{
    i = 7;
}
```

```

void Start ()
{
    int localInt = 0;
    someFunction(out localInt);
    print(localInt);
}

```

If we were to create a variable that existed only within the `Start ()` function, for example, we'd need to use only the `out` keyword. This limits where the variable is being changed. However, if we need to use a variable with a wider scope which more functions are going to see, then we'd want to use the `ref` keyword.

```

int classInt = 0; //can be seen by both functions
void someFunction(ref i)
{
    i = 7;
}
void Start ()
{
    someFunction(ref classInt);
}
void Update ()
{
    print(classInt);
}

```

Should a variable be accessed across more than one function, then we'd need to modify the original variable, so `ref` is more useful in this situation. Of course, there are other ways to do the same task. The `someFunction()` function can return a value, or it can modify a variable using the `ref` keyword as shown in the following code fragment:

```

int classInt = 0;
int someFunction()
{
    return 7;
}
void Start ()
{
    classInt = someFunction();
}
void Update ()
{
    print (classInt);
}

```

This works perfectly well, but then we're limited to a single return value. If we had something more complex in mind, we'd need to use either `out` or `ref` should we need more than a single return value. For instance, if we needed two or three values, we might have to use something like this:

```

void someFunction(out x, out y)
{
    x = 7;
    y = 13;
}
void Start ()
{
    int firstInt = 0;
    int secondInt = 0;
}

```

```

        someFunction(out firstInt, out secondint);
        print(firstInt + " " + secondint);
    }

```

As we've seen, there are plenty of options when we begin to get familiar with how C# works. It's important to not let all of the options confuse us. At the same time, it's good to know how to do a single task in different ways, should we find one more convenient than another.

In addition, `ref` also allows you to use the variable within the function. This means that `ref` has both incoming and outgoing availability.

In the end, though, it's up to you to decide how to write and use functions and how they change values. How you decide to do this allows you to express your own ideas and methods through code. Different programming languages have different levels of expressiveness or expressivity. Basically, some languages are more strict, while others allow for more than one way to carry out any given task.

The expressivity of C# is fairly high, allowing you to try out many different methods to do the same thing. There are more common practices, but by no means should you feel limited to a commonly used syntax. If you feel that a less commonly used syntax is more helpful, by all means use it.

6.20 Type Casting Numbers

We need to know more about a variety of different data types and how they're used before we can begin to understand how and why type casting is important. At this point, we need more casting from one type of data into another. Now that we've had a chance to use a wide variety of different data types, it's time we started learning why converting from one type to another is necessary.

We casted from object to float, but why is this required? C# is a *type-safe* programming language. What this means is that your code is checked for mixed types before it's compiled into machine-readable code. This is done to prevent any errors that show up while the code is running. When converting between types that are similar, we don't usually see too many errors. In the following code sample, we demonstrate why we don't want to use an integer for setting an object's position.

```

using UnityEngine;
using System.Collections;
public class TypeConversion : MonoBehaviour
{
    public int Zmove;
    //Use this for initialization
    void Start ()
    {
    }
    //Update is called once per frame
    void Update ()
    {
        transform.position = new Vector3(0,0,Zint);
    }
}

```

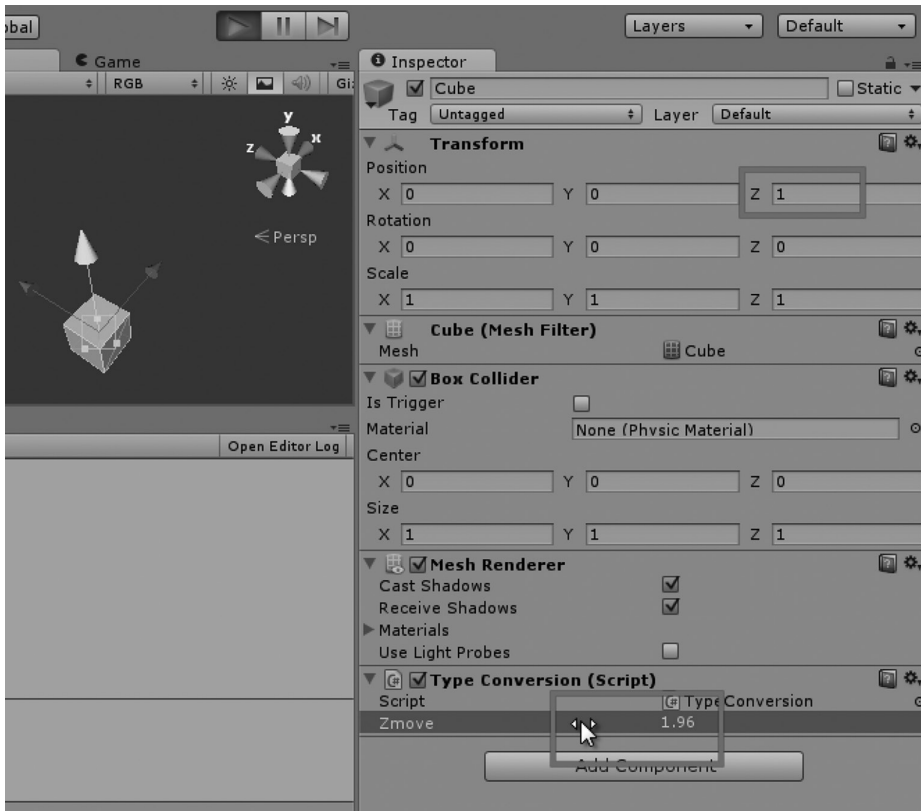
The result of this is a cube that snaps from one location to another. Assign a script named `TypeConversion.cs` to a cube in a new scene. Start the game and then use the Scene editor to observe the cube's movement when the `Zmove` is modified. The cube hops from point to point. Integers can't hold fractions. However, there are other types that do. The `float` type is a number type that stores the position of a decimal point.

Simply put, an integer is a whole number such as 1, 2, and 3. A fraction can have a value between 1 and 2, for example, 1.5625. Therefore, if we change from an `int` to a `float`, we'll get a

more smooth movement when we modify the Zmove value. However, what happens when we cast a float to an int?

```
void Update ()
{
    int Zint = (int)Zmove;
    transform.position = new Vector3(0, 0, Zint);
}
```

In this code fragment, we start with a modifiable Zmove, and then in the Update (), we create a cast from Zmove to Zint. This results in the same choppy movement we observed before. However, you may see that there's some information lost in translation.



The input from Zmove is at 1.96; however, the actual Z position of the cube in the scene is sitting at 1. This is what is meant by data loss. Everything after the decimal is cut off when moving from a float to an int. In Section 4.7, we looked at some math operators and what happens when 1 is divided by 5. The results ended up giving you different number of places after the decimal. When converting from int to float, we don't need to use a cast.

```
public float Zmove;
//Update is called once per frame
void Update ()
{
    float Zfloat = Zmove;
    transform.position = new Vector3(0, 0, Zfloat);
}
```


In particular, notice that `float Zfloat = Zmove;` didn't need an `int-to-float` cast like `(float) Zmove` even though `Zmove` was an `int`. This is because there's no data lost. A smaller data type like an `int` will fit into a `float` without any chance of data being lost. When data can convert itself from a smaller type into a larger type, this is called an *implicit cast*.

6.20.1 Number Types

This leads us to numbers having a size. In fact, so far we've used `int` and `float`, and there was even mention of a `double`. However, why do these have a size related to them and how is an `int` smaller than a `float`? This is a computer science question, and to explain this, we need to remember how computers count.

In the old days, when computers had moving parts, instructions were fed to them using a flat piece of physical media with holes punched into it. Each hole in the media represented either a 0 or a 1, which was then converted back into decimal numbers which we use in our everyday life. The primary limitation in these old computers was size, and to hold big numbers, computers needed to be physically bigger.

Each number was stored in a series of switches. In 1946, the Electronic Numerical Integrator and Computer (ENIAC) used flip-flop switches to store a number; imagine a light switch, with *on* being a 1 and *off* being a 0. Each group of switches was called an accumulator and it could store a single 10-digit number for use in a calculation. Altogether with 18,000 vacuum tubes and weighing over 25 tons, the ENIAC could store a maximum of 20 numbers at a time.

Because programming languages have such a deep-rooted history with limited data space, numbers today still have similar limitations. On your PC, you might have the memory to store many hundreds of billions of numbers, but this doesn't hold true for every device. Every year new smaller, more portable devices show up on the market, so it's good to be aware that forming bad habits of wasting data space will limit the platforms you can write software for.

6.20.2 Integers

It's important to know how computers convert 1s and 0s into decimal numbers. To explain, let's start with a 4-bit number. This would be a series of four 1s or 0s. A zero would be as simple as 0000. Each place in the number is a different value. The first place is either a 1 or a 0, the second 2 or 0, the third 4 or 0, and the fourth 8 or 0. Each whole number between 0 and 15 can be represented with these 4 bits.

To demonstrate 0101 means 2 + 8 or 10. An interesting change is to shift both 1s to the left to get 1010 that translates to 1 + 4 or 5. This is called a bit shift, or in this case a shift left. The number 15 looks like 1111. It just so happens that a 4-bit number is called a *nibble*. A nibble also happens to easily represent a single hex number.

Hex numbers are used often when dealing with assigning colors on a web page. Hex numbers range from 0 to 9 and include an additional A through F to fill in the last six digits. A color is denoted by three 8-bit numbers for red, blue, and green. An 8-bit number is called a *byte*. Each color gets a range from 0 to 255. This turns into a two-digit hex number. Therefore, 0 is 00 and 255 is FF in hex.

6.20.2.1 Signed Numbers

Today C# uses numbers starting with the *sbyte* up to the decimal. The *sbyte* is a number from -127 to 127, and the *byte* is a number from 0 to 255. The *s* in the *sbyte* is an abbreviation of the word *signed*. Therefore, you can actually call an *sbyte* a *signed byte*.

This is an important difference: Signing a number means it can be either positive or negative. However, you lose half the maximum range of the number since one of the 1s and 0s is used to tell the computer if the number is positive or negative. If we use the first bit to represent the signedness of a number in terms

of our nibble, this turns 0100 into positive 1, while 1100 turns into negative 1. Negative 3 is 1110 and negative 7 is 1111, or “negative + 1 + 2 + 4.”

sbyte	8 bits	−128 to 127
Byte	8 bits	0 to 255
Short	16 bits	−32,768 to 32,767
Unsigned short	16 bits	0 to 65,535
Int	32 bits	−2,147,483,648 to 2,147,483,647
Unsigned int	32 bits	0 to 4,294,967,295
Long	64 bits	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
Unsigned long	64 bits	0 to 18,446,744,073,709,551,615

The above table represents most of the useful varieties of integer numbers. None of these have a decimal point in them to represent a fraction. Numbers that do have a decimal are called floating point numbers. So far we’ve used `float` and `double` as well; only that it hasn’t been so obvious.

6.20.3 Floating Point

Floating point numbers have been a focus of computers for many years. Gaining floating point accuracy was the goal of many early computer scientists. Because there are an infinite possibilities of how many numbers can follow a decimal point, it’s impossible to truly represent any fraction completely using binary computing.

A common example is π , which we call pi. It’s assumed that there are an endless number of digits following 3.14. Even today computers are set loose to calculate the hundreds of billions of digits beyond the decimal point. Computers set aside some of the bits of a floating point number aside to represent where the decimal appears in the number, but even this is limited.

The first bit is usually the sign bit, setting the negative or positive range of the number. The following 8 bits is the exponent for the number called a mantissa. The remaining bits are the rest of the number appearing around the decimal point. A float value can move the decimal 38 digits in either direction, but it’s limited to the values it’s able to store.

We will go further into how numbers are actually stored in terms of 1s and 0s in Section 8.9. For now just understand that numbers can represent only a limited number of digits. Without special considerations, computers are not able to handle arbitrarily large numbers.

To cast a `float` into an `int`, you need to be more explicit. That’s why C# requires you to use the cast and you need to add the `(int)` in `int Zint = (int)Zmove;`. The `(int)` is a cast operator; or rather `(type)` acts as a converter from the type on the right to the type needed on the left.

6.20.4 What We’ve Learned

There’s still a bit left to go over with numbers, but we’ll leave off here for now. A CPU is a collection of transistors. The computer interprets the signal coming from each transistor as a switch which is either on or off.

Computers collect these transistors into groups to accomplish common tasks. A floating point unit (FPU) is a collection of transistors grouped together to assist in floating point number processing. A graphics processing unit (GPU) is a variety of different groups of transistors specifically used to compute computer graphics.

These groups of transistors are built with highly complex designs. Their organization was designed by software to accomplish their task quickly. Software was used to build the hardware designed to run more software. It all sounds a bit of catch-22, but at least we have some historic reference where all of this comes from.

6.21 Types and Operators

Most of the operators we’ve been using compare equality. This works just fine when dealing with numbers. However, we can use operators on types that are not numbers, but which comparative operators can we use?

For number types, we can use relational operators. This affords us the ability to set booleans by using greater than $>$, less than $<$, greater than or equal to \geq , and less than or equal to \leq . All of these operators produce a boolean result.

```
float a = 1.0f;
float b = 3.0f;
bool c = a > b;//false
bool d = a < b;//true
```

As you might imagine, all of the math operators such as add $+$, subtract $-$, multiply \times , divide $/$, and modulo $\%$ work on any number type data. However, we can also use the $+$ on the string type.

```
//Use this for initialization
void Start ()
{
    string a = "hello";
    string b = ", world";
    print (a + b);//prints hello, world
}
```

In the `print()` function, we use `(a + b)` to join the two strings together. However, we cannot use multiply, subtract, divide, or modulo on strings. But, we can use some of the comparative operators.

```
//Use this for initialization
void Start ()
{
    string a = "hello";
    string b = ", world";
    print (a == b);//prints False
}
```

In the `print(a == b);` function, we get `False` printed to the Console in Unity 3D. Likewise, we can use numbers and compare their values as well. We can also use the not equal `(!=)` operator on strings.

```
//Use this for initialization
void Start ()
{
    string a = "hello";
    string b = ", world";
    print (a != b);//prints True
}
```

The comparative operator `(!=)`, or not equal operator, works on strings as well as numbers. Knowing which operators takes a bit of intuition, but it's important to experiment and learn how types interact with one another.

6.21.1 GetType()

We can also compare data types. This becomes more important later on when we start creating our own data types. To know how this works, we can use the data types that are already available in C#.

```
//Use this for initialization
void Start ()
{
    int a = 7;
    string b = "hello";
    print (a.GetType() != b.GetType());//prints True
}
```

The built-in types in C# have a function called `GetType()`, which allows us to check against what type of data each variable is. In this case, an `int` is not equal to a `string`, so the Console prints out `True` when the game is started.

```
//Use this for initialization
void Start ()
{
    int a = 7;
    string b = "7";
    print (a.ToString() == b); //prints True
}
```

Even though we have two different types here, we can convert an `int` to a `string` by using the `ToString()` function in the `int` data class. Here we've got an `int 7` and we're comparing it to the `string "7"`; when the `int` is converted to a `string`, the comparison results in `True` printed to the Unity's Console panel when run. We can do some more simple type conversions to confirm some basic concepts.

```
//Use this for initialization
void Start ()
{
    int a = 7;
    float b = 7.0f;
    print (a == b); //prints True
}
```

6.21.2 More Type Casting

We can compare `ints` and `floats` without conversion. This works because C# will convert the `int` to a `float` implicitly before the comparison is made. However, C# does know that they are different types.

```
//Use this for initialization
void Start ()
{
    int a = 7;
    float b = 7.9f;
    print (a == b); //prints False
    print (a == (int)b); //prints True
}
```

We can force the cast using the `(int)` on `b` that was assigned `7.9`, which we know is clearly greater than `7`. However, when we cast `float 7.9` to an `int`, we lose the numbers following the decimal. This makes the cast version of the comparison true.

Can we cast a `string` to a number data type?

```
//Use this for initialization
void Start ()
{
    int a = 7;
    string b = "7";
    print (a == (int)b); //doesn't work
}
```

No, there's no built-in method to allow us to change a `string` data type into an `int` or any other number type. Therefore, type conversion has limitations. However, don't let this get in your way. Comparing values should be used with like types to begin with. What else can we compare?

```
GameObject a = GameObject.CreatePrimitive(PrimitiveType.Capsule);
GameObject b = GameObject.CreatePrimitive(PrimitiveType.Capsule);
print (a == b); //False?
```

What is being compared here? Well, these are actually two different objects in the scene. Even though they share a good number of attributes, they are not the same object. A more clear way to compare two instances of a game object is to use the following fragment:

```
GameObject a = GameObject.CreatePrimitive(PrimitiveType.Capsule);
GameObject b = GameObject.CreatePrimitive(PrimitiveType.Capsule);
print (a.GetInstanceID());
print (b.GetInstanceID());
print (a.GetInstanceID() == b.GetInstanceID()); //False
```

Here we're being more specific as to what we're comparing. If every object in the scene has a unique instance ID, then we're going to more clearly debug and test what's being compared when we need to check objects in the scene for matches. When we compare objects in the scene, which don't have a clear comparison, there's usually a method to allow us to make a more readable difference between objects.

```
GameObject a = GameObject.CreatePrimitive(PrimitiveType.Capsule);
GameObject b = a;
print (a == b); //True
```

In this case, yes, they are the same object, so the behavior is correct. However, again we should do the following to ensure that we're comparing something more easily debugged.

```
void Start () {
    int a = GameObject.CreatePrimitive(PrimitiveType.Capsule).
GetInstanceID();
    int b = a;
    print (a);
    print (b);
    print (a == b); //True
}
```

The above code produces the following output in the Console panel:

```
-3474
UnityEngine.MonoBehaviour:print(Object)
TypeCasting:Start () (at Assets/TypeCasting.cs:10)
-3474
UnityEngine.MonoBehaviour:print(Object)
TypeCasting:Start () (at Assets/TypeCasting.cs:11)
True
UnityEngine.MonoBehaviour:print(Object)
TypeCasting:Start () (at Assets/TypeCasting.cs:12)
```

Now we're looking at something that makes more sense. It's always important to understand what our code is doing. In some cases, it's unavoidable, but there's nearly always a method that's available to give us data that we can read.

In the case where we are looking to see if they are `GameObjects`, we'd need to check for type. This means that we can make sure that we're looking for `GameObjects`. To do this, we need to check if one object's type matches another object's type.

```

void Start ()
{
    GameObject a = GameObject.CreatePrimitive(PrimitiveType.Capsule);
    GameObject b = GameObject.CreatePrimitive(PrimitiveType.Capsule);
    print (a);
    print (b);
    print (a.GetType() == b.GetType()); //True
}

```

There is a very important difference here. We're starting with two objects of type `GameObject`. This allows us many more options than an `int` type. `GameObjects` contain many more functions that allow for many more complicated comparisons. `GameObject a` and `GameObject b` can be checked for their type. Therefore, `a.GetType()` and `b.GetType()` will return the same `GameObject` type. The significant difference here is the fact that the type of data in memory has a type, that is to say, there's a form or shape that each data has in the computer's memory.

Therefore, `GameObjects` have an associated type; this also means that we can make our own types. So far with every class we write, we are creating new types of data. To see this concept in operation, it's best to write a couple of simple classes to test. Let's look at what other types are for a moment.

```

int c = 1;
Debug.Log(c.GetType());

```

The above code sends the following output to the Console panel:

```

System.Int32
UnityEngine.Debug:Log(Object)
TypeCasting:Start () (at Assets/TypeCasting.cs:14)

```

We've discovered that `c` is a `System.Int32`, so what are `a` and `b`?

```

UnityEngine.GameObject
UnityEngine.Debug:Log(Object)
TypeCasting:Start () (at Assets/TypeCasting.cs:14)

```

The `GameObject a` is a `UnityEngine.GameObject`, which is the same as `b`. Because they are of the same type of data, the check from `a == b` is true. We should investigate a few other types. Let's see the following code fragment:

```

float c = 1.0f;
Debug.Log(c.GetType());

```

With the above code, we get the following debug information sent to the Console panel :

```

System.Single
UnityEngine.Debug:Log(Object)
TypeCasting:Start () (at Assets/TypeCasting.cs:14)

```

Even though the type we're using is called `float`, C# interprets it as a `System.Single`.

```

System.Single c = 1.0f;
Debug.Log(c.GetType());

```

6.21.3 Type Aliasing

We could use the above code in place of `float`, but `float` is the naming convention that started with C#. Therefore, we'll give in to convention and use `float` rather than `System.Single`, even though both are acceptable by C#. The word `float` came from someone early on in C# who decided to add a type alias. We can add our own type aliases with a `using` statement.

```
using UnityEngine;
using System.Collections;
using MyOwnIntType = System.Int16;
```

Along with the directives we're used to seeing at the top of our classes, we can add in another `using` directive. Adding `using MyOwnIntType`, we can assign it to `System.Int16`; which turns our identifier `MyOwnIntType` into a `System.Int16`.

```
MyOwnInt c = 1;
Debug.Log(c.GetType());
```

Using the above fragment produces the following Console output:

```
System.Int16
UnityEngine.Debug:Log(Object)
TypeCasting:Start () (at Assets/TypeCasting.cs:15)
```

Therefore, the `float` was assigned to `System.Single` somewhere in the depths of Unity 3D. It's unlikely that we're going to find out where the convention of using `float` is a `Single`, but it's important to know how the keyword was assigned. The fact is that the `float` keyword, even though it has become superfluous in programming, is still just the alias given to a `System.Single`.

Likewise, `int` or `double` and some other keywords we commonly use are just aliases to simplify our code. Other systems have aliased the system types with words such as `bigint`, `smallint`, and `tinyint` as mappings to different numbers of bytes held for the integer value.

Therefore, now that we know a bit where types come from and how to convert one type into another, it's time to put this to some use.

6.21.4 Boxing and Unboxing

Boxing is the term that programmers use when a generic container is used to assign a multitude of types. Once it's discovered what has been "put in a box," we can decide what to do next. When the data is pulled from the variable, you "unbox" the data once you know what it is and what to do with it.

For instance, we can get all of the objects in a scene and put them into an array `GameObject[] allObjects`. This includes the Main Camera and any lights in the scene as well as any monsters, players, and environment objects. If you look through the list for anything that has a `Zombie()` class attached, you can then take the closest `GameObject` with a `Zombie` component and carry out any operations involving that particular object. This unboxes that one `Zombie` from the array and allows you to interact with it.

6.22 Operator Overloading

What happens if you'd like to do something specific when you add one zombie to another? Operators, like anything else it seems in C#, can have additional functionality added to them. When it comes to dealing with new classes, we need to manage data using methods which we feel fit. In most cases, operator overloading should be considered a bit like voodoo.

```
class Zombie
{
}
void Start ()
{
    Zombie a = new Zombie();
    Zombie b = new Zombie();
    Zombie c = a + b;
}
```

Therefore, this might not be a common scenario, but it's likely that we might want to tell our game what to do in case we would like to add two zombies together to make a third. Say for instance, our zombie had a damage number.

```
class Zombie
{
    public int damage = 10;
}
```

With this value, we might want to add zombie's damages together if we add one zombie to another. In that case, if we had zombie a and b in a scene and a designer decided that it would be cool for them to merge into a bigger zombie, we could go through a process of adding one zombie to another through a function. Intuitively, we might want to use an operator instead.

```
class Zombie
{
    public int damage = 10;
    public static Zombie operator + (Zombie a, Zombie b)
    {
    }
}
```

We start by adding a new function using the pattern we've got used to with accessor property identifier. This time we follow with a type, in this case `Zombie` followed by `operator +` to tell C# that we're intending to overload the `+` operator. In our argument list that we use, enter the left and right sides of the operator we're overloading.

The problem with operator overloading between classes of a nonnumerical value is the fact that there is a loss in readability. For purposes of learning how to do this, we'll ignore this fact; however, do consider operator overloading something best avoided. Most of the time, the operators used on numbers are sufficient and should be left alone. When working on classes you've created, the common math operators should be left to operating on numbers.

6.22.1 A Basic Example

With the `OperatorOverloading` project, open the `Example` component on the `Main Camera` and open that in `MonoDevelop`.


```

public static Zombie operator + (Zombie a, Zombie b)
{
    Zombie z = new Zombie();
    int powerUp = a.damage + b.damage;
    z.damage = powerUp;
    return z;
}

```

Therefore, our designer decides that we need to add `Zombie a`'s damage to `Zombie b`'s damage to produce `Zombie c`'s damage. We add the required statements to fulfill the request into the `+` operator overloading function. Start by creating an object to return, which fulfills the requirement of the data type inferred after the `public static` keywords.

Therefore, once we're done adding `a`'s damage to `b`'s damage, we can set our new zombie's `z.damage` to the added result. Once we're done with all of our adding behaviors, we can `return z` to the statement where it was called from.

In this case, `Zombie c = a + b`; where variable `c` now gets the result of our overloaded `+` operator.

```

void Start ()
{
    Zombie a = new Zombie();
    Zombie b = new Zombie();
    Debug.Log(a.damage);
    Debug.Log(b.damage);
    Zombie c = a + b;
    Debug.Log(c.damage);
}

```

With the above code, we get the following debug information sent to the Console panel:

```

10
UnityEngine.Debug:Log(Object)
OperatorOverloading:Start () (at Assets/OperatorOverloading.cs:18)
10
UnityEngine.Debug:Log(Object)
OperatorOverloading:Start () (at Assets/OperatorOverloading.cs:19)
20
UnityEngine.Debug:Log(Object)
OperatorOverloading:Start () (at Assets/OperatorOverloading.cs:21)

```

To add some more meaningful context to apply operator overloading, we'll look at a more interesting example.

```

class Supplies
{
    public int bandages;
    public int ammunition;
    public float weight;
    public Supplies(int size)
    {
        bandages = size;
        ammunition = size * 2;
        weight = bandages * 0.2f + ammunition * 0.7f;
    }
}

```

If we have a supply box that is a standard item in our game, we might want to have a simple method to add one supply box to another. In our code, this becomes something pretty interesting when we add a constructor that has a bit of math involved to automatically calculate the weight based on the items in the supply box. Here we're adding some arbitrary weight from the bandages and ammunition to the weight of the supply box.

After the Supplies constructor is set up, we're going to add in a behavior for the + operator.

```
public static Supplies operator + (Supplies a, Supplies b)
{
    Supplies s = new Supplies(0);
    int sBandages = a.bandages + b.bandages;
    int sAmmunition = a.ammunition + b.ammunition;
    float sWeight = a.weight + b.weight;
    s.bandages = sBandages;
    s.ammunition = sAmmunition;
    s.weight = sWeight;
    return s;
}
```

This simply takes the combination of a's and b's contents and adds them together and sets a new Supplies object to the combined values. At the end, we return the new object.

```
void Start ()
{
    Supplies supplyA = new Supplies(3);
    Supplies supplyB = new Supplies(9);
    Supplies combinedAB = supplyA + supplyB;
    Debug.Log(combinedAB.weight);
}
```

If we create two copies of the Supplies object and add them together, we get a final result that reacts how you might expect. This works as you might expect with any other math operator. The following code also produces an expected result:

```
Supplies supplyA = new Supplies(3);
Supplies supplyB = new Supplies(9);
Supplies combinedAB = supplyA + supplyB;
Debug.Log(combinedAB.weight);
Supplies abc = supplyA + supplyB + combinedAB;
Debug.Log(abc.weight);
```

The final weight of supplyA + supplyB with another supplyA and another supplyB is 38.4 units of weight. The overload is required to be static because this is a function that acts on the Supplies object at a class-wide scope. Should we need to go lower into the structure to give us the ability to add bandage C = bandageA+bandageB;, we'd need to make a class for that with an operator overload of its own.

6.22.2 Overloading *

We're not restricted to overloading a class by a class. As in the above example, we're adding supplies to more supplies. We can overload our Supplies operator * with a number.

```
public static Supplies operator * (Supplies a, int b) {
    Supplies s = new Supplies(0);
    int sBandages = a.bandages * b;
    int sAmmunition = a.ammunition * b;
    float sWeight = a.weight * b;
```

```

        s.bandages = sBandages;
        s.ammunition = sAmmunition;
        s.weight = sWeight;
        return s;
    }

```

In the above example, we take supplies and multiply it by `int b`. To make use of the new operator, our code would have to look something like the following:

```

Supplies sm = new Supplies(5);
Debug.Log(sm.weight);
sm = sm * 3;
Debug.Log(sm.weight);

```

In this case, our `sm = sm * 3;` takes the original and then multiplies it by 3 and accepts the new value for itself. The log for this sends 8 followed by 24 to the Console panel. This also means that we can use the following modification to get 32 printed to the Console panel: `sm = sm + sm * 3;`

In addition to the previous examples using `+` and `*`, we can also overload the `true` and `false` keywords. For operators such as `>` and `<`, we can test if one zombie is more dangerous than another.

6.22.3 Overloading <

Operators that return a `bool` value can also be overridden. The operators `<` and `>` can be overridden to return a `true` or `false` value. The code fragment that accomplishes this is as follows:

```

public static bool operator < (Zombie a, Zombie b)
{
    if (a.damage < b.damage)
    {
        return true;
    } else {
        return false;
    }
}

```

Before we can use this, Unity 3D will likely remind us that we're forgetting something with the following error message:

```

Assets/OperatorOverloading.cs(46,36): error CS0216: The operator
'OperatorOverloading.Zombie.operator <(OperatorOverloading.Zombie,
OperatorOverloading.Zombie)' requires a matching operator '>' to also be defined

```

To comply we'll add in the opposite version of the less than `<` operator overload. Using this we would compare the two zombies in our `Start ()` function with the following statement:

```

public static bool operator >(Zombie a, Zombie b)
{
    if (a.damage > b.damage)
    {
        return true;
    } else {
        return false;
    }
}

```

A quick copy/paste and a replacement of less than < to greater than > is all it takes to allow us to use the less than < and greater than > overload. In our `Start ()` function, we can use the following few statements to check if our overload is working.

```
Zombie a = new Zombie();
Zombie b = new Zombie();
a.damage = 9;
if (a < b)
{
    Debug.Log("a has less damage!");
}
```

If the default initialization of the zombie's health is set to 10, then `Zombie a` does indeed have less damage left than `Zombie b`. In this `if` statement, we do get the "a has less damage!" printed to our Console panel. Does this mean that he's more dead than `Zombie b`? I guess this question is left to the designer on what to do with this information.

However you decide to use overloaded operators is left to how you've decided to work with your classes. In many cases, if you find yourself comparing a few variables between two objects of the same class to execute some block of code, then you should consider using an operator overload. Likewise, if you find yourself adding multiple things together using the `+` for multiple attributes, then that's another candidate for an overload.

6.22.4 What We've Learned

With all of what we just learned, you should be able to see how overriding less than < and greater than > can be useful in terms of sorting. You'd be able to make a sort based on specific properties between classes. Comparing distances and threat levels can be a very interesting behavior.

Some of what we're doing here is a bit awkward and mostly contrived. In general, greater than > and less than < should be used to compare more numeric values. In the above code, comparing two zombies directly is misleading. Are we comparing size, armor, and number of brains eaten? It's impossible to know with the operator alone. In this case, `a.armor > b.armor` would make far more sense.

It's important to always leave the code in a readable state. Simply comparing two classes against one another makes it impossible to know exactly what it is we're comparing. This also goes for any other math operator. In general, math operations are best suited for numbers, not zombies.

6.23 Controlling Inheritance

Dealing with various uses for inheritance is key to using any OOP. The key features for OOP include keeping control over each inherited object. Some features should not be directly overridden. It's up to you to decide on how this is maintained, but once you're working in a group, this control is going to be lost.

If someone needs to add behaviors to a class, but you require something like movement to remain consistent between all classes, then you'll have to implement various systems to keep people from breaking your code. Sealed class, often referred to as a final class, cannot be inherited from. This sort of data ends the line as far as inheritance is concerned. This class is used when you want to finalize a class and prevent any further inheritance.

Class inheritance works by layering changes on top of a base class. Each layer can either add new functions or modify functions that are inherited from its base. In general, all of the functions found in the base class are going to exist in any class inheriting from it.

The common metaphor used to describe inheritance is a family tree; however, what more closely describes class inheritance is biological classification or scientific taxonomy. This is a system to organize and categorize organisms into groups such as species, family, and class. Each category is a taxonomic rank.

To biologists the species *Agaricus bisporus*, the average button mushroom, belongs to the genus *Agaricus* of the family Agaricaceae of the order Agaricales in the phylum Basidiomycota in the kingdom Fungi. That's quite a mouthful, even if you don't like them on pizza.

In a very similar way, when you create a `Zombie : MonoBehaviour`, you are creating a class `Zombie` based on `MonoBehaviour`, `Behaviour`, `Component`, `Object`, and `object`. Notice that `Object` with an uppercase `O` is based on `object` with a lowercase `o`.

In this ranking and categorizing of mushrooms and zombies, the hierarchy that is formed means there are common traits shared among objects. A death cap mushroom is a fungus like a button mushroom. Both share common traits, reproduce by spreading spores, and have similar structures made of chitin. One of these makes pizza deadly, and the other just makes pizza yummy.

In a similar way, all of the classes we've written based on `MonoBehaviour` share common features, for example, `Start ()` and `Update ()`. From `Behavior`, they inherit `enabled`. From `Component`, they inherit various messaging and component properties. `Object` gives them the ability to instantiate or destroy themselves and find one another. Finally, `object` gives them all `ToString()`.

Nature might take aeons to create a new class of life. For a programmer, a new class in Unity 3D just takes a few mouse clicks. Programmers also have the ability to override how inherited functions behave. However, unexpected behaviors usually result in bugs. To control this, it's best to put preventative measures around inherited functions through encapsulation and special controls.

6.23.1 Sealed

The `sealed` prefix to a function is one of the systems that allows you to control how a function is inherited. Once you've written a well-constructed class, it's a good idea to go back and prevent anyone from breaking it. The `sealed` keyword prevents a class member from being misused by a child inheriting the member.

One commonly written class is a `timer`. The `timer` should have some features such as setting the length of time for the timer, asking for how much time is left, pausing and restarting the timer, and events for when the timer ends and starts.

If you want all of the timers to behave exactly the same, then you'll have to prevent anyone from making any unnecessary modifications to the `timer` class. This is why they need to be sealed. A sealed class is meant to prevent any further modifications from the point where it has been sealed.

6.23.1.1 A Basic Example

Adding in the `sealed` keyword before the `class` keyword does the trick. Once this is added, no class is allowed to inherit from the class. In the `Sealed` project, let's look at the `FinalizedObject` class attached to the Main Camera.

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

Once in place, you can create another class in an attempt to inherit from it. The following class tries to inherit from the sealed class:

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

This, of course, produces the following error in the Unity's Console panel:

```
Assets/InheritFromSealed.cs(4,14): error CS0509: 'InheritFromSealed': cannot
derive from sealed type 'FinalizedObject'
```

It would be easy to remove the sealed keyword from the class and open it up for basing more classes on it. However, this does inform you that you are indeed making a change that isn't intended. This behavior is meant to limit the amount of tampering of a class written to do something specific. Usually, classes written with sealed are for very basic and very widely available functions that shouldn't require any other class to inherit from it.

The key notion is that a class that is widely available should be stable and unchanging. Reliability is key for preventing too many bugs from creeping into your code. All of the different accessibility modifications such as sealed are intended to help prevent unexpected behavior. Of course, this doesn't by any stretch of the imagination mean that it prevents bugs completely.

6.23.2 Abstract

Related to how classes inherit behavior from their parent class, the abstract keyword is another clever keyword. The abstract keyword is used to tell inheriting classes that there's a function they need to implement. If you forget to implement a function that was marked as abstract, Unity 3D will remind you and throw an error.

6.23.2.1 A Basic Example

To inform another programmer, or yourself that you need to implement a specific function, the abstract keyword is used. Setting up a class to inherit from helps you plan your classes and how the classes are intended to be used.

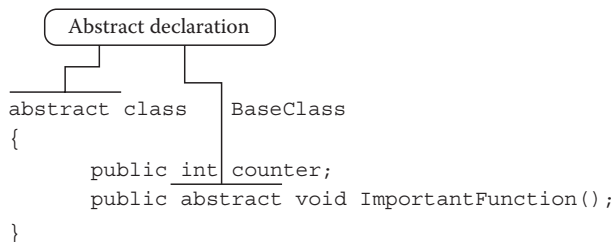
```
using UnityEngine;
using System.Collections;
public class Abstraction : MonoBehaviour
{
    abstract class BaseClass
    {
        public int Counter;
        public abstract void ImportantFunction();
    }
    class ChildClass : BaseClass
    {
        public override void ImportantFunction()
```

```

        {
            Counter++;
            Debug.Log(Counter);
        }
    }
    void Start () {
        ChildClass c = new ChildClass();
        c.ImportantFunction();
        c.ImportantFunction();
        c.ImportantFunction();
        c.ImportantFunction();
    }
}

```

The example above creates two nested classes in the `Abstraction.cs` class based on `MonoBehaviour`. The first class is called `BaseClass`. This provides two things: an `int` to count with and an abstract function. The `int` is called `counter` and the function has been named `ImportantFunction()`. A class with an abstract function in it is required to be declared as `abstract` as well.



The `abstract` keyword implies that the function declared is a stub. This tells the programmer who is inheriting from the `BaseClass` that there's an important function which he or she has to implement. The implementation, however, is left up to the author of the child class.

For instance, if we're making monsters, we'd have a monster base class and some sort of `attackHuman()` function that all monsters will need to do. Vampires, zombies, and werewolves all need to do some `attackHuman()` function. How and what that function entails differs between each monster. How this is implemented depends on the monster and who is writing the function.

When a function is preceded by the `abstract` keyword, the function cannot have a body. This might look like the following:

```

abstract class BaseClass
{
    public int Counter;
    public abstract void ImportantFunction()
    {
        //no code allowed
    }
}

```

Should there be any code at all in the `abstract ImportantFunction()`, you'll get the following error:

```

Assets/Abstraction.cs(9,38): error CS0500:
'Abstraction.BaseClass.ImportantFunction()' cannot declare a body because it
is marked abstract

```

This only means to inform you that you cannot implement an abstract function. You must rely on a child class making the implementation. The intent here is to provide a clear and specific structure for all classes inheriting from the `BaseClass`.

```
class ChildClass : BaseClass
{
    public override void ImportantFunction()
    {
        Counter++;
        Debug.Log(Counter);
    }
}
```

The function in `ChildClass` that is inheriting from `BaseClass` needs to use `override` to inform C# that it is indeed writing the implementation for the important function. The result of this `ChildClass` is to increment the `Counter` up by 1, with each call to `ImportantFunction()` after a `Debug.Log()` call. When `Start ()` is called in the `Abstraction.cs` class, we make an instance of the `ChildClass` and call the important function a few times to get 0, 1, 2, and 3 printed to the Console panel.

To continue we should make a sibling class similar to `ChildClass` with a different implementation of the `ImportantFunction()` call.

```
class SiblingClass : BaseClass
{
    public override void ImportantFunction()
    {
        Counter-- ;
        Debug.Log(Counter);
    }
}
```

This time we decrement `Counter` by 1 each time the function is called. Therefore, if the following lines are added to the `Start ()` function, we can get some set of numbers going up and another set going down.

```
void Start () {
    ChildClass c = new ChildClass();
    c.ImportantFunction();
    c.ImportantFunction();
    c.ImportantFunction();
    c.ImportantFunction();
    SiblingClass s = new SiblingClass();
    s.ImportantFunction();
    s.ImportantFunction();
    s.ImportantFunction();
    s.ImportantFunction();
}
```


This results in the following Console output:



If the classes are considered final, they can be sealed.

```
sealed class ChildClass : BaseClass
{
    public override void ImportantFunction()
    {
        Counter++;
        Debug.Log(Counter);
    }
}
```

This means that the `ChildClass` can no longer be derived from. However, this doesn't apply to the `SiblingClass` that hasn't been sealed. The `Sibling` is considered a branch from `BaseClass`. One interesting test case we can observe is using `sealed` on an abstract class. Of course, but adding `sealed` to the `BaseClass` looks like the following:

```
abstract sealed class BaseClass
{
    public int Counter;
    public abstract void ImportantFunction();
}
```

This results in both `ChildClass` and `SiblingClass` breaking with the following error:

```
Assets/Abstraction.cs(12,22): error CS0709: 'Abstraction.ChildClass': Cannot
derive from static class 'Abstraction.BaseClass'
Assets/Abstraction.cs(21,15): error CS0709: 'Abstraction.SiblingClass':
Cannot derive from static class 'Abstraction.BaseClass'
```

The `sealed` keyword really means it. Once you've sealed a class, any classes deriving from it will immediately break. Keep this in mind when you start structuring how classes derive from one another. As another interesting experiment, what happens if we want to create the instance of the base class?

```
void Start ()
{
    BaseClass b = new BaseClass();
}
```

We can create instances of each child class based on `BaseClass()`, but we cannot use the `BaseClass` itself. We get an error that looks like the following:

```
Assets/Abstraction.cs(34,46): error CS0144: Cannot create an instance of the
abstract class or interface 'Abstraction.BaseClass'
```

C# is informing us that it cannot create an instance of the abstract class or interface `BaseClass`. An interface works in a similar way to an abstract class, but with a few more restrictions. All of these mechanisms are there to prevent problematic issues that have come up with different languages. We'll get into the hows and whys of the interface in Section 7.6.4.

6.23.3 Abstract: Abstract

The topic should be expanded to look at additional abstract classes inheriting from abstract classes. These can be used to add more diversity to a simple base class. By creating a branch that is also abstract, you can add more data fields and functions to provide a variation on the first idea.

```
abstract class BaseClass
{
    public int Counter;
    public abstract void ImportantFunction();
}
abstract class SecondaryClass : BaseClass
{
    public int Limit;
    public abstract bool AtLimit();
    public abstract void SetLimit(int l);
}
```

Here we have the `abstract class SecondaryClass` that is based on the `abstract class BaseClass`. We are allowed to add some additional fields and functions to this secondary class. The advantage here is that we don't need to make any modifications to the `BaseClass`, as there might be other classes relying on its stability. Keeping code intact is important, and diving into base classes and making changes could cause problems to ripple through the rest of the project.

Changing inheriting classes is simple; change the `: BaseClass` to `: SecondaryClass` and C# will tell you what you need to fix.

```
sealed class ChildClass : SecondaryClass
{
    public override void ImportantFunction()
```

```

    {
        Counter++;
        Debug.Log(Counter);
    }
}

```

With the inheritance changed here, we get a simple warning:

```

Assets/Abstraction.cs(18,22): error CS0534: 'Abstraction.ChildClass' does not
implement inherited abstract member 'Abstraction.SecondaryClass.AtLimit()'
Assets/Abstraction.cs(19,22): error CS0534: 'Abstraction.ChildClass' does not
implement inherited abstract member 'Abstraction.SecondaryClass.SetLimit(int)'

```

We need to implement the abstract member `AtLimit()`; and `SetLimit(int)`; which is great; this means that we get to keep our current implementation of `ImportantFunction()` and we need to add only the missing functions.

```

sealed class ChildClass : SecondaryClass
{
    public override void ImportantFunction()
    {
        Counter++;
        Debug.Log(Counter);
    }
    public override bool AtLimit()
    {
        return Counter >= Limit;
    }
    public override void SetLimit(int l)
    {
        Limit = l;
    }
}

```

Here we've written some simple implementations of `AtLimit()`; and `SetLimit(int)`; for the `ChildClass`. To check if we are indeed at the limit, we use `return Counter >= Limit;`. This works only because `Limit` is a member of the `SecondaryClass` and `Counter` is found in `BaseClass`. The inheritance allows the `ChildClass` to use both of these data fields together.

Next we have `SetLimit(int)`; that accepts a number to set the limit. In the `Start()` function in the `Abstraction.cs` class, we can add the following code to test our modifications to `ChildClass`:

```

ChildClass c = new ChildClass();
c.SetLimit(2);
c.ImportantFunction();//prints 1
Debug.Log(c.AtLimit());//prints False
c.ImportantFunction();//prints 2
Debug.Log(c.AtLimit());//prints True

```

This sends the following output to the Console panel in Unity 3D:

```

1
UnityEngine.Debug:Log(Object)
ChildClass:ImportantFunction() (at Assets/Abstraction.cs:23)
Abstraction:Start () (at Assets/Abstraction.cs:52)
False
UnityEngine.Debug:Log(Object)
Abstraction:Start () (at Assets/Abstraction.cs:53)

```

```

2
UnityEngine.Debug.Log(Object)
ChildClass:ImportantFunction() (at Assets/Abstraction.cs:23)
Abstraction:Start () (at Assets/Abstraction.cs:54)
True
UnityEngine.Debug.Log(Object)
Abstraction:Start () (at Assets/Abstraction.cs:55)

```

6.23.4 Putting This to Use

We'll start with a construct that's used fairly often for many different game systems.

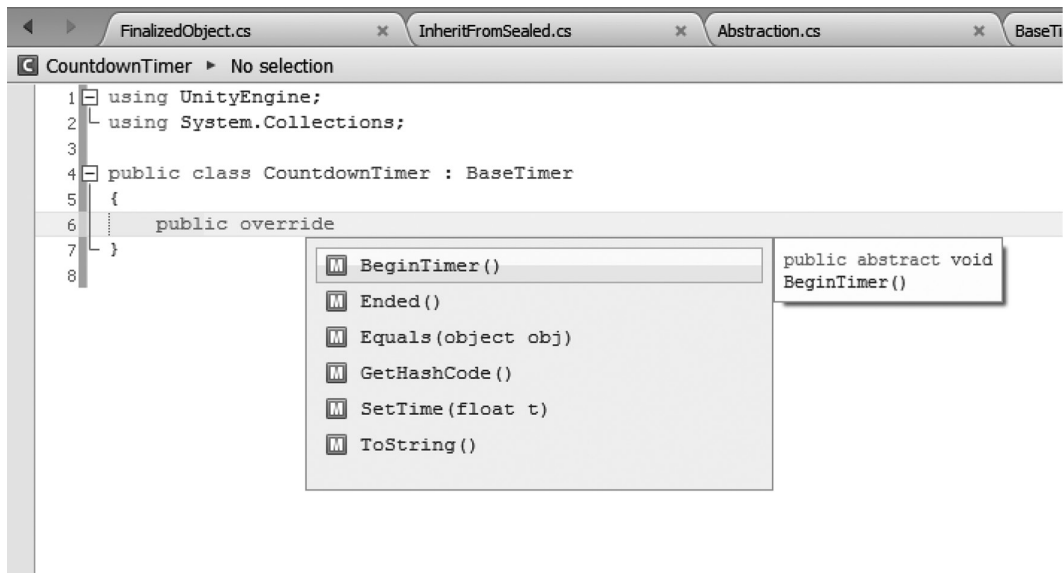
```

using UnityEngine;
using System.Collections;
public abstract class BaseTimer
{
    public float time;
    public float endTime;
    public float remainingTime;
    public float normalizedTime;
    public abstract void SetTime(float t);
    public abstract bool Ended();
    public abstract void BeginTimer();
}

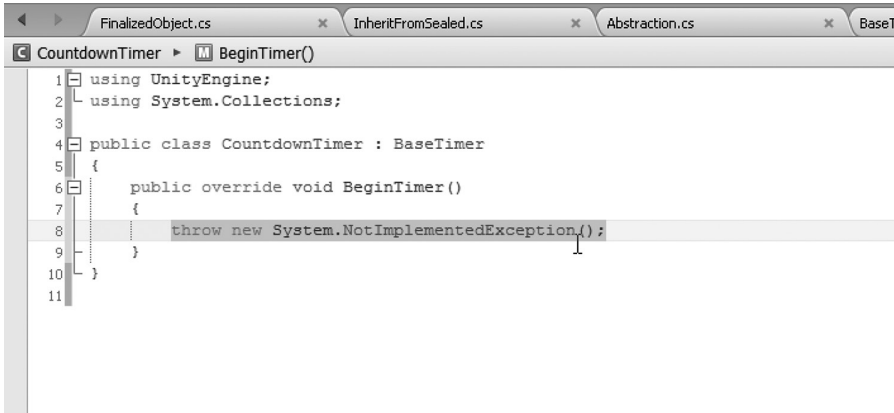
```

An abstract `BaseTimer` gives us a few interesting things to work with. First off, the big difference here is the fact that we're not using `MonoBehaviour` as our base class for the `BaseTimer`. We're going to make this a base class for use in any number of other classes that will have their own update functions, so we won't need to use an `Update ()` function in this class.

Next we'll make an implementation of this abstract class called `CountdownTimer()`; in which we can set a number of seconds and check for the `Ended()` boolean to become true.



As we begin to populate our new CountdownTimer with public override, we can make use of MonoDevelop's handy pop-up.



Selecting `SetTime` automatically fills in the rest of the function with some placeholder code. As the functions are stubbed in, the number of functions appearing in the pop-up is reduced to only the remaining functions waiting to be implemented. The seemingly extra functions `ToString()`, `Equals()`, and `GetHashCode()` are described in `Object()`, which is what every class created in Unity 3D is based on by default. These are provided by default and can be ignored.

```

using UnityEngine;
using System.Collections;
public class CountdownTimer : BaseTimer
{
    public override void BeginTimer()
    {
        throw new System.NotImplementedException();
    }
    public override bool Ended()
    {
        throw new System.NotImplementedException();
    }
    public override bool Equals(object obj)
    {
        return base.Equals(obj);
    }
    public override int GetHashCode()
    {
        return base.GetHashCode();
    }
    public override void SetTime(float t)
    {
        throw new System.NotImplementedException();
    }
    public override string ToString()
    {
        return string.Format("[CountdownTimer]");
    }
}
  
```

With all of the functions we're concerned with stubbed in, we can begin to flesh them out. The first function sets how long the timer will last. For this we'll simply add the following code to replace the system error:

```
public override void SetTime (float t)
{
    time = t;
}
```

This sets the float time to `t` in `SetTime(float);`. From here we can move on to the next function.

```
public override void BeginTimer ()
{
    endTime = Time.fixedTime + time;
}
```

This turns into a pretty simple process of setting up each function with a simple implementation of each function. The only thing to watch out for here is the automatic word replacement happening in MonoDevelop trying to change time to Time. Next up is the bool for the `Ended()` function.

```
public override bool Ended ()
{
    return Time.fixedTime >= endTime;
}
```

When `Time.fixedTime` is larger than or equal to the `endTime`, then the function returns true, otherwise the function returns false. Now we're about ready to give this a try.

```
CountdownTimer countdown;
void Start ()
{
    countdown = new CountdownTimer();
    countdown.SetTime(3.0f);
    countdown.BeginTimer();
}
void Update ()
{
    if (countdown.Ended())
    {
        Debug.Log("end");
    }
}
```

In the `FinalizedObject.cs` class that was attached to the Main Camera at the beginning of this chapter, we can make use of our new `CountdownTimer` class. We need to make a variable for the `CountdownTimer` so it can be shared between our `Start ()` and our `Update ()` functions. Therefore, a public `CountdownTimer countdown;` is added at the class scope.

Once in the `Start ()` function, we can add in the statement to instantiate a new `CountdownTimer()` with `countdown = new CountdownTimer();`. Follow this with a statement to set the length of the timer and a statement to start it.

Once in the `Update ()` loop, we use the `countdown.Ended()` to check on each update whether or not the countdown has ended. Once it has, we begin to send end to the Console panel. To restart the countdown, we just need to call the `BeginTimer()` function again and wait for `Ended()` to be true again.

6.23.5 What We've Learned

We can continue to make variations on the `BaseTimer` class fairly easily. We can make an implementation called `CountupTimer` that will be true until the timer has ended. All we'd have to do is switch one operator in the `Ended()` function and one operator in the `BeginTimer()` function to the following:

```
public override void BeginTimer ()
{
    endTime = Time.fixedTime - time;
}
public override bool Ended ()
{
    return Time.fixedTime < endTime;
}
```

This code changes the behavior and adds a variety of systems which we can use in our game. We added a normalized variable which we can use to show us a value between 0 and 1 when it's queried.

```
public override bool Ended ()
{
    normalizedTime = (endTime-Time.fixedTime)/time;
    return Time.fixedTime >= endTime;
}
```

We can set this by using the above addition to the `Ended()` check. In the `Abstraction.cs` class, we can look at the `normalizedTime` value with the following debug statement.

```
void Update () {
    Debug.Log(countdown.normalizedTime);
    if(countdown.Ended()) {
        countdown.BeginTimer();
    }
}
```

With this change, we can watch the values start near 1.0 and decrease toward 0.0, and reset based on what was used in the `SetTime()` function. This is a very basic timer, but its use can be shared between many different classes. Each class can have any number of timers set up to trigger various changes in behavior.

Zombies can wait for the timer to end before climbing out of the ground after they've spawned. They can set a different timer to tell them how often to search for a new target. Timers can be used for any number of different tasks.

Once a timer has been used in more than a couple of classes, it's time to seal it off to prevent its modification. Any modifications that might change its behavior could result in many different unexpected behaviors from any classes using it.

If any new types of timers are required, it's better to create a new branch and make a new implementation of the `BaseTimer` rather than tweak the timer already in use. It's easy to say that having too many different implementations of a timer class can create a headache for anyone coming in late to the project. If not wisely implemented, each programmer might end up with his or her version of a timer, which isn't optimal.

The coordination of all of this is dependent on how you've structured your team of programmers. Any individual change can have wide-reaching problems if unchecked. C# in general is built to allow for these sorts of behaviors, both better and worse. In the end, it's up to your ability to communicate and share your plans with the rest of your team to keep your code under control.

6.24 Leveling Up

Building up some skills. We're getting pretty deep into C#. This section contains a great deal of the things that make C# powerful and flexible. A great deal of the code we've learned up to this point allow us to get functional and get some basic work done. Even some of the more complex tasks can be completed with what we already know.

We had a short introduction to the more interesting constructs that make up C#. It's about time we had a short review of constructors, namespaces, and inheritance. We'll also look at more uses of arrays, enums, and `goto` labels. All of these systems make up a part of the skills that every C# programmer uses.