

4

Basics: The Building Blocks of Code

After covering the small parts that make up code, it's time to start learning how all of the parts come together to create a behavior. Creating data is easy; keeping the data organized is harder. Adding logic to do something with the data is harder still. That said, it's not impossible; otherwise, no one would write code.

This chapter covers how a statement of code works and where the power that drives the code originates. Programming is much less enigmatic when you understand the mechanisms that drive it. To understand why code behaves the way it does, it's important to know your way around the language.

4.1 What Will Be Covered in This Chapter

This chapter is about getting past the details and getting an understanding about the structure of code. This process is somewhat comparable to learning the grammar of the C# programming language. We'll cover the basics about the different types of logic statements and looping statements.

Logic allows you to pick different actions based on the result of a calculation. If you want to tell a human to run away from a zombie or run to a human for help, you'll need logic, to control that human's behavior. Logic takes data, makes a comparison, and chooses which block of code to execute.

Looping statements are able to read and react to more than one or two sets of data. If you're in a room with any number of zombies, you'll want to take them into account all at once. Looping statements are an integral part of any programming language. We'll see why they're so useful in this chapter, in which we discuss the following topics:

- Building up basic concepts
 - Creating a new class
 - Directives: The keyword `using`
 - Functions
 - An introduction to scope
 - The keyword `this`
 - Order of operation
 - Logic and operators
 - Loops
 - Warnings and errors
-

4.2 Review

Before getting ahead of yourself, it's important to remember some of what we just covered. We will be learning many new tokens in this chapter. A token, as we know, is the smallest element that makes up a statement.

Statements are organized tokens that accomplish a task. Tokens can be keywords, or words reserved by C# for special purposes. Some tokens are used to separate statements, and others are used to assign values to variables.

A keyword cannot be used for anything other than what it has been reserved for. Some keywords can be used only in a specific context. Several keywords are used only if you've defined a specific library, where they have been defined; we'll discuss some of these topics later in this chapter.

When you create a variable, you've created a new token. Some variables exist only in the context they were created in; the control of this existence is called scope, something which is coming up in this section as well. As a reminder, when you invent a word to identify a variable that word becomes a new token.

Once created, the token can be used in many different ways; you can apply an operation to a token and use it to assign values to other variables. These operations all happen in a very specific order. Create a variable, execute an operation or an expression, assign the value. This is what makes C# an imperative language.

4.3 Building Up a Game Idea: Working with What You Know

Often, when you come up with a game idea it's usually something that sounds familiar yet it's never been done before. When I'm at a convention or a comic book, game, or developer conference, people often like to tell me about their new game ideas. In about 20 years of these often impromptu game pitches, I've heard mostly about World War II (WWII) games usually involving zombies. Hopefully, you've thought of something more original.

At the same time, it's necessary you prune your ideas down to the most simple core game elements play possible. Starting off with an epic massive multiplayer free to play role-playing game (RPG) with app purchases of user-generated content is a mammoth undertaking.

To be honest, I wouldn't know where to start on such a titanic project. With that in mind, when learning any programming language and learning a new game engine, your goals should be in scale to what you know.

Starting off with the programming goal "I'd like to have a game controller move a character around" means you stand a higher chance of success. From there, you can move on to "I'd like the character to shoot a projectile when I press a button." This step-by-step approach gives you the building blocks necessary one day for creating your MMORPG, and it's not recommended that you start with building the next MMORPG/realtime strategy/WWII first-person shooter (FPS) from the get-go.

4.3.1 Design from Experience

When programmers think about game design, it's usually in the form of a technical specification. If there are zombies, then we need character controls and some zombie behaviors. If you need a fully automatic chainsaw rocket launcher, you'll need some physics and a pretty cool weapons system. Unity 3D might have the physics and rendering covered, but the rest is up to you.

After accruing a great deal of experience, you can begin to think about more complex game mechanics. It's great to have pie in the sky ideas, but not for your first project. The best games are finished games. The awareness of what is involved with writing a complete game usually comes only after finishing your first project. Only then is it possible to comprehend what's involved with the "Triple A" or high-budget titles that are produced at large game companies.

You should either finish your project or start on something new. I've started and stopped countless projects. With each project started I get closer to finishing before coming up with another better idea.

This habit is fine, and I encourage the behavior. With each project you tend to approach a similar problem in a slightly different, and often more efficient, way. Because of this approach, you're also learning and growing as a game developer.

4.3.1.1 Know Yourself

When you're just getting started it's better to start with your own coding skill. Think to yourself, "I can make a guy that can run, jump, and shoot. I can probably figure out how to make some simple monsters to

shoot at. I should make a game where you run, jump, and shoot at monsters.” This sounds like a reasonable plan.

Where you begin your code and what you decide to tackle first is determined by what the player does the most often in your game. If you’re focused on making a third-person zombie shooter, then you should probably begin with moving a character around on screen with a camera over his shoulder. Once your camera works, move on to the next activity. After shooting, move on to shooting at monsters. Sometime after that, make the monsters react to getting shot.

It’s an incremental process; layer one activity onto the next. When the game begins to be playable, you’ve reached a point where you can begin to add on more interesting behaviors. This process of adding features comes only after the core activity has been achieved.

4.3.2 Primary Activity

Deciding what it is that you do most of the time in building a game is usually based on the type of game you’re aiming to build. Action games tend to require a great deal of moving around and looking. Puzzle games tend to require a great deal of watching objects move around and looking for patterns.

It’s necessary to break your game down into something overly, simplified like moving and looking or watching and pattern matching. The primary activity your player will be doing should be the best functioning part of your code. Quite often, once you start on getting code written, you discover new game play ideas.

As the ideas begin to flow, however, you need to remind yourself that none of them will work unless you get your camera working or your pattern matching down. Maintaining focus is the hardest thing to do as a new programmer. It’s also important to remain focused on your ultimate goal: writing a game.

The players’ primary activity should be the focus of your game code. You should boil down your ideas into a solid obtainable goal. Only when you’ve reached the minimal activity to start with, you’re ready to move on.

Reading through completed code projects may help you see what you should be working toward. Reading forums online, asking questions, and looking at examples of code helps immensely. Everything you’re trying to do in code has most likely been done before, and in many different ways. It’s up to you to find the method you’re most comfortable with and that matches what you’re doing most.

The Unity 3D Asset Store is complementary to your achieving this goal. Many collections of code for various purposes can be found here. If you’re looking for a system to control zombies, there might be a good starting point in the store. Reading code downloaded from the store means you’ll have Unity-compatible source code, and many examples to learn from for working on your own project.

4.3.3 Moment to Moment

Once you’ve managed to get your primary activity understood, it’s time to move on to the moment-to-moment game play. Moment to moment involves thinking about what it is that your player does while doing his or her primary activity.

For a third-person zombie shooter, you’ll be mouse clicking to shoot weapons. If you’re moving tiles around to match colors, you’ll be clicking on tiles. This first action dictates the next set of functions that need to be written to reach your game-making goals.

4.3.4 Actions to Functions

The first step to building code is thinking in terms of taking interactions and turning them into code. When you need to read mouse movement, you need to look for something that does that. In most game systems there’s usually some input method that exposes the various input systems that the engine can read.

Mouse input, touch input, and in many cases accelerometer, gyro, and even a global positioning system (GPS) can be accessed through functions coded by the engine developers. From there you need to figure

out what you want to do with the input. Mouse movements are usually in x and y coordinates that correspond to a position on the screen. Systems like joysticks often return an x and y value between -1 and 1 .

At first, nothing has to move. You just need to print out the text informing the accomplishment of listening to a mouse click, or recording a mouse position on screen. After you've accomplished reading your input sources, you'll need something to click on, or shoot at. This never has to be fully realized artificial intelligence (AI), or puzzle piece when a simple cube will suffice.

With every step, use a printout to the Console panel to prove you're getting the data you are looking for. Then you should always be using primitive shapes to test out a basic interaction. There's nothing worse than spending a few weeks modeling, rigging, and animating a zombie, only to find out you need to rewrite your code because of something unexpected.

4.3.5 Compromise

As mentioned, you might figure out that building a MMORPG with WWII vehicles, zombies, crafting, and user-generated content might be a bit too big for your first project.

At least I hope you figure this out. Once you've decided you need to change plans you need to cut features. Ultimately, this all comes down to how much time you are willing to dedicate to your project.

If you're willing to stick with Unity 3D and C# for the next 20 years then sure, go ahead and make that MMORPGWWII-with-zombies game you've never seen before. Unfortunately, it's likely that you will not see it; no one else will see it either. I have heard of one project where a lone programmer has literally spent over 20 years on one game project. He started in the early 1990s and is still working on his game. He has yet to produce a demo, and will probably spend another 20 years on it. I hate to see people do this, but it's their choice. Once you've written enough code for your first game, you might discover that just wandering around and shooting zombies might be enough to constitute a game. In reality, it is enough to be a game, and you might even be able to finish it before your computer becomes obsolete. If the game isn't fun, however, with this simple activity, then maybe the core of the game isn't actually fun. In any case, you made a game and it's time to start writing another game but maybe this time you will start off with a more interesting game play core.

As an independent developer, or someone working on your own, it's also important to weigh your code time against your art time. Producing art and code takes time, and it's quite difficult to rework and change after a great deal of work has been done. Although code would remain largely the same, to make changes in code to accommodate a minor change in art might take days.

Renaming each joint in a zombie uniquely per requirement of your code would take only a few minutes to correct in your 3D modeling package. Conversely, you the programmer might have to invent a new system for joint location to deal with joints of all the same name, based on some clever concept you haven't even thought of yet. Inventing a new system, or correcting some joint names: hopefully the compromise is easy to make.

4.3.6 Starting with Controls

At the beginning of almost every game development project, the player controls are usually the first focus of the game. After all, most games are played by players, so they'll need to be able to control how they play your game.

This should seem obvious, but I've been a witness to some game development that started with realistic physics for breaking objects or amazingly lifelike human motion. In both cases, however, that amazing never-before-seen technology was greatly ignored by the player, or at best taken for granted.

Only a small tech-savvy group of the gaming audience would notice, for example, how glass breaks in a realistic manner, or how a character's feet plants when he turns around. Many developers think that they're breaking ground on realism in gaming. To their player, they're spending thousands of hours not making the game more fun.

Input management is our first goal. You should decide if you're dealing with either keyboard and mouse or touch screen taps and gestures. Testing and iterating on a good input management system takes time and effort. The amount of time usually spans the length of the entire project. Constant tweaks and updates create a constant loop between game play and input.

In Chapter 3, we added a cube to a scene and attached an `Example.cs` script to it. This is actually nearly enough to call a sand box. Depending on your game, you might want to add in a floor and walls. This can be done by adding in some cubes and stretching them out to build a simple room to run around in.

There are many different books already on building games and environments in Unity 3D, and you can defer to those as reference. However, it can't be overlooked that Unity 3D has many different assets already for you to use on the Asset Store. This is a huge bonus for you as a programmer as well.

The Asset Store includes source code written in C#. Downloading free assets from the Asset Store, or even trials, will reveal a wealth of C# techniques both good and bad. After downloading free assets from the Asset Store, it would be difficult to tell what's good and what's bad.

In both cases of good and bad code, code is a very creative art. As in visual crafts like painting and drawing, there are good artists and there are bad ones. The differences also apply to programming. There's plenty of bad code to learn from, or rather learn what not to do. There's also plenty of code that is useful and clever.

Later on, when you've learned more, you'll be able to tell the differences more easily. Large unreadable code assets tend to stand out, as so clean tidy code assets. In general, if you see a great deal of unreadable, uncommented, and messy formatting, you're looking at garbage. If you see clear simple short statements, the chances are you're looking at a winner.

4.3.7 What We've Learned

So far we're just getting started. Along our road toward zombie destruction, we're going to need to do a great deal of footwork and studying. Some of what we're about to get into might seem a bit mundane, but it's only the beginning.

We're learning a new language with new words and concepts. With the help of Unity 3D we're going to have a foundation on which pretty much any game can be built. Stick to it and you'll be writing code in no time.

4.4 Creating a Class

A class is the set of instructions and tools necessary to build an object. This was mentioned before, but we didn't look at what's involved with writing a new class. To write a new class, open Unity 3D, start a new Project named `MyNewClass`.

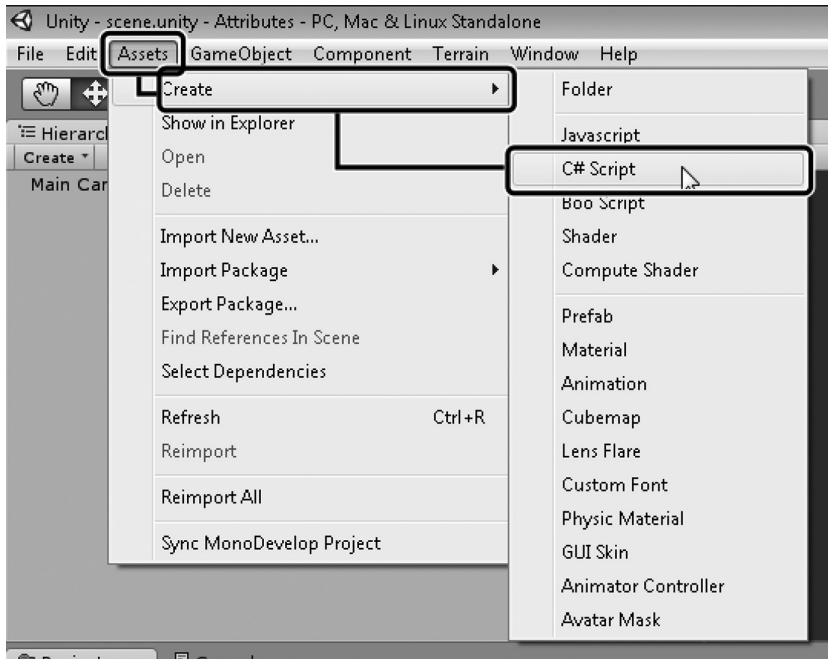
To get our heads around what a class is, we're going to write a class inside of a class. This is what is called a nested class. Open the `FirstExample.cs` to follow along.

To write a nested class you start with a *class declaration*. A class declaration is basically the keyword `class` followed by an identifier. As a reminder, an identifier is simply a word you use to name any data you've created. Once a class has been declared, it's a matter of filling it with all of the useful things that classes can do.

```
class MyNewClass
{
    //code goes here...
}
```

Where does this go? Well, to get started we'll want to have Unity 3D do a bit of footwork for us. Much of this work is done for you by the Unity 3D Editor when you use the Asset → Create → C# Script menu in the Project panel.

This takes a bit of the work out of starting a new file from scratch. There's no difference between the files Unity 3D creates for you this way and the new C# files you create.



Unity 3D creates a new file, writes some directives, and declares a class, then adds in some base functions to get started. We'll go through the same steps to understand why the file is written like it is. And we'll start with the class declaration.

4.4.1 Class Declaration

Classes contain a variety of different objects called members, or *class members*. Variables are accessed by identifiers. When referencing a class the *data members* are sometimes called *fields* or *properties*.

When you fill out a form to log into a web page, you may need to put your name and a password into empty boxes, which are often called text fields because they are fields of data to a programmer. They are open spaces in your computer's memory, into which you put data.

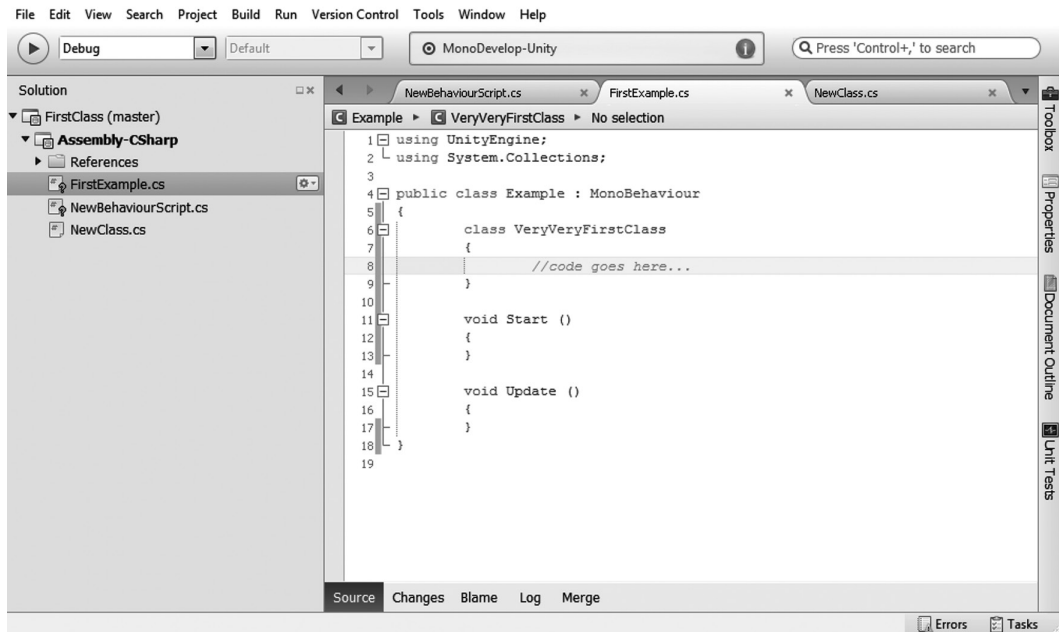
The sections of a class that contain logic and code statements are called *function members*. These are also accessed by their identifiers. Depending on what a function does for the class, it can be categorized as a *method*, a *constructor*, a *destructor*, an *indexer*, a *operator*, a *property*, or an *event*. How these functions differ from one another will be discussed in Sections 5.4.3, 7.13, 4.7.1.1, 7.5, and 7.15.

Classes need to talk to one another for any number of reasons. Zombies need to know where humans are, and bullets need to take chunks out of zombies. Suppose a zombie has a number of chunks left before slain, and the human has a position in the world where the zombie needs to chase him; this information needs to be shared. The availability of the information is called *accessibility*.

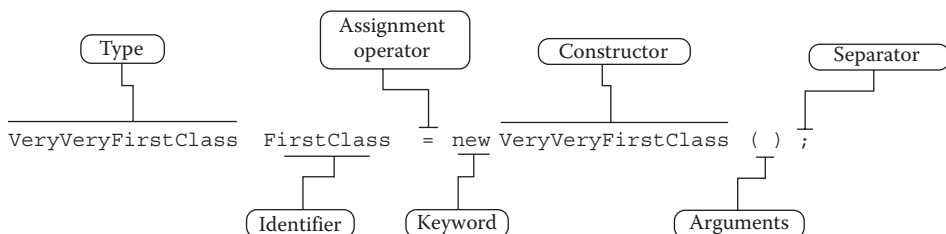
Data and function members can be made accessible to each other using the *public* keyword. We'll observe how this works in a moment.

4.4.1.1 A Basic Example

C# allows us to create classes within a class. Shown above is a class declaration for `VeryVeryFirstClass`. This is a nested class inside of the `Example` class. For ease of use we'll use a new `Example` class to build a `VeryVeryFirstClass` object. The declaration of the class looks like `class VeryVeryFirstClass {}` that creates a new class that has no members.



Look just below that and you'll find a line that has the statement `void Start ();`; this statement creates a function that is a member of the `Example` class. To create an instance of this class we add `VeryVeryFirstClass FirstClass = new VeryVeryFirstClass ();` to the `Start ()` function.



Let's examine the first two words, or tokens. The first token `VeryVeryFirstClass` indicates the type of object we intend to create. The class we created is now a type of data. Once `VeryVeryFirstClass` has been declared as a class, it becomes an instruction to create a new object that can be found by its identifier.

We know a variable is declared by a keyword followed by an identifier like `int i;`; a class variable is declared in a similar way: `VeryVeryFirstClass FirstClass`. However, unlike an `int` a class *cannot* be assigned easily.

What differentiates an `int` from a class is how the type is created and then assigned. We'll have to go into detail a bit later when we would have a deeper understanding about data types, but for now, we'll consider an `int` as a fundamental type.

4.4.1.2 Value and Reference Types

Although we declared class `VeryVeryFirstClass` we have yet to give it anything to do. For now, there's nothing in `VeryVeryFirstClass`. Once we start adding features to `VeryVeryFirstClass` the task of assigning a value to the class becomes ambiguous. In context:

```
VeryVeryFirstClass FirstClass = 1; //what should this do?
```

What would that mean if `VeryVeryFirstClass` was a zombie? Is the `1` his name, how many hit points does he have, or what is his armor rating? Because C# can't be sure, the computer has no hope of making a guess.

This differentiates a *value type*, like an `int`, over a *reference type*, such as `MyClass`. However, before we go into a chapter about types we need to know more about classes, so we'll get back to the topic at hand.

Therefore, when creating a new instance of a reference type we need to use the form `new SomeClass ()`; to assign a variable of type `SomeClass`. Thus, `MyClass myClass = new MyClass ()`; is used to create a variable for `MyClass` and assign the variable `myClass` a new instance of the correct class.

To make things clear, `int i` can be assigned in the exact same way we assign `myClass`.

```
int i = new System.Int32();
```

We can use the above statement if we want to make our `int` declarations look the same as our class declarations. This is unorthodox and unnecessary, but it's possible. It's important to know that things like this do exist and that C# is flexible enough to allow us to do this.

4.4.2 Adding Data Fields

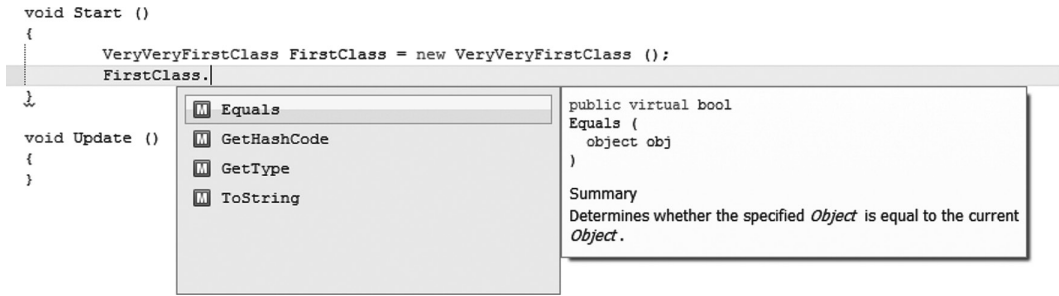
A data field is any type of information that lives inside of a class. We'll cover what data type really means in Section 6.5.3 but for now we'll assume that data can be pretty much anything, including a number. When a class has data fields added to it, we're giving the class a place that holds onto information and allows the class to remember, or store, some data. For instance, were we to write a zombie for a game we'd need the zombie to know, or store, how many brains it has eaten, or how many whacks it has taken from the player.

```
class VeryVeryFirstClass
{
    int num;
}
```

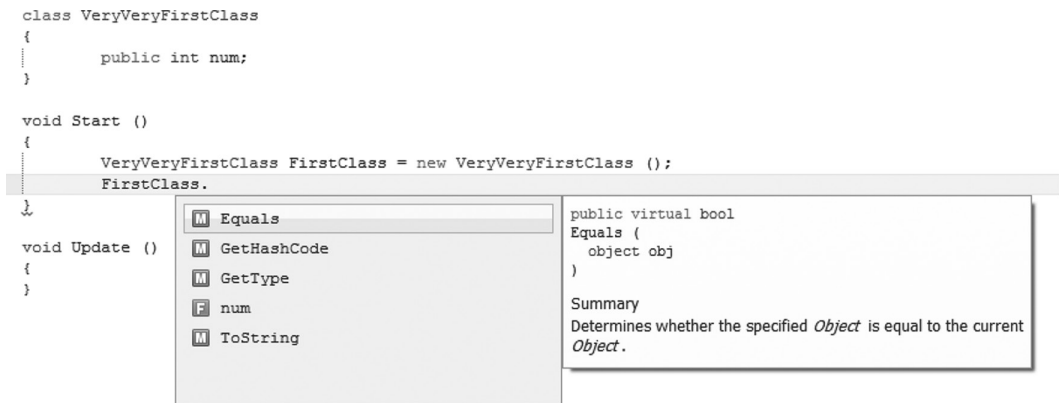
To add a data field to `VeryVeryFirstClass` we use the same declaration as we would for any other variable. This should be obvious, but since we're writing a class inside of a class this might be a bit confusing. However, when this is declared as `int num`; we're not allowed to use it yet.

4.4.3 Access Modifiers and the Dot Operator

To talk to members of a class we use the *dot operator* (`.`) to tell C# we want to access a member of a class. When a new data field is added to a class it's assigned a type and an identifier. The identifier turns into the word we use after the dot to request access to that data or function.



If we try to find the `num` data field when using MonoDevelop we won't see the `num` data field in the list of members. To make the `int num` accessible we need the `public` keyword. The `public` keyword is similar to several other keywords that change the accessibility of a variable within a class; such keywords are called *access modifiers*. There are several access modifiers we'll come to use, but for now we'll just get familiar with the `public` access modifier.



Adding the keyword `public` before the `int num` changes the accessibility to the classes outside of `MyClass` and allow for other objects to see the data field. The data fields that can be changed this way are called *instance variables* that are unique to each instance of the class.

An instance is a unique version, or object, based on the class from which it was created from. Each instance stands on its own as a new object created from the class it's based on. All of the class members are unique to each instance. There are systems that circumvent this behavior, but we'll get to that in Section 5.5.

To observe this point, we'll create a few instances of the class and set the data fields to different values.

```

class VeryVeryFirstClass {
    public int num;
}

//Use this for initialization
void Start () {
    VeryVeryFirstClass FirstClass = new VeryVeryFirstClass ();
    FirstClass.num = 1;
    VeryVeryFirstClass SecondClass = new VeryVeryFirstClass ();
    SecondClass.num = 2;
    VeryVeryFirstClass ThirdClass = new VeryVeryFirstClass ();
    ThirdClass.num = 3;
}

```

Here, we created three instances of `VeryVeryFirstClass`. Each one has its own `num` data field and each one can be assigned a unique value. To check this, we could go through a process and print out each one.

```
Debug.Log (FirstClass.num);
Debug.Log (SecondClass.num);
Debug.Log (ThirdClass.num);
```

We could add in these three lines after `ThirdClass.num` was assigned. We could do something a bit more clever by adding a function into the class that does this for us. A class member can be something other than a variable. Functions can also be class members accessible through the dot operator.

```
class VeryVeryFirstClass
{
    public int num;
    public void PrintNum()
    {
        Debug.Log(num);
    }
}
```

Start off by adding a `public void PrintNum()`, and then add in `Debug.Log(num);` to the function, following the example above. We'll get a better look at what function really is in Chapter 5, but this example serves as a quick and dirty introduction come back to a simple function nonetheless.

Next, we can use the function member by using the same dot operator.

```
void Start ()
{
    MyClass FirstClass = new MyClass();
    FirstClass.num = 1;
    MyClass SecondClass = new MyClass();
    SecondClass.num = 2;
    MyClass ThirdClass = new MyClass();
    ThirdClass.num = 3;
    FirstClass.PrintNum();
    SecondClass.PrintNum();
    ThirdClass.PrintNum();
}
```

The `PrintNum()` function is now usable through the dot operator for each instance of `MyClass`. There are many nuances that were skipped over, but we'll go into much greater detail in the rest of this book.

4.4.4 Class Scope

Data fields can appear anywhere in your class, though a proper system requires that we keep them together at the beginning of a class. However, it's sometimes simple to organize data with functions that need to use them.

```
class VeryVeryFirstClass
{
    public int num;
    public void PrintNum()
    {
        Debug.Log(num);
    }
    //some other class data field.
    public int OtherNum;
    public void PrintOtherNum()
```

```
    {  
        Debug.Log (OtherNum) ;  
    }  
}
```

With the above example we can see that `OtherNum` isn't grouped with `num`, but rather its declaration is written just before the function that prints it. Where and when class data fields are declared have no effect on whether or not a function can see it. As a matter of fact, the placement of the declaration statements can happen at the bottom of the class, as seen here:

```
class MyClass  
{  
    public void PrintNum()  
    {  
        Debug.Log (num) ;  
    }  
    public void PrintOtherNum()  
    {  
        Debug.Log (OtherNum) ;  
    }  
    //declared at the end.  
    public int OtherNum;  
    public int num;  
}
```

The functions and data members of a class can appear in any order. This feature might seem contrary to intuition. However, the C# code is fully compliant and functions just fine, ignoring how the different members are arranged. This rule doesn't always hold true, especially once we get into how code itself is executed, but at the class level, or at the class scope, as programmers like to say, the ordering isn't as important.

Not all programming languages will allow this. UnrealScript, for example, enforces all variables live at the beginning of a function. Declaring them anywhere else raises an error. C# understands that formatting is less important than clarity. Declaring variables close to where they are used, however, is important for readability.

4.4.5 Class Members

As we add functions and variables to a class, we're adding *members* to the class. In the previous example, `num` is a member of `MyClass`. So too are `PrintNum()` and any other functions or variables included in the class. We can refer to some of the members as `public`, which means that other classes can access them, while the inaccessible members are `private`.

In most cases, if `public` is not used then you can assume the member belongs to `private`. Private members cannot be accessed from outside of the class, at least not directly. However, functions on the inside or local to the class can modify them freely.

4.4.6 What We've Learned

We just covered some of the very basic concepts of what makes up a class. There are plenty of other keywords and some complex concepts that we have yet to even touch on. Later on, we'll get into some of the more clever tricks that classes allow for.

On your own, you should practice creating different data fields and add more functions to your class to get a feel for how they work. As you learn how to use more keywords, you should add them to your own classes to check that you understand how the keywords work.

`Example.cs` is a class. In this chapter, we created `MyClass` as a class inside of the `Example` class. `MyClass` is an example of a nested class, or a class inside of a class. You're allowed to write multiple nested classes.

```

using UnityEngine;
using System.Collections;
public class Example : MonoBehaviour
{
    class VeryVeryFirstClass
    {
        public int num = 0;
        public void PrintNum()
        {
            Debug.Log(num);
        }
    }
    class AnotherClass
    {
        public int anotherNum = 1;
        public void PrintNum()
        {
            Debug.Log(anotherNum);
        }
    }
    //Use this for initialization
    void Start ()
    {
    }
    //Update is called once per frame
    void Update ()
    {
    }
}

```

We can add as many classes as necessary to any other class. Nested classes can have more classes inside of them.

```

class AnotherClass
{
    public class InsideAgain
    {
        public int insideNum = 0;
        public void PrintInside()
        {
            Debug.Log(insideNum);
        }
    }
    public int anotherNum = 0;
    public void PrintNum()
    {
        Debug.Log(anotherNum);
    }
}

```

We can add class InsideAgain to AnotherClass if we needed to. So long as the classes are all public we're allowed to use them from outside of the class.

```

void Start ()
{
    AnotherClass.InsideAgain ACIA = new AnotherClass.InsideAgain();
}

```

```
        ACIA.insideNum = 10;
        ACIA.PrintInside();
    }
```

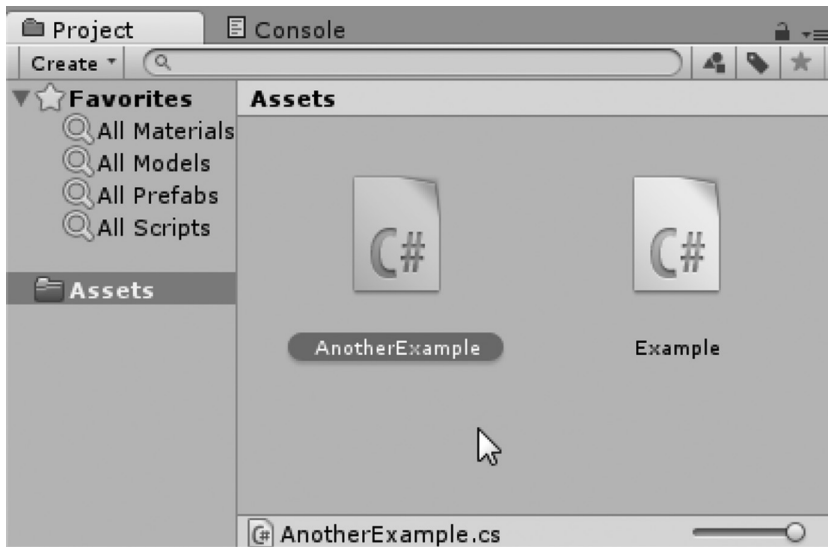
Using the dot operator from `AnotherClass` we can get to the classes inside of it. This sort of class-in-a-class layout isn't too common, but there's nothing keeping you from doing it. It's important to remember to keep things as simple as possible. With every layer of obscurity, you add a layer of complexity.

The dot operator allows us to access the contents of a class, including classes within classes. The behavior is used throughout C#, and all of the features rely on classes and class members. We'll see how to access the many functions available to us in Chapter 5.

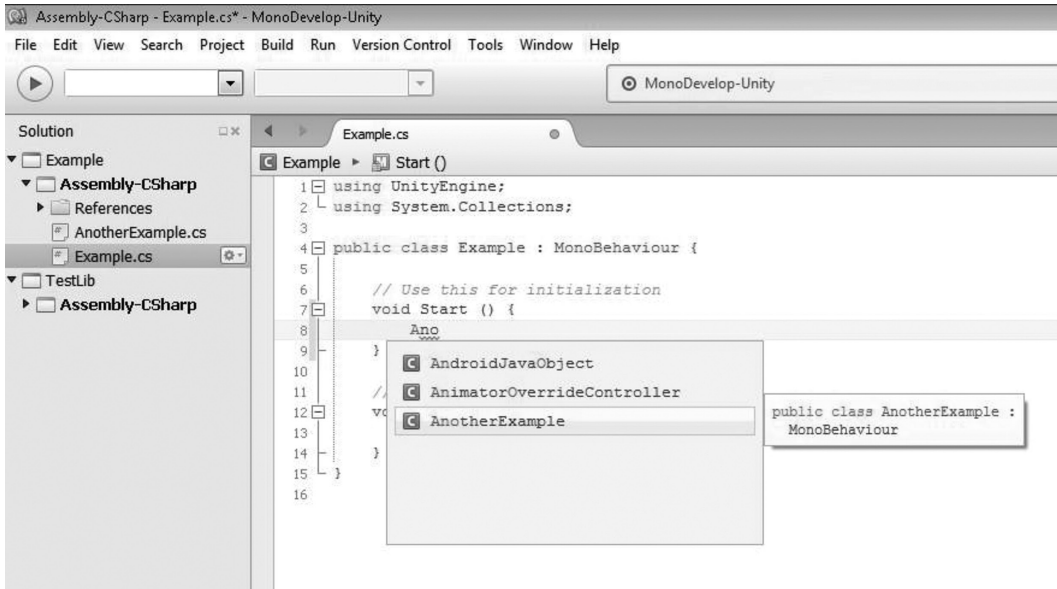
We've been working in a nested class all this time. To show how this works in another class, we can go back to Unity 3D and create a new class next to `Example.cs` called `AnotherExample.cs`. If we open `Example.cs` we have access to the `AnotherExample` class.

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

The above class is the automatically generated class from Unity 3D. This means that in the `Example.cs` `Start ()` function we're allowed to use the class as though it were any other publically available class. This does mean that there's a completely new C# file alongside the original `Example.cs` file we have been working in.



When we write code in the `Example.cs` class the `AnotherExample` class can be accessed from within the `Example` class.



```
void Start ()
{
    AnotherExample aClass = new AnotherExample();
}
```

The above statement in the `Start ()` function of `Example.cs` will create a new instance of the `AnotherExample` class we just created. Classes become accessible throughout the rest of your Unity 3D project. This allows objects and classes to send messages to one another. As soon as they are created, they are able to communicate with one another.

Classes are able to communicate with one another thanks to MonoDevelop's understanding how C# works. Once a class is created its contents can be public or private depending on how they are declared. The scope of a class and its fields are controlled by where they appear.

A class is also able to create and assign its own data. Inside of a function you can create data that's scoped to the function, or to the rest of the class. Once a class is instantiated, it becomes an object that allows you to see and manipulate its instanced contents.

With these basic concepts we'll want to explore the different classes which Unity 3D and Microsoft have created to allow us to access the various abilities that are contained in the Unity 3D game engine. Accessing the classes and fields which Unity 3D has provided is quite simple, as we will see in Chapter 5.

4.5 Directives

When you start writing code in Unity 3D, or any other programming environment, you must consider what tools you have to begin with. For instance, we might consider that with the .NET (read as "dotnet") framework we get things like `int`, `bool`, `float`, and `double`. These tools are given to us by adding `using System;` to our code.

We also have been taking for granted the use of `using UnityEngine;`; this function gives us another massive set of tools to work with including a ton of work that allows us to use functions and events that are triggered by the Unity 3D game engine itself.

All of these classes, functions, and events are stored in software called a *library*. Basically, this is code that has been written to enable your code to communicate with Unity 3D. These resources provide a gigantic foundation to build your game on. Much of the difficult math and data management functions in relation to Unity 3D has been written for you.

Knowing what they are and how they are used is important. It's like knowing what tools you have in your toolbox. In Chapter 5, we'll go over their basic use and how to find out more about what they offer.

Directives provide the connection between your class and a selected library. Many hundreds of classes, written by the programmers who created Unity 3D, allow you to communicate between the player, the characters, and every other object in the game. In short, a library is software for your software.

```
using UnityEngine;  
using System.Collections;
```

In the first two lines of the class we had Unity 3D prepare for us are the directives we needed to get started. The connection between Unity 3D and your code is made, in part, by using directives. A directive makes calls to outside or external resources called libraries.

The libraries that the directives call upon are connected to compiled code. Unity 3D is also connected to the libraries that your code uses. The connection is made by identifying the name of the library after the keyword `using` is entered.

Libraries live in paths, like a file in a folder on your computer, only they are compiled into a few tightly bundled binary files.

NOTE: The initialism *DLL* refers to *dynamically linked library* that is a complete set of software compiled ahead of time for your use. Pretty much every software released these days include many different DLLs that allow the software to be updated and modified by only updating the DLLs they come with. Often some DLLs are located in your computer's operating system's directory but depending on the software involved it too will bring along its own DLLs.

The first line, `using UnityEngine;`, pulls all of the resources that deal with specific Unity 3D function calls. For instance, moving an object, playing sounds, and activating particle systems are found inside of the `UnityEngine` library. The functions necessary for making bullets and magic spells work are built into `UnityEngine`.

As the resources that make up your game are imported into Unity 3D they're made available to your code. The libraries provided by Unity 3D expose all of the imported assets in such a way that they can be found like files on your computer.

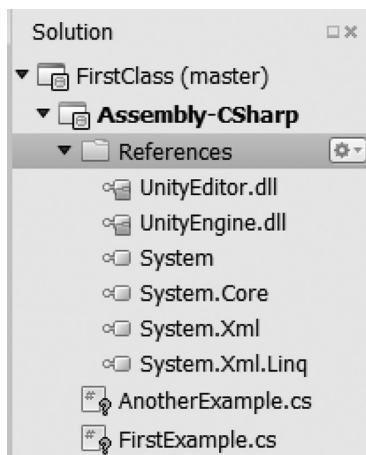
The programmers at Unity 3D have taken on the bulk of these complex tasks for you, so you don't have to deal with all of the physics and PhD-level science of rendering 3D graphics. The keyword `using` is how we tell C# that we're going to ask for a specific library to access.

Many other programming languages do the same thing; for instance, Python uses the statement `import module` to do the same thing; JavaScript is more wordy, by requiring a path like `src = "otherScripts/externalLibrary.js"` to access external code.

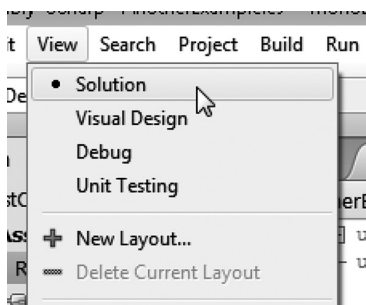
C++ and C both do the same as C# with the keyword `using` to access external libraries. In the end, your code has the ability to work with preexisting code stored in nicely packaged libraries. For Unity 3D, this is pretty much necessary.

4.5.1 Libraries

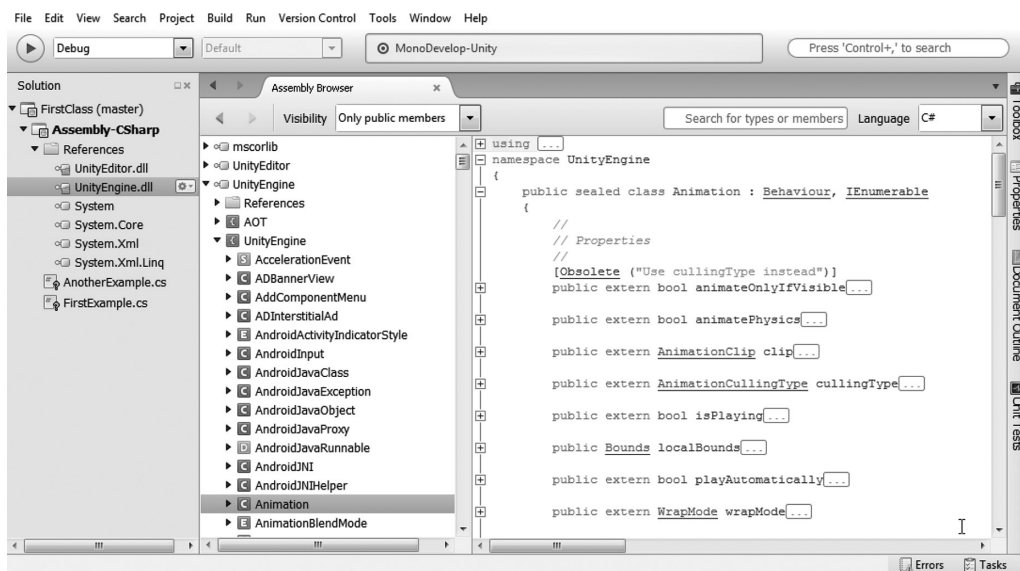
Libraries are collections of classes and data that have been bundled into a convenient package with Unity 3D. You can consider them to be similar to a compiled application. A library contains classes that are all accessible through the dot operator, like the classes we wrote in Chapter 3. Inside of MonoDevelop you can take a look at some of the other libraries we can use.



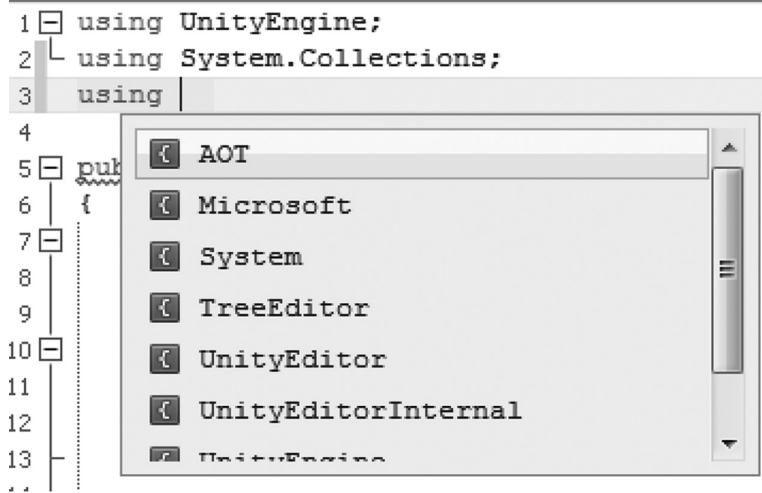
Inside of MonoDevelop, expand the References icon in the Solution explorer. In case the Solution explorer panel isn't open, then use the following menu to show the panel: View → Solution.



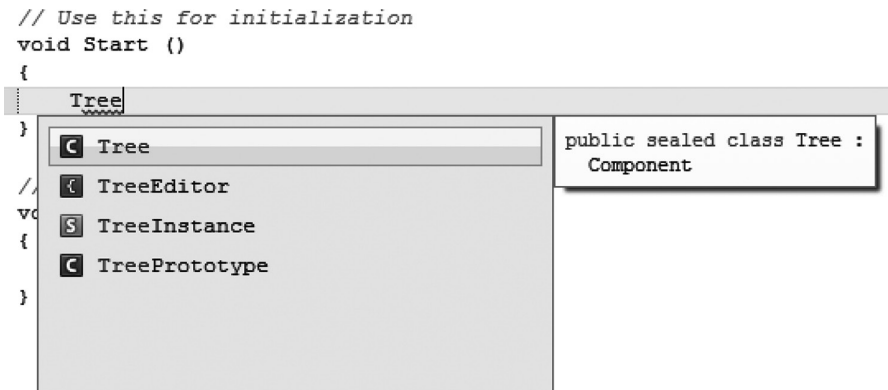
Double click on the UnityEngine.dll under the References folder.



The Assembly Browser will open showing you the various libraries you have access to. In your FirstExample.cs class, entering the using keyword will prompt a pop-up. The pop-up window prompts you with the many different libraries available to Unity 3D.

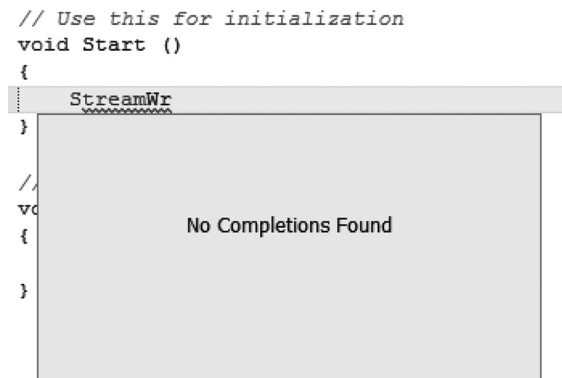


You're then prompted by the various libraries you can add to the project. By adding in UnityEditor we can access new functions not previously available without the directive.

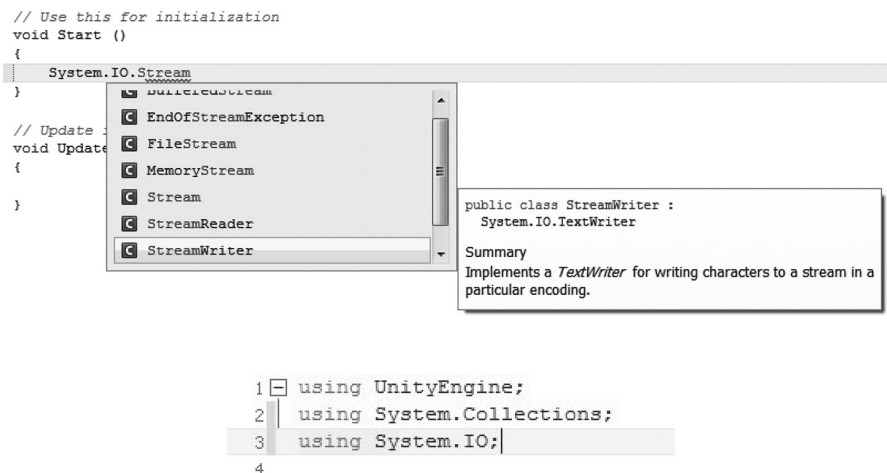


We won't go too far into how to use these other functions. We are just now getting to know the functions we already have. Between both UnityEngine and System.Collections we've got quite a large set of tools already. Later on, we'll be able to create our own libraries. Within our own libraries, we'll be able to write useful functions that can be used like anything found in UnityEngine.

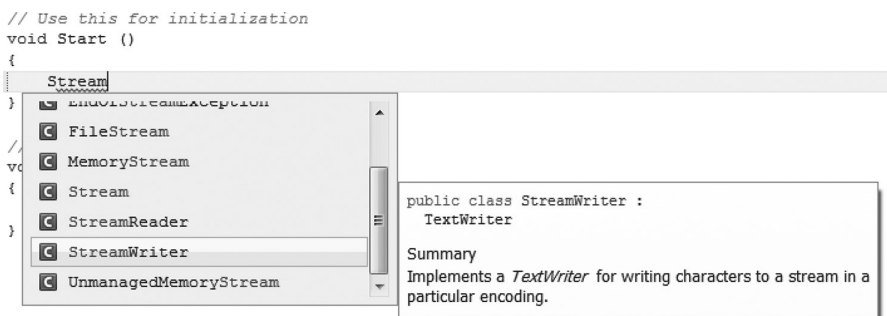
In our FirstExample.cs class, we'll use the Start () function and look for a StreamWriter function.



However, the auto-complete pop up shows “No Completions Found” as we start to enter the function we’re looking for. Should we use a long-winded version, we would see that it is hidden within System. Within System.IO we find several objects that have Stream in them.



However, if we add in the `using System.IO;` directive we can shorten our bit of code. The `using` keyword helps us create a link between the classes within the libraries and our code. The more explicit we are about the objects we’re after, the less writing we need to do after the directives have been added.

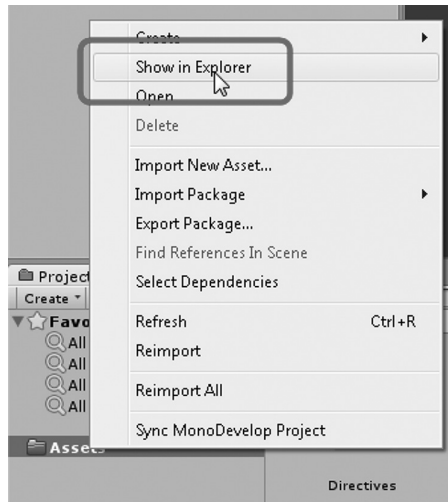


For this trick, we’re going to use the StreamWriter to create a text file.

```
void Start () {
    StreamWriter writer = new StreamWriter("MyFile.txt");
    writer.WriteLine("im a new text file.");
    writer.Flush();
}
```

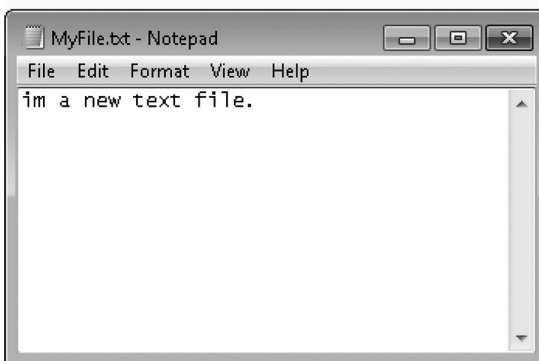
Using the StreamWriter we create a new object called writer. When the StreamWriter is instantiated we pass a parameter with a string used for the file name; in this case, MyFile.txt is given to the new writer object. With the object instantiated with the identifier, we use the object and call upon the WriteLine function and give it some text to write into the new file. The line `im a new text file.` will appear in the MyFile.txt file that Unity 3D will write for us. After that we need to use the Flush(); call to tell the writer to finish its work.

Attach the script onto the Main Camera in the scene and press the Play button to execute the `Start()` function in the C# class. To find the file, right click on the Assets directory in the Project view.



This action should bring up the directory where the new `MyFile.txt` has been written.

| | | | |
|---------------------------|-------------------|------------------------|------|
| Assets | 7/21/2013 4:06 AM | File folder | |
| Library | 7/21/2013 4:06 AM | File folder | |
| ProjectSettings | 7/21/2013 2:48 AM | File folder | |
| Temp | 7/21/2013 4:06 AM | File folder | |
| Assembly-CSharp.csproj | 7/21/2013 2:49 AM | Visual C# Project f... | 3 KB |
| Assembly-CSharp-vs.csproj | 7/21/2013 2:49 AM | Visual C# Project f... | 3 KB |
| Directives.sln | 7/21/2013 2:49 AM | Microsoft Visual S... | 2 KB |
| Directives-csharp.sln | 7/21/2013 2:49 AM | Microsoft Visual S... | 2 KB |
| MyFile.txt | 7/21/2013 4:06 AM | Text Document | 1 KB |



Opening the `MyFile.txt` will open the file in your usual text editor; in my case, it was Notepad. You could imagine writing various bits of information in this, or even begin to read from a text file into your game! This might make writing dialog for an RPG game a bit easier. You wouldn't have to write all of the text into each object or even code in your game. Reading from a text file would make more sense.

Of course, we'll have to do more work to get the most out of reading and writing to text files, but this is just the beginning. We'll move on to more sophisticated tricks later on. This is just a glimpse into what we can do by accessing the different parts of the libraries available to us.

4.5.2 Ambiguous NameSpaces

Therefore, we're looking at names of functions from libraries; in theory, each library should have unique names for each function. However, what happens if Unity 3D wrote a function with a specific name that might have existed in another Library?

```
using UnityEngine;
using System.Collections;
public class Directives : MonoBehaviour
{
    //Use this for initialization
    void Start ()
    {
        int UnityRand = Random.Range(0, 10);
    }
}
```

Therefore, if we look at the above code, we have two directives that provide us with a vast wealth of tools to start working with. One of the many tools is a simple random number generator. The `Random.Range()` function was written by the Unity 3D programmers and requires a minimum and a maximum number.

```
using UnityEngine;
using System.Collections;
using System;
```

However, if we add `using System;` to the list of directives, as in the above code sample, we'll get an error in Unity 3D.

```
Assets/Directives.cs(9,33): error CS0104: 'Random' is an ambiguous reference
between 'UnityEngine.Random' and 'System.Random'
```

This means that there's an identically named `Random` function in `System` as well as `UnityEngine`. To resolve this error, we need to be more specific when we use `Random`. Since we might need to use other parts of the `System;` library, we'll want to keep this included; otherwise, we wouldn't have this problem at all.

```
void Start ()
{
    int UnityRand = UnityEngine.Random.Range(0, 10);
}
```

If we change `Random.Range` to `UnityEngine.Random.Range()`, we'll fix the problem of conflicting functions. There will be many times where you might have to change the names of a specific library as well.

4.5.3 What We've Learned

Knowing where to look for these functions is something that can be done only by having an investigative hunch. Usually, a quick search on the Internet can drum up a few clues as to what function you need to search for to get a specific task done.

Different libraries provide us with systems which help us add additional functionality to our classes. By default, we're given a rich set of tools to build into our game. Once we need to start reading and

writing files to disk for saving lengthy game info or perhaps loading information from a web server, we will require functions that reach beyond the built-in tools that Unity 3D starts us off with.

It should also be mentioned here that the free version of Unity 3D will allow you to use only the System and Unity 3D DLLs. The pro version will allow you to use many external libraries for various purposes. With a pro license, you'll be able to use open source libraries for playing with the Microsoft Kinect, or Nintendo Nunchuk.

In addition, it's also possible to write your own DLLs creating libraries for your game. This might allow you to add in platform-specific functions to accelerate various aspects of the game outside of the abilities of the built-in Unity 3D tools or engine.

4.6 Functions

Functions, sometimes called methods, contain statements which can process data. The statements can or cannot process data. Methods can be accessed by other statements. This action is referred to as *calling a function* or *making a function call*. In this chapter we touch on the `Start ()`, but we've yet to figure out why the `Start ()` functions work.

So far we've created a class and we gave the class some places to store data. We've even used some of those variables in Unity 3D. We saw how to add a C# class to our game objects in a scene. In Chapter 3, it was a cube in a scene. Then we exposed some more variables to the editor.

This coding, however, hardly makes for a very interesting video game. To add any sort of behavior, we'll need to add in logic. Logic can't be determined without a function to run the logic. The logic needs to process data for anything in the game to happen.

To coordinate this merging of data and logic, we'll need to write some functions.

4.6.1 What Are Functions?

In Chapter 3, we talked about the process of shampooing. A second look at the idea of a function is to comprehend the words *wash*, *rinse*, and *repeat* in more depth. If you've never heard of the word *wash* before then you're probably dirty, or a computer. Computers need every word defined for them before they understand it.

```
void Wash ()
{
}
void Rinse ()
{
}
void Repeat ()
{
}
```

On a superficial level, we've used the above code to define the three words used to shampoo hair. However, none of the definitions contains any actions. They define the words or identifiers, but when they're used nothing will happen. However, the computer will be able to `Wash (); Rinse (); Repeat ();` without any problems now.

In the `MyClass` we wrote:

```
public void PrintNum ()
{
    Debug.Log (anotherNum);
}
```

which is quite a simple function.

Now that we can write some functions for use in Unity 3D let's get to some real coding. To make use of all of these variables, we need functions. Functions contain the logic in every class. The concept is the same for most object oriented programming languages.

NOTE: Functions may look different in different programming languages, but the way they work is mostly the same. The usual pattern is taking in data and using logic to manipulate that data. Functions may also be referred to by other names, for example, methods.

The major differences come from the different ways the languages use syntax. Syntax is basically the use of spaces or tabs, operators, or keywords. In the end, all you're doing is telling the compiler how to convert your instructions into computer-interpreted commands.

Variables and functions make up the bulk of programming. Any bit of data you want to remember is stored in a variable. Variables are manipulated by your functions. In general, when you group variables and functions together in one place, you call that a class.

In Chapter 3, we discussed a bit about writing a class starting with a declaration and the opening and closing curly braces that make up the body. Functions are written in a similar fashion.

We start with the declaration `void MyFunction`; then we add in both parentheses `()` and the curly braces `{}`, indicating to ourselves that we may or may not fill this in later. The first pair of parentheses `()`, doesn't necessarily always require anything to be added. However, the curly braces `{}`, or the body of the function, do need code added for the function to have any purpose.

When writing a new function, it's good practice to fill in the entirety of the function's layout before continuing on to another task. This puts the compiler at ease; leaving a function in the form `void MyFunction` and then moving on to another function leaves the compiler confused as to what you're planning on doing.

The integrated development environment, in this case MonoDevelop, is constantly reading and interpreting what you are writing, somewhat like a spell checker in a word processor. When it comes across a statement that has no conclusive form, like a variable, function, or class definition, its interpretation of the code you're writing will raise a warning or an error. MonoDevelop might seem a bit fussy, but it's doing its best to help.

4.6.2 Unity 3D Entry Points

The Unity 3D developers provided us with `MonoBehaviour` which is a class that allows our C# classes to have direct access to the inner workings of the Unity 3D game scene. The `MonoBehaviour` class is a collection of all the functions and data types that are available specific to Unity 3D. We'll learn how to use these parts of Unity 3D in the following tutorials.

NOTE: It's also important to know that when learning the specifics of Unity 3D, you should be aware that these Unity 3D-specific lessons can be applied to other development environments. When you use C# with Windows or OS X, you'll be able to write applications for other operating systems for software other than Unity 3D.

Think of `MonoBehaviour` as a set of tools for talking to Unity 3D. Other systems have different sets of tools for making apps for other purposes. Learning how to use the other systems should be much easier after reading this book and learning how to talk to Unity 3D.

When dealing with most other C# applications, you'll need to use `Main()` as the starting point for your C# application. This is similar to many other development environments like C++ and C.

When the class we asked Unity 3D to create was made, Unity3D automatically added in `void Start ()` and `void Update ()` to the class. These two functions are called *entry points*. Basically, the base `MonoBehaviour` class has several functions, that include `Start ()` and `Update ()`, which are called based on events that happen when your game is running.

There are other functions that are also automatically called when `MonoBehaviour` is used: `Awake ()`, `OnEnable ()`, `Start ()`, `OnApplicationPause ()`, `Update ()`, `FixedUpdate ()`, and `LateUpdate ()`; these functions are commonly called when the object is created and your game is

running. When the object is destroyed, `OnDestroy()` is called. There are also various rendering functions that can be called, but we don't need to go over them here.

To better understand what happens during each frame of the game it's best to imagine the game operating a bit like a movie. At the very beginning of the game, when the player presses Start, every actor in the scene with a script based on `MonoBehaviour` has its `Start()` function called.

Calling a function basically executes the lines of code that live inside of the curly braces that follow the function's declaration. Before the end of the first frame, the `Update()` functions of each script in the scene are called in no particular order.

At the beginning of the next frame, if there are no new scripts in the scene, the `Start()` functions are not called. Only `Update()` functions are called from that point on. Only if new scripts are introduced to the scene are `Start()` functions called on scripts new to the scene. If there is no code in the `Start()` function or if there is no `Start()` function in the script, then nothing happens.

To benefit from anything happening on each frame you'll need to put code into the `Update()` function. When a class is first introduced into the game world you need to add code into the `Start()` function. There are several other entry points that we'll be working with. And you'll be able to make use of them all as you see fit. It's important to know that without these entry points your class will be functioning on its own.

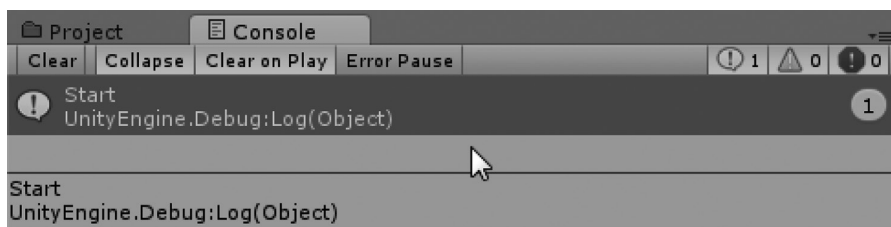
Functions all have specific names or identifiers. So far we've seen `Start()` and `Update()`, but there are many more functions which we'll make use of throughout this book. The first function we'll make use of the `Log()`, which is a function found inside of the `Debug` class. We'll return in a moment to how this function is used, but first we'll demonstrate how it's used.

If `Debug` is a class, this means that `Log()` is a member of `Debug`. The dot operator allows us to access the `Log()` function found in the `Debug` class.

```
void Start () {  
    //This will print out "start" in the Unity Console!  
    Debug.Log("start");  
}
```

Inside of the `Start()` function add in `Debug.Log("start");`, as seen in the example above. When the game is started the class will execute the code found inside of the `Start()` function. This means that the `Debug.Log("start");` statement is executed.

Continuing from the previous `FirstExample` project, click on the Play button and then click on the Console panel to bring it to the top. You should see the following lines of text in the Console panel.



The first line says `Start`, followed by `UnityEngine.Debug:Log(Object)`, which is the expected result. If not, then double check a few different things. First check that the line of code you wrote ends with a semicolon (;) and is not left empty. Next, make sure that you're using quotes around the word: "Start."

Then, check that your spelling and case are correct: `Debug` and `Log`. Also make sure you didn't forget the dot operator (".") between `Debug` and `Log`. All of the punctuation matters a great deal. Missing any one of these changes the outcome of the code. This is why syntax matters so much to programmers. Missing one detail breaks everything.

There are a few important things happening here, though many details left out for now. By the end of this chapter everything should be clear.

4.6.3 Writing a Function

A function consists of a declaration and a body. Some programmers like to call these *methods*, but semantics aside, a function is basically a container for a collection of statements.

4.6.3.1 A Basic Example

Let's continue with the `FirstClass` example project.

```
void MyFunction ()
{
}
```

Here is a basic function called `MyFunction`. We can add in additional keywords to modify the function's visibility. One common modifier we'll be seeing soon is `public`. We'll get further into the `public` keyword in Section 4.13 on accessibility.

```
public void MyFunction ()
{
}
```

The `public` keyword needs to appear before the return type of the function. In this case, it's `void`, which means that the function doesn't return anything. Return types are something else that we'll get into in Section 6.3.3, but functions can act as a value in a few different ways. For reference, a function that returns an `int` would look like this. A return statement of some kind must always be present in a function that has a return type.

```
public int MyFunction ()
{
    return 1;
}
```

The `public` modifier isn't always necessary, unless you need to make this function available to other classes. If this point doesn't make sense, it will soon. The last part of the function that is always required is the parameter list. It's valid to leave it empty, but to get a feeling for what an arg, or argument in the parameter list, looks like, move on to the next example.

```
public void MyFunction (int i)
{
}
```

For the moment, we'll hold off on using the parameter list, but it's important to know what you're looking at later on so it doesn't come as a surprise. Parameter lists are used to pass information from outside of the function to the inside of the function. This is how classes are able to send information from one to another.

We've been passing an argument for a while now using the `Debug` class. The `Log` member function of the `Debug` class takes a single argument. In `Debug.Log("start");` the "start" is an argument we've been sending to the `Log` function. The `Debug` class is located in the `UnityEngine` library and made available because of the `using UnityEngine;` statement.

We'll see how all of these different parts of a function work as we get to them. Functions are versatile but require to be set up. Luckily, code is easy to build up one line at a time. It's good to know that very few people are capable of writing more than a few lines of code at a time before testing them.

To use a function, you simply enter the identifier followed by its argument list.

```

void SimpleFunction ()
{
    Debug.Log ("simple function is being called");
}
void Start ()
{
    SimpleFunction();
}

```

The placement of `SimpleFunction ()` in the class has no effect on how it's called. For the sake of argument we can use `SimpleFunction ()` anywhere in the class as long as it's been declared at the class scope.

```

void Start ()
{
    SimpleFunction();
}
void SimpleFunction ()
{
    Debug.Log ("simple function is being called");
}

```

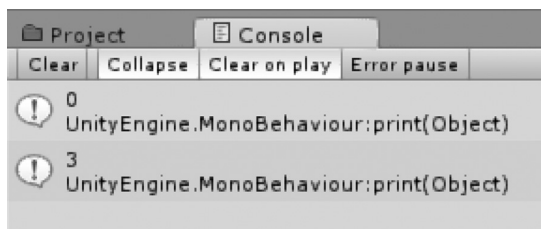
Here, we've used `SimpleFunction()` before it was declared. You may have noticed that `Debug.Log();` looks similar to our simple function. The identifier `print` followed by `()` with text inside of the parentheses is a commonly used function.

```

int a = 0;
void SetAtoThree ()
{
    a = 3;
}
void Start ()
{
    Debug.Log (a);
    SetAtoThree();
    Debug.Log (a);
}

```

For a function to work, we need to give it some instructions. If we declare an `int a` and set it to 0, we'll have some data to start with. Create a new function called `SetAtoThree()`, and then in its code block tell it to set `a` to 3. If we print `a` in the `Start ()` function we'll get 0, the value which `a` was set to in the beginning. Then the function `SetAtoThree()` was executed, which set `a` to 3, so when `a` is printed again its new output is 3.



4.6.4 More on White Space and Tabs

Tabs are used to delineate the contents of a function. So far, we've been seeing each function written with its contents tabbed to the right with a single tab.

```

void MyFunction()
{
→   //tabbed over once
→   int i = 0;
→   //while statement tabbed over once as well
→   while (i < 10) {
→       →   //contents of the while statement tabbed twice
→       →   print (i);
→       →   i ++;
→   }
}

```

The contents of the function are tabbed over once. The contents of the while statement inside of the function are tabbed over twice, as indicated by the arrows. This presentation helps to clarify that the contents of the while statement are executed differently from the rest of the function.

To help understand how the previous code fragment is read by the computer, we will step through the code one line at a time. This emulates how the function is run when it's called upon. Of course, a computer does this very quickly, but it's necessary for us to understand what the computer is doing many millions, if not billions, of times faster than we can comprehend.

```

void MyFunction()
{
    //tabbed over once
    int i = 0;
    //while statement tabbed over once as well
    while (i < 10) {
        //contents of the while statement tabbed twice
        print (i);
        i ++;
    }
}

```

NOTE: For a moment imagine your central processing unit (CPU) was made of gears spinning more than 2 billion times per second. The 2.4 Ghz means 2,400,000,000 cycles per second. A cycle is an update of every transistor in the silicon on the chip: over 2 billion pulses of electrons running through the silicon, every second turning on and off various transistors based on your instructions. Of course, not all of these updates are dedicated to your game; many of the cycles are taken up by the operating system running Unity 3D, so performance isn't always what it should be.

The first line `void MyFunction()` is identified by the computer and is used to locate the code to execute. Once the function is started the contents of the code begin at the first statement. In this case we reach `//tabbed over once`, which is a comment and ignored by the computer.

Next we reach `int i = 0;`, which is a declaration and assignment of a variable that is identified as `i`, an identifier commonly used for integers. This declaration tells the computer to create a small place in memory for an integer value and it's told that we're going to use `i` to locate that data.

Following the variable declaration, we get to another comment that is also ignored by the computer. The `while (i < 10) {` line of code follows the comment and this opens another space in memory for some operations. This also creates a connection to the `i` variable to check if the `while` statement will execute. If `i` is a value less than 10 then the contents will be executed; otherwise, the contents of the while statement are ignored.

Another important character the computer finds is the opening curly brace (`{`) that tells it to recognize a new context for execution. The computer will read and execute each line in the `while` statement till it finds a closing curly brace: `}`. In this new context we'll return to the first curly brace until the `while` statement's condition is false.

Because `i` is less than 10, the `while` statement's condition is true, so we will proceed to the first line of the `while` statement. In this case, it's a comment that is ignored. This is followed by a `print` function,

which also has a connection to the `i` variable. Therefore, this line prints to the console the value which is being stored at `i`. Once the value has been printed to the console the computer moves to the next line down.

The `i++;` statement tells the computer to look at the contents of `i`, add 1 to that value, and assign that new value back to `i`. This means that the new value for `i` is 1 since we started at 0. Once the computer reaches the closing curly brace we jump back to the top of the `while` statement.

The contents of the curly braces are repeated until the value for `i` reaches 10; once the iteration returns 11, the condition of the `while` statement changes from *true* to *false*. Because 11 is not less than 10, the statements found between the curly braces will not be read. Therefore, the code in the `while` statement will be skipped. Once the computer reaches the closing curly brace of `MyFunction()`, the computer stops running the function.

Computers do exactly what they're told and nothing more. The compiler runs using very explicit rules which it adheres to adamantly. Because of this, any strange behavior, which you might attribute to a bug, should point to how you've written your code. In many cases, small syntactical errors lead to errors which the computer will not be able to interpret.

It's greatly up to you to ensure that all of your code adheres to a format the computer can understand. When you write in English, a great deal of the sentiment and thought you put into words can be interpreted by each reader differently. For a computer there's only one way to interpret your code, so either it works or it doesn't.

4.6.5 What We've Learned

Function declaration is similar to class declaration. Functions are the meat of where logic is done to handle variables. How variables make their way into and out of a function is covered in Section 6.18. There are many different ways to handle this, but we'll have to handle them one at a time.

On your own experiment with functions like the following:

```
void ATimesA() {
    a = a * a;
}
```

You might want to set `int a` to a value other than 0 or 1 to get anything interesting out of this. Later on, we'll be able to see how logic and loops can be used to make functions more powerful.

4.7 Order of Operation: What Is Calculated and When

Variables and functions work together to make your class operate as a part of your game. As an example, we're going to move an object through space using some simple math. To use numbers correctly, we need to be very specific. Remember, computers do exactly what they're told and nothing more.

Code is executed only when the function the code lives in is called. If the function is never called then nothing in the function will run. Once a function is called, the operation starts with evaluating the code at the top and works its way down to the end of the function one line at a time.

The first order which we've been working with so far is by line number. When code is evaluated each line is processed starting at the top; then it works its way down. In the following example, we start off with an integer variable and perform simple math operations and print the result.

```
void Start ()
{
    int a = 1;
    print (a);//prints 1
    a = a + 3;
    print (a);//prints 4
    a = a * 7;
    print (a);//prints 28
}
```

As you might expect the first line prints 1, followed by 4, and then by 28. This is a rather long-handed method to get to a final result. However, things get a bit strange if we try to shorten this to a single line, as math operators work in a different order than we just used them.

```
void Start ()
{
    int a = 1 + 3 * 7;
    print (a); //prints 22
}
```

After shortening the code to a single line like in this example, we get a different result, 22, which means 3 and 7 were multiplied before the 1 was added. Math operators have a priority when being evaluated, which follows the order of precedence. The multiply and divide operators have a higher precedence than add and subtract.

When evaluating `int a`, C# looked at the math operators first, and then did the math in the following order. Because the `*` was seen as more important than the `+`, `3 * 7` was computed, turning the line into `1 + 21`; then, the remaining numbers around `+` were evaluated, resulting in `a` being assigned to 22, and then ultimately printing that to the Console output.

We'll observe the other behaviors and learn to control how math is evaluated. This section might not be so exciting, but it's worth the while to have this section as a reference. When your calculations begin to have unexpected results, it's usually because of order of operation coming into play and evaluating numbers differently from what you expected.

4.7.1 Evaluating Numbers

Calculation of variables is called *evaluation*. As we had mentioned before, each object added to the scene in Unity 3D's editor has several components automatically added when it's first created. To make use of these components, you simply use their name in your code. The first component we'll be using is transform attribute. Transform contains the position rotation and scale of the object. You can see how these are changed when you select the object in the scene and move it around.

You can also enter numbers using the Inspector panel, or even click on the variable name and drag left and right as a slider to modify the data in the field. Moving objects around with code works in a similar manner as entering values in the Inspector panel. The position of the object is stored as three float values called a `Vector3`.

Operators in C# are the tools that do all of the work in your code. The first set of operators we'll discuss is arithmetic operators.

4.7.1.1 Math

`+` `-` `*` and `%`

The `+` and `-` do pretty much what you might expect them to: $1 + 3 = 4$ and $1 - 3 = -2$; this doesn't come as a surprise. Divide uses the `/`, putting the first number over the second number. However, the number *type* is important to sort out before we get too much further.

```
28 // Use this for initialization
29 void Start ()
30 {
31     Debug.Log (10 / 3);
32     Debug.Log (10f / 3f);
33     Debug.Log (10.0 / 3.0);
34 }
```

The first line above is a $10/3$, which might not return what you would expect. Writing a number without any decimal tells the compiler you're using integers, which will give you an integer result. Therefore, the final result is simply 3, with the remaining one-third cut off since it's rounded down. The second line has an `f` after the number, indicating we want a float value. The third line without the `f` indicates a double value. This has twice the number of digits as a float.

```

! 3
  UnityEngine.MonoBehaviour:print(Object)
! 3.333333
  UnityEngine.MonoBehaviour:print(Object)
! 3.333333333333333
  UnityEngine.MonoBehaviour:print(Object)

```

The result here shows the different number of trailing 3s after the decimal. We mentioned in Section 4.4.1.2 that there are different types of numbers. Here is an example of what that really means. An integer is a whole number without any decimal values. A float is a decimal type with only 7 digits in the number; a double is quite a lot bigger, having a total of 15 digits in the number.

```

28 | // Use this for initialization
29 | void Start ()
30 | {
31 |     Debug.Log (10 / 3);
32 |     Debug.Log (10f / 3f);
33 |     Debug.Log (10.0 / 3.0);
34 |     Debug.Log (10000000.0f / 3.0f);
35 |     Debug.Log (1000000000000000.0 / 3.0);
36 | }

```

Doing this experiment again with larger numbers exposes how our numbers have some limitations.

```

! 3
  UnityEngine.MonoBehaviour:print(Object)
! 3.333333
  UnityEngine.MonoBehaviour:print(Object)
! 3.333333333333333
  UnityEngine.MonoBehaviour:print(Object)
! 3333333
  UnityEngine.MonoBehaviour:print(Object)
! 3333333333333333
  UnityEngine.MonoBehaviour:print(Object)

```

Like the integer, the large doubles and floats only have so many places where the decimal can appear. The number for the double is huge to begin with, but we should have more 3s after the number than are shown with a decimal; where did they go? Keep in mind there *should* be an indefinite number of trailing 3s but computers aren't good with indefinite numbers.

NOTE: Computers without special software can't deal with numbers with an unlimited number of digits. To show bigger numbers, computers use scientific notation like `E+15` to indicate where significant numbers begin from the decimal point. Games, in particular, rarely ever need to deal with such large numbers. And to keep processes running fast, numbers are limited.

Software applications designed to deal with particularly huge numbers tend to work more slowly as they need to run with the same limitations but instead perform operations on the first part of a number and then another operation on the second half of a number. This process can be repeated as many times as necessary if the number is even bigger. Scientific computing often does this and they have special number types to compensate. There are some additional number types, as well as other data types that aren't numbers. We'll leave the other data types for later (Section 5.3.4). The operators we're dealing with here are mainly for numbers.

Next up is the multiply operator (*). All of the limitations that apply to the divide operator apply to the multiply operator as well. One important thing to consider here is that multiplication is technically faster than division. Therefore, in cases where you want to do $1000/4$, the computer is going to be faster if you use $1000 * 0.25$ instead.

This difference is a result of how the computer's hardware was designed. To be honest, modern CPUs are so much faster than they used to be, so using a / versus a * will usually make no noticeable difference. Old school programmers will beg to differ, but today we're not going to need to worry about this technicality, though I feel every classically trained programmer wants to scratch out my eyes right now.

Last up is the %, which in programming is called modulo, not percent as you might have guessed. This operator is used in what is sometimes called clock math. An analog clock has 12 hours around it. Therefore, a clock's hours can be considered %12, or modulo twelve, as a programmer might say.

To use this operator, consider what happens when you count 13 hours on a 12 hour clock and want to know where the hour hand will be on the 13th hour. This can be calculated by using the following term: $13\%12$; this operation produces the value 1 when calculated. Modulo is far easier to observe in operation than it is to describe.

The output from the code below here repeats from 0 to 11 repeatedly. Clocks don't start at 0, but computers do. Therefore, even though *i* is not being reset and continues to increase in value. On each update *i* is not being reset to 0, instead it continues to increment until you stop the game. A programmer would say "int *i* mod twelve" to describe $i\%12$ shown here.

```
int i = 1;
void Update ()
{
    int mod12 = i % 12;
    Debug.Log(i + "mod12 = " + mod12);
    i++;
}
```

Programmers are particular about their vocabulary and syntax, so it's best to understand how to think and talk like one. Programmers will usually assume you already know something like this, but it is worthwhile elaborating on the topic. Compilers, however, don't all think in the same way. Therefore, if you're curious and feel like learning other programming languages in the future, you might find that other languages will behave differently.

4.7.1.2 Operator Evaluation

As we saw before with $1 + 3 * 7$ you may end up with different results, depending on the order in which you evaluated your numbers. To prevent any human misinterpretation of a calculation you can reduce the preceding numbers to $4 * 7$ or $1 + 21$. To make our intent clear we can use parentheses.

To gain control, we use the open parenthesis and close parenthesis to change the order of evaluation. For example, we can do this: $1 - 2 - (3 + 4)$. This turns into $1 - 2 - 7$, a whole different result: $-1 - 7$ yields -8 . A simple change in the order of operation can result in a completely different number when evaluated. This sort of thing becomes more important once we start shooting at zombies and dealing out damage. Some monsters might have thicker skin, and you'll want to reduce the damage done by a specific amount based on where he was shot.

4.7.1.2.1 A Basic Example

As we saw before, $1 + 3 * 7$ will result in 22. We could use the long-winded version and start with a, then add 1 to it, then add 3 to that, and then multiply that by 7 to get 24. There's an easier way.

```
void Start () {
    int a = (1 + 3) * 7;
    Debug.Log (a);
}
```

By surrounding a pair of numbers with parentheses, we can tell the compiler to evaluate the pair within the parentheses before the multiplication. This results in 28, similar to the long-handed version of the same math. To elaborate we can add in another step to our math to see more interesting results.

```
void Start () {
    int a = 1 + 3 * 7 + 9;
    Debug.Log (a);
}
```

The code above results in 31 printed to the Console panel. We can change the result to 49 by surrounding the $7 + 9$ with parentheses, like in the following.

```
void Start () {
    int a = 1 + 3 * (7 + 9);
    Debug.Log (a);
}
```

The compiler computes the $7 + 9$ before the rest of the math is computed. Parentheses take precedence over multiplication, regardless of where they appear. It doesn't matter where the parentheses appear, either at the beginning or at the end of the numbers: The parentheses are computed first.

Adding another set of parentheses can change the order of computation as well. Parentheses within parentheses tell the computer to start with the innermost pair before moving on. For instance $(11 * ((9 * 3) * 2))$ starts with $9 * 3$; this turns into $(11 * ((27) * 2))$, which then yields $27 * 2$, which turns into $(11 * (54))$; $11 * 54$ turns into 594. This can be switched round by shifting the placements of the parentheses to a different number. Try it out!

```
void Start () {
    int b = (11 * ((9 * 3) * 2)) ;
    Debug.Log (b);
}
```

At this point, you will need to be very careful about the placement of the parentheses. For each opening parenthesis, there needs to be a corresponding closing parenthesis. If you're missing one, then Unity 3D will let you know. It's important to have a sense of what sort of math to use and when to use it.

In this case, where the parentheses encapsulate the entire expression their presence is redundant. Taking out the first and last parentheses will have no effect on the final result. This is true for most operations, but for clarity it's useful to know that they are indeed optional. To be clear about this we can add any number of extra parentheses to the operation and there will be no other effect to the final result, though it may look a bit strange.

```
void Start () {
    int b = (((11 * ((9 * 3) * 2)))) ;
    Debug.Log (b);
}
```

To make things more clear, we can rearrange this expression to be a lot more readable by setting up different variables. For each set of operations we can create a new variable. The operations begin with the first variable; then, after the value is stored, the value is carried to the next time it's used. Just remember, a variable cannot be used until it's been created.

```
void Start () {  
    int a = 9 * 3;  
    int b = a * 2;  
    int c = b * 11;  
    Debug.Log (c);  
}
```

In this example, we've created a different variable for each part of the math operation. First, we created a $9 * 3$ variable assigned to `a`. After that we multiplied `a` by 2 and assigned that to `b`. Finally, we multiplied `b` by 11 and assigned that result to `c`. To view the final result, we printed out `c` to get 594, the same result as the long, fairly obscure operation we started with. This example shows a simple use of variables to make your code more readable.

This seems like we're back to where we started with, at the beginning of this section. To be honest, both methods are perfectly valid. The decision to choose one system over another is completely dependent on which you find easier to understand.

What we should take away from this is the order in which the lines are read. The operations start at the top and work their way line by line to the bottom of the function. As variables are assigned new values, they are kept until they are reassigned.

It's important to note that by creating more than one variable, you're asking the computer to use more space in your computer's memory. At this point we're talking about very small amounts of memory. A classically trained programmer might want to fix this code upon reading it, and then convert it to a single line. To be honest, it does take up plenty more space, not only visually.

When starting a new set of operations, it's sometimes helpful to use a more drawn-out set of instructions, using more variables. When you're more comfortable that your code is doing what you expect it to, you can take the time to reduce the number of lines required and do a bit of optimization. There's nothing wrong with making things work first and cleaning things up later.

4.7.2 What We've Learned

Therefore, you've been introduced to variables and you've learned about functions, two big components of programming needed to make a game. Directives will allow your code to talk with Unity 3D, and put meaning behind your variables and functions.

We're using C# and Mono, which is an object oriented programming environment. We just learned objects are all the little bits of info that make up your different C# classes. When we're done with writing some code, Unity 3D will automatically interpret your code for your game.

At this point we've got a pretty strong grasp of the basics of what code looks like and the different parts of how code is written. We've yet to make use of any logic in our functions but we should be able to read code and know what is and isn't in a function.

It would be a good idea, by way of review, to make a few other classes and get used to the process of starting new projects, creating classes, and adding them to objects in a Unity 3D Scene. Once you're comfortable with creating and using new classes in Unity 3D you'll be ready to move on.

4.8 Scope: A First Look

Encapsulation is the term given to where data can be accessed from. The amount of accessibility is called scope. The visibility of a variable to your logic changes depending on where it appears.

Encapsulation can hide, or change the accessibility of data between statements, functions, and classes. Another way to think about it is keeping data on a need-to-know basis. This invisibility is intended for several reasons. One reason is to reduce the chances of reusing variable names.

```
void Start ()
{
    for (int i = 0; i < 10; i ++) {
        print (i);
    }
    // i only exists between the { and }
```

For this example, we used the `print();` function. Thanks to a long history of programming, almost every language has some version of `print`; this function does the same thing as `Debug.log()`; but it just means less typing. There are often many ways to achieve the same result; it's just up to you to pick which one to use.

4.8.1 Class Scope

To get a better look at how scope and encapsulation work together, we're going to work with the Scope Unity project. Attached to the camera is the Scope Unity project which can be found in the downloaded materials provided. When the scene is run, the code attached to the camera will execute as normal.

```
using UnityEngine;
using System.Collections;
public class Scope : MonoBehaviour {
    int MyInt = 1;
    //Use this for initialization
    void Start () {
        Debug.Log(MyInt);
    }
}
```

The `MyInt` variable exists at the class scope level. `MyInt` can be used in any function that lives in the class scope. The example code you're looking at will produce the following Console output:



Both `Start ()` and `Update ()` live in the class scope so both can see the `MyInt` variable. Therefore, the following code will send a 1 to the console. Both `Start ()` and the `Update ()` functions will keep sending 1s to the Console window.

```
using UnityEngine;
using System.Collections;
public class Scope : MonoBehaviour
{
    int MyInt = 1; //this int is in the class scope
    void Start ()
    {
        Debug.Log (MyInt); //Start can see MyInt
    }
}
```

```

void Update ()
{
    Debug.Log (MyInt); //Update can see MyInt
}
}

```

Placement of a variable at the class scope has little effect on where it's used. The code above can be rearranged easily. The code below will behave in exactly the same way as the code above.

```

public class Scope : MonoBehaviour {
    void Update ()
    {
        Debug.Log (MyInt);
    }
    void Start ()
    {
        Debug.Log (MyInt);
    }
    int MyInt = 1;
}

```

The functions and variables act in the same way, irrespective of where they appear in the class. However, class scope is easy to override with other variables of the same name. This is called variable collision, the effects of which you should be aware of, as discussed in Section 7.4.

```

public class Scope : MonoBehaviour
{
    public int MyInt = 1; //declares MyInt
    void Start ()
    {
        int MyInt = 2; //declares a new MyInt, stomps on the first one
        Debug.Log (MyInt);
    }
}

```

If we declare a variable of the same name in more than one place then it will be overridden by the closest version of our variable. If you were to run this script in your scene, you'd have the Console printout 2 because of the line that preceded the print function call.

There are problems with how variables can collide with one another. And the best way to avoid this problem would be to come up with good names for your variables. Therefore, can we use both versions of MyInt?

```

public class Scope : MonoBehaviour
{
    public int MyInt = 1; //declares MyInt
    void Start ()
    {
        Debug.Log (MyInt); //being used before it's declared.
        int MyInt = 2; //declares a new MyInt, stomps on the first one
        Debug.Log (MyInt);
    }
}

```

The answer is no. The result of such a declaration will be an error: "A local variable MyInt cannot be used before it is declared." You may have expected the first print function to print out the class scope version of MyInt, but this isn't the case.

The variables that live at the class scope happen to still exist even if you stomp on their name with variables of the same name in a function. The keyword `this` preceding the variable unhides the variable that is living at the class scope.

```
void Start ()
{
    Debug.Log(this.MyInt); //class scope version of MyInt
    int MyInt = 3;
    Debug.Log(MyInt);
}
```

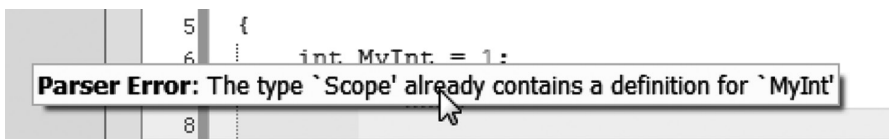
By creating an `int MyInt` inside of the `Start ()` function in your class you have effectively hidden the class scoped `MyInt` from the function. Although it can be accessed as `this.MyInt`, the `MyInt` in the function takes precedent.

This example should also highlight that a variable can be used only after it's been initialized. This is a situation best avoided, so it's good to remember what variable names you've used so you don't use them again.

To elaborate, a function starts from the first curly brace (`{`) and computes one line at a time, working its way from the top, moving toward the bottom till it hits the closing curly brace (`}`). If a variable is used but if a statement uses a variable before it's declared, an error results.

```
void Start ()
{
    Debug.Log(MyInt); //MyInt doesn't exist yet.
    int MyInt = 3;
}
```

Something that might be obvious but should be mentioned is the fact that you can't reuse a variable name. Even if the second variable is a different type, we'll get the following error in MonoDevelop explaining our error.



Another effect of creating a variable inside of a function is the fact that another function can't see what's going on. In the following code we declare a `StartInt` in the `Start ()` function. After that we try to use it again in the `Update ()` function, which results in an error.

```
void Start ()
{
    int StartInt = 100;
    print(MyPersonalInt);
}
void Update ()
{
    print(StartInt);
}
```

C# is telling us that we can't use a variable before it's declared. Of course, we did declare it once, but that was in a different function. The `Update ()` function can't see inside of the `Start ()` function to find the `MyPersonalInt` created in the `Start ()` function.

You'll have to keep this in mind once you start writing your own functions. There are two common ways to work with declaring variables. The first is to place all of your variables at the top of the function. This way you ensure that everything you may need to use is already declared before you need them. The second method is to declare the variable just before it's needed.

```
void Start ()
{
    int MyFirstInt = 100;
    int MySecondInt = 200;
    print(MyFirstInt);
    print(MySecondInt);
}
```

In this case, we're being very clear inside of this function about what variables we may use. For short functions this can sometimes be a lot easier to deal with. You can easily sort your variables and change their values in one place. It's also easier to check if you're accidentally reusing a variable's name.

```
void Start ()
{
    int MyFirstInt = 100;
    print(MyFirstInt);
    int MySecondInt = 200;
    print(MySecondInt);
}
```

In this second case, we're using the variable only after it's created. When writing long functions you may find the second method a bit more clear as you can group your variables together near the logic and functions that will be using them.

This is a small example of code style. When reading someone else's code, you'll have to figure out how they think. In some cases, it's best to try to match the style of existing code. However, when you're writing everything on your own, it's best to keep to your own style and decide how you want to arrange your code.

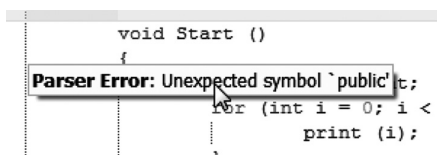
If we declare a variable at the class scope, all functions inside of the class can see, and access, that variable. Once inside of a function, any new variables that are declared are visible only to that function. Some statements can declare variables as well, and those variables are visible only inside of the statement they are declared. Once we get to know about more statements, we'll see how they work, but we'll need to keep scope in mind when we get there.

4.8.2 Function Scope

Variables often live an ephemeral life. Some variables exist only over a few lines of code. Variables may come into existence only for the moment a function starts and then disappear when the function is done. Variables in the class scope exist for as long as the class exists. The life of the variable depends on where it's created.

In a previous exercise, we focused on declaring variables and showing them in the Unity 3D's Inspector panel. Then we observed when variables need to be declared to be used without any errors. The placement and `public` keywords were necessary to expose the variables to the Unity 3D editor.

The `public` keyword can be used only at the class scope. This is called class scope as the variable is visible to all of the functions found within the class. Making the variable `public` means that any other class that can see this class can then also have access to the variable as well. We'll go into more detail on what this means and how it's used in a bit.



```
void Start ()
{
    Parser Error: Unexpected symbol `public`
    for (int i = 0; i <
        print (i);
}
```


Adding the `public int` within the `Start ()` function will produce an error. You're not allowed to elevate a variable to the class scope from inside of a function. MonoDevelop will inform you of the error: `Unexpected symbol 'public';` the reason is that variables within a function cannot be made accessible outside of the function.

There are systems in C# to make data within a function accessible, but using the keyword `public` is not it. We'll look into how that's done in Section 6.1. A variable declared inside of a function exists only as the function is used. Variables declared at the top of a function are declared at the function scope. This is to say that the variable is visible to any statement within the function.

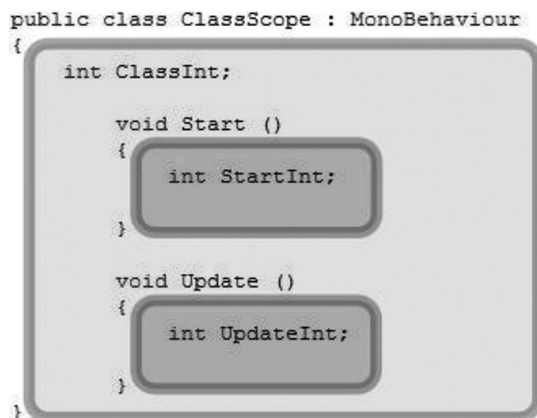
```
//Use this for initialization
void Start () {
    int StartInt = 1;
    Debug.Log(StartInt);
}
```

In a function, declaring `int StartInt` is valid, but its scope is limited to within the function. No other functions can see or directly interact with `StartInt`. There are methods to allow interaction within the function, but we'll get into that in Section 6.3.2.

The functions declared in the class are also a part of the class scope, and at the same level as the variables declared at the class scope level. If a function in a class is preceded by the `public` keyword, it's accessible by name, just like a variable. Without the `public` keyword the function will remain hidden. We'll make use of `public` functions in Section 6.3.2, but it's important to know that a function can be made publicly available, like variables.

Referring to the `Scope.cs` file we created for this chapter, we're going to make use of the functions given to us by the programmers at Unity 3D. The location of a variable in code changes where it can be seen from and how it can be used.

As noted, limiting accessibility of any variable is called encapsulation. The curly braces, `{}`, and parentheses, `()`, have a specific context. This context can keep their contents hidden from other sections of code. This means that `Start ()` can have one value for `StartInt`, and if you create a new `StartInt` in `Update ()`, it will not be affected by the `StartInt` in `Start ()`.



If we look at the above figure we can visualize how scope is divided. The outer box represents who can see `ClassInt`. Within the `Start ()` function we have a `StartInt` that only exists within the `Start ()` function. The same is repeated for the `UpdateInt`, found only in the `Update ()` function. This means that `Start ()` can use both `ClassInt` and `StartInt` but not `UpdateInt`. Likewise, `Update ()` can see `ClassInt` and `UpdateInt` but not `StartInt`.

In the diagram below the boxes represent different levels of scope.

```

public class ClassScope : MonoBehaviour
{
    int ClassInt;

    void Start ()
    {
        int StartInt;
        for( int i = 0; i < 10; i++ )
        {
            Debug.Log( i );
        }
    }

    void Update ()
    {
        int UpdateInt;
        for( int i = 0; i < 10; i++ )
        {
            Debug.Log( i );
        }
    }
}

```

Within each function lies a simple for loop. We'll go into more detail about how a for loop works, but in short, for loops create and use their own variables, which have limited scope and exist only within the confines of the loop. This also means that the for loop inside of the `Start ()` function has access to `ClassInt` as well as `StartInt`. Following the same restrictions, the for loop inside of `Update ()` can see `ClassInt` and `UpdateInt`, but not `StartInt` or `int i`, which exist in the for loop that's been written into the `Start ()` function.

4.8.3 What We've Learned

Encapsulation describes a system by which a variable's scope is defined. Depending on where a variable is declared, the scope of the variable changes. Encapsulation prevents variables from overlapping when they're used. Encapsulation, therefore, allows one function to operate independently from another function without our needing to worry about a variable's name being reused.

We'll cover more situations where scope becomes a useful tool to help classes message one another. Encapsulation and scope define a major principle of object oriented programming, and understanding this concept is important to learning how to write C#.

4.9 This

What happens when a class needs to refer to itself? In most cases, a class can use its own properties and fields and not get confused what variable is being referred to. For instance, we can take a look at a basic class that might look like the following.

```

public class MyThis {
    float MyFloat;
    public void AssignMyFloat(float f) {
        MyFloat = f;
    }
    public void ShowMyFloat() {
        Debug.Log(MyFloat);
    }
}

```

Here, we have a class which we can add to an `Example.cs` file. We'll look at `public class MyThis` and see that it's got a field called `MyFloat`. We've also got two functions inside of the class that assigns

the `MyFloat` field within the `MyThis` class. In the first function, we assign `MyFloat` to a parameter, and in the second function, `ShowMyFloat ()` we simply send the `MyFloat` from within the class to the Console panel. In use, the full code would look like the following.

4.9.1 A Basic Example

In the `KeywordThis.cs` file in the `Scope` project, we'll find the following code.

```
using UnityEngine;
using System.Collections;
public class KeywordThis : MonoBehaviour {
    public class MyThis {
        float MyFloat;
        public void AssignMyFloat(float f) {
            MyFloat = f;
        }
        public void ShowMyFloat() {
            Debug.Log(MyFloat);
        }
    }
    //Use this for initialization
    void Start () {
        MyThis mt = new MyThis();
        mt.AssignMyFloat(3.0f);
        mt.ShowMyFloat();
    }
}
```

In the `Start ()` function we print out 3 to the console. There's no confusion because `MyFloat` is never used in a situation where its name may be confused with another identifier. The situation changes should we change a parameter name in `AssignMyFloat`, for example, from `float f` to `float MyFloat`.

```
public void AssignThisMyFloat(float MyFloat) {
    MyFloat = MyFloat; //ambiguous MyFloat
}
```

If we add the `AssignThisFloat(float MyFloat)` function to the `MyThis` class, we'll get some rather unexpected behavior. If we use this to assign `MyFloat` in the `MyThis` class using the following statements, we will not get the same behavior as before. Unity 3D lets us know something might be misleading by prompting us with the following warning:

```
Assets/KeywordThis.cs(21,49): warning CS1717: Assignment made to same variable; did you mean to assign something else?
```

To solve this problem, we'll look at some simple solutions.

```
void Start () {
    MyThis mt = new MyThis();
    mt.AssignThisMyFloat(3.0f);
    mt.ShowMyFloat();
}
```

This sends a 0 to the Console panel. This is because in the function `AssignThisMyfloat`, where we assign the `MyThis` variable `MyFloat` to the `AssignThisFloat` argument `MyFloat`, the function assigned the class variable back to itself. The local scope within the function took precedence over the value stored inside of the class. We can correct the behavior by adding the keyword `this` to the function.

4.9.2 When This Is Necessary

```
public void AssignThisMyFloat(float MyFloat) {
    this.MyFloat = MyFloat;
}
```

With the addition of the keyword `this` to the function above, we can specify which version of the `MyFloat` variable we're talking about. Running the `Start ()` function again, we'll get the expected 3 being printed to the Console panel. It's easy to avoid the requirement of the `this` keyword. Simply use unique parameter names from the class's variables and you won't need to use the `this` keyword.

There is no unexpected behavior if we superfluously use the `this` keyword anytime we want to. The keyword does as you might expect it will. In both of the versions of the following function, we get the exact same behavior:

```
public void AssignMyFloat(float f) {
    MyFloat = f;
}
public void AssignMyFloat(float f) {
    this.MyFloat = f;
}
```

Of course, we don't want to use both in the same class at the same time, but it's important to know that either version can be used, just not at the same time. The use of `this` can help make the code more readable. In a complex class where we have many different variables and functions, it sometimes helps to clarify if the variable being assigned is something local to the function we're reading or if the variable is a class scoped variable. Using `this` we can be sure that the variable is assigned to a class variable, and not something only in the function.

4.9.3 Awkward Names

Naming variables can be difficult. Below we have an awkwardly named variable `int MyBadlyNamedInt` in an awkward class.

```
class MyAwkwardClass
{
    int MyBadlyNamedInt = 0;
}
```

Then we have a function that also happens to have a variable named `MyBadlyNamedInt` inside of it.

```
class MyAwkwardClass
{
    int MyBadlyNamedInt = 0;
    void PoorlyNamedFunction()
    {
        int MyBadlyNamedInt = 7;
    }
}
```

Things get strange if we were to need to call upon the class's badly named `int` rather than the function's badly named `int`. The class scoped `int MyBadlyNamedInt` still exists; it hasn't gone away. The same `int MyBadlyNamedInt` inside of the `PoorlyNamedFunction` is interfering with the same `int` at the class scope.

```
class MyAwkwardClass
{
    int MyBadlyNamedInt = 0;
```

```
void PoorlyNamedFunction()
{
    Debug.Log(this.MyBadlyNamedInt);
    int MyBadlyNamedInt = 7;
    Debug.Log(this.MyBadlyNamedInt);
}
```

The `Debug.Log` before or after the declaration of the function's bad `int` will send a 0 to the Console panel. Without the `this` keyword added before the identifier, we assume that we're using the function's named version of the variable.

If the `this` keyword had not been used, then we'd get a conflict with the first use of `MyBadlyNamedInt`. Within the `PoorlyNamedFunction()`, C# has noticed that the `BadlyNamedInt` is declared again and ignores the class scope `int` of the same name.

4.9.4 What We've Learned

In general, `this` is a somewhat superfluous keyword. The necessity of using the keyword `this` arises only when a name of a parameter in a function matches the name of a variable in the class where the function's parameter appears.

When naming variables and functions, it's best to keep to a consistent pattern. In general, many people use longer names at the class scope. For instance, the following code uses a pattern of shorter and shorter variable names depending on the variable's scope:

```
class MyClass {
    int MyInt = 0;
    void MyFunction(int mi) {
        int mInt = mi;
        MyInt = mInt;
    }
}
```

At the scope of the entire class the `int` is `MyInt`. Within the function `MyFunction` in `MyClass` a local variable only visible to `MyFunction` is called `mInt`. The variable from the function's argument list is simply `mi`.

There are obvious limitations to using a convention like shortening variable names. It's very easy to run out of letters and end up with unclear variable names, but it's a simple place to get started.

This isn't a common practice convention by any means; it's one that I've personally become used to using. These sorts of personal conventions are something that everyone needs to invent for his or her individual use. I'd expect you to find your own convention for naming variables.

The flexibility of C# allows everyone to have a personal style of writing code. Identifiers aren't the only place where one can find self-expression through code. Another way to express ones self through code is deciding to use a `for-while` or `do-while` loop. Both can be used for the very similar tasks, but the decision is ultimately left to the programmer writing the code.

For any reason should you feel like you prefer one method over another you should follow your own intuition and write how you like to. Programming is a creative process and writing expressive code is a process just like any other form of art.

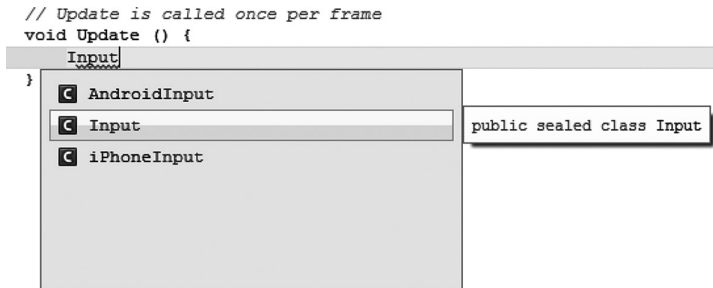
4.10 Turning Ideas into Code—Part 1

When you need to take a task and turn it into code, you need to start off with the first action taken by the player and turn that into something that the computer can use. Or rather collect incoming data and then process it. Often, when this is the input, the input needs to be recorded and dealt with in a meaningful way.

If your primary form of control involves mouse and keyboard, then it's best to start by making sure you can read both the mouse and keyboard inputs. If you're using a touch screen then you need to make sure you can read where and when touch input happens. When using Unity 3D you've got the `Input` class to start with; everything you need will be found in there.

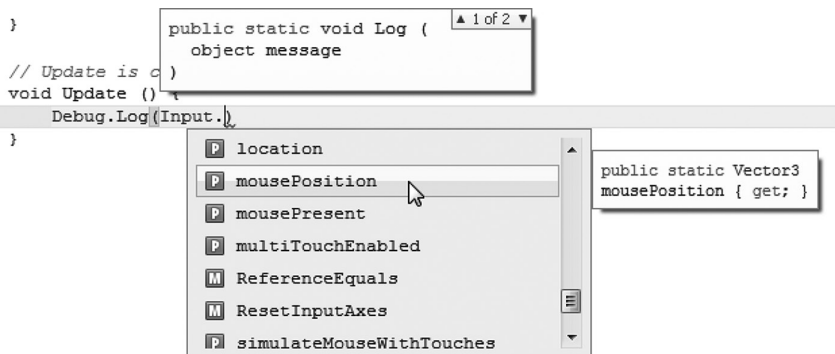
To follow along, start with the `IdeasPartOne` project from the downloaded projects file provided. In here, we'll look at the `Example.cs` class in the `Assets` directory.

MonoDevelop makes looking for the code you need easier by guessing what it is you're looking for. Start entering the word `input` and then MonoDevelop guesses what class it is you're going to pull from. If you're unsure where to start looking, a search through a web search engine is a good place to start. Chances are the question you have in your head has been asked on a forum and answered in some way, often with some code to help you toward your goal.



A simple Google search for “Unity 3D How do I read the keyboard” takes you to either the Unity 3D online documentation or to <http://answers.unity3d.com>; either of these web pages will at least give you a clue or two about what function or class to look at.

Say, for a moment, we want to read the mouse movement; to check we are able to do this, we'd be able to use the `print` function to see what sort of data we get from the `Input` class.



The closest thing we're able to find which might lead to a mouse position seems to be `Input.mousePosition`. The `Input.mousePosition` returns a `Vector3` type variable. A `Vector3` is made of three parts, a float `x`, `y`, and `z` value and we can expect that this is related to the `Input` in some way. To find out what values the `Input.mousePosition` returns we can use the `Debug.Log(Input.mousePosition);` entered into the `Update ()` function of the `Example.cs` class found in the provided project. To make sure that the `Update ()` function is called we'll attach the `Example.cs` class to the main camera in a new empty scene.

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

While the game is playing, the first and second values printed out to the Console window in Unity 3D change when the mouse is moved. The connection between the mouse and the value returned by `Input.mousePosition` make sense, but how do we use this to control a camera or even a character? We get two fairly big numbers which represent some point on the screen.

```
(445.0, 422.0, 0.0)
UnityEngine.MonoBehaviour:print (Object)
Example:Update () (at Assets/Example.cs:9)
```

Turning the data from `Input.mousePosition` into something, we can use to move a character or camera for a game that requires a bit of manipulation. Before we start using the `Input.mousePosition` values we need to decide what we're going to do with them. In a classic FPS the camera is always under control by the mouse. Therefore, we should start by doing the same.

With the script attached to the camera, we can begin to manipulate various aspects of the object that the code is attached to.

In a new Unity 3D project, we should have the `Example.cs` attached to the Main Camera in the scene. This file will allow us to manipulate the camera with mouse input.

4.10.1 Mouse Input

```
using UnityEngine;
using System.Collections;
public class Example : MonoBehaviour {
    //Use this for initialization
    void Start () {
    }
    //Update is called once per frame
    void Update () {
        transform.rotation =
Quaternion.Euler(new Vector3(Input.mousePosition.x, Input.mousePosition.y, 0));
    }
}
```

To get started, we'll use a single line of code in the `Update ()` function. The line `transform.rotation = Quaternion.Euler(new Vector3(Input.mousePosition.x, Input.mousePosition.y, 0));` is a long line of code. And it doesn't quite work how you might expect. When you add a few objects into your scene and move the mouse with the game running, you'll see that the camera is indeed spinning around and the mouse is rotating the camera's point of view. However, the behavior isn't exactly the same as that of an FPS.

This is a start, and you might be able to see how quickly we can begin to manipulate objects in a game with familiar systems. However, the refinement to get from here to a full-blown first-person controller takes much more than one line of code. Even so, there are many things to explain in that single line; no doubt there are many questions you might have as to what's really going on here, and that's good.

We will learn what all of that means, but one thing at a time. We still have many different terms to figure out before we get to that.

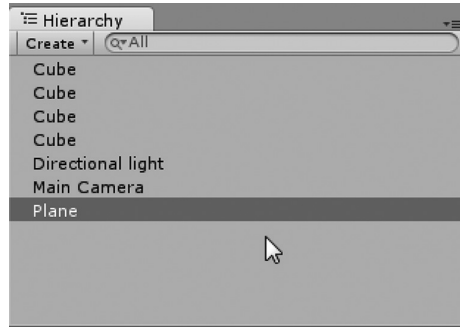
4.10.2 GameObject

If we take a step back from mouse input, we would ask the following question: What is the rotation of the camera? Rotation data is easily accessed by any script attached to the camera object in the scene, but why? When a script is attached to an object in Unity 3D, it becomes a component of that object.

Classes, as we know, can be created as an aggregate of other classes. An object is created from a blueprint, or rather a class. That process of creating an object from a class is called instantiation. And by adding components to a `GameObject` in the `UnityEditor`, we're providing Unity 3D with `GameObject`

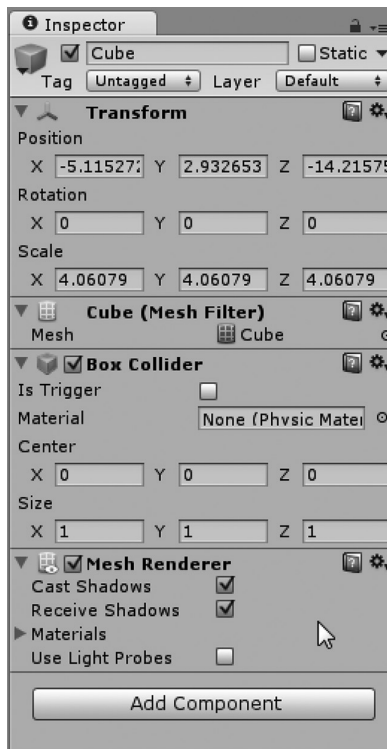
construction information. This info is used to instruct Unity 3D what objects to instantiate for the `GameObject`.

Specifically, in code, the C# file you attach is instantiated by the `GameObject` when the game starts. As a matter of fact, practically every object in a Unity 3D Scene is a `GameObject`. The camera, lights, boxes, and so forth, everything you see in the scene, is a different instance of the `GameObject` class, each one with different attachments.



Everything you see in the Hierarchy panel is a `GameObject`. They are all instances of the `GameObject` class found in the `UnityEngine` library. This point is significant because this means that Unity 3D has done some automatic code generation for you. This is, of course, what a game engine is supposed to do. The Unity 3D engineers put a great deal of work building tools with a pretty User Interface. As a level designer you're able to manipulate your values in your code while the game is running.

`GameObject` is made up of several other classes. You can see what other classes are being used in the `GameObject` when it's selected. The Inspector panel shows you some of the classes that are a part of the selected `GameObject`.



The Transform, Mesh Filter, Box Collider, and Mesh Renderer are all classes that are all members of the GameObject in the scene. To dig another level deeper, Transform has Position, Rotation, and Scale as data members. Each other class listed in the Inspector panel has data members as well. For every script you add to each GameObject, you're adding new data members to the GameObject.

So, to a certain extent, when you add objects to the scene, you're adding code to the scene. When you attach components to a GameObject, you're writing code to instantiate your class in that GameObject. At all levels, every action you take in Unity 3D is calling some class function, or manipulating the data of a member of a class. This goes the same for every other piece of software you use every day; it's all code.

This might seem somewhat existential, but it's to our advantage to understand that a GameObject is a class. Every time we add our code as a component, we're telling GameObject to instantiate our class. Once instantiated, our code has become a part of the Unity 3D Scene and can both read and write data to the scene it's in.

Once we get a better grip of the rest of the terms required for writing code, we'll start adding and removing objects in the Unity 3D Scene, but for now, just get used to the idea that everything you do is simply executing some code in Unity 3D.

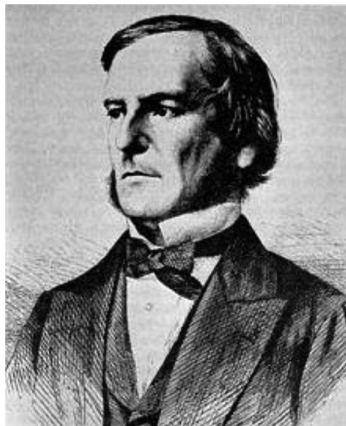
4.10.3 What We've Learned

Figuring out what we have to work with is pretty simple. MonoDevelop assists in various ways, from telling us when we've done something wrong, or what might break. MonoDevelop also provides us with plenty of clues to help us figure out what options we have so far as functions and classes are concerned.

Learning the tricks required to make these clues into functions which do what we want takes a bit of learning and practice. Asking the right questions on forums will only get us so far. By spending time to learn how to use the data we have and how to apply the data through logic, we'll be able to accomplish quite a lot on our own.

To learn how to do all of these things takes time and effort. It's also a good idea to do plenty of research on your own before asking for help. Many problems have already been solved in many different ways. It's up to you to find one that you can work with.

4.11 Logic and Operators



Logic allows you to control what part of a function is evaluated based on changes to variables. Using logic, you'll be able to change which statements in your code will run. Simply put, everything you write must cover each situation you plan to cover. Logic is controlled through a few simple systems, primarily the `if` keyword and variations of `if`.

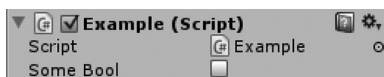
4.11.1 Booleans

Using George Boole's logic is a basic tool in nearly every programming paradigm. In C# booleans, or bools for short, are either true or false. It's easiest to think of these as switches either in on or in off position. To declare a var as a bool, you use something like the following.

To start off, we'll begin with the Bool project in Unity 3D.

```
public class Example : MonoBehaviour
{
    public bool SomeBool;
```

By using `public` you'll expose the boolean variable to Unity 3D's editor. This declaration will also make the variable appear in the Inspector panel like so. The boolean needs to appear here to be seen by the rest of the class in Unity 3D. We'll learn why this is so in Section 4.13 on scope, but we'll digress for now.



`Some Bool` appears in the Inspector as an empty check box. This decision means that you can set the bool as either true by checking it on or false by leaving it unchecked. This allows you to set your public variables before running the assigned script in the game.

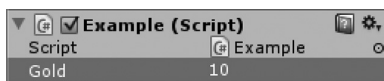
NOTE: C# does a pretty good job of automatically initializing many of its built-in types. Number values are always initialized to 0 when they are not given a value. Booleans are initialized as false when not given a value. This is handy to know since not all languages do this.

For instance, Lua, another commonly used game programming language, assumes nothing about the value until it's given one. Therefore, when any variable is first declared, it's initialized as nil, which is similar to C#'s behavior when creating a variable of a more complex data type. Though C# calls this uninitialized state null, this means the same as nothing. For good measure, UnrealScript uses the keyword *none* to mean the same as null.

Changes made before the game is run will be saved on that object. For making a game, for example, you can place treasure chests throughout a dungeon and set the amount of gold in each one. This is great since you will not need to make a different script for each treasure chest.

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

For instance setting a number for gold in the treasure chest will allow you to change the gold in each instance.



4.11.2 Equality Operators

Equality operators create boolean conditions. There are many ways to set a boolean variable. For instance, comparisons between values are a useful means to set variables. The most basic method to determine equality is using the following operator: `==`.

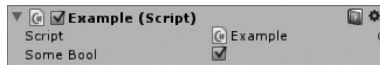
There's a difference between use of a single and a double equals to symbol. `=` is used to assign a value whereas `==` is used to compare values.

4.11.2.1 A Basic Example

When you need to compare two values you can use the following concept. You'll need to remember that these operators are called equality operators, if you need to talk to a programmer. The syntax here may look a bit confusing at first, but there are ways around that.

```
void Start ()
{
    SomeBool = (1 == 1);
}
```

There are other operators to be aware of. You will be introduced to the other logical operators later in the chapter. In this case, we are asking if two number values are the same. When the game is started, you'll be able to watch the `SomeBool` check itself to true. To explain how this works, we'll need to look at the order of operation that just happened. First, the right side of the first single `=` was calculated. `1 == 1` is a simple comparison, asking if 1 is the same as 1, which results with a true value. Test this out and check for the result in Unity 3D.



To make this a more clear, we can break out the code into more lines. Now, we're looking at a versus b. Clearly, they don't look the same; they are different letters after all. However, they do contain the same integer value, and that's what's really being compared here.

```
void Start ()
{
    int a = 1;
    int b = 1;
    SomeBool = (a == b);
}
```

Evaluations have a left and a right side. The single equal to operator (`=`) separates the different sides. The left side of the `=` is calculated and looks to the value to the right to get its assignment. Because `1 == 1`, that is to say, 1 is equivalent to 1, the final result of the statement is that `SomeBool` is true. The check box is turned on and the evaluated statement is done.

```
void Start ()
{
    int a = 1;
    int b = 3;
    SomeBool = (a == b);
}
```

As you might expect, changing the value in one of the variables will change the outcome of this equality test. `SomeBool` will be clicked off because the statement is no longer true. Or in plain language, Is a equal to b? The answer is no, or as far as `SomeBool` is concerned, false.

4.11.3 If and Branching

In the bool project from the downloaded content, we can control the rotation of the cube in the scene by adding an `if` logic statement. `If` statements are sometimes called branching statements. Branching means that the operation of code will change what code is executed.

```

void Start ()
{
    SomeBool = (1 == 1);
    int a = 1;
    int b = 3;
    SomeBool = (a == b);
    if (SomeBool) {
        transform.Rotate (new Vector3 (45f, 0f, 0f));
    }
}

```

The variable `SomeBool` becomes `true` once the right side of the statement is evaluated. Before the assignment, it remains in the state it was before the assignment. If you leave the `SomeBool` variable in the Inspector panel in the editor unchecked, it will check itself on when you run the game. Likewise, we can also show the following.

```

void Start ()
{
    if (true) {
        transform.Rotate(new Vector3(45f, 0f, 0f));
    }
}

```

The `if` statement is always followed by a pair of parentheses, `()`, which is followed by an opening and a closing curly brace: `{}`. The contents of the parentheses tell the compiler whether or not to execute the contents of the following curly braces. If the contents of the parentheses are `false`, then the contents of the curly braces are ignored.

```

public class Example : MonoBehaviour {
    public bool SomeBool;
    //Use this for initialization
    void Start ()
    {
        if (SomeBool) {
            transform.Rotate(new Vector3(45f, 0f, 0f));
        }
    }
}

```

As you might expect here, by clicking `SomeBool` to `true` before the game is run, the object this script is attached to will rotate 45 degrees on the *x*-axis. If not, then the cube will not be rotated when the game is started. Test this out on your own and observe the results.

```

void Update ()
{
    if (SomeBool) {
        transform.Rotate(new Vector3(45f, 0f, 0f));
    }
}

```

Move the code into the `Update ()` function and you'll be able to turn the cube while the game is running; we don't have code yet to turn it back so it's a trick that only works once. We'll find out later how to make this more interesting.

There are many ways to control how booleans are set. The keyword `if` is used most often to control the flow of your logic. Simply put, if something is `true`, then do something; otherwise, C# will ignore the statement and move on, and move on, as we have observed. What if we want to do the opposite?

4.11.3.1 Not!

If we are looking for a false value to trigger an evaluation of code, we'd need to look for a *not true* to evaluate the `if` statement. This code will look slightly different, as it includes the `!` operator.

```
void Start ()
{
    if (!SomeBool) {
        transform.Rotate(new Vector3(45f, 0f, 0f));
    }
}
```

This change reverses the behavior of the `if` statement. If you leave `SomeBool` as false when the `if` statement is evaluated, then code inside of the curly braces is evaluated. In this case the `!` is called a “not.” The not looks for a false value to determine if the `if` statement will be evaluated, or rather it reads the `SomeBool` as what it is not.

We can use this in another way with another equality operator.

```
void Start () {
    int a = 1;
    int b = 1;
    SomeBool = (a != b);
}
```

The `!=` operator is read as “not equal,” which looks at the following and returns false. When `a = 1` and `b = 1`, they are equal. `SomeBool` is asking, Is `a` not equal to `b`? The answer is false: `a` is not *not equal* to `b`. The logic can get a bit awkward, but it's the same as a double negative in English grammar.

4.11.4 Flowcharts

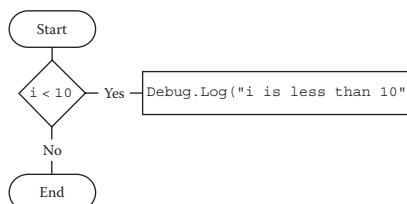
Flowcharts, sometimes called activity diagrams, are often used to graph out an algorithm, or flow of logic. This makes a whiteboard in a programmer's office a sort of visual work space for testing out a concept to create a piece of code. There is software available that can reverse engineer code and build a flowchart diagram to help visualize what the code is doing.

In the 1970s, Paul Morrison of IBM wrote a book on flow-based programming. This book takes the idea of following data between nodes connected by lines to write software. Unfortunately, the concept has yet to take hold, and traditional written code remains prominent. The concept is still powerful, and learning how flowcharting works is helpful.

The flowchart involves three parts: a beginning and end, logic, and a sequence. Code we've written so far is like the below:

```
void Update ()
{
    int i = 1;
    if (i < 10) {
        Debug.Log("i is less than 10");
    }
}
```

This code can be translated into nodes with the following diagram:



With some diagramming schemes, however, it's easy to lose some clarity. The more the data added to a diagram, the more confusing it can be. However, in a general form, logic used and the result of that application are easier to understand.

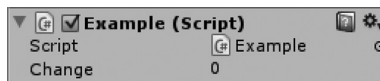
It's because of this simplicity some programmers opt to diagram a complex problem with a flowchart. Once you are able to trace the behavior of your data from start to end, it becomes more clear what your code must do. Using a flowchart as a guide, you can have a better idea what your code must accomplish when you start writing.

Game designers should also be able to explain a specific design decision in terms of a flowchart. When a monster shows up the game designer must be able to explain with simple logic what happens next. User interfaces and even the computer's AI behavior. All of these game events can benefit from having a flowchart as a basis for design decisions.

4.11.5 Relational Operators

Bool values can also be set by comparing values. The operators used to compare two different values are called relational operators. We use `==`, the is equal symbol, to check if values are the same; we can also use `!=`, or not equal, to check if two variables are different. This works similarly to the `!` in the previous section.

Programmers more often check if one value is greater or lesser than another value. They do this by using `>`, or greater than, and `<`, or less than. We can also use `>=`, greater or equal to, and `<=`, less than or equal to. Let's see a few examples of how this is used. First, let's add in a public variable `int` called `change`. This should appear in the Inspector panel when you select the cube with the attached `Example.cs` script.



Then, add in the `if` statement with the following parameter in parentheses. Then fill in the curly braces with what we've learned so far, setting the `transform.position` to a new vector. This time we're adding the code to the `Update ()` function since it's easier to observe when evaluated frame by frame.

```
public class Example : MonoBehaviour
{
    public int change;
    void Update ()
    {
        bool changed = change > 10;
        if (changed) {
            transform.position = new Vector3(3f, 0f, 0f);
        }
    }
}
```

Create a `bool` called `changed`, then assign its value to `change` being greater than 10. In the `if` statement, we can decide to evaluate the contents based on `changed` being true or false. We can shorten this code by skipping the creation and assignment to `change`, by adding the condition directly to the `if` statement's parameter, as in the following.

```
void Update ()
{
    if (change > 10) {
        transform.position = new Vector3(3f, 0f, 0f);
    }
}
```

This bool can make the code more or less readable, depending on how easily you can read and interpret what the code is doing. Most programmers would prefer that the extra `changed = change > 10;` be omitted, as this means there are fewer lines of code to read. Both versions are perfectly valid; it's up to you to decide on which you need to use based on your own skill.



When you run the game, you should be able to click and drag on the `change` variable to increase or decrease the value of `change`. Once the value of `change` is greater than 10, the cube will jump to the location you set in the `if` statement.

Lowering the `change` below 10 will not return the cube. This is because we have a statement to move it only when the `change` value is above 10. To execute another set of code when `if` returns not true, we need to use the `else` keyword.

4.11.5.1 Else

The `else` keyword follows the curly braces that `if` is evaluating. Here, we add in `else transform.position` is set to `new Vector3(0,0,0)`, setting the cube's position to the origin of the world.

```
void Update ()
{
    if (change > 10) {
        transform.position = new Vector3(3f, 0f, 0f);
    } else {
        transform.position = new Vector3(0f, 0f, 0f);
    }
}
```

Running the code again and playing with the value of `change` will result in the cube jumping to a new position any time `change` is above 10, and then moving back to the origin when the value is less than 10. However, setting the value to 10 exactly will not fulfill the `if` statement. To do this, we need to use the following code.

```
void Update ()
{
    if (change >= 10) {
        transform.position = new Vector3(3f, 0f, 0f);
    } else {
        transform.position = new Vector3(0f, 0f, 0f);
    }
}
```

Setting the `if` statement to `change > = 10` means that if the number is 10 or above, then the condition of the statement is true and the first part of the `if` statement is evaluated. Otherwise, the section encapsulated in the `else` part of the statement is evaluated. This logic allows for two states to exist. If we need to check for multiple conditions to be set, then we're going to need another case to exist.

```
void Update ()
{
    if (change > = 10) {
        transform.position = new Vector3(3f, 0f, 0f);
    }
}
```

```

        if (change < 10) {
            transform.position = new Vector3(0f, 0f, 0f);
        }
    }

```

We could use two `if` statements to do the same thing, as with the `if-else` statement. This code is going to do the same thing, but this will drive a programmer crazy. The code will also allow for more errors and odd behavior if you get one of the `if` statements wrong.

4.11.5.2 Else If

So far, we've observed `if` and `else`. To allow for more than two states, we can use `else if` to further add conditions to our logic.

```

void Update ()
{
    if (change >= 10) {
        transform.position = new Vector3(3f, 0f, 0f);
    } else if (change <= -10) {
        transform.position = new Vector3(-3f, 0f, 0f);
    }
}

```

We can leave the code here for you to test. Slide `change` back and forth to numbers greater and less than 10 and -10. This looping will make the cube jump between two positions. If we want to have a default when the `change` value is between -10 and 10. Next, we'll want a third condition when neither `if` nor `else` is true.

```

void Update ()
{
    if (change >= 10) {
        transform.position = new Vector3(3f, 0f, 0f);
    } else if (change <= -10) {
        transform.position = new Vector3(-3f, 0f, 0f);
    } else {
        transform.position = new Vector3(0f, 0f, 0f);
    }
}

```

With the addition of a final `else`, we catch the case when neither `if` nor `else if` is true. Now, we've got three places for the cube to jump to when playing with the value of the variable `change`. If this logic seems obscure, it's a good exercise to think the steps out loud using plain English. If `change` is greater than or equal to 10, then transform the cube to 3, 0, 0. Else if the change is less than or equal to negative 10, then set the position to negative 3, 0, 0. Else set the position to 0, 0, 0.

`if` statements need to begin with `if`. The last statement needs to end with either `else if` or `else`. An `else` that comes before an `else if` will create an error. The final `else` needs to come at the end of a long chain of `else if` statements.

There are no limits to what can be in each statement, though the logic might lose some of its sensibilities when using different arguments in each statement.

4.11.6 Rearranging Logic

Once we start using a multitude of variables combined with logic, we might start to use more variables. It becomes important to revisit a bit about scope, so we can remember how long and where a variable exists.

Download the project from the website <https://github.com/badkangaroo/UnityProjects/archive/master.zip> and open the LogicScene. Open the `Logic.cs` found in the Project panel that has been attached to a cube found in the game hierarchy. MonoDevelop should launch. A few public variables have been created and should already have `A_Cube`, `B_Cube`, and `C_Cube` assigned in the scene. If not, then it's easy to drag and drop the objects into their slots in the Unity Editor's Attribute Editor panel.

Now, let's focus on the `Update ()` loop, to add some behavior to our little cube friend. A quick search on Google for "Unity change material color" reveals `renderer.material.color = Color.red`. To see if this function works, we'll add it to our first `Update ()`;

```
void Update () {
    renderer.material.color = Color.red;
}
```

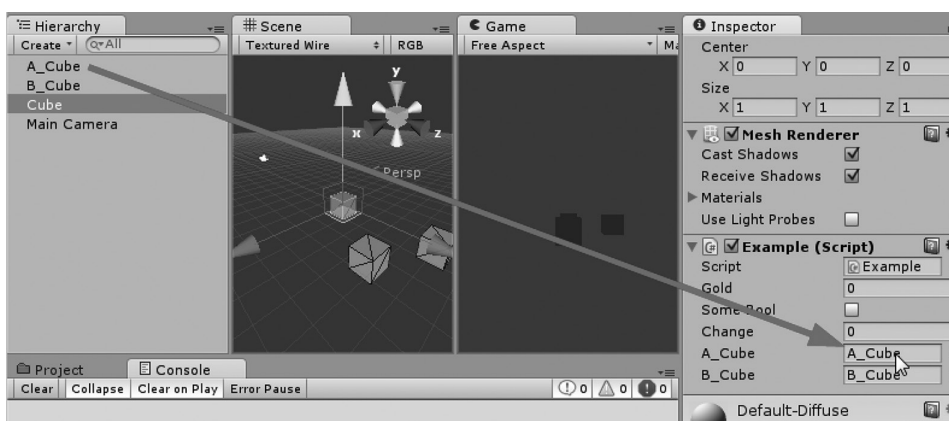
Sure enough, our cube changes to a red color when the game is started. Okay, so we know this function does what we need it to.

```
void Update () {
    Color col = Color.red;
    renderer.material.color = col;
}
```

To have a visible change in the game while it's playing we need to change the color assignment into a variable. However, in this example, we've set it once dynamically when it was declared. Checking again in the game confirms that this works just as well.

```
public GameObject A_Cube;
public GameObject B_Cube;
void Update ()
{
    Color col = Color.red;
    float Ax = A_Cube.transform.position.x;
    float Ay = A_Cube.transform.position.y;
    float Bx = B_Cube.transform.position.x;
    float By = B_Cube.transform.position.y;
    renderer.material.color = col;
}
```

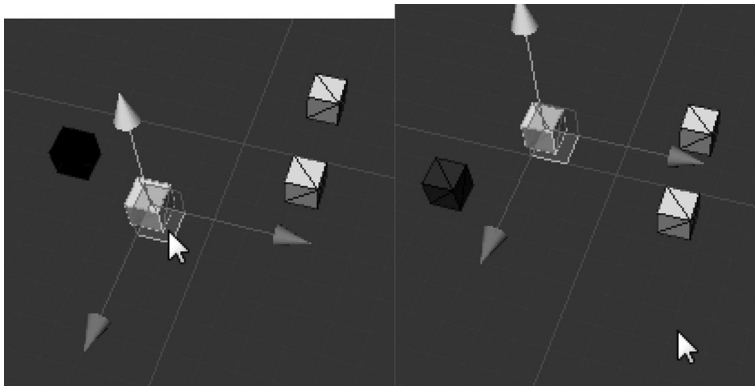
Next, we're going to want to collect some data from the scene from the other cubes placed in the world. We've added an `A_Cube` and a `B_Cube` to the class. To assign these, create a couple more cubes in the scene and drag them from the hierarchy into the slots in the Inspector panel.



We now have *x* and *y* coordinates from both *A_Cube* and *B_Cube*. Now, to create a simple placement puzzle, we can use a bit of logic to manipulate the color of the cube. We're going to check the *x* coordinate of *A* against the *x* coordinate of *B*. When *Ax* is greater than *Bx*, we're going to change the color of the cube from black to blue.

```
public GameObject A_Cube;
public GameObject B_Cube;
void Update ()
{
    Color col = Color.red;
    float Ax = A_Cube.transform.position.x;
    float Ay = A_Cube.transform.position.y;
    float Bx = B_Cube.transform.position.x;
    float By = B_Cube.transform.position.y;
    if (Ax > Bx) {
        col = Color.black;
    } else if (Ax <= Bx) {
        col = Color.blue;
    }
    renderer.material.color = col;
}
```

To see this code in action, go to the Scene panel, select either the *A* or the *B* cube, and move it around following the *x* or red arrow.



Now moving around either the *A_Cube* or the *B_Cube* around on the *x* will switch the color of the cube from blue to black. The placement of the *if* statement is important. The order in which *if* statements appear in the code has a direct effect on the final color of the material we're setting. Focus on the first *if* statement.

```
if (Ax > Bx) {
    float d = Ax + Bx;
    if(d > 5.0f) {
        col = Color.yellow;
    }
    col = Color.black;
} else if (Ax <= Bx) {
    col = Color.blue;
}
```

In the first statement in the code below, we set *col = Color.black*, before the *if (d > 5.0f)* statement. If the distance between *Ax* and *Bx* is greater than 5, we set the color to yellow. However, in this example, nothing happens.

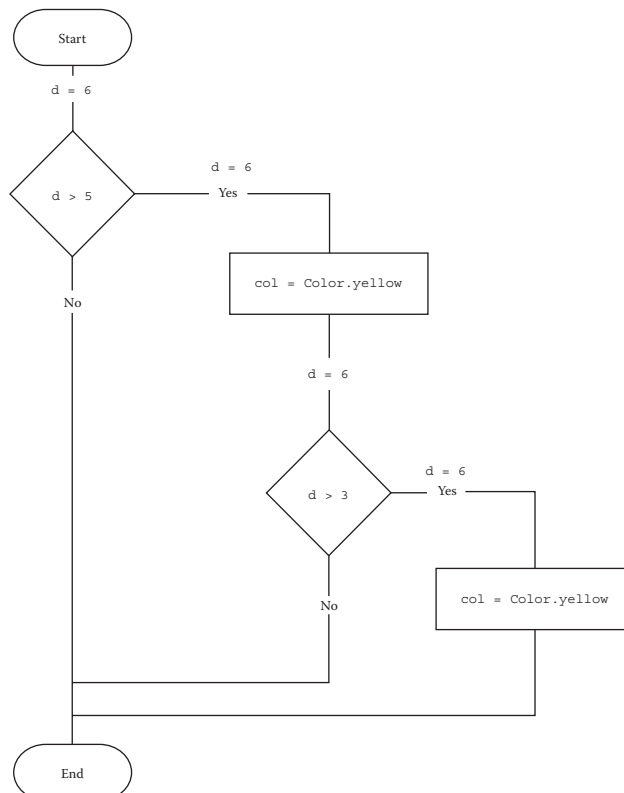
```
if (Ax > Bx) {  
    col = Color.black;  
    float d = Ax + Bx;  
    if (d > 5.0f) {  
        col = Color.yellow;  
    }  
} else if (Ax <= Bx) {  
    col = Color.blue;  
}
```

To obtain any effect, we need to change the placement of the code that sets the color. Placing the second `if` statement inside of the first `if` statement ensures that the color stays set at `yellow` before exiting the first `if` statement. When the second `if` statement is run, `col` is set to `Color.yellow`. Once the statement is finished, the statement exits up one level and then `col` is set to `Color.black` and the rest of the `if` statement is skipped, setting the color to `black` if the first `if` statement is true. At first, this might seem obvious to some, but it's important to see firsthand.

4.11.6.1 Flawed Logic

```
if (d > 5.0f) {  
    col = Color.yellow;  
    if (d > 3.0f) {  
        col = Color.cyan;  
    }  
}
```

To better understand this code, we might look at a flowchart:



If we look at the second `if` statement, we find that we can add in more logic. Adding in another `if` statement to change the color to cyan if `d` is greater than 3.0f will always prove true when found inside of the `d > 5.0f` `if` statement. This means that the cube will never turn yellow. To allow for both yellow and cyan, we'd need to think of a different way to pick when each color appears.

```
if(d > 5.0f) {
    col = Color.yellow;
    if (d > 10.0f) {
        col = Color.cyan;
    }
}
```

With the code changed such that one follows the other, we can set the color to cyan when `d` is greater than 10.0f. However, there's no reason for doing the code like this.

```
if(d > 5.0f) {
    col = Color.yellow;
}
if (d > 10.0f) {
    col = Color.cyan;
}
```

This code is another version of the same code with one `if` statement not being inside of the other. When using multiple `if` statements, the logic is easier to follow if the `if` statements don't live within one another. There are some additional concepts to avoid when figuring out how to add in another step in logic.

4.11.6.2 Unreachable Code

```
if(d > 5.0f) {
    col = Color.yellow;
    if (d < 3.0f) {
        col = Color.green;
    }
}
```

When we add in an `if` statement which checks distance `d` inside of another distance, check it's easy to create code which will never be reached. Checking if `d` is less than a value will never be seen if its value is never less than 3.0f when we're checking for the value being greater than 5.0f. Such statements are considered unreachable because they are something that the compiler will not be able to catch. It's up to your ability to think out the logic to see when code statements are unreachable.

```
if (Ax > Bx) {
    col = Color.black;
    float d = Ax + Bx;
    if (d < 3.0f) {
        col = Color.green;
    }
    if(d > 5.0f) {
        col = Color.yellow;
    }
    if (d > 10.0f) {
        col = Color.cyan;
    }
} else if (Ax <= Bx) {
    col = Color.blue;
}
```

Rearranging the code to be as flat as possible helps to limit errors like unreachable code. When `if` statements are added inside of another `if` statement we add additional layers of complexity. To reduce the complexity of the code separate the `if` statements to make them easier to read and interpret. Later on, we'll look at some different ways to make this appear a lot cleaner and easier to read.

4.11.7 Another Look at Scope

We still have a little bit to review in relation to scope. We've been using the `d` variable a few times in the first `if` statement. We added the `float d = Ax + Bx;` inside of the first `if` statement. Every frame this is updated to a new value that is then looked at for each `if` statement that follows. This raises the question how often can `d` be used?

```
if (Ax > Bx) {
    col = Color.black;
    float d = Ax + Bx;
    if (d < 3.0f) {
        col = Color.green;
    }
    if (d > 5.0f) {
        col = Color.yellow;
    }
    if (d > 10.0f) {
        col = Color.cyan;
    }
} else if (Ax <= Bx) {
    col = Color.blue;
    if (d < 3.0f) {
        col = Color.magenta;
    }
}
```

Adding in `d` within the `else if` statement produces an error. After `d` is declared, its scope is limited to the block of code it was created in. The new `else if` statement creates a new code block, which contains variables that are declared either in the function or within the statement.

We could change `d` from `d = Ax + Bx;` to `d = Ax - Bx.` The declaration is valid, but the use of `d` is different from before.

```
} else if (Ax <= Bx) {
    col = Color.blue;
    float d = Ay - By;
    if (d < 3.0f) {
        col = Color.magenta;
    }
}
```

If we declare a new `d` and set it to `Ay - By`, we don't get any conflicts, but it is confusing to see in your code. At first, `d` is `Ax + Bx`, and then it's `Ay - By`. When you start debugging code, this difference might be missed and can lead to long nights hunting for a single error. If you move the `float d` declaration outside of the first statement, it becomes visible to both the `if` and the `else if` statements.

Just as awkward, you can use `Ax + Bx` in both the `if` and the `else if` statements for `d`. This would be redundant and can also lead to problems if you wanted to change the value for `d`.

```
float d = Ax + Bx; //moved here for wider scope visibility
if (Ax > Bx) {
    col = Color.black;
    if (d < 3.0f) {
        col = Color.green;
    }
}
```

```

        if (d > 5.0f) {
            col = Color.yellow;
        }
        if (d > 10.0f) {
            col = Color.cyan;
        }
    } else if (Ax <= Bx) {
        col = Color.blue;
        if (d < 3.0f) {
            col = Color.magenta;
        }
    }
}

```

With `d = Ax + Bx;` moved just before the first `if` statement, it's no longer scoped to within its original code block. This can get deeper by moving `float d = Ax + Bx;` before both `if` and `else if` statements, the value of `d` can be seen by both logic statements.

```

if (Ax > Bx) {
    col = Color.black;
    float d = Ax + Bx;
    if (d > 10.0f) {
        col = Color.cyan;
        float e = Ay + By;
        if (e > 1.0f) {
            col = Color.green;
        }
    }
} else if (Ax <= Bx) {
    col = Color.blue;
}

```

We can add in another `float e`, within the `if (d > 10.0f)` statement.

```

if (Ax > Bx) {
    col = Color.black;
    float d = Ax + Bx;
    if (d > 10.0f) {
        col = Color.cyan;
        float e = Ay + By;
        if (e > 1.0f) {
            col = Color.green;
        }
    }
    if (e > 2.0f) {
        col = Color.gray;
    }
} else if (Ax <= Bx) {
    col = Color.blue;
}

```

Adding a second `if` statement outside of `if (d > 10.0f)`, which uses the `e` variable declared before, will produce an error. The variable `e` exists only within the block of code inside of the `if (d > 10.0f)`. The accessibility of variables follows the same rules as before. Both `e` and `d` are no longer visible in the second `else if` statement since the scope for those variables is contained within the blocks of code they were declared in.

The rules for variable scope are strict. Scope is managed thus to reduce the occurrence of duplication. If a variable wasn't managed like this, then you might end up stomping on a variable declared early on in your code with an unintended value.

4.11.8 What We've Learned

Thinking through logic is one of the most difficult parts of programming in general. For now, the logic has been pretty straightforward, but logic can get very tricky quickly. Once things get more difficult, it may be easier to set the booleans ahead of time, like in the first example below, where we use `bigChange` and use the value after it was set.

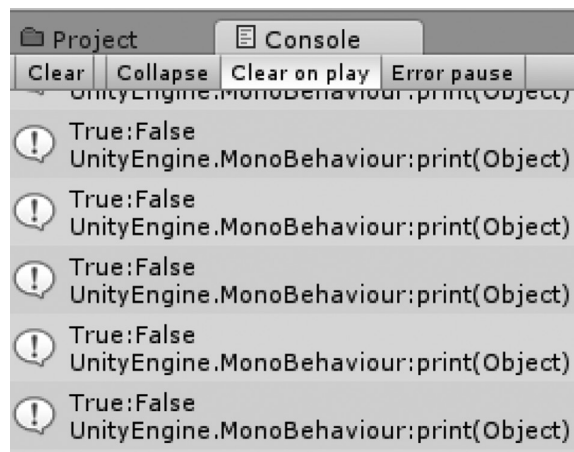
One technique to ensure you're forming your logic correctly would be to avoid combining everything into one large `if else if` statement. Start by making several smaller `if` statements and then combining them afterward.

```
void Update ()
{
    if(change >= 10)
    {
        transform.position = new Vector3(3,0,0);
    }
    if (change <= -10)
    {
        transform.position = new Vector3(-3,0,0);
    }
}
```

In this code block, we can know that each `if` statement is being evaluated properly. Finally, we can also set booleans ahead of time and watch how they are behaving using the `print` function.

```
void Update ()
{
    bool bigChange = change >= 10;
    bool lowChange = change <= -10;
    print(bigChange + ":" + lowChange);
}
```

With the addition of the `+"+"`, we can add in the two true and false values together on the same line. To join values together in the `print` function we use the `+` operator to merge values. `":"` turns the contained `:` into a string value. Therefore, here, we're merging three values together as strings, or as a programmer might say, "We're appending values" for the `print` function.



This approach becomes a valuable trick to help you work out your logic when you're writing your code. Try out a few different variations on the code yourself to make sure this all makes sense to you. We'll

revisit a few different ways to control logic as we move on to using additional logical operators within the `if` statement's parameters.

The ability for Unity 3D to allow you to interact with your code through public variables enables you to quickly and easily see how your code behaves. Without this ability, you're left to interacting with code through more code. This gets dull quickly. After a few more programming basics, we're going to dive into the more interesting parts of Unity 3D.

4.12 Loops

To follow along, we'll be using the Loops project and looking at the `Example.cs` file in the Assets directory. The `Example.cs` has been attached to the Main Camera so that the code will execute when the game is started.

This section is something you may need to return to several times for reference. Learning these different loops is all very different. Most of them accomplish the same thing, but with subtle differences. In general, the `for(;;)` loop is the easiest to use, and sometimes the fastest.

The `Update()` function is run once every frame. Within that function, our logic is evaluated, starting at the top moving toward the bottom. Once the function is complete, the next frame begins, and the `Update()` function is called again to restart the evaluation at the top. This process is then repeated for every frame while the game is running.

To further understand how this process affects our code, we should be able to count the number of frames and thus the number of times the `Update()` function has been called. To do this, we'll simply create a `public int` to count the number of times the `Update()` function has been called.

4.12.1 Unary Operators

The term *unary operator* means there's only one operation required. In C#, and many other programming languages, unary operators work only on integers. Integers are best at counting whole values. In this case, we're going to keep track of how many times the `Update()` function has been called.

Integers are whole numbers, like 1, 2, and 1234. Unlike how numbers are commonly used in written English, in code, no commas are used: "1,234," for example, should be written "1234." Commas are special characters in C#, so the computer will look at 1234 as a 1 followed by a different number 234. We'll go further into the differences between numbers, but we need to know a bit about integers before proceeding. In a new project in Unity 3D start, with the Loops project.

```
public int counter = 0;
void Update ()
{
    counter = counter + 1;
}
```

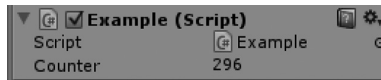
To start, we should look at how the process looks like in long hand. Adding `counter = counter + 1` means we're going to take the previous value of `counter` and add 1 to it. Some programming languages require this counting method. Lua, for example, needs to increment variables in this manner.

More complex languages like C# can shorten this to a single statement with the use of a unary operator.

```
public int counter = 0;
void Update ()
{
    counter++;
}
```

The unary operator `++` works the same as the long-hand version from the previous statement. When you run the game you can watch the counter value update in the Inspector panel in Unity 3D. After only a

few seconds, you can see the `counter` running up very quickly. Each time the `Update ()` function is called, the `counter` is incremented by 1.



Another unary operator is the `--`, which decrements a value rather than increments. Replace `++` with `--` and you can watch the `counter` running down into negative numbers. Each one of these has important uses.

4.12.2 While

If we want to run a specific statement more than once in a loop, we need another method. To do this, C# comes with another couple of keywords, `while` and `for`. The `while` statement is somewhat easier to use. It needs only one `bool` argument to determine if it should continue to execute.

This statement is somewhat like the `if` statement, only that it returns to the top of the `while` statement when it's done with its evaluations.

```
using UnityEngine;
using System.Collections;
public class Example : MonoBehaviour
{
    //Use this for initialization
    void Start ()
    {
    }
    //Update is called once per frame
    public int counter = 0;
    void Update ()
    {
        while(counter < 10)
        {
            counter++;
            print(counter);
        }
    }
}
```

We start again with `public int counter`. The `while()` loop checks if the `counter` is less than 10. While the `counter` is less than 10, then, we run the contents of the `while` statement that increments the `counter`. Once the `counter` is larger than 10, the `while` statement doesn't evaluate its contents. As a programmer might say, the `while` loop's code block will not be executed.

There is one danger using `while` statements. If the `while` statement remains true, it will not stop. In this case, it means that the `Update ()` function will not finish and the game will freeze on the frame. This also means that any input into Unity 3D will respond very slowly as it's still waiting for the `Update ()` function to end. We'll observe how runaway loops behave later on in this chapter.

4.12.3 For

To gain a bit more control over the execution of a loop, we have another option. The `for` loop requires three different statements to operate. The first statement is called an initialization, the second is a condition, and the third is an operation.

```
for (initialization ; condition ; operation)
{
    //code block
}
```

The first argument allows us to set up an integer. After that, we need to have some sort of boolean condition to inform the `for` loop when to stop. The last argument is something that operates on the initialized variable from the first statement in the `for` loop. The following code block is a typical example of the `for` loop statement.

```
void Update ()
{
    for (int i = 0; i < 10; i++)
    {
        print(i);
    }
}
```

Inside of the `Update ()` loop, we can add a `for` loop that looks like this. After the keyword `for`, we need to add in our three arguments, each one separated by a semicolon (;). The first part, `int i = 0`, is the initialization of an `int` for use within the `for` loop. The scope for `int i` exists only within the `for` loop.

```
void Update ()
{
    for (int i = 0; i < 10; i++)
    {
        print(i);
    }
    print(i); //error
}
```

The variable `i` we wrote into the `for` loop is gone once you leave the `for` loop. This point is related to scope. Scope is basically how a variable's existence is managed and when it's allowed to be used. This is just a simple example, and scope has a greater meaning than that shown in the example here.

If you try to use values in the loop outside of the encapsulating curly braces (`{}`), you'll get an error stating you can't use `i` before it's created. At the same time, you're not going to want to declare an `int i` before using one in the `for` loop.

```
void Update ()
{
    int i = 0; //i is going to be used in the for loop
    for (int i = 0; i < 10; i++)
    {
        print(i);
    }
}
```

The second part of the `for` loop is `i < 10`, which is a condition letting the `for` loop know when to stop running. This is similar to the `while` loop we just took a look at. The `for` loop offers alternative uses of its arguments, some uses may seem awkward. This is unorthodox, but you can leave out the initialization section if an `int` value has been declared before the loop.

```
void Update ()
{
    int i = 0; //i is going to be used in the for loop
    for (; i < 10; i++)
    {
        print(i);
    }
}
```

There's not much of a reason for this but it's not going to break anything. The last argument of the `for` loop is an operation which runs after the first two arguments are evaluated. If the second part of the `for` loop is true, then the contents of the curly braces are evaluated. The last part of the expression is `i++` that can also be moved around.

```
void Update ()
{
    int i = 0;
    for(; i < 10 ;)
    {
        print(i);
        i++;
    }
}
```

Unlike the `while` loop that relies on an external variable, the `for` loop contains everything it needs to operate, although, as we've just seen, we can still move the parts of the `for` loop around. If only the condition argument of the `for` loop is used then the `for` loop acts the same as a `while` loop. Although rearranging the parts of a `for` loop like this is not recommended, this does serve as a way to show how flexible C# can be.

Just so you know, a classically trained programmer might have a migraine after looking at this `for` statement, so give him or her a break and just do things normally.

The `while` loop runs only once after the `counter` reaches 10 after the first time it's evaluated. Each time the `Update ()` function is called, the `for` loop starts over again. We can use a slight variation of this using `float` values, though this isn't any different from using integers. Again, it's best to use integer values in a `for` loop, and this is only to show how C# can be changed around freely.

```
void Update ()
{
    for (int i = 0.0f ; i < 10.0f ; i = i + 1.0f)
    {
        print(i);
    }
}
```

4.12.4 Do-While

Do-while is another loop to add to our toolbox. We took a look at the `for` and the `while` looping statements in Section 4.12.3. The `for` and `while` loops are the most common and the most simple to use. There are, however, variations on the same theme that might come in handy.

The great majority of the tasks can be handled by the simple `for` loop. However, variations in looping statements are created either to support different programming styles or to conform to changing requirements in different programming styles or to allow for different conditions to terminate the loop.

The `while` and the `for` loops both check various conditions within their parameter list before executing the contents of the loop. If we need to do the check after the loop's execution, then we need to reverse how the loop operates.

```
int i = 0;
do{
    Debug.Log("do " + i.ToString());
    i++;
}while(i < 10);
```

The `do-while` loop has some subtle differences. For most programmers, the syntax change has little meaning and, for the most part, `do-while` can be rewritten as a simple `while` loop.

It's worth noting that most programming languages use the same keywords for the same sort of task. Lua, for instance, uses `for` in a similar fashion:

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

However, Lua likes to shorten the conditions of the `for` loop by assuming a few different things in the condition and the operation of `i`. Other languages like Java, or JavaScript, use `for` and `while` more like C#. Things get really strange only if you look at something like F#:

```
for I = 1 to 10 do
    printfn "%d" i
```

Syntax aside, once you know one version of the `for` loop, it's easier to figure out how another language does the same thing. The more basic concepts like booleans and operators also behave the same across languages. Therefore, you won't have to relearn how to use operators now that you know how they're used in C#.

4.12.5 Postfix and Prefix Notation

The placement of a unary operator has an effect on when the operation takes place. Placement, therefore, has an important effect on how the operator is best used. This is a difficult concept to explain without seeing in use, so we'll skip right to some sample code.

```
void Start ()
{
    int i = 0;
    Debug.Log(i); //before the while loop = 0
    while(i < 1)
    {
        Debug.Log(i++); //in the while loop = 0
        Debug.Log(i); //called again!= 1
    }
}
```

For various reasons, the `i++` is the more standard use of the unary operator. When this code is executed, the first `Debug.Log(i);` before the while loop, will produce a 0, just as when `i` was first initialized. The first time the `Debug.Log(i);` inside of the while loop is called, another 0 is produced in the Unity 3D Console panel. Where things become interesting is at the third `Debug.Log(i);` statement, where we have a different number.

The statement where `i++` appears will use the value which `i` held before it was incremented. After the statement where `i++` appears, the value of `i` will have been incremented. This is the difference between postfix and prefix, where `i++` is a postfix notation and `++i` is prefix notation. To illustrate this difference, observe the change when we use a prefix notation:

```
void Start ()
{
    int i = 0;
    Debug.Log(i); //before the while loop = 0
    while(i < 1)
    {
        Debug.Log(++i); //in the while loop = 1
        Debug.Log(i); //called again!= 1
    }
}
```

The first version, with postfix notation, produces two 0s and a single 1 before the `while` loop terminates. The second version produces one 0 and two 1s before the `while` loop terminates. The choice of which notation to use depends mostly on what value you need `i` to take when you use it. The syntax difference is subtle, but once it's understood, the effects can be controlled. To make the difference more drastic, we can move the placement of where the unary operation occurs.

```
void Start ()
{
    int i = 0;
    Debug.Log(i); //0
    while(++i < 1)
    {
        Debug.Log(i); //doesn't get reached
    }
}
```

The above statement will produce a single 0 in the Unity 3D's Console panel. The condition to execute the `while` loop begins as false, where the value for `i` is not less than 1, thus the `Debug.Log()` statement is never called. However, if we change things around, we'll get a 0 and a 1 printed to the Console panel.

```
void Start ()
{
    int i = 0;
    Debug.Log(i); //0
    while(i++ < 1)
    {
        Debug.Log(i); //1
    }
}
```

This code block means that the `while` loop started off true, since the value of `i` was 0 when evaluated, and only after the evaluation was complete was `i` incremented. Using the `while` loop in this way is unconventional, and it might be best avoided even though it's clever.

4.12.6 Using Loops

Making use of the loops is pretty simple. In the following use of the `while` loop, we're counting up a number of cubes. To test out our `while` loop, we'll create a new cube for each execution of the `while` loop. To do this, we'll create a new `GameObject` named `box`.

To initialize the `box`, we start with `GameObject.CreatePrimitive(PrimitiveType.Cube)`. This initialization has a couple of new concepts. The first is the `GameObject.CreatePrimitive()` function. `CreatePrimitive()` is a function found inside of the `GameObject` class.

To let `CreatePrimitive` know what to make, we're using the `PrimitiveType.Cube`, which is of `PrimitiveType`; technically, it's an enum, but more on that later. Then, as we've seen before in the `while` loop, we're incrementing the `numCubes` variable to make sure that the `while` loop doesn't run uncontrolled.

```
int numCubes = 0;
//Update is called once per frame
void Update ()
{
    while (numCubes < 10)
    {
        GameObject box = GameObject.CreatePrimitive (PrimitiveType.Cube);
        box.transform.position = new Vector3 (numCubes * 2.0f, 0f, 0f);
    }
}
```

```

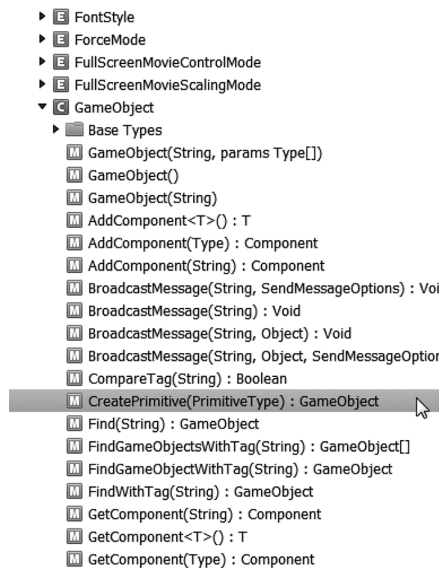
        numCubes++;
    }
}

```

To tell where each cube is created, we move each cube to a new position by setting the `box` to new `Vector3`. In `Vector3()`, we'll take the value of `numCubes` and multiply it by `2.0f`, making sure that each cube is moved to a different `x` location. Logically, each iteration of `numCubes` will increment `numCubes` by 1. Therefore, on the first execution, we'll have `0 * 2.0f` that ends with 0. Then, the second time through the `while` loop, the result will be `1 * 2.0f`, then `2 * 2.0f`, and so on, until `numCubes` is larger than 10.

Each time the `while` loop is evaluated, the `box` points to a new instance of a cube primitive. The previous iteration of the `while` loop forgets about the previous instance of the `box`, which instance `box` is referencing becomes very important as we don't necessarily want to deal with every instance of the `box` game object we're thinking about.

An inquisitive person should be asking, Where did `CreatePrimitive` come from? The answer to that is found inside of the `UnityEngine.dll`. `GameObject` is going to be the primary focus of creating any object in the game, as the name might suggest. Expanding the `GameObject` found inside of the Solution Explorer in MonoDevelop, we find a promising `CreatePrimitive` function. This is related to how we found `Vector3()` in an earlier section.



The `CreatePrimitive()` function expects a `PrimitiveType` as an argument. Clicking on the `PrimitiveType` found in the Assembly Browser window, you are presented with the following set of code.

```

using UnityEngine;

namespace UnityEngine
{
    public enum PrimitiveType
    {
        Sphere,
        Capsule,
        Cylinder,
        Cube,
        Plane,
        Quad
    }
}

```

Here is an enum called `PrimitiveType`. Inside this is `Sphere`, `Capsule`, `Cylinder`, `Cube`, and `Plane`. Each one is a value inside of the `PrimitiveType`. Enums are another generic type that C# makes extensive use of, which we'll introduce in Section 6.6.1. The `CreatePrimitive` function inside of the `GameObject` class is followed by: `GameObject`. This means that it's creating a `GameObject` for you to access. We'll learn about this once we start writing our own classes and functions with return types.

NOTE: Looking things up on the Internet is also a huge help. There are many people asking the same questions as you are. If you don't find what you're looking for, then you might just be asking the wrong question, or phrasing it in an obscure way. To find information on `CreatePrimitive`, search Google using "CreatePrimitive Unity" to get some quick possible answers.

In most cases, the answers are going to be brief and obscure, but they at least serve as an example for what you might need to look for in your own code to complete the task you're trying to complete.

This discussion might be a bit much to swallow right now, so we'll stick with the small bite we started with and get back to the deeper picture later on. For now, let's get back to figuring out more on loops.

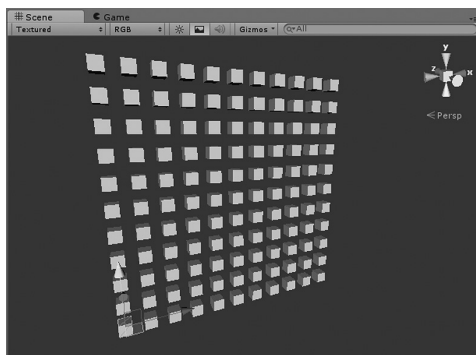
4.12.7 Loops within Loops

We've created a line of cubes, but how would we go about making a grid of cubes?

Let's move our focus back to the `Start ()` function, so our code is carried out only once. Then we'll write two `for` loops, one within the other. The first `for` loop will use `int i = 0` and the second will use `int j = 0` as their initialization arguments. Then we'll leave the `numCubes` as a public `int` we can change in the editor. If we set the `numCubes` in the inspector to 10, then each `for` loop will iterate 10 times.

```
public int numCubes = 10;
//Use this for initialization
void Start ()
{
    for (int i = 0; i < numCubes; i++)
    {
        for (int j = 0; j < numCubes; j++)
        {
            GameObject box =
            GameObject.CreatePrimitive (PrimitiveType.Cube);
            box.transform.position =
            new Vector3 (i * 2.0f, j * 2.0f, 0f);
        }
    }
}
```

Only in the second `for` loop do we need to create a box; then, in the `x` and `y` values of the `Vector3`, we will do some of the same math we did before. This time we'll use the `int i` variable for `x` and `j` for `y` to create a grid. This declaration creates a row and column relation between the two `for` loops. As one iterates through the row, the other inside of the row fills in the column with cubes.



Hop to the Scene panel in the Unity 3D editor to take a look at the grid of cubes that was produced when you pressed the Play button. When the first `for` loop begins, it gets to the second `for` loop inside of it. Only after the second `for` loop is done with its statements of creating 10 boxes will it go back to the first `for` loop that moves on to the next iteration.

4.12.8 Runaway Loops

Loops can often lead to runaway functions. A `while` loop without any condition to stop it will make Unity 3D freeze. Here are some conditions that will immediately make Unity 3D stop on the line and never finish the frame. Adding any of these will halt Unity 3D, and you'll have to kill the Unity 3D app though the Task Manager.

```
while(true)
{
}
for(;;)
{
}
```

In some cases, these conditions are used by some programmers for specific reasons, but they always have some way to break out of the function. Using either `return` or `break` can stop the loop from running on forever. It's worth mentioning that the keyword `return` is usually used to send data to another function. If there isn't a function waiting for data, it's better to use `break`.

```
for(;;)
{
    return;
}
while(true)
{
    break;
}
```

There is a specific behavioral difference between `return` and `break`. With `return` you're stopping the entire function and restarting at the top of the function. With `break` you're stopping only the block of code the `break` is in.

```
void Start ()
{
    Debug.Log("at the start");
    for(;;)
    {
        Debug.Log("before the return");
        return;
        Debug.Log("after the return");
    }
    Debug.Log("at the bottom");
}
```

Use the above code in the `Start ()` function of a new script in use in a Unity 3D Scene. First of all, we'd get a couple of warnings informing us of some unreachable code. However, that doesn't keep us from testing out the results printed on the Console panel.

```
at the start
UnityEngine.Debug:Log(Object)
Loops:Start () (at Assets/Example.cs:8)
before the return
UnityEngine.Debug:Log(Object)
Loops:Start () (at Assets/Example.cs:10)
```


Therefore, we do get the first line of the `Start ()` function as well as the line before the `return` executed. However, the two lines following `return` are not reached. This behavior changes if we replace `return` with `break`.

```
void Start ()
{
    Debug.Log("at the start");
    for(;;)
    {
        Debug.Log("before the return");
        break;//not return!
        Debug.Log("after the return");
    }
    Debug.Log("at the bottom");
}
```

We get only one warning of unreachable code, but we do get a third line printed to the Console window.

```
at the start
UnityEngine.Debug:Log(Object)
Loops:Start () (at Assets/Loops.cs:8)
before the return
UnityEngine.Debug:Log(Object)
Loops:Start () (at Assets/Loops.cs:10)
at the bottom
UnityEngine.Debug:Log(Object)
Loops:Start () (at Assets/Loops.cs:14)
```

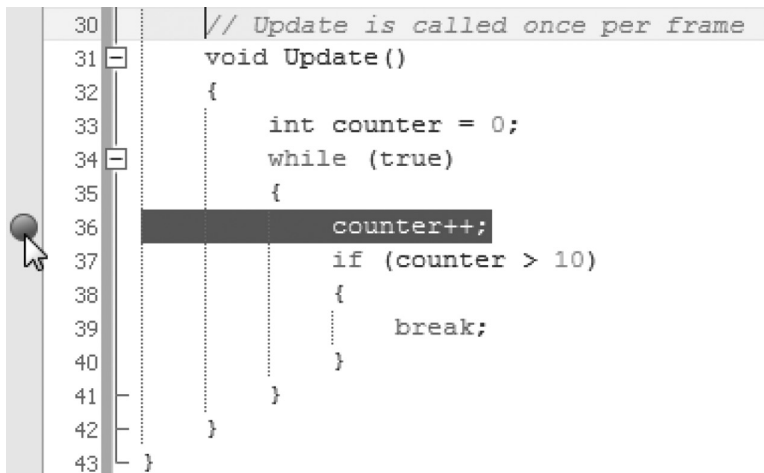
Therefore, we do get the line after the `for` loop, but we don't get the rest of the contents from the `for` loop after the `break` statement. We'll go into more detail on the specifics of how these functions are used later on, which provides a very short example for `break`; For instance, we can use the following code to see how a `while(true)` can be stopped with a specific instruction.

```
void Update ()
{
    int counter = 0;
    while (true)
    {
        counter++;
        if (counter > 10)
        {
            break;
        }
    }
}
```

The counter builds up while we're trapped in the `while` loop. Once the counter is above 10, we break out of the `while` loop with the `break` keyword.

4.12.9 Breakpoints: A First Look

To have a better understanding on what we're seeing here, we're going to use a very important feature of MonoDevelop. With the code we just entered in the previous `Update ()` function, we have a `while(true)` and a `counter++` incrementing away inside of it.

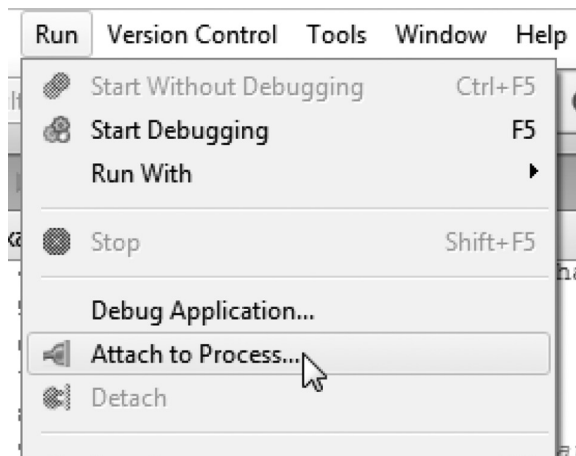


```
30 // Update is called once per frame
31 void Update()
32 {
33     int counter = 0;
34     while (true)
35     {
36         counter++;
37         if (counter > 10)
38         {
39             break;
40         }
41     }
42 }
43 }
```

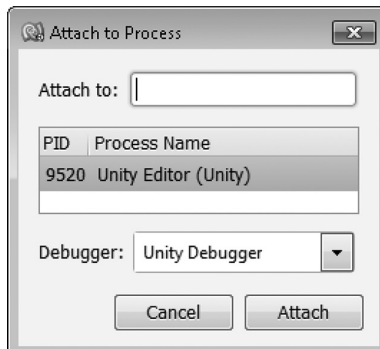
A screenshot of a code editor window showing a C# script. The code is for an `Update()` method. A breakpoint, represented by a small square icon, is set on line 36, which contains the statement `counter++;`. The line is highlighted in black. The left margin of the editor is visible, showing the line numbers from 30 to 43.

There's a narrow margin to the left of the numbers. The margin is for adding breakpoints. Breakpoints are small bookmarks that help you see exactly what's going on inside of your loop. Click on the line with the `counter++;` and you'll see the line highlight.

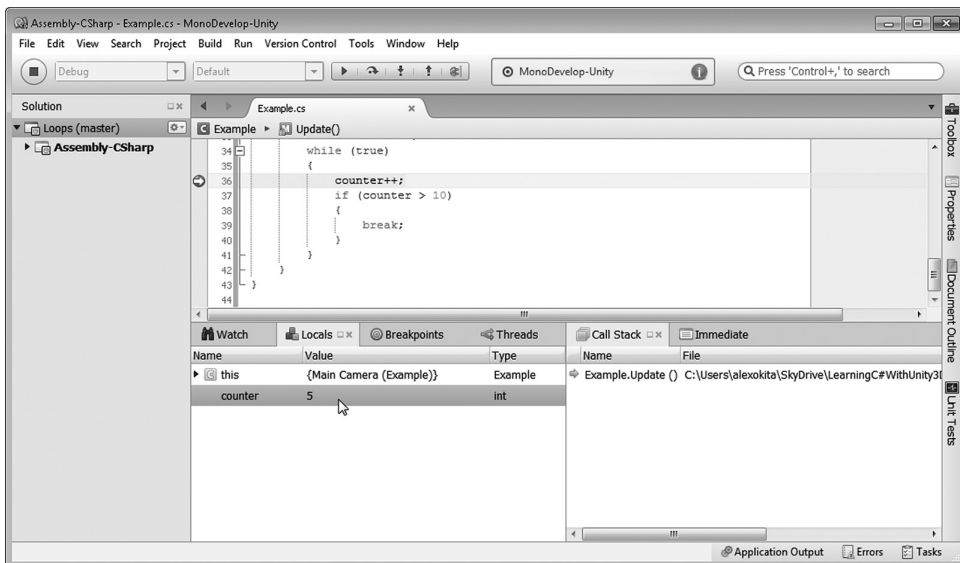
Run the code in the scene with the `Example.cs` attached to the Main Camera. While the game is running, select **Run** → **Attach to Process** in MonoDevelop. This action will open a dialog box with a list of Unity 3D game engines you might have open. In most cases, you would have only one.



The button is pretty small, but it's there. You can also use the menu (**Run** → **Attach to Process**) to open the following dialog.



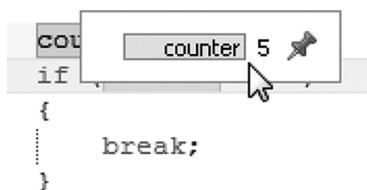
Press Attach with Unity 3D Editor selected. You may have to bring Unity 3D to the foreground by clicking on it so it updates at least once, but then it will freeze. To speak in programmer jargon, this means “you’ve hit a breakpoint.”



You’ll see that the line in MonoDevelop is now highlighted, and a new set of windows has appeared at the bottom of the panel. Select the Locals panel and take a look at the counter. There are three columns: Name, Value, and Type, which tell you the counter is at 0, which will tell you the value for counter and that it’s an int.

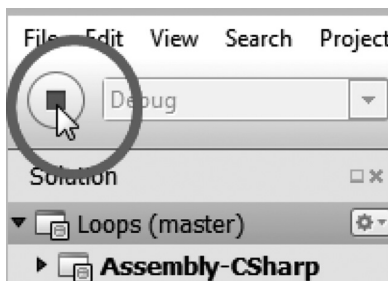
Press F5 to release the break and you’ll see the count increment by 1. As soon as the code gets back to the breakpoint, it will stop again. What you’re looking at is your code being updated only when you press the F5 key. This is called *stepping through code*. And it’s a very important reason why MonoDevelop is to be used with Unity 3D, and not something like Notepad.

You can also hover the cursor over the value you’re interested in observing.



Pressing F5 will show you this value updating with each step. *Debugging* code is a major part of observing how your code is actually working. Breakpoints, and debugging tools like this and debugging tools like the Locals tab are indispensable when it comes to writing complex code.

To stop debugging the code, and let Unity 3D go about its normal work, press the Stop button.



Pressing the little Stop icon will unlink MonoDevelop from Unity 3D and the information that we were watching will stop updating. Unity 3D will also return to regular working order.

4.12.10 What We've Learned

The `for` loop is a tight self-contained method to iterate using a number. The `while` loop may be shorter, and use fewer parameters, but its simplicity requires more external variables to control. When faced with different programming tasks, it can be difficult to choose between the two loops.

In the end, each loop can be used to do the same thing using various techniques we have yet to be introduced to. Don't worry; we'll get to them soon enough. At this point, though, we have learned enough to start thinking about what we've learned in a different way.

The last major concept we learned was the breakpoint, used to inspect what's going on inside of a loop. Breakpoints can be set anywhere and bits of information can be picked out and inspected while your code is running.

Debugging code is a part of the day-to-day routine of the programmer. Using a debugger that allows you to step through your code's execution allows for a greater understanding of what your code's activity involves. The debugger shows you what your code is actually doing.

4.13 Scope, Again

Scope was explored for a bit in Section 4.8, but there is some explaining left to do. We know that some variables exist only for the scope they were declared in. Other variables end up being shown in the Unity 3D Properties editor. This point doesn't explain how one class can talk to another class through a variable, or even whether this was even possible.

4.13.1 Visibility or Accessibility

Normally, variables and class definitions are considered to be *private*. This is usually implied and if you don't tell C# otherwise, it will assume you mean every variable to be private information.

```
private
public
protected
internal
```

There are only four keywords used to change the accessibility of a variable. Each one has a specific use and reason for being. For now, we'll examine the first two keywords in more depth. The keywords `private` and `public` are the two most-simple-to-understand ways to change a variable's accessibility.

We've used `public` before when we wanted a `bool` to show up in the Inspector panel in Unity 3D.

```
using UnityEngine;
using System.Collections;
public class Example : MonoBehaviour
{
    public bool SomeBool;
    void Start ()
    {
        SomeBool = true;
    }
}
```

We have a `public bool SomeBool;` declared at the beginning of this `Example.cs` code. Because `private` is assumed, we can hide this by simply removing the `public` keyword. At the same time, we can be very explicit about hiding this `bool` by adding the `private` keyword.

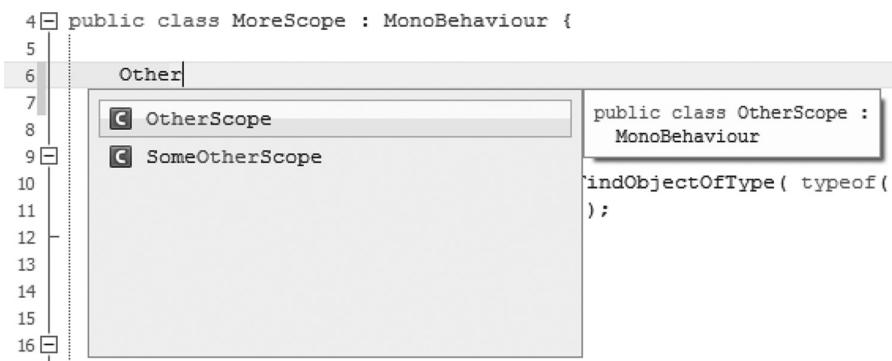
```
using UnityEngine;
using System.Collections;
public class Example : MonoBehaviour
{
    private bool SomeBool; //hidden
    bool AnotherBool;      //also hidden
    void Start ()
    {
        SomeBool = true;
    }
}
```

To the editor, there's no real difference between `SomeBool` and `AnotherBool`. The differences will come in later when we look at inheritance. It's important to know about the other keywords that can precede both class and variable statements.

4.13.1.1 A Basic Example

For a better observation of a variable's scope, we're going to want to start with a scope project different from before. This time, we'll start with the `MoreScope` project. In this project, we have a scene file where our Main Camera has the `MoreScope` component added. On a cube in the same scene, we have another script component added called `OtherScope`.

Here's where things get interesting. From the camera, we're able to see the `OtherScope` class. This is indicated by MonoDevelop allowing us to access the `OtherScope` class as we type.



As we begin to type `Other` the class `OtherScope` appears in the Autocomplete pop up. This tells us that there are accessible objects, which we can access from class to class. This is referred to as the global scope.

4.13.2 Global Scope

When each class is created in Unity 3D, C# is used to make additions to the global scope of the project. This means that throughout the project, each class will be able to see every other class. This also means we can't have two class files with the same name. If we manage to create two scripts with the same name, we'll get the following error.

```
Assets/OtherScripts/OtherScope.cs(4,14): error CS0101: The namespace
'global::' already contains a definition for 'OtherScope'
```

Here, I've created another directory in Assets called `OtherScripts`. Inside of this directory, I've added another `OtherScope.cs` file, which means that in the project as a whole, there are two `OtherScope` classes. As each new file is created, they're added to the global namespace. This is similar to what would happen should we define two variables with the same name in a class. Changing the name of one of the classes resolves the problem.

Going back to the `MoreScope` class from the beginning of the tutorial we added an `OtherScope` other; variable. Next we want to assign a variable within other a value. In the game scene assign the `OtherScope.cs` class to a new `Cube` `GameObject` before running the game. In the `MoreScope`'s `Start ()` function, use the following statement.

```
OtherScope other;
void Start ()
{
    other = (OtherScope)GameObject.FindObjectOfType(typeof(OtherScope));
    Debug.Log(other.gameObject.name);
}
```

Unity's `GameObject` class has a function called `FindObjectOfType()` that will search the scene for any object of the type we're requesting. The statement following the assignment to `other` is `Debug.Log(other.gameObject.name);`, which prints the name of the `gameObject` that the component is attached to in the Console panel. When the game is run, we get the following output.

```
Cube
UnityEngine.Debug:Log(Object)
MoreScope:Start () (at Assets/MoreScope.cs:11)
```

This output indicates that we've correctly found and assigned the variable `other` to the component attached to the cube object in the scene. Now let's add some meaningful variables.

```
using UnityEngine;
using System.Collections;
public class OtherScope : MonoBehaviour
{
    public float Size;
    Vector3 mScale;
    //Use this for initialization
    void Start ()
    {
    }
    //Update is called once per frame
    void Update ()
    {
    }
}
```

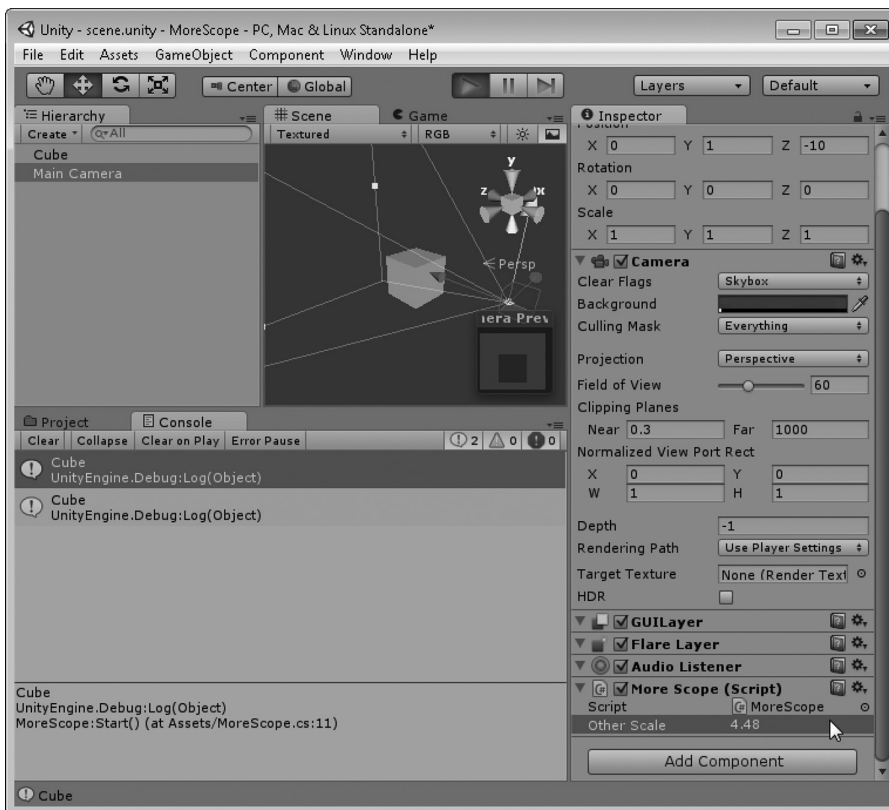
In the `OtherScope` class, we'll create two variables, one `public float` named `Size` and a `Vector3` called `mScale`. The "m" notation is commonly used by programmers as a short version of "my." Thus "mScale" refers to "MyScale." In the `Update ()` function of the `OtherScope` class, we'll add the following statements:

```
//Update is called once per frame
void Update ()
{
    mScale = new Vector3(Size, Size, Size);
    gameObject.transform.localScale = mScale;
}
```

This will set the size of the cube object's `transform.localScale` to the public `Size` variable on the `x`-, `y`-, and `z`-axes. In the `MoreScope` class attached to the `Camera`, we'll add a new public variable called `otherScale`.

```
//Update is called once per frame
public float otherScale;
void Update ()
{
    other.Size = otherScale;
}
```

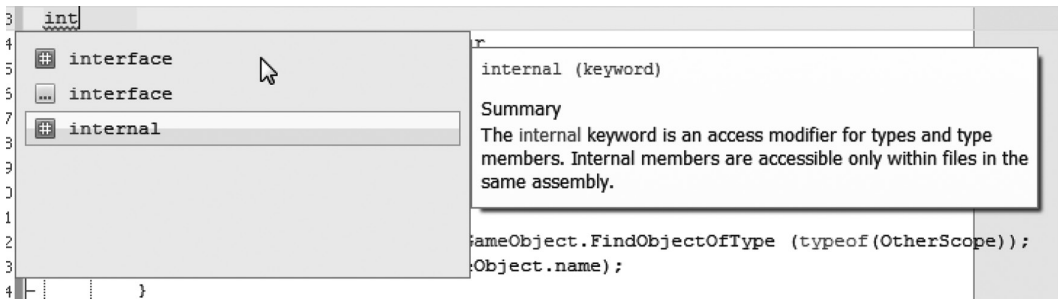
The `Update ()` function in the `MoreScope` class attached to the `Main Camera` should look like the above code. Because the `Size` variable in the `OtherScope` class was made public, it's now accessible through the dot operator. When the scene is run, the size of the cube is controlled by the `otherScale` variable assigned to the `Main Camera`.



You can click-drag on the `otherScale` variable that has been exposed to the Inspector panel with the `Main Camera` selected. This action directly affects the size of the cube in the scene. Even though there's no visible connection between the two objects, their variables have been connected to one another through a global connection.

Accessing a variable is not the only way a class can manipulate another class. Functions can also be accessed by another class, if they've been made public.

The importance here is that both `MoreScope` and `OtherScope` could see one another. Once they can see each other, the variables inside one another are accessible if they're made public. Could it be possible to declare variables that are accessible globally?



We could try to put a variable outside of a class. This might make it visible to any class that might want to access it. This, unfortunately, won't work. However, this doesn't mean that there are no systems that can make a variable globally accessible. However we'll have to leave off here for now. The systems that allow for globally accessible variables require some steps along the way to fully understand, so we'll get to that stuff soon enough.

4.13.3 What We've Learned

Defining public variables allows for interobject communication, which is one of the principles behind object oriented programming. This enables objects to communicate to one another in many unique ways.

Scope and object oriented programming are related. Variable scope is related to object oriented programming. By defining and maintaining scope, you're better able to control how other objects are allowed to communicate.

When working with a team of programmers, it's important to inform all on how your code behaves. You should communicate which variables in your class are allowed to be modified by how they are defined. Making variables public informs everyone that these are the variables allowed to be altered.

By protecting them, you're able to tell others which variables they are not allowed to touch. Much of the time, it's simple to leave things public while testing and making decisions on what needs to be made public. Once you've isolated what can and cannot be altered, it's best to hide variables that shouldn't be changed by outside objects.

4.14 Warnings versus Errors

There are two basic events that we will encounter early on in C#. The first are errors, and then we will get to the many warnings. Compile-time errors will stop all your classes from compiling. Because one class can look at and interact with another class, we need to make sure that every class is free from errors before our code can be compiled.

Most of these compile-time errors occurs from typos or trying to do something with the code that can't be interpreted by the lexer. Most of these errors deal with syntax or type-casting errors, which will prevent the rest of the code from working, so it's best to fix these errors before moving on.

At the same time, we'll get many warnings. Some of these warnings can be considered somewhat harmless. If you declare a variable and don't do anything to either assign something to it or use it in any way, then the lexer will decide that you're wasting resources on the variable. This produces a simple unused variable warning.

In some cases, code can still be compiled, but there will be some minor warnings. Warnings usually inform us that we've either written unused code, or perhaps we've done something unexpected. Code with a few warnings will not stop the compilation process, but it will let us know we've done something that might not necessarily work with Unity 3D.

4.14.1 Warnings

The most common warning is an alert bringing to your attention a variable declaration without any use. The MonoDevelop IDE is quite good at reading your code and interpreting how it's used. A function like the following will produce a warning:

```
void MyFunction()
{
    int j = 0;
}
```

This simply states that `j` is unused, which is true. Nothing is doing anything with it, so is it really necessary? Often, in an effort to write code, you'll be gathering and preparing data for use later on. When we do this, we might forget that we created some data that might have been useful but never did anything with it.

In this case, we might do something with it later on, so it's sometimes useful to comment it out.

```
void MyFunction()
{
    //int j = 0;
}
```

This leaves the variable around in case we need to use it, but it also clears out the warning. Most warnings don't lead to any serious problems.

4.14.2 Errors

One most common error occurs when we are dealing with missing variables or incorrect types. These errors show up fairly often when we first start off writing our own functions and aren't used to working with different data types or know how they interact.

```
float f = 1.0;
```

This is a commonly written statement that produces the following error:

```
Assets/Errors.cs(32,23): error CS0664: Literal of type double cannot be
implicitly converted to type 'float'. Add suffix 'f' to create a literal of
this type
```

The fix for the error is included in the error message. The message includes "Add suffix 'f' to create a literal of this type," which means we need to do the following.

```
float f = 1.0f;
```

This fixes the error. Many of the error messages give a clue as to what's going on and how to fix the error. Many of these errors show up before the game is run. These are called compile-time errors. Errors that show up while the game is running are called run-time errors.

Run-time errors are harder to track down. Often, this requires a debugging session to figure out where and why an error is happening.

4.14.3 Understanding the Debugger

Writing code that works is hard enough. To make sure that the code works, it's often easier to make it readable. To understand how the code works, we often need to add in additional statements to make it easier to debug as well.

To start, we'll want to begin with the `DebuggingCode` project. We haven't covered declaration of multiple variables in one line; this is because it's bad form, but it's not illegal.

```
//Use this for initialization
void Start () {
    int a, b = 0;
}
```

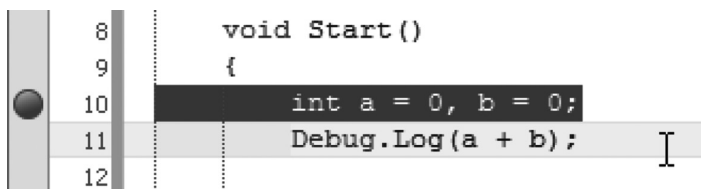
We can declare both a and b in one statement. In the above, a remains unassigned, and b is now storing 0.

```
//Use this for initialization
void Start () {
    int a, b = 0;
    Debug.Log(a);
}
```

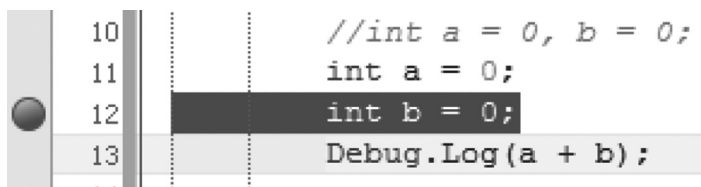
This now produces an error, where a is used before it's assigned; however, the error might not be clear because in the line there certainly is a declaration and an assignment. The assignment is harder to read since it's all happening on one line.

```
//Use this for initialization
void Start () {
    int a = 0, b = 0;
    Debug.Log(a + b);
}
```

This is also valid, but we've done nothing for readability or debugging.



Setting a breakpoint on this line highlights both variables being declared and assigned. This creates a problem if we're trying to find a problem with just one of the variables. This also highlights why we need to change our writing style to help debug our code.



Breaking up the declaration across multiple lines allows for setting a breakpoint on a specific statement rather than a multiple declaration statement. Likewise, debugging a statement like the following is no better.

```
int c = a; int d = b;
```

This line has two statements, but does not have a line break between them. This creates the problem of both readability and debuggability. When you're trying to isolate a problem, it's important to keep each statement on its own line so a breakpoint can be applied to it.

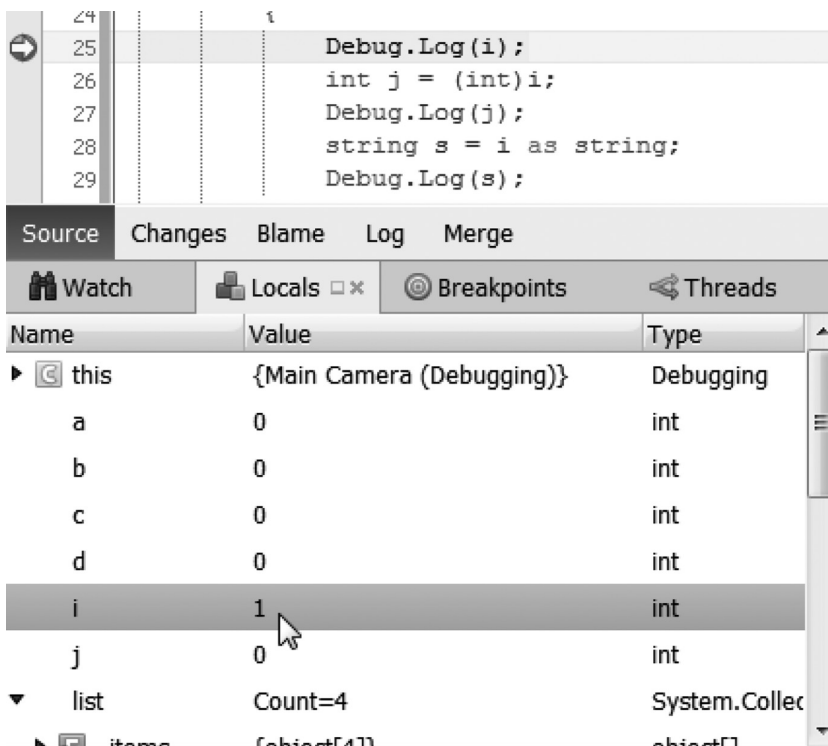
If we continue this exercise, we can take a look at an `ArrayList` that contains different types of data.

```

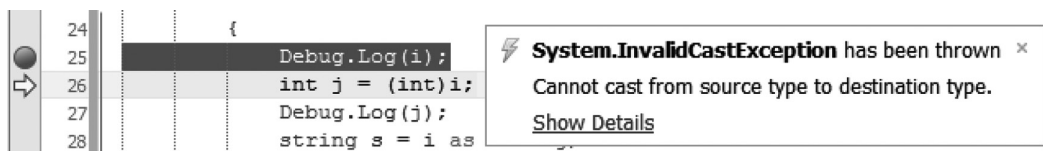
void Start () {
    ArrayList list = new ArrayList();
    list.Add(1);
    list.Add("this");
    list.Add(1.0);
    list.Add(1.0f);
    //lists can have anything in them!
    foreach(int i in list)
    {
        Debug.Log(i);
    }
}

```

In the list, we have an int of 1, a string this, a double 1.0, and a float 1.0f. The foreach loop requires ints and this works fine with the first iteration.



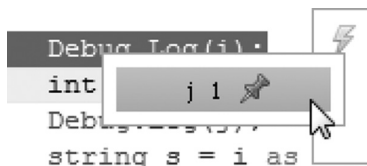
Setting a breakpoint here shows in the Locals panel an int `i` with a value of 1. So far so good. When we step through the code to get to the next item in the list, we get an error.



`System.InvalidCastException` shows up in the Locals panel. Since string `this` isn't an int, we get an error telling us what error was raised and in what line the error occurred on. This sort of debugging is only a contrived example setup to fail in a known location.

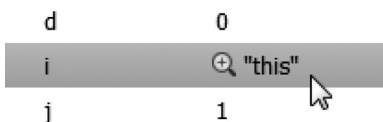
In most instances, problems like this aren't so easy to spot. The warnings and errors are often described with a line number, but it's not always easy to know why the exception was raised. Stepping into the code and checking out values is the only tool we really have to search for problems.

To make things easier on yourself, it's sometimes better to break things out even more.



The object type allows us to cast from the `ArrayList` into different types. By stepping past the assignment of `(int)i` to `j`, we can see what value has been assigned, by hovering over the variable. After this line, string `s` is null.

On the next loop through the `foreach`, we get the expected exception raised, but we have more information.



We can see that `i` is storing the value `this` that cannot be cast to an `int`. By making a separation of the value from where it's used, we can get a better idea of what type of data we're looking at. Adding additional lines of code just for the purpose of debugging allows for easier debugging.

Of course, this isn't as efficient as before. We're creating many small variables to be created and stored. This isn't the most optimized way to keep your code, so it's important to learn how to clean your code up to use less memory, but that's for Sections 6.3.2 and 6.15.6.

4.14.4 What We've Learned

Working through problems takes a step-by-step approach. Setting your code up for debugging requires some formatting changes to your code to help the debugger do its work. Once the code has been laid out for debugging, it's sometimes easier to read but our code can lose some efficiency.

It's always a good idea to keep your code clean and readable, but it's just as important to help your debugger do its work as well. Seeing the debugger step through your code also helps you understand how your own functions do their work.

When you have a chance to observe your code in operation, it's sometimes easier to see where code can be optimized—for example, when you see the same data being used over and over again. Passing values between variables in a visual sense can help reveal patterns. Once the patterns are seen, it can be easier to produce code that creates and reuses the same value less often.

4.15 Leveling Up: Fundamentals

At this point, I can say that you've gained enough of the basic skills to feel confident in your ability to read code and begin to ask the right questions. Just as important, the basics we've covered are shared among most programming languages.

Coming up with clever systems to control data is only a part of the task of programming. A great deal of work is coming up. It's also important to think of clear and concise names for variables. One thing is for sure: There have been plenty of pages written by people trying to wrangle in rampant names for identifiers.

At worst, you might have a situation where humans use health points, zombies hit points, vampires blood units, werewolves silver shots, and each one is simply a record of how many times they take damage before going down. It would be unfortunate for you to have to write functions for each human to shoot at every other type of monster and vice versa.

If someone offers help with your game, looks at your code, and you never hear back from that person again, it's surely due to the confused mess of code he or she got to look at. Some simple guidelines to help with naming have been outlined a number of times by various groups.

4.15.1 Style Guides

When faced with the fact that your code will have to eventually be seen by other programmers, it's best to try to avoid some of the embarrassment of explaining every identifier you've written. Furthermore, as a beginner, it's important that you gain some insight into the common practices that classically trained programmers have used throughout their careers. With any luck, books will help you keep up.

A style guide is a reference that often involves preferences involving everything from the placement of white space to naming conventions of identifiers. Official Microsoft style guides for C# provide details on how to name every different type of identifier. This enables us to tell identifiers apart from one another based on their use for variables, functions, or any other construct that C# uses.

As we learn about the different things that identifiers are used for, we'll also want to know how to best name the construct we're creating. A simple Internet search for "C# style guide" will bring up various links, for example, coding standards, guidelines, and coding guidelines.

Once you start working with any number of programmers, it's best to come to some agreement on the practices you'll choose to follow. Everyone learned slightly different methods and ideologies when he or she was introduced to a programming language, even if C# wasn't the first programming language that person learned. This leads to many different styles and conventions.

