Lesson Three

I Ain't Afraid of No Ghosts

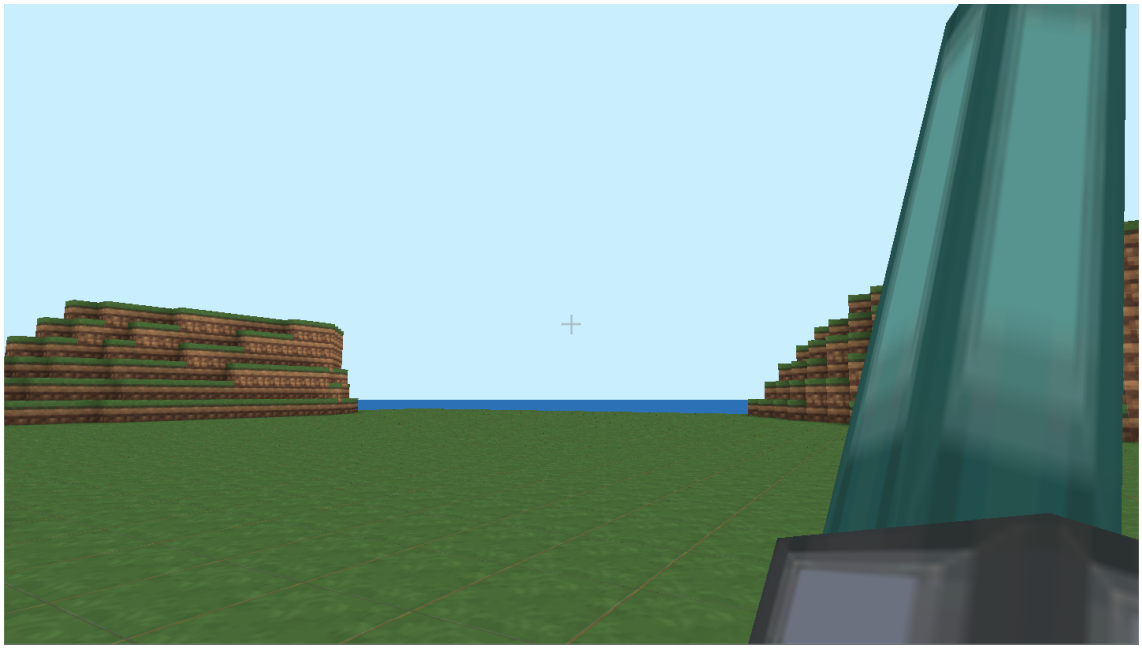

Figure 1: A section of the island that comprises Lesson 3.

1 Introduction to Lesson Three

In the past two lessons, you've learned about the building blocks of code (variables) and how to have your code make different decisions based on the situation (conditional statements). In this lesson we'll bring both of these pieces together and expand upon them in order to fix a *very* broken "beat 'em up" game. This lesson builds on lessons one and two, so we don't recommend attempting it until you've finished both of those.

2 Lesson Three: I Ain't Afraid of No Ghosts

If you closed Unity after Lesson Two, open it back up. Once Unity is opened, select the project from the drop down menu that pops up and wait for it to fully load. From there, go to the "Project" view, open the Lessons folder, and double click the "LessonThree" scene file.

Click the "play" button at the top of Unity to check out the currently broken "beat 'em up" game. Remember you can hit the Escape or Backspace key to get your mouse back and stop the game at any point (this will be necessary to do as the game doesn't end). Note: Windows users may not have the mouse disappear upon play without bringing up the pause menu with Escape then clicking "Resume".

When you boot up the game you should find yourself on a small island. Around the island you will find a number of randomly scattered green ghosts. Sadly, these ghosts don't seem to want to do anything.

However, the first thing you probably noticed was that the player moves much slower than in

```
1 using UnityEngine;
2
3 public class PlayerInfo : MonoBehaviour {
4     //This variable controls the player's speed
5     public float speed = 2;
6     //This variable controls the player's jump height
7     public float jumpValue = 1;
8     //This variable controls the impact of gravity on the player
9     public float gravityValue = 2;
10 }
```

Figure 2: PlayerInfo.cs before any changes have been made.

the last lesson. Additionally, falling seems to go much more slowly. Before we can fully explore the island, we need to deal with these issues. Let's open up the first script (of many) we'll be working with in this lesson. It's name is "PlayerInfo.cs". You'll find it in the Codebase folder, and then within the "PlayerScripts" folder. Double click on the "PlayerInfo.cs" file to open it up in MonoDevelop. As a reminder, MonoDevelop may take some time to open if it was closed after LessonTwo.

2.1 Script Usage and PlayerInfo

In both lesson one and two we only edited a single script. However, even the simple games of lessons one and two relied on a large number of scripts. Typically, each script contains a single class. Classes can, in turn, serve a variety of purposes in a game. For example, a class may be used to store information (like in lesson one where LessonOneGame held values for player speed) or a class may be used to store behavior information (like in lesson two where we added the behavior for the locked door to LessonTwoGame). PlayerInfo falls into the first of these two categories, storing variables that relate to the player.

Definition!

ACCESS MODIFIER: A keyword that limits what other code can "see" a variable, function, or class.

Figure 2 displays how PlayerInfo appears to begin with. By now you should be pretty familiar with the structure of classes. There's still the "using UnityEngine" line at the top of the script, followed by the class information "public class PlayerInfo : MonoBehavior". From there we have three variable definitions. However, these are a bit different than what we've seen before as all three have what's referred to as **access modifiers**. An access modifier is a keyword that determines whether some variable, function, or class can be seen by *other* scripts. The two most common access modifiers are "public" and "private". The access modifier public indicates that a variable, function, or class can be accessed by all other scripts in a project. We've seen it used in all classes thus far (including PlayerInfo) so that they can be used by other classes in the project. While we didn't see the word, it was also used in every single function and variable from other classes we've used so far (so like LockedDoor.OpenDoor() from LessonTwo and Time.deltaTime from LessonOne.

This might seem a bit complicated, so let's show an example. Go ahead change the values in the speed and gravity variables so you can move around a bit easier. We recommend "5" for speed and "20" for gravityValue, but feel free to play with other values for these variables and jumpValue.

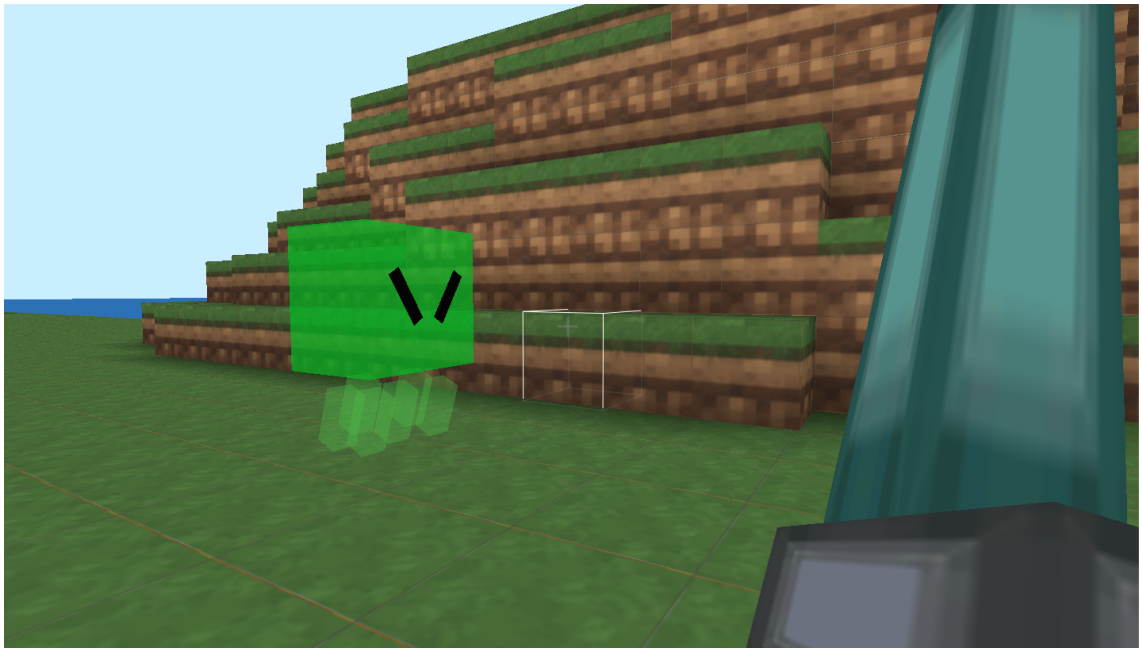

Figure 3: An example of an unmoving ghost.

Make your changes and play the game again, note the improved speed and falling! Feel free to continue to make changes to all three values, testing out their effects on the game world and player motion as you do.

But how did the variables in PlayerInfo.cs actually have this effect? The variables just exist in PlayerInfo.cs. Well in this case, because they're public *another* class (CharacterMotor.cs in this case) reads in the variable values to determine how quickly to move the player. That's the power of the public access modifier!

There's actually an opposite to the public access modifer called "private". We haven't seen it used yet, but it's actually the default access modifier for variables, functions, and classes. So unless otherwise specified, variables, functions, and classes can only be accessed inside the script where they are defined. This is the default as it helps programmers to control how information can be accessed, to be sure it isn't changed when it shouldn't be.

2.2 Fixing Enemy A.I.

Enemy Artificial Intelligence (AI) is a staple of "action" games, where various different enemies create challenge for a player based on their attributes and behaviors. However, our "beat 'em up" presently has no apparent enemy behaviors! All they do is stand around, as portrayed in Figure 3. You can go up to them and hit them with the sword (by left clicking). Why don't we try that?

If you haven't already, play the game so you can run up to an enemy and hit it with your sword.

That's not all that exciting, is it?

At the heart of AI is the idea that an entity will act differently depending on the situation.

For example, if the player is close, then a ghost should attack. But if the ghost doesn't see the player, the ghost should search for the player. That structure kind of looks like the conditionals from Lesson Two, don't you think? In the AI we'll implement for these enemies we'll translate statements like that into code.

If you'll recall, we previously stated that classes served two main purposes, either holding information (like PlayerInfo) or behaviors. The next script we'll be working with is largely of the latter type, holding the behavior of the enemies. The script in question is Enemy.cs. It can be found in the Codebase folder, and then within the "NPC" folder. Double click on it to open it up in MonoDevelop.

When you load up Enemy.cs you should see something like Figure 4. The class Enemy looks very much like other classes we've seen before. It has "Update" and "Start" functions like we've seen before, we'll be using those to store the code to handle the enemies behavior. However, it also has a couple new features.

Definition!

RETURN:
A keyword which "returns" a value from a function to the line of code which called the function.

The first of the two new features can be seen in Enemy's other functions "GetSpeed" and "GetFiringRate". As you can see these functions are public, unlike Update and Start, meaning that they can be called by other code outside of this class. However, the new feature is the teal **return** keyword that both functions use.

Definition!

RETURN TYPE:
Part of a function definition, determines the type of value that function will return, or that it won't return anything if "void".

Though we haven't brought it up before, all functions have what's referred to as a **return type**. This return type specifies the type of value that a function will return, and it's determined by the word directly before a function name in a function definition. So for "Start" and "Update" this word is "void", which means that the function doesn't return anything (void is equivalent to "nothing"). However, both "GetSpeed" and "GetFiringRate" have "float" as their return type, meaning that they return float values when they are called. You may still be a bit confused about return types, but we'll give several examples further in the lesson.

2.3 Class Variables and Moving Enemies with movementController

The second new feature of Enemy is its use of two different new variables "movementController" and "attackController", both of which have types that are classes: NPCMovementController for movementController and RangeAttackController for attackController. It may seem odd to have a variable with the "type" of a class, rather than a primitive type (like float, boolean, or string) but it's actually very typical! We previously referred to classes as "complex variable types", but didn't explain why. If you think back to our other lessons, you might remember that classes had variables inside them (LessonTwoGame has seesPlayer, for example). The reason classes are referred to as "complex" variable types is that they can store *other variables inside them*. We therefore can have variables like movementController, that store *other* variables.

We'll demonstrate why this is important with an example. NPCMovementController handles (as you might guess from the name), an enemy's movement. It handles movement via setting a goal for the enemy and then calculating a path to get to the goal.[1] We'll use the function "SetTargetPositionGoal" as it takes in a GameObject and calculates a path to that GameObject's present position.

Go ahead and add the below line to Start:

```
movementController.SetTargetPositionGoal (player);
```

[1] You can check out a full list of publicly accessible functions in NPCMovement at the end of this lesson, though we'll go over most individually as they come up.

```
1  using UnityEngine;
2
3  public class Enemy : MonoBehaviour {
4      //A reference to this enemy's id (generated at start)
5      public string id;
6      //The speed this enemy will move at
7      private float speed = 3;
8      //The firing rate when the enemy is still
9      private float firingRate = 2;
10     //A reference to the GameObject of the player
11     public GameObject player;
12     //A reference to this enemy's NPCMovementController, which handles movement
13     public NPCMovementController movementController;
14     //A reference to this enemy's RangeAttackController, which handles attacking
15     public RangeAttackController attackController;
16
17     //Returns the speed of this enemy
18     public float GetSpeed(){
19         return speed;
20     }
21
22     //Returns the firing rate based on whether or not the enemy is moving
23     public float GetFiringRate(){
24         //Returns different values if the enemy is moving
25         if (movementController.moving) {
26             return firingRate * 0.5f;//Firing rate is half when moving
27         }
28         else {
29             return firingRate;
30         }
31     }
32
33     // Use this for initialization
34     void Start () {
35
36     }
37
38     // Update is called once per frame
39     void Update () {
40
41     }
42 }
```

Figure 4: Enemy.cs when it's first loaded into MonoDevelop.

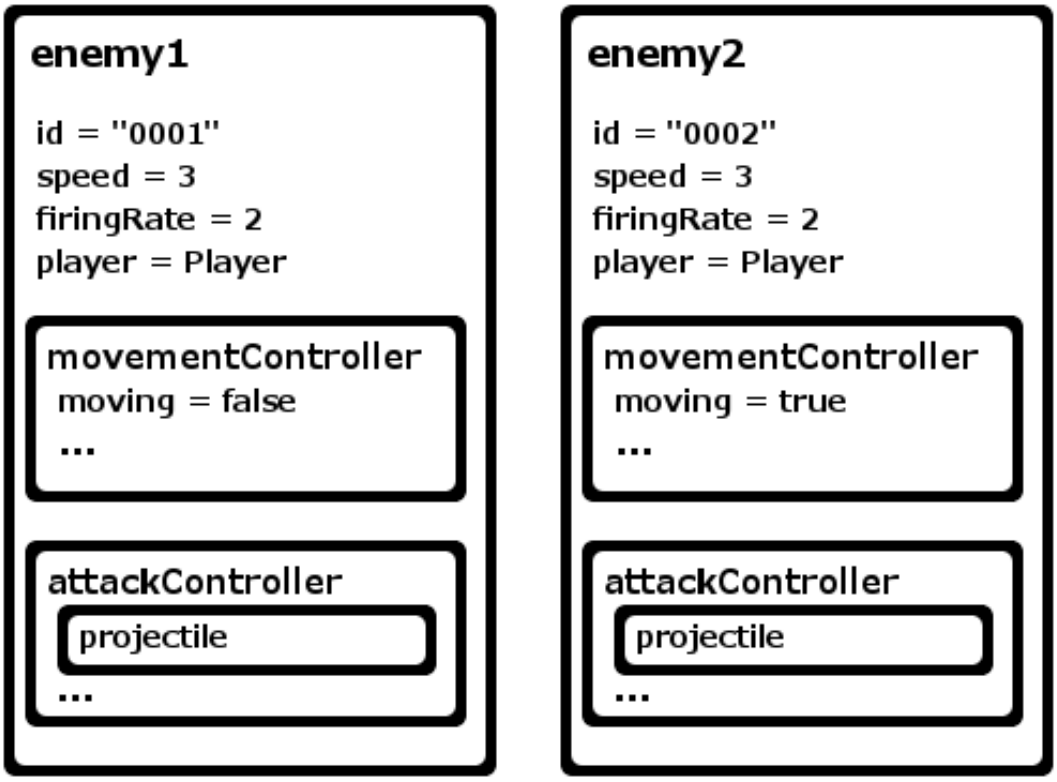

Figure 5: A diagram of two different Enemy objects during gameplay.

Play the game and wait a few moments without moving to find yourself surrounded by enemies.

Well that's something! But how is it working? We can actually do a small test to demonstrate.

Play the game again, but this time as soon as the game begins, move at least a few blocks away from the starting position.

The enemies didn't surround you that time, can you think why? The enemies instead surrounded the spot where you were to begin with. This is because the function call "SetTargetPositionGoal" was only ever called in Start, meaning that the function calculated a path for each enemy to the player's position *when Start was called*.

2.4 Class Variables and Functions

You may have wondered why we're called a function (SetTargetPositionGoal) by using a "class variable" (movementController.SetTargetPositionGoal) instead of just a class (NPCMovementController.SetTargetPositionGoal) as we have previously. This has a lot to do with the way that class variables work. We need to have different values stored in the different variables of movementCon-

troller. In the example that you just played through you probably saw the enemies taking very different paths to reach you and stopping at different times. That makes sense that they'd have to take different paths, given that they all start in different positions, right? They were able to store the value of this path, and follow it, as they could store *different* values!

Classes can have variables as we've seen before. A variable of a class type can actually store *different* values in the variables that it uses. That's how the enemies can have different paths. Take a look at Figure 5 which shows two different Enemy variables (the enemies you see in the game) during gameplay. Each enemy variable as variables of it's own, id, speed, and the GameObject "player" variable, which stores "Player" the actual object a player controls. These enemies are both "built" from the Enemy class that we've been modifying! It serves as their "blueprint" to use the same metaphor as we did with LessonTwo.

As you can see in Figure 5 not only do the two different enemies (enemy1 and enemy2 are their variable names) have different id values, but their movementController variables store different information as well. One is moving while the other is not!

This is the power of variables of a class-type (or "class variables). They will behave in the same way (have the same variables and functions), but can store very different values. That's why we call SetTargetPositionGoal with movementController and not with NPCMovementController. If we had to call that function with the class, instead of a class variable, then we could only store one of "path" value, and all the enemies would have to follow the same path.

Think of it of in terms of the "blueprint" metaphor we used in the previous lesson. A single "house" blueprint could be used to build multiple houses, as a single class can make multiple class variables. However, if we wanted to open a door on one of those houses, we wouldn't try to open the blueprint's door. Similarly, when we wanted to tell an enemy to compute a path to a goal, we don't tell NPCMovementController to do it, but an individual movementController variable. movementController is the house, NPCMovementController is the blueprint.

If you're still finding class variables to be confusing, continuing to fix the game should help in understanding what's going on.

2.5 OPTIONAL: Using Unity to Show the difference between objects

You can actually see these different values in Unity as well. If you play the game again, and then use "Escape" to get the mouse and then press the "pause" instead of the "play" button, you can see "0000" to "0009" in the "hierarchy" view. These are the enemy variables in Unity! If you click on them the "Inspector" view will show something that looks like 6. You can un-click the play button to get out of this mode. We'll use this way of interacting with Unity later, but for now it's just useful as a way to see the differences between the different enemies.

2.6 Making Enemies Move Intelligently

So far we've managed to make the enemies move all to the exact same point just once, after which they will again stay in place forever. That's not particularly useful. In the last section we were able to write what were essentially "rules" for how a door should act. Let's see if we can do the same here!

To begin with, let's see if we can just get the enemies to continually follow the player. To do that we'll need two things: the "ReachedGoal" function of movementController and an if statement. The "ReachedGoal" function returns a boolean value representing whether the enemy has reached its goal (true) or not (false). That means it has a "return type" of boolean. We can use this fact to

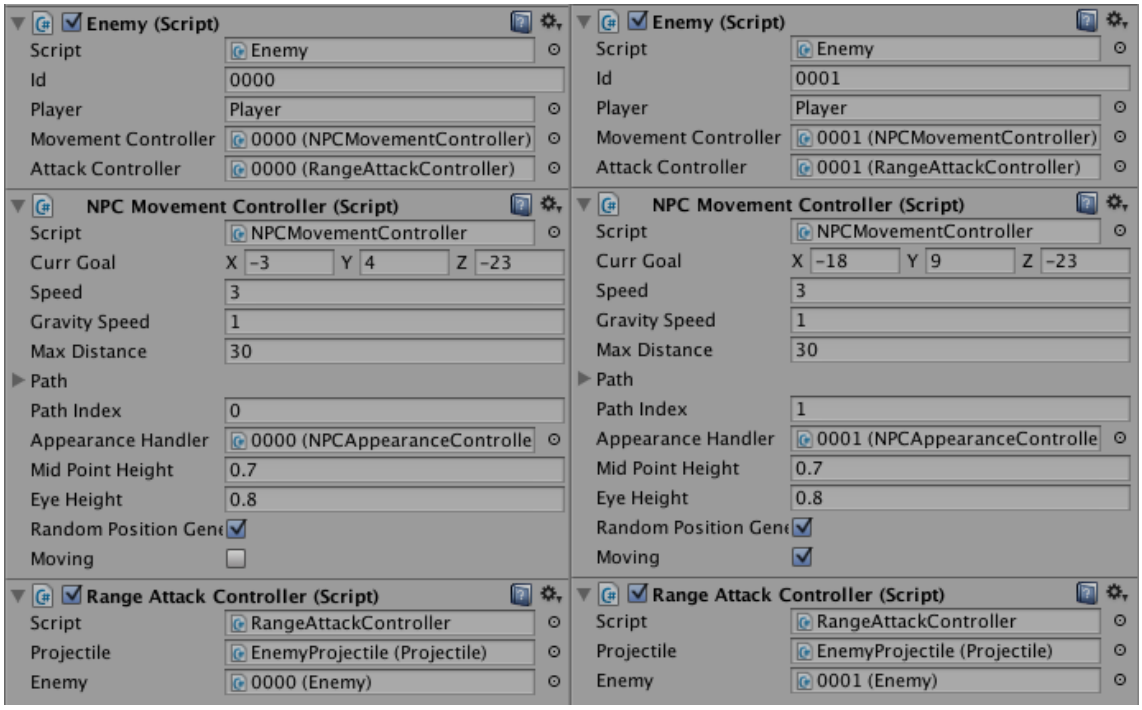

Figure 6: Unity's display of the public Enemy variables for "0000" and "0001".

determine when we need to use "SetPlayerPositionGoal" again. We'll want to put this new check and call in Update, so it occurs more times than just once at the start of the game. Place the following lines of code into Update:

```
bool reachedGoal = movementController.ReachedGoal();
if(reachedGoal){
        movementController.SetTargetPositionGoal (player);
}
```

We can't just call "movementController.SetTargetPositionGoal (player);" in Update as then the enemy would just continually recalculate a path, but never actually move along it. It also takes a lot of time to calculate a path and this would slow down the game.

Play the game again with the code added and saved in Enemy's Update. It may take some time (as they'll have to get to the starting position of the player before they'll plan out a new path), but eventually you'll have a small crowd of enemies following behind you.

You've done it! You've successfully managed to make the enemies follow you. Note that this Update code is being called in *each* Enemy individually as each of these enemies represents a different Enemy class variable. That's a lot to wrap your head around! But let's show you that it's working in game.

Play the game again. This time pay attention to the enemies at the "back" of the line following you as you move around. Try to see the moment they stop going to a different location and turn to where you are. That's the moment that "movementController.ReachedGoal()" is set to true as the enemy reached their last goal, and thus "movementController.SetPlayerPositionGoal (player);" is called and they head to your current location.

This shows you that this Update code is being called in each different enemy individually, as the moment they turn to face you, when "movementController.ReachedGoal()" returns true is different for each one! That's the trouble with writing good enemy behavior in code, it has to be able to work for different enemies in different situations!

However, this isn't particularly good enemy behavior. First of all, having the enemies immediately hone in on you seems pretty unfair. We probably want it so an enemy only chases the player if they can see the player. Can you think of a way to represent in code whether or not an enemy can see a player? Some kind of variable that could store whether or not something was true?

The answer is to add a new boolean variable to track whether or not the enemy can see the player. It'll need to have global scope (defined outside of a function in Enemy), can you think why? Let's name the variable "seesPlayer" with an initial value of false. Define it above the GetSpeed function.

Alright now we have a new variable, but that's not really particularly useful on its own. In trying to determine what to do next let's start with the effect we want to achieve and what we have so far, that's a typical approach for programmers in trying to determine what code to write. We know that we want the enemy to "chase" the player (as it has already) if it can see the player. Therefore it might make sense to go ahead and put the code in Update that we have so far (that we know makes the enemy chase the player) into an if statement with the "seesPlayer" variable as the condition. When you're done the script should look like Figure 7.

But we still don't quite have the behavior we want. For one, "seesPlayer" never gets set to true. Secondly, we haven't yet decided what an enemy should do *before* its seen a player (the code in Start still sends the enemy to the player's position). We could have it do nothing, so an enemy would just sit still until it saw the player. But then a ghost may never move! That's probably not

```
3  public class Enemy : MonoBehaviour {
4      //A reference to this enemy's id (generated at start)
5      public string id;
6      //The speed this enemy will move at
7      private float speed = 3;
8      //The firing rate when the enemy is still
9      private float firingRate = 2;
10     //A reference to the GameObject of the player
11     public GameObject player;
12     //A reference to this enemy's NPCMovementController, which handles movement
13     public NPCMovementController movementController;
14     //A reference to this enemy's RangeAttackController, which handles attacking
15     public RangeAttackController attackController;
16
17     bool seesPlayer = false;
18
19     //Returns the speed of this enemy
20     public float GetSpeed(){
21         return speed;
22     }
23
24     //Returns the firing rate based on whether or not the enemy is moving
25     public float GetFiringRate(){
26         //Returns different values if the enemy is moving
27         if (movementController.moving) {
28             return firingRate * 0.5f;//Firing rate is half when moving
29         }
30         else {
31             return firingRate;
32         }
33     }
34
35     // Use this for initialization
36     void Start () {
37         movementController.SetTargetPositionGoal (player);
38     }
39
40     // Update is called once per frame
41     void Update () {
42         if (seesPlayer) {
43             bool reachedGoal = movementController.ReachedGoal();
44             if(reachedGoal){
45                 movementController.SetTargetPositionGoal(player);
46             }
47         }
48     }
49 }
```

Figure 7: The class Enemy with the seesPlayer variable and additional if statement.

```
40      // Update is called once per frame
41      void Update () {
42          if (seesPlayer) {
43              bool reachedGoal = movementController.ReachedGoal ();
44              if (reachedGoal) {
45                  movementController.SetTargetPositionGoal (player);
46              }
47          }
48          else {
49              bool reachedGoal = movementController.ReachedGoal();
50              if(reachedGoal){
51                  movementController.SetRandomGoal();
52              }
53          }
54      }
```

Figure 8: Update of Enemy.cs after adding everything needed to keep the enemies constantly moving to random positions.

the best idea. How about instead we have the enemies move randomly across the island "searching" for the player?

Let's go ahead and add in the functionality to have the enemies move around the island randomly at first, even if they won't "switch" to seeing the player at this point. The function we'll want to use for this is "SetRandomGoal" of movementController. It works much in the same way as "SetTargetPositionGoal" except that it doesn't have any arguments. Replace the line in Start with:

```
movementController.SetRandomGoal();
```

This will cause each enemy to create an initial path to a random point (each point will be unique to each enemy), instead of to the player.

Play the game, and note that changed enemy behavior.

As you saw, enemies now move to a random position and then come to a stop. This is similar to how the enemies initially acted when we only had the "SetPlayerPositionGoal" line in Start. This should make sense if you think about it, as the only other code in Enemy.cs is in Update, and none of it will run as "seesPlayer" is not ever set to true (at the moment). However, what we'd like to have happen is for the enemy to keep moving to random positions until it sees the player. Can you think of how to keep the enemies moving to random positions if seesPlayer isn't true? Maybe if we say that we'll need to use an "else statement" in Update? And that you can use the code to have enemies chase the player as an example?

We'll need an else statement to handle when the enemy can't see the player. We'll then need to nearly replicate the code within the if statement to cause the enemy to find a new random path after it reaches its current goal. The difference is we'll need to call "SetRandomGoal" instead of "SetPlayerPositionGoal". When you're done, your Update function should look like Figure 8.

Play the game and note that changed enemy behavior.

Now you've got enemies moving everywhere! The last thing we're missing is figuring out when the enemy sees the player. To do that we'll make use of the "CanSeeTarget" function of movementController. CanSeeTarget will return a boolean value if the movementController in question can "see" the passed in GameObject. This is exactly what we need for updating the "seesPlayer" variable! We can use the below line in Update:

```
seesPlayer = movementController.CanSeeTarget(player);
```

Insert the "movementController.CanSeeTarget" line above the if statement in Update and play the game. Try and see if you can get an enemy to first start following you then "lose track" of you. As a warning, this will be very difficult.

It's possible to get the behavior where an enemy starts "chasing" you, but then "loses track" of the player. But it's actually really tough to tell! Can you think why that might be? Let's walk through a possible example to try to illustrate what's happening here.

Check out Figure 9 for an illustration of four different moments with just a single enemy to explain what's happening with the code at this stage. Before Start, the enemy has no path as it was just created (1).
After Start, the enemy has a path to a random point and begins to travel along it (2).
While traveling along the path, the ghost might see the player (seesPlayer will now be equal to true) (3).
But the ghost isn't done with its path (ReachedGoal would not return true) so it continues along it until the path is done (4).
However, the ghost cannot "see" the player in (4), so seesPlayer will be false. Therefore when it reaches the end of its path, it will pick a random goal instead of choosing the player as the goal (it will run the else statement since seesPlayer is false, instead of running the if statement).

That's no good! What we need is something where as soon as the enemy sees the player, it begins to chase the player. Essentially, we need to be able to set the enemy's goal to the player when it sees the player after previously *not seeing the player*. What we need to know is when seesPlayer switches from false to true.

There's a couple different ways to solve this. We could change where we call "CanSeeTarget" to be within the else statement. If we did that then we'd know that seesPlayer was false (so we got into the else statement), and then if seesPlayer became true we'd know it had just switched from false to true. However, that mean that we'd have run the other code in the else statement when that wasn't necessary. We'd essentially have had the enemy acting like it hadn't seen the player that frame even though it had!

To avoid running unnecessary code we'll use the "not" logical operator (!). As you might recall from the last section, logical operators work on conditions like how mathematical operators (+, -, etc) work on numbers. We'll start by storing the return value of "CanSeeTarget" into a temporary variable, let's call it "nowSees". Replace the "movementController.CanSeeTarget" line at the top of Update with the following two lines (they are separated on purpose, as we'll explain):

```
bool nowSees = movementController.CanSeeTarget(player);

seesPlayer = nowSees;
```

What we want is when nowSees is true *and* seesPlayer is false (before seesPlayer is set to nowSees). As that will mean that previously (as in the last time Update was called) the enemy couldn't see the player, but now it can. To say that a bit more formally we want a condition where

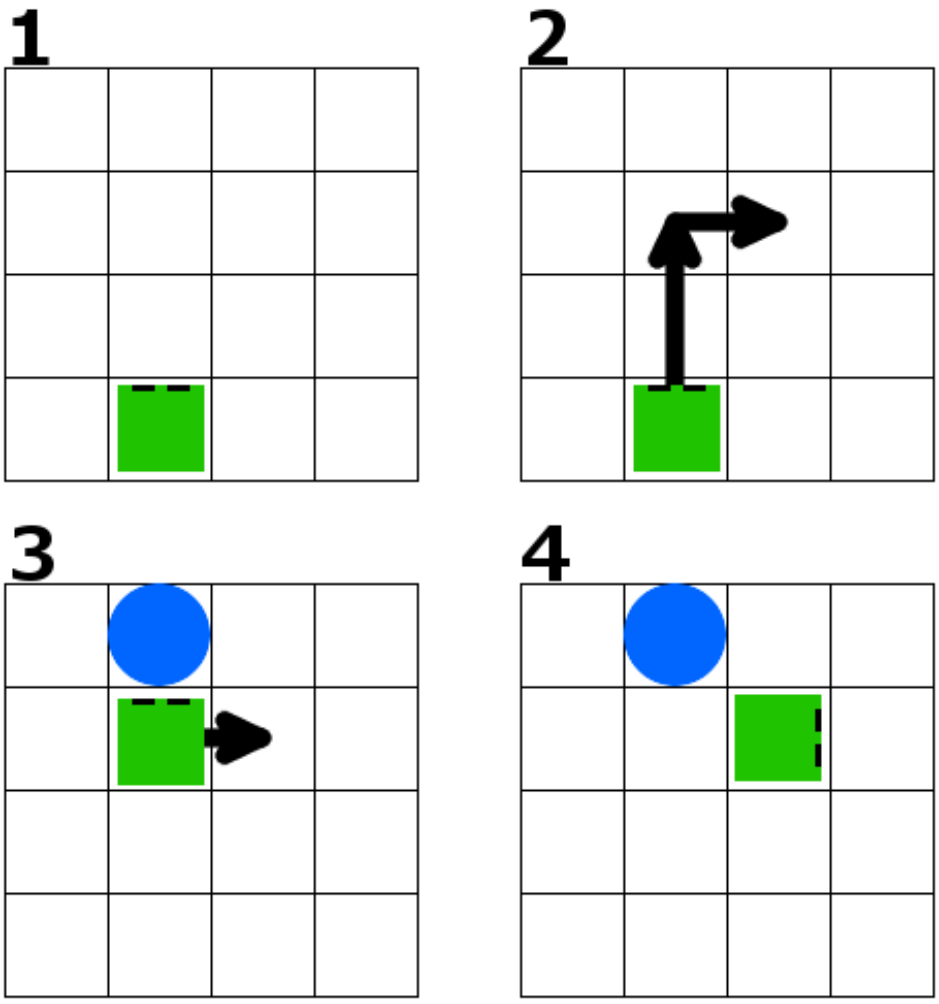

Figure 9: Example of how the current Enemy.cs script works. The green square is a ghost, the arrows are its path, and the blue circle is the player.

```
// Update is called once per frame
void Update () {
    bool nowSees = movementController.CanSeeTarget (player);
    if (nowSees && ! seesPlayer) {
        movementController.SetTargetPositionGoal(player);
    }
    seesPlayer = nowSees;

    if (seesPlayer) {
        bool reachedGoal = movementController.ReachedGoal ();
        if (reachedGoal) {
            movementController.SetTargetPositionGoal (player);
        }
    }
    else {
        bool reachedGoal = movementController.ReachedGoal();
        if(reachedGoal){
            movementController.SetRandomGoal();
        }
    }
}
```

Figure 10: Update of Enemy.cs after adding everything needed to have enemies search for then track the player.

nowSees is true and seesPlayer is not true. If we translate that into code we get the following condition:

```
nowSees && ! seesPlayer
```

This condition will be true when nowSees is true *and* seesPlayer is *not* true. If we put that condition into an if statement, then we can be sure the code in the if statement will run only when the enemy now can see the player, but could not see the player previously. Inside this if statement we could then put a call to "movementController.SetPlayerPositionGoal (player);" to make the movementController immediately calculate a path to the player and begin to follow that path. When you're done your Update function should look something like Figure 10.

Play the game and get some enemies chasing you. It should be much easier this time!

And with that you've done it! You've got the enemy moving intelligently. At this stage you could try to add additional behavior to make the enemy more intelligent. For example, you could add a special behavior in for when the enemy loses sight of the player (when seesPlayer is true and nowSees is false), perhaps having the enemy go ahead and move to the player's current location as that's close to where the enemy had last "seen" the player? That's up to you!

It might be surprising, but a lot of AAA games make use of rules-based enemy artificial in-

```
// Update is called once per frame
void Update () {
    bool nowSees = movementController.CanSeeTarget (player);
    if (nowSees && ! seesPlayer) {
        movementController.SetTargetPositionGoal(player);
    }
    seesPlayer = nowSees;

    if (seesPlayer) {
        if (movementController.ReachedGoal ()) {
            movementController.SetTargetPositionGoal (player);
        }
    }
    else {
        if(movementController.ReachedGoal ()){
            movementController.SetRandomGoal();
        }
    }
}
```

Figure 11: Update of Enemy.cs after removing the unnecessary "reachedGoal" variable.

telligence very similar to this! Of course, at the moment the ghosts running around aren't really "enemies" in that they make no attempt to attack the player. We'll change that though!

2.7 Cleaning up Code

Before making the enemies attack the player, let's take a moment to clean up the code we have so far.

If you'll recall, we don't actually need to use a boolean variable as a condition in an if statement. In the previous section we put a direct boolean value (true and false) inside an if statement.

Using this information it should be pretty easy to see that we don't need to use either "reachedGoal". We can simply use the fact that "ReachedGoal" returns a boolean value to place the call to that *inside* the if statement! Go ahead and remove both the "reachedGoal" variables from Update and just used the "ReachedGoal" function instead. When you're done Update should look like Figure 11. We can do the same thing with two less lines of code, not bad!

We'll continue to use returned values directly from functions (instead of storing them in unnecessary variables) in the next section.

2.8 Making the Enemies Attack!

We used movementController in order to control Enemy movement. As you might guess, we'll be using attackController to handle attacking. Go ahead and insert the following line near the top of

Update:

```
attackController.Fire(player);
```

This will cause the enemy to shoot off a projectile at the passed in GameObject (player). Let's see what the effect of this is!

Go ahead and play the game. Notice the behavior of the enemy's and the projectiles they shoot almost constantly.

This behavior isn't particularly intelligent, is it? Since the enemies just fire constantly it tends to leave stray shots firing in every direction. One simple change we could make is to have the enemies only shoot if they can currently see the player. Move the call to "Fire" to inside the "seesPlayer" if statement.

Move the "Fire" line to only be called when the enemy can "see" the player. Play the game, and see what effect this has!

This behavior is definitely improved, but its still not perfect. The biggest problem seems to be that the projectiles "explode" before hitting the player. That's because these projectiles are being shot when the player is out of range!

We can get the projectiles range by using attackController's "GetProjectileRange", which will return the float value for the max distance the projectile can travel. We can then use movementController's "GetDistanceToTarget", which takes in a reference to a GameObject, and calculates the distance between it and the enemy. These two functions together will help us change the enemy's behavior to only shoot when the player is in range.

Definition!

RELATIONAL OPERATOR:
A symbol or symbols that represents a boolean relationship between two values.

What we want is to only shoot (have Fire be called) if the distance between the player and enemy (GetDistanceToTarget) is less than the distance the projectile can fly (GetProjectileRange). That phrasing should make it sound like an if statement might be useful. However, we'll have to use something new called a **relational operator**. A relational operator is a symbol or symbols that represents a boolean relationship between two values. For examples there's ">" which is the greater than symbol. In code if we had a line:

```
bool test =  4 > 2;
```

Then test would store a value of true. Since it's "true" that 4 is greater than 2. You may have run into this symbol and it's opposite less than (<) in a math class. They work in code in a very similar way, as you can guess! For example if we used less than instead of greater than:

```
bool test =  4 < 2;
```

Then test would store a value of false.

But how is this useful to you? Well remember the phrasing we used earlier: "if the distance between the player and enemy (GetDistanceToTarget) is less than (GetProjectileRange)". Now that you know about relational operators, you should see that we can make use of them to check for this exact condition and to only "Fire" when it is true. Replace the line that calls "Fire" (inside the seesPlayer if statement) with the following lines (as a note, both GetDistanceToTarget and attackController should be on the same line in update, we just can't fit those on the same line here):

```
bool inRange = movementController.GetDistanceToTarget(player)
         < attackController.GetProjectileRange();
if(inRange){
        attackController.Fire(player);
}
```

```
void Update () {
    bool nowSees = movementController.CanSeeTarget (player);
    if (nowSees && ! seesPlayer) {
        movementController.SetTargetPositionGoal(player);
    }
    seesPlayer = nowSees;

    if (seesPlayer) {
        if(movementController.GetDistanceToTarget(player)<attackController.GetProjectileRange()){
            attackController.Fire(player);
        }
        if (movementController.ReachedGoal ()) {
            movementController.SetTargetPositionGoal (player);
        }
    }
    else {
        if(movementController.ReachedGoal ()){
            movementController.SetRandomGoal();
        }
    }
}
```

Figure 12: Update with Fire now only firing when it makes sense to do so.

Play the game, and notice the effect of these changes!

Does what's happening make sense? Now Fire will only be called when the player is closer to the enemy than that enemy's max range. This doesn't mean the enemy will always hit, but it does mean that the enemy won't just shoot wildly. And that looks more intelligent!

Remember that you don't need to put an variable into an if statement, just a boolean value. That means we can actually don't need inRange, we can put "movementController. GetDistanceToTarget(player) < attackController.GetProjectileRange()" directly into the if statement! Check out Figure 12 for how Update should end up looking.

You've done it! Now you have an enemy that moves and shoots at least somewhat intelligently! At this point you could add more if statements to make it behave more intelligently on your own. Think about the condition that you want to create (for example: if the player is within range, the enemy should always move towards the player as if it heard it), then translate that into code (for example: if(movementController.GetDistanceToTarget(player) < attackController.GetProjectileRange())), and determine where in Update it makes sense to put this if statement. Feel free to add as many behaviors as you like, see how smart of an enemy you can make!

2.9 Creating End Conditions

You may have already gone and "destroyed" all the enemies in the game and been left to nothing to do but stop playing. You may have also noticed that no matter how many of the projectiles hit you, you can never lose! If not its worth noting at this point that there is no way to win or lose the game! For both we'll make use of brand new scripts and we'll need to make use of the relation operators as well.

2.10 Losing

We'll start with making it possible to lose. To do that we'll want to open up a new script "Projectile.cs". As you can probably guess, this script handles the behavior for projectiles. It can be found in the Codebase folder, and then within the NPC folder. Double click "Projectile" to open it up.

There's a lot going on in the "Projectile" class! Feel free to take a deeper look at it, but the function we need is called "HitPlayer" and it's right at the top of the class. In this function we can specify what should happen when a projectile hits the player. Right now all that happens is that the projectile is destroyed via the line "Destroy (gameObject);".

For now, let's go ahead and make the player lose if they get hit by just one shot (we can make this less difficult later). We can end the game by calling the line:

```
GameManager.EndGame ("You Lose!");
```

EndGame will immediately stop play and display whatever string value is passed to it. Therefore, we can use it for both winning and losing! Go ahead and insert that line somewhere *before* the line that destroys the projectile in HitPlayer.

Play the game! You'll almost immediately see that this is an unreasonable level of difficulty.

Because the enemies move randomly, it can be only seconds before the first time the player is hit by a projectile. Just one shot isn't even enough time to act!

Many games use a "hitpoints" system. Hitpoints typically represent a player or enemy's health, and the player or enemy doesn't die till they lose all of their hitpoints. Let's see if we can use something like that here! To do that, we'll actually need to introduce a new variable to track the player's hitpoints. Can you think of where might be a good place to store this value?

Open PlayerInfo.cs back up and add a new integer variable named "hitpoints" (there's no need for decimal points here). We'll store this information here since that's where the rest of the player information is, and we have access to it inside HitPlayer (since PlayerInfo playerInfo gets passed into the function). Since we'll want to be able to both get and alter the value of this variable, it will need to be public. Set its initial value to something high at first, perhaps a hundred? In total the line should look like:

```
public int hitpoints = 100;
```

Add that to PlayerInfo then return to Projectile.cs.

What we want now is to have the value of hitpoints get lower every time a Projectile hits the player. Remember how we can alter the value of a variable as we did in LessonOne, we'll want to use a line like:

```
playerInfo.hitpoints = playerInfo.hitpoints -1;
```

This will change the value of hitpoints stored in playerInfo to be one less every time the player is hit by a projectile. We used a similar technique in LessonOne. It works because the right side of the equals sign is evaluated first, meaning that it gets the value of playerInfo.hitpoints and subtracts one from it. Then the equals sign stores that new value into the same variable. This has the effect of making playerInfo.hitpoints go down by one every time this line is called!

The last thing we'll want to do is add a call to EndGame with "You Lose!" (feel free to change the text used) only if hitpoints is less than 0. Translating that statement into code we get:

```
if (playerInfo.hitpoints < 0) {
```

```
//This function is called when the player is hit
private void HitPlayer(PlayerInfo playerInfo){
    //TODO; What should happen when you hit the player?
    playerInfo.hitpoints = playerInfo.hitpoints -1;
    if (playerInfo.hitpoints < 0) {
        GameManager.EndGame ("You Lose!");
    }
    //Destroy the projectile
    Destroy (gameObject);
}
```

Figure 13: HitPlayer when it is complete.

```
        GameManager.EndGame ("You Lose!");
}
```

Insert both the line changing hitpoints value and the new condition into HitPlayer and play the game (your HitPlayer should now look like Figure 13. If you're willing to wait a very long time, you may still be able to die, but a hundred projectiles is a lot of projectiles.

You've done it! You've created a lose condition! Feel free to alter the starting value of hitpoints to make it a little bit more challenging, and avoiding ghosts more important. Try values below 10 or so for a pretty challenging experience! While you're in Projectile.cs feel free to change the "projectileSpeed" or "maxLifeTime" variables as well, to make the projectiles easier or harder to dodge. Play around with it!

2.11 Winning

Last but not least, we have to create a win condition. For this we'll need to open the last of the scripts we'll be accessing in this lesson Sword.cs. Sword.cs can be found in the "Codebase" folder and then within the "Items" folder.

While Sword mostly looks like the code we've seen before, for the first time we have a class that doesn't "inherit from MonoBehavior" (: MonoBehavior) instead it inherits from "Item". This allows the "Use" function be called when the item is held and used. But what we really care about is the HitEnemy function. Currently all it does (as you can see) is destroy the Enemy on a hit "Destroy (enemy.gameObject);".

What sort of win condition should we have? Maybe something like if the player has destroyed a certain number of enemies? There are ten enemies in the game, but to begin with let's say the player need only defeat 1 enemy to win. So let's go ahead and put in a new call to "GameManager.EndGame" but this time with text related to winning:

```
GameManager.EndGame ("You Win!");
```

Insert the EndGame line into HitEnemy and play the game. Destroying just a single enemy seems a bit easy, doesn't it?

```
int enemyCount = 0;

//This method is called when the sword hits an enemy
private void HitEnemy(Enemy enemy){
    Destroy (enemy.gameObject);

    //TODO; Add code here for handling the win condition
    enemyCount = enemyCount + 1;

    if (enemyCount > 9) {
        GameManager.EndGame ("You Win!");
    }
}
```

Figure 14: HitEnemy when it is complete.

What if the player instead had to kill many of the ghosts, say all ten to win? That sounds like it's a better win condition, but how could we represent that in code? Well we'll need some way to store the value of how many enemies have been destroyed so far. To do that, we'll need to add a new variable to Sword with global scope. Since we're counting something, we have no need of decimals so we can use an integer value. Let's call it "enemyCount" and it'll need to have an initial value of 0 so we count up. We don't need to access this variable outside of Sword.cs so feel free to make it private, or simply not specifying an access modifier.

```
int enemyCount = 0;
```

Add that line to above HitEnemy. We'll also need to add one to it every time HitEnemy is called. Within HitEnemy we'll need the line:

```
enemyCount = enemyCount +1;
```

Once we've done that, let's figure out what the condition can be before the Win message should be called. It should be something like: enemyCount is greater than nine (since there are ten ghosts we want to be destroyed). To translate into code we'd use something like:

```
if (enemyCount > 9) {
        GameManager.EndGame ("You Win!");
}
```

Add everything to HitEnemy such that it ends up looking like Figure 14. Play through the game and try to get all the enemies!

And that's it! You've created a win condition! There's other things you could add here. For example, you could of course add hitpoints to enemies as well, decrement the enemy hitpoints in HitEnemy until they fall too low then call "Destroy (enemy.gameObject);". But that's up to you! You have a full, working "beat 'em up" game.

2.12 End of Lesson Three

That's all we have for Lesson Three! Feel free to play with your game more, altering the speeds and behaviors of the various pieces till you're happy with them. In this lesson you learned how to:

1. Use access modifiers (private/public) to restrict access
2. Make use of functions that "return" values
3. Create complex enemy behavior with if statements
4. Call on class variables to run similar behavior on individual objects
5. How to use relational operators to get boolean values from numbers

In the next lesson we'll focus on the actually building up and creating your own unique world to play a simple version of an adventure game.

2.13 movementController Functions

Function Name	Arguments	Behavior	Return Type
SetTargetPositionGoal	GameObject value of the target	Sets the enemy's goal to be the target's current position.	[Not Any]
GetDistanceToTarget	GameObject value of the target	Calculates distance between the target and this enemy.	Returns a float value of the distance between the target and this enemy.
CanSeeTarget	GameObject value of the target	Calculates whether or not the enemy can see the target GameObject.	Returns a boolean value, true if can be seen, false otherwise
SetRandomGoal	[Not Any]	Calculates a random position and sets the enemy's current goal to it.	[Not Any]
ReachedGoal	[Not Any]	Determines whether or not the enemy has reached its current goal.	Returns a boolean value, true if the enemy has reached the goal, false otherwise.
Stop	[Not Any]	Immediately stops the enemy and sets its goal to its current position.	[Not Any]

2.14 attackController Functions

Function Name	Arguments	Behavior	Return Value
GetProjectileRange	[Not Any]	Calculates the largest possible distance a projectile could travel	Returns a float value of max projectile distance.
Fire	GameObject value for target	Creates and fires a projectile at the target, if it can.	[Not Any]

2.15 Lesson Three Glossary

Vocabulary Word	Definition	Examples
access modifier	A keyword that limits what other code can "see" a variable, function, or class.	public and private
public	The access modifier that allows access of a class, function, or variable outside of the class where its defined.	public float GetSpeed()
private	The access modifier that doesn't allow access of a class, function, or variable outside of the class where its defined..	private float speed = 2.0f;
return	A keyword which "returns" a value from a function to the line of code which called the function.	return speed;
return type	A keyword which "returns" a value from a function to the line of code which called the function.	void
getter	A slang term for a type of function that is publicly accessible and returns the value of some private variable	GetSpeed
relational operator	A symbol or symbols that represents a boolean relationship between two values.	> and <