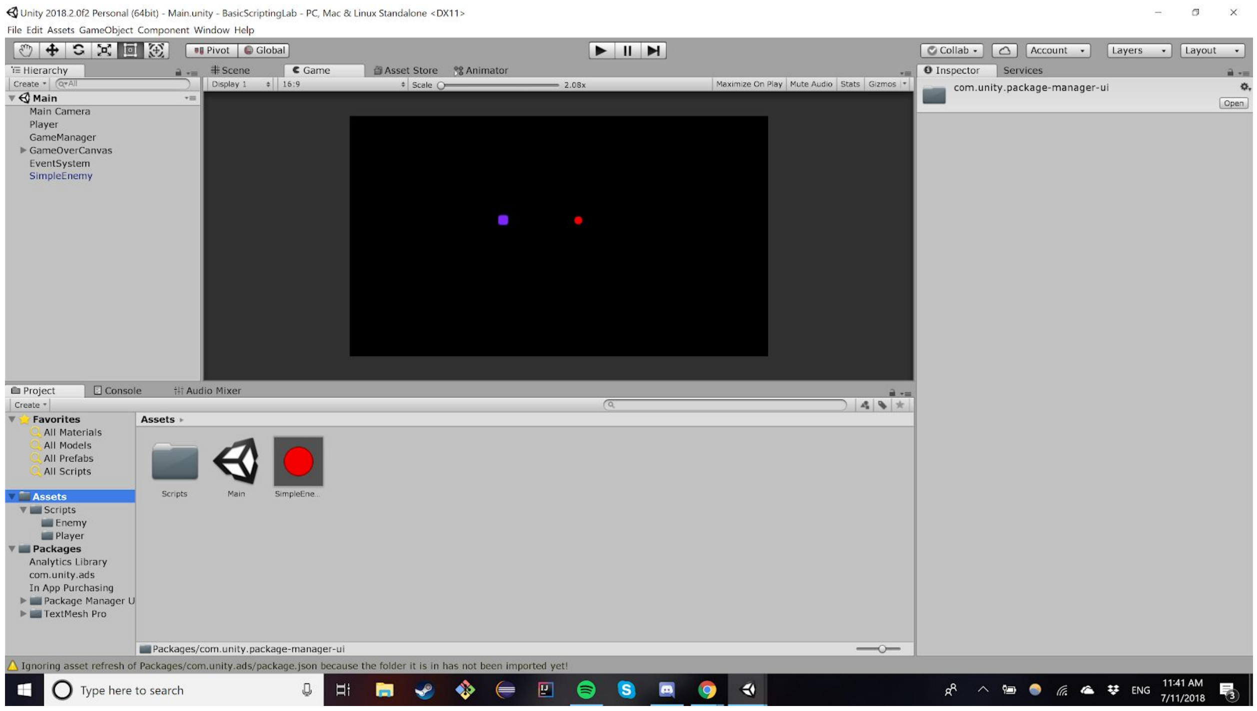
Basic Scripting Lab

Information only relevant to artists will be in blue
Information only relevant to programmers will be in red
Information relevant to all will be in black

Lab Start

In this lab we will be taking a surface level look at scripting by adding a new enemy type to a minigame. No actual coding will be involved.



```
Ignoring asset refresh of Packages/com.unity.ads/package.json because the folder it is in has not been imported yet!
```

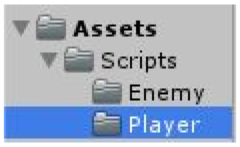
First off select the *Main* scene from the *Project* tab. You should see a purple and red square on a black background .

Now hit play and see how the game currently plays. You should see red circles coming towards you. If you get hit once you die.

Dying in one hit isn't very fun! Let's figure out how to fix that!

Opening a script

Scripts are found in the *Project* tab under *Assets.* Navigate to the *Player* folder and inside you should see a script called *Health*

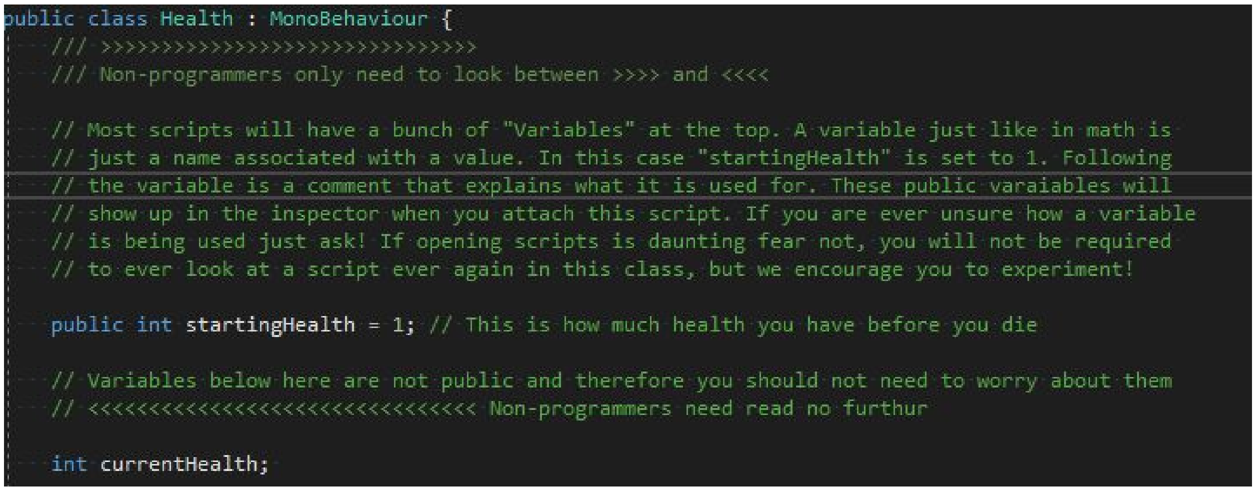


Double click the script to open it. (This sometimes takes a while)

It is good practice to keep all your variable definitions at the top of the class and provide a brief comment explaining what it does. Some variable protection level explanations to get you started:

- **Private** - This is default and means no other script can access it, not even children
- **Public** - All scripts can access this variable **AND it shows up as a field in the inspector when this script is attached to an object**
    - If you need a variable to be public that should NOT show in the inspector write "**[HideInInspector]**" (no quotes) before or on the line above
- **Protected** - Child scripts can access this but no other scripts can

As an artist you probably won't be editing scripts much if at all but it can be useful to know the basics. Towards the top of the file you should see a handful of comments. Read through them. The image below contains everything you would need to read in the script



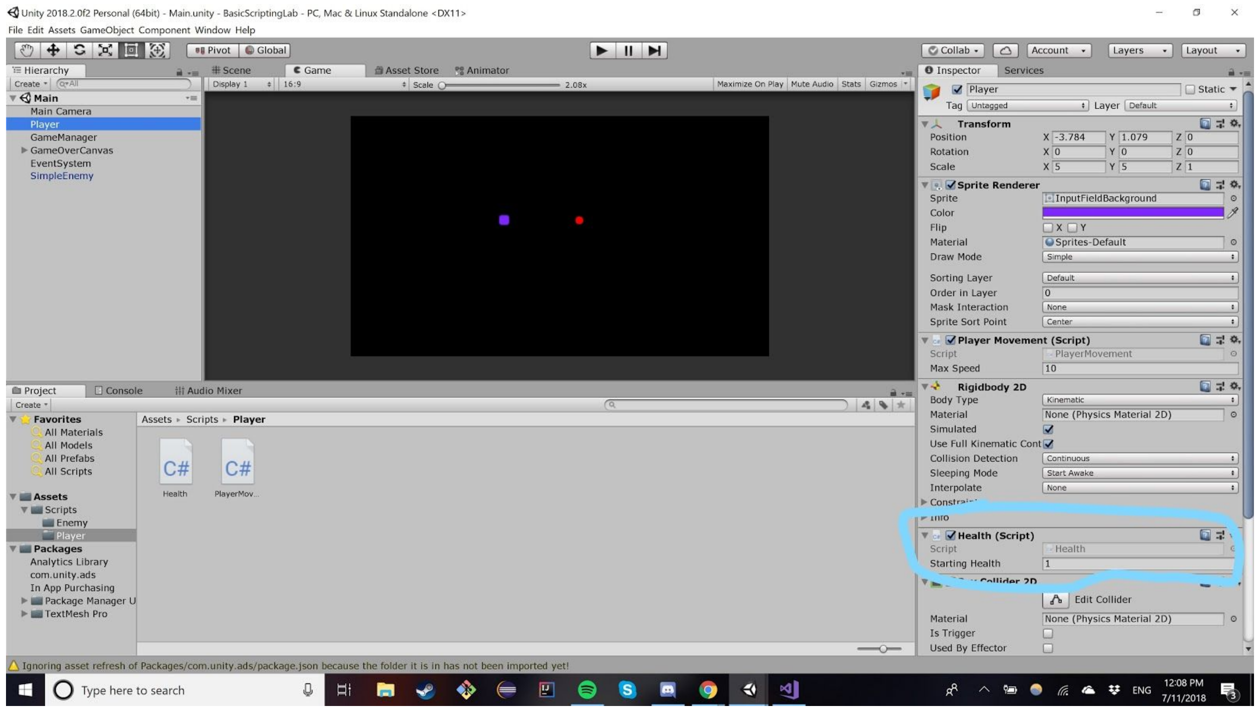
```
public class Health : MonoBehaviour {
    /// >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
    /// Non-programmers only need to look between >>>> and <<<<

    // Most scripts will have a bunch of "Variables" at the top. A variable just like in math is
    // just a name associated with a value. In this case "startingHealth" is set to 1. Following
    // the variable is a comment that explains what it is used for. These public varaiables will
    // show up in the inspector when you attach this script. If you are ever unsure how a variable
    // is being used just ask! If opening scripts is daunting fear not, you will not be required
    // to ever look at a script ever again in this class, but we encourage you to experiment!

    public int startingHealth = 1; // This is how much health you have before you die

    // Variables below here are not public and therefore you should not need to worry about them
    // <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< Non-programmers need read no furthur

    int currentHealth;
```

So now we all know about public variables, but how can we change them? You can change them within the script but this will only affect future objects that use this script, what about the ones already using it?

Each object that has a Health script attached to it will have its own copy of the variables within (unless they are static variables). Public variables will take on their default values unless they are set to another value via the inspector. Inspector changes to public variables override script values!

The better way to modify these public variables is through the inspector. So lets exit out of this script and go back to Unity. Now we can change our starting health to something other than one.

To do this select the *Player* object in the hierarchy. In the *Inspector* look for the component titled *Health (Script)* You should see that it has one modifiable field “Starting Health.” Change this from a 1 to a 3.



Unity 2018.2.0f2 Personal (64bit) - Main.unity - BasicScriptingLab - PC, Mac & Linux Standalone <DX11>

Player
Tag: Untagged Layer: Default

Transform
Position X -3.784 Y 1.079 Z 0
Rotation X 0 Y 0 Z 0
Scale X 5 Y 5 Z 1

Sprite Renderer
Sprite: InputFieldBackground
Material: Sprites-Default
Draw Mode: Simple
Sorting Layer: Default
Order in Layer: 0
Mask Interaction: None
Sprite Sort Point: Center

Player Movement (Script)
Script: PlayerMovement
Max Speed: 10

Rigidbody 2D
Body Type: Kinematic
Material: None (Physics Material 2D)
Simulated: ✓
Use Full Kinematic Cont: ✓
Collision Detection: Continuous
Sleeping Mode: Start Awake
Interpolate: None

Health (Script)
Script: Health
Starting Health: 1

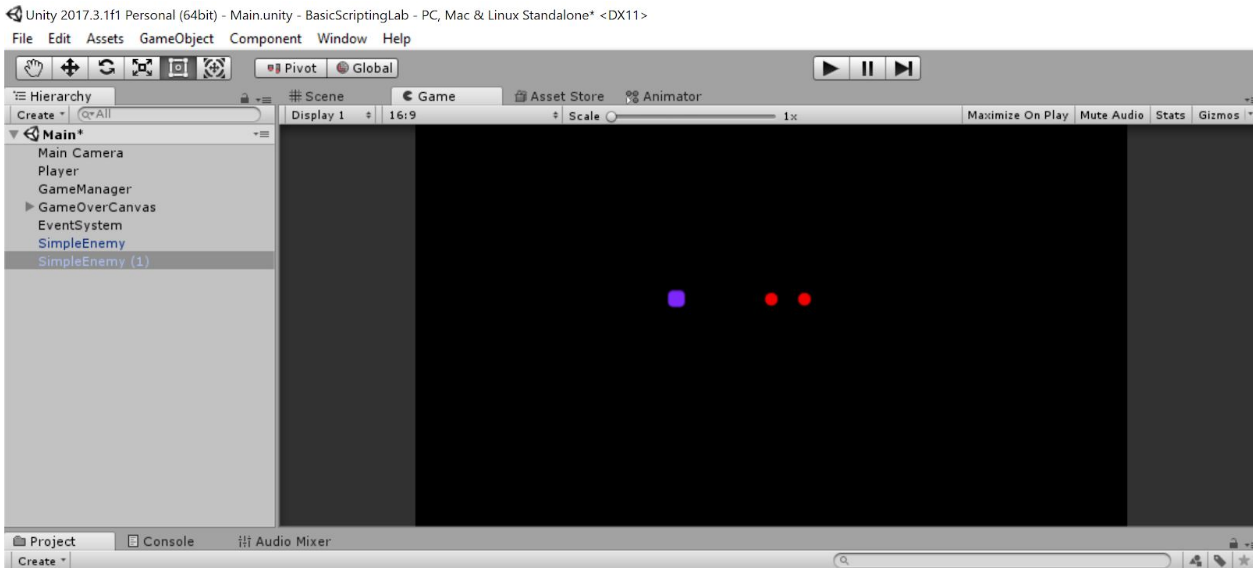
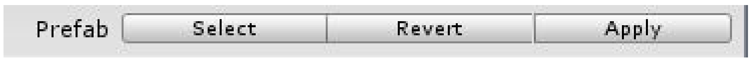
Material: None (Physics Material 2D)
Is Trigger
Used By Effector

Ignoring asset refresh of Packages/com.unity.ads/package.json because the folder it is in has not been imported yet!

Prefabs
A *Prefab* is essentially a *GameObject* that you save as a *Resource* / *Asse*t (like a script or character art). To create a *prefab* you can either right click in the *Project* window and select *Create* > *Prefab* or you can simply drag an existing *GameObject* from the *Hierarchy* tab into the *Project* window.

Once you have a *Prefab* you can repeatedly drag it into the scene to create copies of the object. Try doing this with the *SimpleEnemy prefab*. You will notice a few things:

- The color in the hierarchy changed and the names are followed by a number

- 
Unity 2017.3.1f1 Personal (64bit) - Main.unity - BasicScriptingLab - PC, Mac & Linux Standalone* <DX11>
File Edit Assets GameObject Component Window Help
Pivot Global
Hierarchy
Create
Main*
Main Camera
Player
GameManager
GameOverCanvas
EventSystem
SimpleEnemy
SimpleEnemy (1)
Scene Game Asset Store Animator
Display 1 16:9 Scale 1x
Maximize On Play Mute Audio Stats Gizmos
Project Console Audio Mixer
Create
- When you select one of these objects there are new buttons at the top of the inspector. These are used to keep the different prefab copies in sync. Hitting apply will apply the changes made to this copy to all copies. Revert will undo any changes on this copy. Select just shows you the prefab this copy was made from


Creating the new enemy type
Delete all but one of the *SimpleEnemy* **hierarchy** objects **(DON'T DELETE THE PREFAB)**. We will use this last *SimpleEnemy* to create a new enemy type!

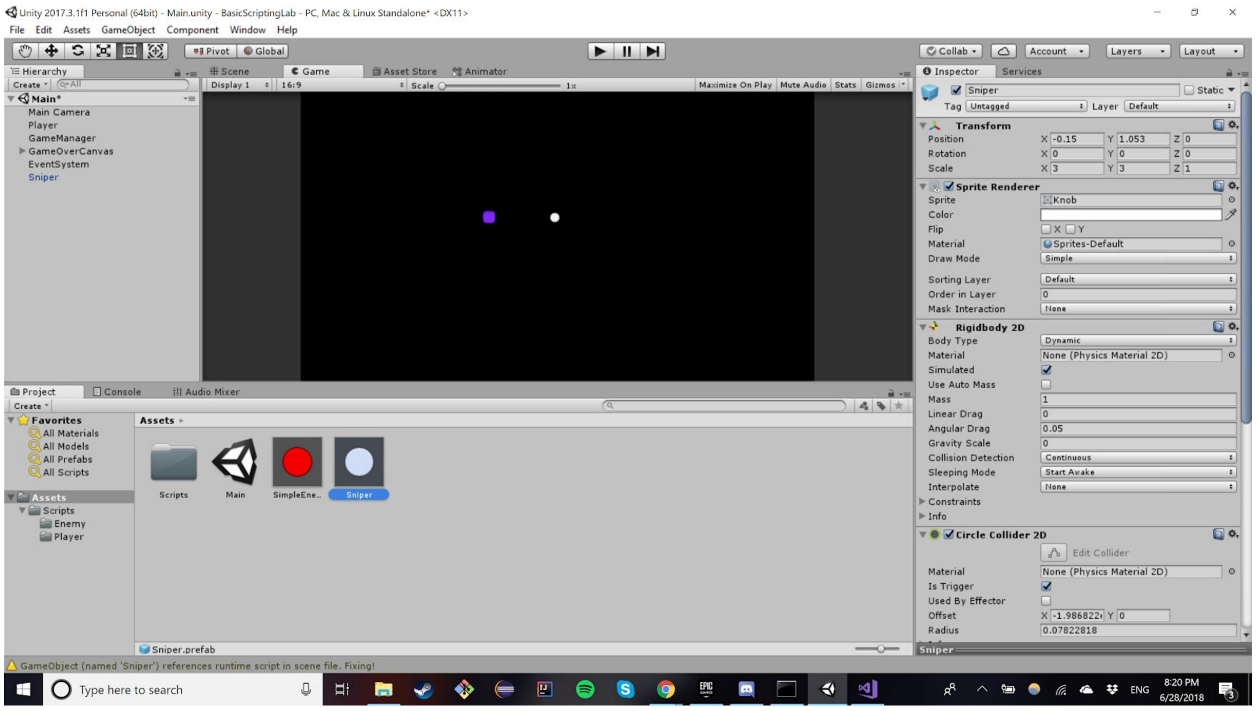
First we need to create a new prefab by right clicking in the *Assets* folder of the *Project* tab. Then selecting *Create* > *Prefab*. Give it the name "Sniper"; we have now created an empty template for our new enemy.

Next rename the *SimpleEnemy* object to "Sniper" from the hierarchy tab, then drag the object into the prefab you just created. This will break the link with the *SimpleEnemy Prefab* and now we are free to change things without messing up the original enemy type.

Here is a list of things you should change:

- Change the color of the *SpriteRenderer* component to whatever you'd like
- **Do not remove the SimpleEnemyMovementScript**
- Change the damage on the "Attack" script to be 2.
- Click *Add Component* and search for "Sniper Movement"
    - I have found a speed of 16 and a spawn distance of 10 works well

Now rename the enemy to "Sniper" and drag the enemy you just created from the hierarchy down below next to SimpleEnemy. This will automatically create a prefab.


Unity 2017.3.1f1 Personal (64bit) - Main.unity - BasicScriptingLab - PC, Mac & Linux Standalone* <DX11>
File Edit Assets GameObject Component Window Help
Hierarchy
Main*
Main Camera
Player
GameManager
GameOverCanvas
EventSystem
Sniper
Scene Game Asset Store Animator
Project Console Audio Mixer
Favorites
All Materials
All Models
All Prefabs
All Scripts
Assets
Scripts
Enemy
Player
Scripts Main SimpleEne... Sniper
Sniper.prefab
Inspector Services

Sniper — Tag: Untagged — Layer: Default

Transform
Position X -0.15 Y 1.053 Z 0
Rotation X 0 Y 0 Z 0
Scale X 3 Y 3 Z 1

Sprite Renderer
Sprite: Knob
Flip: ☐ X ☐ Y
Material: Sprites-Default
Draw Mode: Simple
Sorting Layer: Default
Order in Layer: 0
Mask Interaction: None

Rigidbody 2D
Body Type: Dynamic
Material: None (Physics Material 2D)
Simulated: ✓
Use Auto Mass: ☐
Mass: 1
Linear Drag: 0
Angular Drag: 0.05
Gravity Scale: 0
Collision Detection: Continuous
Sleeping Mode: Start Awake
Interpolate: None

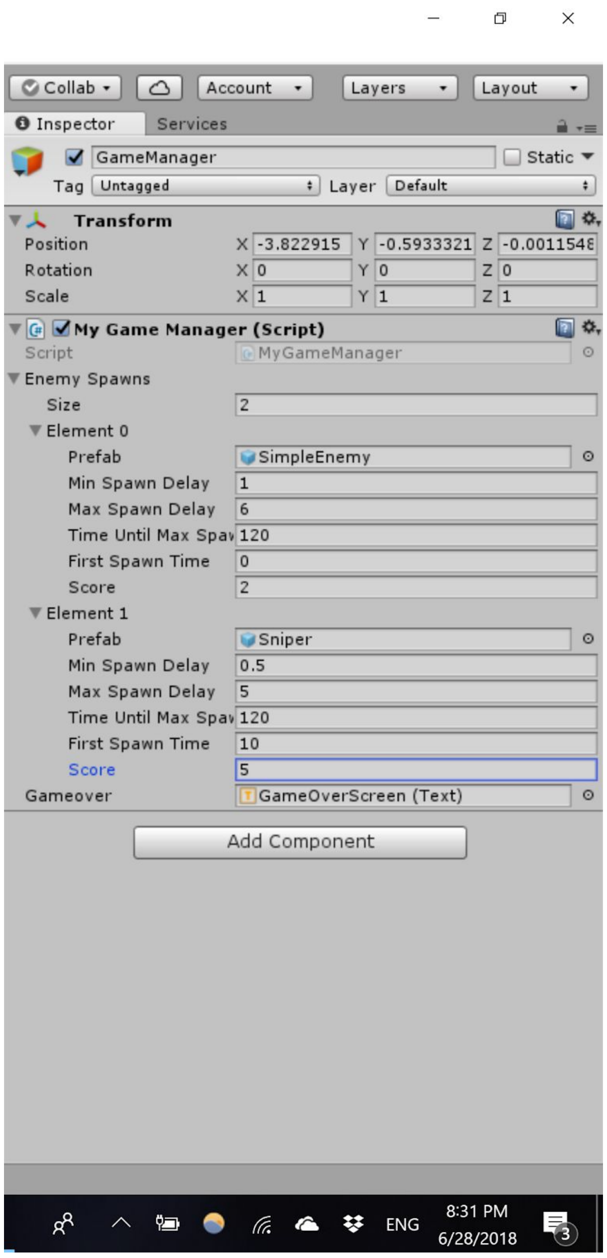
Circle Collider 2D
Material: None (Physics Material 2D)
Is Trigger: ✓
Used By Effector: ☐
Offset X -1.986822 Y 0
Radius: 0.07822818

GameObject (named 'Sniper') references runtime script in scene file. Fixing!

Adding Sniper to the GameManager

Select the *GameManager* object from the *Hierarchy* tab. Edit the *GameManager* component to be as follows:

- Expand *Enemy Spawn* by clicking the arrow on the left
- Change size from 1 to 2
- Expand *Element 1* by clicking the arrow on the left
- Drag the *Sniper Prefab* you created onto the *Prefab* variable (or select it by clicking the circle to the right of where it says prefab)
- Set the remaining variables as follows:



GameManager
Tag Untagged Layer Default

Transform

Position	X	-3.822915	Y	-0.5933321	Z	-0.0011548
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

My Game Manager (Script)

Script	MyGameManager
Enemy Spawns	
Size	2
Element 0	
Prefab	SimpleEnemy
Min Spawn Delay	1
Max Spawn Delay	6
Time Until Max Spav	120
First Spawn Time	0
Score	2
Element 1	
Prefab	Sniper
Min Spawn Delay	0.5
Max Spawn Delay	5
Time Until Max Spav	120
First Spawn Time	10
Score	5
Gameover	GameOverScreen (Text)

For a more detailed description of the variables check the script for the comments

Fixing Bugs

If you try to play the game currently, you may notice...strange behavior among the enemies that are Snipers. They jitter and stutter and seem to break, or they rush the player at the speed of light. So let's fix this!

Go to the "Sniper" Prefab you created, and remove the "Simple Enemy Movement" component. Save the prefab and start the game, and the snipers should now spawn heading initially towards the player, but the they should not follow the player; they should proceed in a straight line.

Be careful when you create scripts, especially when other people are going to be modifying it and or reading it. This is especially troublesome if you modify the same elements, such as movement, because this will lead to unexpected results. Make sure you write comments that thoroughly explain functions and the purpose behind the architecture you create. It is often easy to make a script bloated, and communication is key so two programmers do not create different functions that modify the same variables at various times. A couple of points that I believe are important.

1. Make sure that everyone knows what you are going to use and change

2. Take the time to write good comments that explain functions as well as what variables and components they use, otherwise alot of time will be spent miscommunicating and fixing code others write or your own code.

3. Have clearly defined functions that, if possible, are brief. Use interfaces and do not create one mega function that does everything with one switch statement.

Additional reading for Programmers:
Artists see checkoff below

Take a look at MyGameManager.cs and EnemyData.cs . These scripts combines a few things to make the Inspector for the script more useful. While there are much more complicated ways to modify what the inspector of a script is capable of these are going to be the most useful for you:

- [HideInInspector] Use this when you don't want the inspector to show a variable but you still need it to be public (like in a struct or array)
- Structs - Use these to create convenient groupings of variable names that will stay grouped even in the inspector
  - The [System.Serializable] is required for any struct or array to show up in the inspector. Google this if you are interested in why.
- Arrays/Lists - Use these to make adding new things to your game easier. Instead of having to add a new public variable for each enemy type you can add a new enemy entirely in the inspector!

Take a look at the relationship between *EnemyMovement.cs SimpleEnemyMovement.cs* and *SniperEnemyMovement.cs* . This is a good example of how to utilize *Inheritance* as in 61b.

- The protected keyword comes in handy here
- Functions that you plan on overriding need to be visible to the child (public or protected) as well as meant to be overriden (virtual or abstract)
- When a child class overrides a method you need to use the keyword *override*
- When you override a method it is good practice to call the parent method by using "base.methodName()" *base* refers to the parent class

- Note that we separated Player and Enemy movement entirely because there wasn't really much shared between them in our case

Note how *Health.cs* and *Attack.cs* interact. This is a good example of how to utilize *Composition* which should be a pretty new concept in coding here.

- When an object with an *Attack* script comes into contact with another object it will check to see if that object has a *Health* script and if it does it will call that script's *takeDamage()* function. It also checks if the health is <= 0, so it can tell the GameManager that the game is over.
- *Composition* is this idea of building up behaviors through modular components. An enemy has a movement script and an attack script. These two scripts together make up the behavior of an enemy. In *Inheritance* you build unique things up by adding new features to a parent class, from the top down. In *Composition* you build unique things from the bottom up by assembling different pieces together to get a desired behavior.
- Most games will probably utilize a mix of *Composition* and *Inheritance*. If the abundance of keywords required for *Inheritance* is incredibly daunting and confusing for you, fear not because you can get by on almost entirely *Composition* here

Checkoff:

- For checkoff make the Sniper enemy spawn at 10 seconds (change *First Spawn Time* from 60 to 10)
- The original enemy should still exist and be spawning
- The Sniper should have a unique color
- The Sniper should have a movement pattern different than the regular enemies
- The Player should be able to be hit more than one time before dying
    - Watch out for the clump of enemies following the player, they will do a lot more than 1 damage
- Make sure you fill out the attendance form!

Challenges(Optional):

Please do not hesitate to ask for help if you choose to explore some of these. These are a lot more open ended and it can be easy to get tripped up a weird aspect of Unity.

- Add a new enemy type
    - Straightforward, basically repeat the same steps as the above lab but make a new movement script or a new attacking script
- Make things pretty!
    - Add art assets to the SpriteRenderer
    - Look into particle effects and trail renderers (Will be a later lab on this)
    - Add sound effects! (Will be a later lab on this)
- Add a player attack that can kill enemies

- There are many valid approaches for this (all with their own complications) Here are a handful of starting tips:
- https://docs.unity3d.com/ScriptReference/Physics2D.CircleCast.html
- Currently the enemies have hitboxes that are known as “Triggers” . Any colliders can pass through a Trigger collider and no physics will take place, but a Trigger Event will occur. Think of them as sliding glass door sensors at stores like WalMart.
  - CircleCast will not detect Colliders that are Triggers. You will have to add another collider to the enemies.
- If you use the existing health script you will have to look into ways to make the enemies not attack each other
  - Give the enemies a special tag or add them to a new layer at the top of the inspector
  - Add a public variable in the attack script that says what tag/layer the target has to have in order to be attacked
- To avoid the enemies having physics interactions with their new colliders go to *Edit>Project Settings>Physics 2D* then look at the Layer Collision Matrix. You can specify which layers can collide with which layers