## C# Programming – Classes

## Classes

## As C# is an object orientated programming language, everything in C# is built upon classes.

## A class is a structure, similar to a blueprint, which defines a set of related functions and variables, called methods and members.

## Remember when we looked at types and variables. Well, each of those defined types we looked at are actually a defined class. Each type has a list of functions and properties that we can access. And each variable we created was actually an object of that class’ type

## By knowing how to create our own classes we can create new types of objects, which is really important for game programming. More importantly, we can create multiple instances of the class. This allows us to define the functions and variables for a class once, and then reuse that code over and over.

## A class in Unity is usually defined in a new script with the same name as the class. For example our PlayerControl.cs script is created and only contains our PlayerControl class.

## Let’s take a look at a simple class and break down each part:

1. **using** UnityEngine;
2. **using** System.Collections;
4. **public** **class** PlayerControl : MonoBehaviour {
6. //Prefab to instantiate
7. **public** GameObject ammoPrefab;
8. **int** ammo = 10;
10. **void** FireWeapon(**int** numberOfBullets)
11. {
12. **for** (**int** i=0; i<numberOfBullets; i++) {
14. **if** (ammo >= 0) {
15. Instantiate (ammoPrefab);
16. ammo--;
17. }
18. }
19. }
21. // Update is called once per frame
22. **void** Update () {
24. **if** (Input.GetKeyDown (KeyCode.Space)) {
25. FireWeapon (1);
26. }
27. }
28. }

## Creating Classes

## D:\lachlan.reid\Desktop\2015-08-20_14-47-26.png

## Create a new class by creating a new C# script in the project folder. It should look something like this:

1. **using** UnityEngine;
2. **using** System.Collections;
4. **public** **class** PlayerControl : MonoBehaviour
5. {
6. //Fields, properties, methods and events go here...
8. // Use this for initialization
9. **void** Start () {
11. }
13. // Update is called once per frame
14. **void** Update () {
16. }
17. }

## These lines are automatically written for you when you create a new C# script. Classes also created with two default functions: Start() and Update(). These are created automatically as they are very commonly used functions in Unity, and most classes we create will utilize one, or both, of these functions. Our example script above doesn’t require the Start() function, and so we can delete it.

## The first two lines are using declarations. These import libraries of useful types and functions which we can access.

## The third is the class declaration. First we state whether we want a public or private class, followed by declaring this function as a class. The name that we are assigning to this class is next. This will automatically be set to the name of the file, and cannot be changed. Lastly we set the inheritance of the class. By default this will be MonoBehaviour, which is a Unity specific class. We will talk about class inheritance in another handout.

## Class Members

1. //Prefab to instantiate
2. **public** GameObject ammoPrefab;
3. **int** ammo = 10;

## A class can have a number of members which are unique variables local to every object created of this class’ type. These variables can be set as public or private, and are set to private by default. Public variables can be accessed from other objects and scripts and altered, though this is usually a bad idea. Private variables are only known to the class that houses them. These variables cannot be accessed directly, and usually require a getter/setter function to modify them.

## Any public variable will also appear in the Inspector of the object the script is attached to. This is really useful for easily dragging prefabs and materials from the project into the space provided, as well as testing variables quickly. Public variables are usually not a good idea, and are best to be avoided where possible.

## D:\lachlan.reid\Desktop\2015-08-19_16-34-02.png

## Here we are accessing and mutating both of our class members. You can see that they behave just like other variables.

1. **if** (ammo >= 0) {
2. Instantiate (AmmoPrefab);
3. ammo--;
4. }

## If another script needed to access or mutate a class member, the variable can be accessed by using a period after the object, followed by the member identifier. For example this small script checks if our ammo member is less than 10, and if so, resets the int back to 10.

## NOTE: *This currently would not work with the scripts above. As we have not set ammo to be a public variable, we would need to set up get/set functions to allow the ammo member to be accessed outside of its class.*

1. //Pick up Ammo
2. //Get the playerControl's ammo count and refill it.
3. **if**(playerControl.ammo < 10)
4. playerControl.ammo = 10;

## Class Methods

## A class can also have a number of functions specific to the class’ needs.

## In the playerControl example we have two class methods. The first FireWeapon(), is a new unique function which only exists in this script. It passes through a for loop a number of times, given by its parameter, creating a new ammoPrefab object and altering the ammo member. This method will only be processed when called directly

## The Update() method is a special method that we are inheriting from MonoBehaviour. It automatically runs every frame of our game, whether we want it to or not. Update() methods are really handy for checking input, applying movement, or any other functions that we need to call every frame. Our Update() method is checking for space bar input, and firing our weapon if it was pressed.

1. **void** FireWeapon(**int** numberOfBullets)
2. {
4. **for** (**int** i=0; i<numberOfBullets; i++) {
6. **if** (ammo >= 0) {
7. Instantiate (ammoPrefab);
8. ammo--;
9. }
10. }
11. }
13. // Update is called once per frame
14. **void** Update () {
16. **if** (Input.GetKeyDown (KeyCode.Space)) {
17. FireWeapon (1);
18. }
19. }

## Apart from the Update() or Start() methods (although there are a couple of others), our methods will not run automatically, and can be called at any time during the game. If we are calling the method from within the class just use the function’s identifier, followed by parentheses containing any parameters required.

## In the Update() method shown above, this line is calling the FireWeapon method to be run and passing through an int parameter of 1.

1. FireWeapon (1);

## And if we wanted to call this function from another script, just like a class member, a class method can be called by using its identifier after a period. For example:

1. //Fire the player's weapon three times
2. playerControl.FireWeapon(3);