**Coroutines** – used to make timers/run functions are specific times.

I like to use them to create timers for the AI’s state changes from patrol or idle state.

Coroutines are also useful for creating timers for level completion or different timers for the player to complete certain tasks in a specific amount of time.

To start a Coroutine, you write IEnumerator *coroutineName* (*parameters for coroutine*)

{*code for coroutine.*}

**Example Timer:**

This is the base framework that a coroutine looks like.

Public class CoroutineTimer : MonoBehaviour{

IEnumerator ExampleTimer()

{

}

}

**Add in parameters:**

Public class CoroutineTimer : MonoBehaviour{

IEnumerator ExampleTimer(float timer)

{

While(timer>0)

{

Timer = timer – Time.deltaTime;

Yield return null;

}

}

The above is a basic setup for a coroutine.

Next, you need to Start the Coroutine, using **StartCoroutine()**.

**StartCoroutine(ExampleTimer(4));**

this starts the ExampleTimer coroutine and has a timer set of 4 seconds before it ends.

Coroutines are a useful tool for programmers to use to help time and set up their AI or trigger certain events in an area to happen after the player has entered it.

Knowing how to stop a Coroutine is important to, because if you let it keep going, it can cause problems with the rest of the project, such as

**Lambdas –** quick ways to write functions for code. Able to be used as quick functions. Use the operator =>, stands for goes to. So something like

anonymousDel myDelegate = y => y\*7;

can be considered a lambda expression.

The left side of the lambda specifies the input parameters, while the right side holds the expression or statement block. So the current lambda example above reads left to right as “y goes to y times 7”

Another example of a lambda would be something along the lines of

delegate float randomDel(float a, float b)

randomDel getSmallerFloat = (x,y) =>{if(x<y) return x; else return y;};

print(getSmallerFloat(15,16);

this should print out 15, as it is the smaller float.

Lambdas are useful for writing short simple code, so as to get things done quickly and to test out different things. They can also be used to inject code without needing to write a function.

They are useful for prototyping different things or saving space, but if you aren’t use to writing code in this fashion, it can become difficult to read.

**Delegate –** Also referred to as **Events** or **Callbacks,** delegates can be used to save memory and processing cost for different things. A delegate is like a function pointer in C or C++.

Delegates are used to encapsulate a reference to a function inside of the delegate object. The encapsulated reference can then make reference to the function inside of a different class, without said class ever knowing about it at compile time about which method will be invoked.

For example:

Public delegate void delegateExample(float, int );

delegateExample would be called in the parameters of the method you want to use it in.

The delegate could be used for a function that takes in float and int parameters, so it can be used to call a function deathExample(float x, int y), where the function needs a float for the characters health and an int for the characters’ lives remaining.

**Multi-Threading – (I might need more help in understanding how to do these)**

Multi threading is a way to get multiple different things done at the same time and in a more efficient manner.

Threads in Unity can be called using the System.Threading.

You then can create a Thread variable.

The 2 threads above can be used to call different functions at the same time to run and operate. The threads can be used to get user input, handle background information, and handle multiple streams of input, like if there was a large amount of players or users in an online game.

Worker threads are a type of thread that mostly handles input requests that don’t need to take as many resources as other threads that are running at the same time.