Lab 12 – Multithreading

# Hands-On

In the threading unit, the student will get hands-on experience with starting and stopping multiple threads in a c# program. Although unity will be used to facilitate this, all threading-specific code will be utilized using c#’s System.Threading API.

## Section Two: Functionality

In this section, the student will look at how to pass data to threads using Lambda expressions, how to name threads, the difference between a foreground and background thread, as well as how to set thread priority

### Step One: Lambda?

A lambda expression is an anonymous function, and it is mostly used to create delegates. It’s a method without a declaration (access modifier, return value declaration, name, ect). It is mostly used for convenience. It is a shorthand that allows you to write a method in the same place you are going to use that method. This is especially useful in places where you will only be using the method once, and the method definition is short. It saves you the effort of declaring and writing a separate method to the containing class.

Lambda expression should be short. A complex definition makes the code too difficult to read. Lambda expressions are defined as Parameters => executed Code

For example:

n => n%2 == 1

If you wrote a function to determine if a student was a teenager or not, it would look something like this:

|  |
| --- |
| bool isStudent **(**Student currStudent**)**  **{**  **return** **(**currStudent**.**age **>** 12 **&&** currStudent**.**age **<** 20**);**  **}** |

You can turn this into a lambda expression like:

|  |
| --- |
| currStudent **=>** currStudent**.**age **>** 12 **&&** currStudent**.**age **<** 20 |

Here is an example of a lambda in action:

|  |
| --- |
| **using** UnityEngine**;**  **using** System**.**Collections**;**  **public** class LambdaTest **:** MonoBehaviour **{**  **delegate** bool isTeenager**(**Student student**);**  // Use this for initialization  void Start **()** **{**  Student student1 **=** **new** Student **(**5**);**  Student student2 **=** **new** Student **(**10**);**  Student student3 **=** **new** Student **(**15**);**  Student student4 **=** **new** Student **(**20**);**  Debug**.**Log **(**"Function student1 teenager? : " **+** funcIsATeen **(**student1**));**  Debug**.**Log **(**"Function student2 teenager? : " **+** funcIsATeen **(**student2**));**  Debug**.**Log **(**"Function student3 teenager? : " **+** funcIsATeen **(**student3**));**  Debug**.**Log **(**"Function student4 teenager? : " **+** funcIsATeen **(**student4**));**  isTeenager isATeen **=** s **=>** s**.**age **>** 12 **&&** s**.**age **<** 20**;**  Debug**.**Log **(**"Lambda student1 teenager? : " **+** isATeen **(**student1**));**  Debug**.**Log **(**"Lambda student2 teenager? : " **+** isATeen **(**student2**));**  Debug**.**Log **(**"Lambda student3 teenager? : " **+** isATeen **(**student3**));**  Debug**.**Log **(**"Lambda student4 teenager? : " **+** isATeen **(**student4**));**  **}**  bool funcIsATeen**(**Student currStudent**)**  **{**  **return** **(**currStudent**.**age **>** 12 **&&** currStudent**.**age **<** 20**);**  **}**    // Update is called once per frame  void Update **()** **{**    **}**  class Student  **{**  **public** int age**;**  **public** Student**()**  **{**  age **=** 10**;**  **}**  **public** Student**(**int age**)**  **{**  **this.**age **=** age**;**  **}**  **}**  **}** |

In this example, a student class is declared that has a public integer age.

In the main class, a function is declared as described above. A Student is passed into the function, and the function returns if the students age is over twelve and under twenty (a teenager).

A delegate is created that returns a bool, and is named isTeenager. It takes a student as a parameter.

Inside of the Start() function, the delegate is created with a lambda expression:

isTeenager isATeen = s => s.age > 12 && s.age < 20;

isTeenager isATeen is simply the delegate declaration, isATeen is a “variable” of type isTeenager. The “variable” is assigned to the lambda expression:

s **=>** s**.**age **>** 12 **&&** s**.**age **<** 20

S is the parameter that will be passed into the lambda function. This lambda function is saying

Function**(**s**)**

**{**

Return s**.**age **>** 12 **&&** s**.**age **<**20

**}**

How does the lambda expression know that s is a student? In this particular case it is because of the delegate. The lambda expression is returning into the delete function isATeen, which takes a Student as a parameter.

When you call isATeen(student1) you are calling the lambda expression, and using student1 as the parameter (named s).

If you need multiple parameters with a lambda expression, you can use parenthesis:

**(**s**,** youngAge**)** **=>** s**.**Age **>=** youngAge**;**

You can also give a type of each parameter:

**(**Student s**,** int youngAge**)** **=>** s**.**Age **>=** youngAge

You can have a lambda expression with no parameters;

**()** **=>** Debug**.**Log**(**“This **is** a lambda expression with no parameters”**)**

And lambda’s with multiple statements in the body:

**(**s**,** youngAge**)** **=>**

**{**

Debug**.**log**(**"I am doing multiple things!"**);**

Return s**.**age **>=** youngAge**;**

**}**

As well as have local variables within a lambda:

s =>

{

int youngAge = 18;

Debug.log("I am doing multiple things!");

Return s.age >= youngAge;

}

### Step Two: Magic!

The easiest way to pass arguments to a thread’s target method is to use a lambda expression that calls the method with the desired arguments:

|  |
| --- |
| **using** UnityEngine**;**  **using** System**.**Collections**;**  **using** System**.**Threading**;**  **public** class Arguments **:** MonoBehaviour **{**  void Start**()**  **{**  Thread thread **=** **new** Thread **(()** **=>** DisplayMessage**(**"Hello from the thread!"**));**  thread**.**Start **();**  **}**  void DisplayMessage**(**string message**)**  **{**  Debug**.**Log **(**message**);**  **}**  **}** |



With this approach, you can pass in any number of arguments to the method. You can even wrap the entire implementation into a multi-statement lambda:

|  |
| --- |
| void Start**()**  **{**  **new** Thread **(** **()** **=>**  **{**  Debug**.**Log **(**"I'm running on another thread!"**);**  Debug**.**Log **(**"This is so cool!"**);**  **}**  **).**Start**();**  **}** |

In this statement you create a new thread (since this will be an anonymous lambda thread you don’t need to declare it as a thread).

As the “method” the thread will call, you then start the lambda expression with no parameters – () =>

Then you start a multi-statement lambda the same way you do a normal method – { }

You do what you need to do inside of the lambda expression -- Debug.logs

You close the thread’s parameter and call the Start function

### Step Three: Captured!

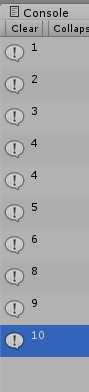
There is one thing you need to be extra wary about when doing this, however. C# has captured variables, which you should be aware of.

Consider the following:

|  |
| --- |
| void Start**()**  **{**  **for(**int i **=** 0**;** i **<** 10**;** i**++)**  **{**  **new** Thread**(** **()=>** Debug**.**Log **(**i**)).**Start**();**  **}**  **}** |

When this is printed, you would expect it to print out 1,2,3,4,5,6,7,8,9,10. You might even expect it to be out of order because of how threads run.

However, something like this may occur (you may have to run this code multiple time to see this happen)



Not only is it out of order, but notice that 4 is printed twice and 7 is never printed at all. This is because c# has something called captured variables. This has nothing to do with multi-threading at all (the out of order does, but printing 4 twice and 7 not at all does not). This has everything to do with the way c# handles capturing variables.

This problem occurs because the I variable refers to the same memory location throughout the entire loop’s lifetime. Therefore, each thread calls Debug.Log on a variable whose value can change as it is executing!

If you wish to learn more about captured variables in c# you can reference here:

<http://blogs.msdn.com/b/matt/archive/2008/03/01/understanding-variable-capturing-in-c.aspx>

And here:

<http://jeremybytes.blogspot.com/2015/03/lambda-expressions-captured-variables.html>

As far as you are concerned in related to multi-threading, the solution to this is to use a temporary variable like so:

|  |
| --- |
| void Start**()**  **{**  **for(**int i **=** 0**;** i **<** 10**;** i**++)**  **{**  int temp **=** i**;**  **new** Thread**(** **()=>** Debug**.**Log **(**temp**)).**Start**();**  **}**  **}** |

This causes variable temp to be local to each loop iteration. Therefore, each thread will capture a different memory location and there is no problem.

### Step Four: Names

Each thread has a Name property that you can set for the benefit of debugging. This is particularly useful in Visual Studio, since the thread’s name is displayed in the Threads Window and Debug Location toolbar. You can set a thread’s name **ONLY ONCE**; attempts to change it later will throw an exception.

The static Thread.CurrentThread property gives you the currently executing thread. In the following example, the main threads name is set:

|  |
| --- |
| void Start**()**  **{**  Thread**.**CurrentThread**.**Name **=** "main"**;**  Thread worker **=** **new** Thread **(**Go**);**  worker**.**Name **=** "worker"**;**  worker**.**Start**();**  Go**();**  **}**  void Go**()**  **{**  Debug**.**Log **(**"Hello from " **+** Thread**.**CurrentThread**.**Name**);**  **}** |

### Step Five: Foreground vs Background

By default, threads you create explicitly are foreground threads. Foreground threads keep the application alive for as long as any one of them is running, whereas background threads do not. Once all foreground threads finish, the application ends, and any background threads still running abruptly terminate.

Note: A thread’s foreground/background status has no relation to its priority or allocation of execution time.

You can query or change a thread’s background status using its IsBackground property. Here’s an example:

|  |
| --- |
| void Start**()**  **{**  Thread worker **=** **new** Thread **(**Go**);**  Debug**.**Log **(**"is worker background? " **+** worker**.**IsBackground**);**  worker.isBackground = true;  Debug**.**Log **(**"is worker background? " **+** worker**.**IsBackground**);**  worker**.**Start **();**  **}** |

### Post-Lab

1. Explain what is displayed in the console and why it is displayed as it is:

|  |
| --- |
| void Start**()**  **{**  string text **=** "t1"**;**  Thread t1 **=** **new** Thread**(** **()** **=>** Debug**.**Log **(**text**));**  text **=** "t2"**;**  Thread t2 **=** **new** Thread **(()** **=>** Debug**.**Log **(**text**));**  t1**.**Start **();**  t2**.**Start **();**  **}** |

### On Your Own!

Create a new script, for movement.

Create three cubes, and place the scripts on each cube.

Create another script to hold health. Create a sphere and add all three scripts to the sphere.

Movement Script:

Inside the movement script, translate the object and change direction that you are translating to every 5 seconds.

The translation should be done in a THREAD

Inside the movement script, if the cube hits the sphere it should deal 1 point of damage.

Health Script;

Holds health in a public variable

Should be set to a small number like 5 or 10

Displays health on screen