**Assignment Description**

In previous modules and courses, we implemented sub-optimal communications between our objects. In one of our games, our fish needed to know about the existence of the HUD; that doesn't really keep our objects as independent from each other as we'd like. One excellent solution to this problem is to implement an event system, so in this assignment you'll be implementing a simple event system.

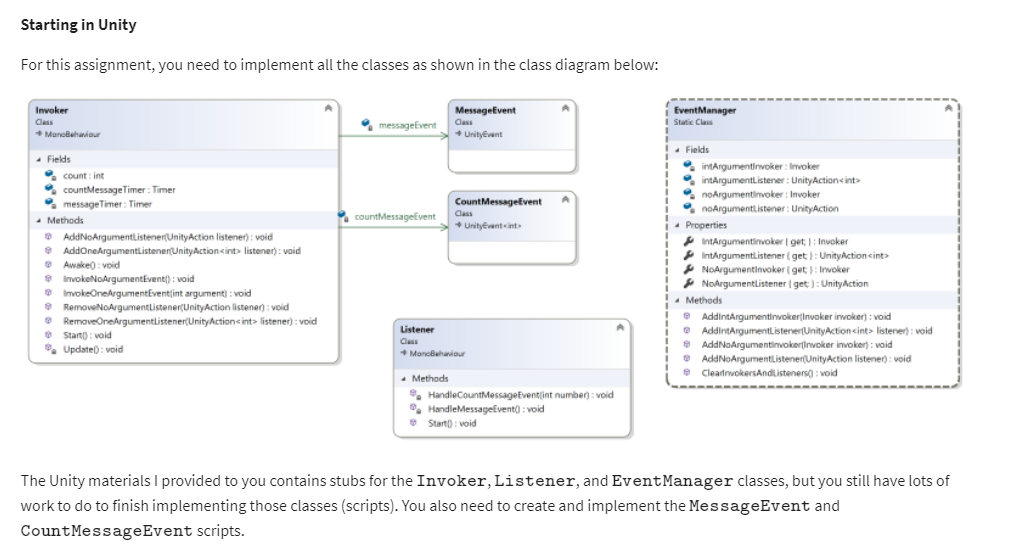
More specifically, in this assignment, you'll implement MessageEvent, CountMessageEvent, Invoker, Listener, and EventManager classes.

**Why do we care?**

Almost every non-trivial game -- at least those that care about good software design -- uses some form of event system. Getting practice with the simple event system in this assignment will help you implement more complicated event systems in your games as you need them.

The contents of the zip file is different from what you've seen in the past, because it contains both Unity materials and a console app project for the autograder.

Unzip the Unity materials zip file somewhere on your computer and open the Unity project in Unity.

****

The Unity materials I provided to you contains stubs for the \tt{Invoker}Invoker, \tt{Listener}Listener, and \tt{EventManager}EventManager classes, but you still have lots of work to do to finish implementing those classes (scripts). You also need to create and implement the \tt{MessageEvent}MessageEvent and \tt{CountMessageEvent}CountMessageEvent scripts.

In the class diagram above, the arrow (called an association) from the \tt{Invoker}Invoker class to the \tt{MessageEvent}MessageEvent class indicates that the \tt{Invoker}Invoker class has a \tt{MessageEvent}MessageEvent field named \tt{messageEvent}messageEvent.

**Note:** You may find that some of the work you did in the Is There Anybody Out There? exercise is useful as you complete this assignment.

Create a \tt{MessageEvent}MessageEvent script that inherits from \tt{UnityEvent}UnityEvent and create a \tt{CountMessageEvent}CountMessageEvent script that inherits from \tt{UnityEvent<int>}UnityEvent<int>. Remember, \tt{UnityEvent}UnityEvent and \tt{UnityEvent<>}UnityEvent<> are in the \tt{UnityEngine.Events}UnityEngine.Events namespace.

In the \tt{EventManager}EventManager script I provided, add code to the \tt{AddNoArgumentInvoker}AddNoArgumentInvoker and \tt{AddNoArgumentListener}AddNoArgumentListener methods to make the appropriate modifications to the provided \tt{noArgumentInvoker}noArgumentInvoker and \tt{noArgumentListener}noArgumentListener fields.

In the \tt{Invoker}Invoker script I provided, declare fields for a \tt{Timer}Timer object and a \tt{MessageEvent}MessageEvent object. Add code to the \tt{Awake}Awake method to create an instance of the \tt{MessageEvent}MessageEvent object in that field. Add code to the \tt{AddNoArgumentListener}AddNoArgumentListener method that adds a \tt{UnityAction}UnityAction delegate as a listener for the \tt{MessageEvent}MessageEvent; remember, \tt{UnityAction}UnityAction is in the \tt{UnityEngine.Events}UnityEngine.Events namespace. Add code to the \tt{Start}Start method to add the \tt{Invoker}Invoker as a no argument invoker in the\tt{EventManager}EventManager, add a Timer component, set its duration to 1 second, and run the timer. Add code to the \tt{Update}Update method to check if the timer has finished; if it has, call the \tt{InvokeNoArgumentEvent}InvokeNoArgumentEvent method and run the timer again. Add code to the \tt{InvokeNoArgumentEvent}InvokeNoArgumentEvent method to invoke the \tt{MessageEvent}MessageEvent.

In the Unity editor, attach the Invoker script to the main camera.

In the \tt{Listener}Listener script I provided, in the \tt{Start}Start method, add code to add the \tt{HandleMessageEvent}HandleMessageEvent method as a no argument listener in the\tt{EventManager}EventManager.

In the Unity editor, attach the Listener script to the main camera.

Run the game. The message should be printed in the Console window every second.

Follow similar steps to add event handling for the one int argument event. I just had the \tt{Invoker}Invoker script increment the value it includes each time it invokes the event, but do whatever you want to provide a different integer each time.

**Moving Over to the Console App**

Now that you have your code working in Unity, it's time to move it over into the console app and make sure it will all work fine in the automated grader. Unzip the autograder materials zip file somewhere on your computer and open the Visual Studio solution.

The Visual Studio project I provided to you has stubs for the \tt{Invoker}Invoker, \tt{Listener}Listener, and \tt{EventManager}EventManager classes; you'll replace those files with the ones you developed in Unity soon. The project I provided also includes a fully-implemented \tt{Program}Program class that tests the classes you've implemented.

**Note:** The autograder materials I've provided will compile, but if you run the project as it was provided it will crash. You need to add your classes to the project before you'll be able to run the code without crashing.

Copy the following files, AND ONLY THE FOLLOWING FILES, into the same folder as the Program.cs file in the console app project: MessageEvent.cs, CountMessageEvent.cs, Invoker.cs, Listener.cs, and EventManager.cs. Confirm that you want to overwrite the Invoker.cs, Listener.cs, and EventManager.cs files when prompted to do so. Do NOT copy the Timer script over; doing that will break the automated grader.

You should now be able to build and run the solution as discussed below.

**Required Output Format**

The Program.cs and the stub I provided for the \tt{Listener}Listener class handle all the output correctly. The output is simply Passed or FAILED for each test case that's selected by the user input.

**Running Your Code**

Because of the code I included to work with the automated grader on Coursera, when you run your program the command prompt window will open and it will sit there doing nothing. To make your code run, type in a test case number (1 through 10, inclusive) and press the <Enter> key; your code should then run so you can check your output.

For example, your input could be

\tt{3}3

to run the third test case.

You can actually run your code again if you want to by typing in a test case number and pressing the <Enter> key again. When you’re ready to stop running your code, type q (for quit).

Here's what running the code multiple times with different inputs should look like (though none of your test cases should fail!). The first line is the test case number, the second line is your output line, and so on:

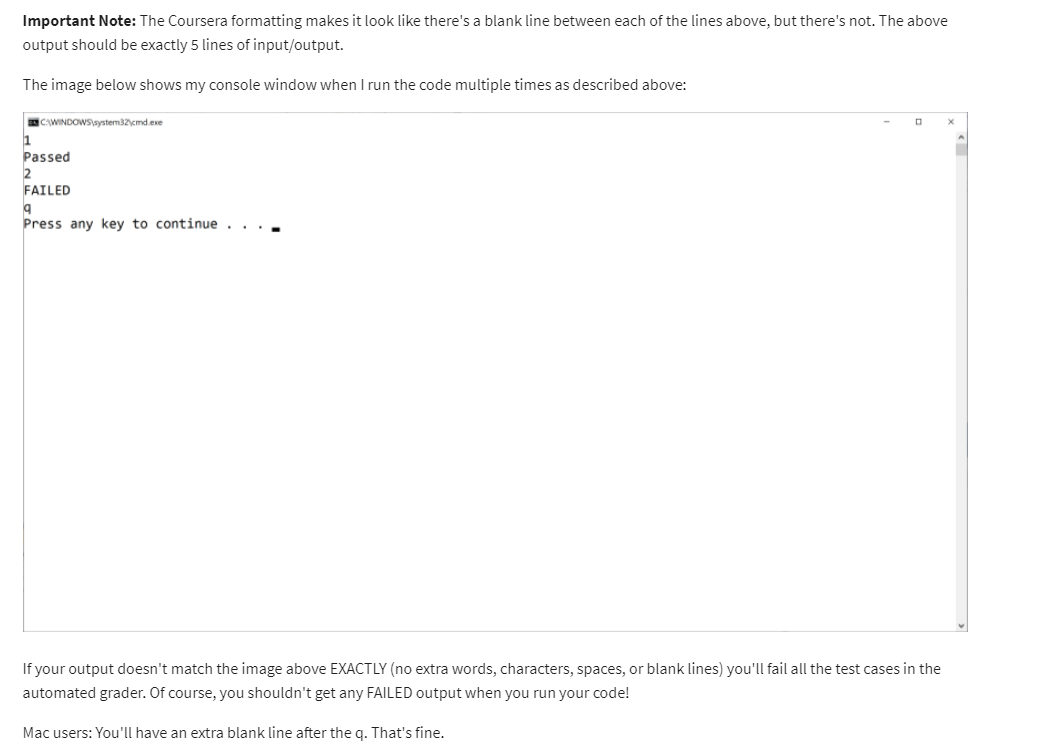
**1**

**Passed**

**2**

**FAILED**

**q**



**Test Case Inputs**

The automated grader uses the set of test cases that are built into the Program.cs file I provided to you.