CS 376 Exercise 1:
Making a simpler serializer

Overview

In this assignment, you'll write a serializer for a simplified subset of the Unity object system. We're serializing to a text file in a format similar to JSON, so you'll be able to read it.

We've provided:

- Simplified versions of GameObject, Component, Transform, CircleCollider2D, and SpriteRenderer, called FakeGameObject, FakeComponent, etc. They'll give you a sense of how to the different objects fit together.
- A Serializer and Deserializer class. To serialize an object *o*, one creates a Serializer and then calls its WriteObject method on *o*. Since *o* likely has pointers to other objects, we're really serializing a whole tree or graph of objects. Similarly, to deserialize, you make a Deserializer and then call its ReadObject method. Again, this is likely to end up reading in a whole graph of objects.
    - There's a built-in static method, Serializer.Serialize() that does those steps for you. It takes an object and gives you back the serialized string.
    - There's a built-in static method, Deserializer.Deserialize() that takes a serialized string and gives you back the object, or rather an equivalent object.
- A set of useful methods in a static class called Utilities that lets you
    - Find out all the fields in an object, the names (strings) of those fields and their values
    - Take an object, the name of a field, and a new value for the field, and update that field to have that value
    - Create a new object given the name (a string) of its class.
- Code to serialize and deserialize
    - Primitive types like int, float, and string
    - Sequence types like arrays and lists
- Unit tests that run inside Unity and log their results to the unity console window

You need to fill in the missing code in the file StudentCode.cs to:

- Write a complex object (a class instance). This needs to write out a serial number for the object, and if it hasn't already been written out, its type, and all its fields.
- Write an arbitrary object. This needs to check the object to see what type it is and call the appropriate code to write it.
- Read a complex object. This needs to read the serial number and possibly the data for the object if that object hasn't already been read in.
- Read an arbitrary object. This needs to read the first character of the description, decide what kind of object it's looking at, and call the appropriate code to read it.

Serialization format

The format we'll use is a variant of JSON. It looks like this:

- Whitespace is ignored, so you can generate newlines as you like
- Floats and ints print as normal decimal numbers: 1.5, 1, etc.
- Strings print with double quotes: "this is a string"
- Booleans print as “True” or “False”
- Lists and arrays print with square brackets. Inside the brackets are the elements of the list, separated by commas
- Complex objects containing their own fields which have not already been serialzed, should print as:

  #*id* { type: *typename*, *fieldname*: *value*, … }

  where
  - *id* is the serial number assigned to this object
  - *typename* is the name of the type of the object. You can get the type of an object by calling its GetType() method, and you can get a type’s name by looking at its .Name field.
  - *Fieldname*s are the names of the different fields in the object and *value*s are their respective values. We’ve given you a method that will tell you what all the field names and values are for a given object
- Complex objects that have already been serialized should be printed just as the #*id*, since all the other data has already been put into the serialization stream.

Example serialized data structure

Here’s the serialization produced by my solution set for the GameObject graph used in the tests. That graph has a gameobject, parent, which two gameobjects in it, the first of which has its own child object inside it. All gameobjects have transform components, but the two children of the parent game object also have other components in them. Here’s the serialization:

```
#0{
    type: "FakeGameObject",
    name: "test",
    components: [
        #1{
            type: "FakeTransform",
            X: 0,
            Y: 0,
            parent: null,
            children: [
                #2{
                    type: "FakeTransform",
                    X: 100,
                    Y: 100,
                    parent: #1,
                    children: [
                        #3{
                            type: "FakeTransform",
                            X: 0,
                            Y: 0,
                            parent: #2,
```

```
                    children: [ ],
                    gameObject: #4{
                        type: "FakeGameObject",
                        name: "child 2",
                        components: [
                            #3
                         ]
                    }
                }
             ],
            gameObject: #5{
                type: "FakeGameObject",
                name: "child 1",
                components: [
                    #2, #6{
                        type: "FakeCircleCollider2D",
                        Radius: 10,
                        gameObject: #5
                    }, #7{
                        type: "FakeSpriteRenderer",
                        FileName: "circle.jpg",
                        gameObject: #5
                    }
                 ]
            }
        }, #8{
            type: "FakeTransform",
            X: 500,
            Y: 550,
            parent: #1,
            children: [ ],
            gameObject: #9{
                type: "FakeGameObject",
                name: "child 3",
                components: [
                    #8, #10{
                        type: "FakeCircleCollider2D",
                        Radius: 200,
                        gameObject: #9
                    }, #11{
                        type: "FakeSpriteRenderer",
                        FileName: "circle.jpg",
                        gameObject: #9
                    }
                 ]
            }
        }
     ],
    gameObject: #0
}
```

```
        ]
}
```

Notice that the serialization starts with “#0”, meaning “here comes object #0!” and ends with “gameObject: #0”, which means “the value of the gameObject field of this object is object #0.” Since object 0 has already been serialized, we don’t have to include another copy of it here (and shouldn’t!).

Running the game

This is only a game in the sense that it’s running under unity. But ignore that fact and open the file Assets/Scenes/SampleScene.unity. This should launch unity and open the level that has the necessary components in it.

Hit the run button. It should print out a message in the Console window about a NotImplementedException. That’s fine; you haven’t implemented any of the code yet. The tests it’s running are in Driver.cs, so you can look there if you want to see what the exact test code is or if you want to put a breakpoint on the test while you’re debugging your code below.

Getting started

Reading other people’s code is one of the most important skills for you as a programmer. So begin by reading all the code we’ve provided. You can do this by choosing “Open C# Project” from the Assets menu in Unity. This will launch Visual Studio on the source files in the game. Do this, and then read over all the .cs files.

Your goal here isn’t to memorize the code, just to see what classes and methods are there so that when you need a method to do something we’ve already implemented, you know where to go looking for it.

IMPORTANT: the code in Utilities.cs uses the reflection features of C#, which we haven’t talked about, and which aren’t central to this class. So while I’m happy to answer any questions about it, don’t feel you need to understand how those methods work. You just need to understand what they do.

Writing the serializer

Note: If you want to add any additional methods or fields to the Serializer class, include them inside StudentCode.cs inside the area that starts saying “public partial class Serializer { … }”. Any code you write for the serializer should be between those { }s. **Do not modify any files** other than StudentCode.cs. If you do so, your peer reviewers probably won’t be able to run your code.

Writing simple objects

Now go to the WriteObject() method in StudentCode.cs. This method actually “inside” the Serializer class, so you can access any of the methods and fields in Serializer.cs from that method. WriteObject is mostly doing a case analysis of the different kinds of objects that might be passed to it. Fill in the code for each of those cases. We’ve put in lines that say “throw new NotImplementedException();” everywhere you need to add code. Replace these lines with the right code to write out the object o in the correct format. Note that strings should be written with " marks before and after them.

After you fill in each case in WriteObject(), try saving your file and running the game and make sure that the associated test has switched from "Failed" to "Succeeded". If you run into problems, jump down to "using the debugger", below.

Writing complex objects

Once you finish filling in WriteObject(), you'll need to fill in WriteComplexObject(). This is the method that gets called to serialize objects that have fields in them. You will need to assign the object a serial number, if you haven't already, and remember that serial number in a hash table (use the Dictionary<object,int> data type; search for "C# Dictionary class" for documentation). If there's already a serial number assigned to the object, then you've already written the object once in this serialization, so just write out a # followed by the number. If you haven't already assigned a number, assign one, remember it, write out # followed by the number, and the write { }s with the type and fields written inside, separated by commas.

Using the debugger

You can breakpoint your code and do the usual debuggery kinds of things by pressing the "Attach to Unity" button in Visual Studio. That will put the unity editor, and therefore you game, under the control of VS. You can set breakpoints in your code in VS and Unity will stop when it gets to them. You can then do all the usual things you do in a debugger: look at the call stack (find the Call Stack window), continue or single step (see the Debug menu), or look at the values of variables (in the locals and auto windows).

Logging debugging data

You can also print your own messages to Unity's console window by calling Debug.Log(). This should be used sparingly, or you will get lost in hundreds of messages.

Writing the Deserializer

Now you'll do the same thing but for the deserializer. We won't make you fill in the ReadObject() method, but you should fill in the blanks in ReadComplexObject(). Again, save your file and try the game to see if your tests are succeeding.

Note: If you want to add any additional methods or fields to the Deserializer class, include them inside StudentCode.cs inside the area that starts saying "public partial class Deserializer { … }". Any code you write for the deserializer should be between those { }s.

If the tests print that your score is 100, then you're done!

Turning it in

For most assignments, you'll turn in your full Unity project. But for this assignment, just turn in your StudentCode.cs file. Upload it to Canvas.