CS 376 Exercise 1:
Making a simpler serializer

Note

This is probably the most difficult of the assignments in the class. Don't leave it to the last minute.

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`. 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 Visual Studio and display their status

You need to fill in the missing code in the code in `Serialization/Serializer.cs` and `Serialization/Deserializer.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: "GameObject",
    name: "test",
    components: [
        #1{
            type: "Transform",
            X: 0,
            Y: 0,
            parent: null,
            children: [
                #2{
                    type: "Transform",
                    X: 100,
                    Y: 100,
    parent: #1,
    children: [
        #3{
            type: "Transform",
            X: 0,
            Y: 0,
            parent: #2,
            children: [ ],
            gameObject: #4{
                type: "GameObject",
                name: "child 2",
                components: [
                    #3
                 ]
            }
        }
     ],
    gameObject: #5{
        type: "GameObject",
        name: "child 1",
        components: [
            #2, #6{
                type: "CircleCollider2D",
                Radius: 10,
                gameObject: #5
            }, #7{
                type: "SpriteRenderer",
                FileName: "circle.jpg",
                gameObject: #5
            }
         ]
    }
}, #8{
    type: "Transform",
    X: 500,
    Y: 550,
    parent: #1,
    children: [ ],
    gameObject: #9{
        type: "GameObject",
        name: "child 3",
        components: [
            #8, #10{
                type: "CircleCollider2D",
                Radius: 200,
                gameObject: #9
            }, #11{
                type: "SpriteRenderer",
                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!).

Unit Testing

This assignment does not run inside of Unity. It's a standalone C# project, which means it can use the standard C# unit testing framework. We've included a set of unit tests, so you don't have to write your own.

When you start the assignment most or all of the tests will fail. As you implement parts of the assignment, more tests will succeed. When all the tests succeed, you're done.

Preparation

Installing .NET 9

Installing Rider will *probably* install .NET 9 for you. If you try to open the assignment in Rider and it says it can't open the projects, then install .NET 9 here:

Download .NET 9.0 (Linux, macOS, and Windows) | .NET

Grab the latest version. If you have a mac, choose macos: arm64. If you have a windows machine, choose windows: x64, unless you have one of the really new machines that uses arm64.

Read the Rider docs

You don't need to know a lot about Rider to do the assignment, but make sure you did the reading we told you to do (see the schedule for week 1 on Canvas).

Getting started

Start by doing a pull on GitHub to make sure you have the most recent version of the repo. Then open Rider, choose File>Open>Open … and choose the file `Serializer.sln`. This is the "solution" file. It contains two "projects", the serializer itself, called "Serializer", and the unit tests, called "SerializerTests". Inside of Serializer, are two folders, `FakeUnity`, which has a classes that implement fake versions of some of the important Unity classes, like GameObject, Transform, and Component, and Serialization, which has the code for the serializer and deserializer.

Familiarizing yourself with the code

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. 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.

Running unit tests

Go to the menu and choose **Tests>Run all tests from solution**. This will pop up the test window, compile the code, and run the tests. You'll see a lot of failed tests.

Writing the serializer

You'll be filling in methods in the files Serializer.cs and Deserializer.cs.

Important: do not modify any of the other files. When we test your code, we will only use your Serializer.cs and Deserializer.cs files. So if they depend on changes you made to other files, your code will fail when it is being graded.

Writing simple objects

Now go to the `WriteObject()` method in Serializer.cs. `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("Fill me in!");`" 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 rerunning the tests, and you should see them gradually start passing. If you did this part correctly, you should find that SerializeBool, SerializeFloat, SerializeInt, SerializeNull, and SerializeString should all work. If not, select the broken test in the test window, and press the bug icon to the left to run the test in debug mode. This will allow you to run it with breakpoints, single-stepping, and so on. It will also cause it to stop if it hits an exception.

Debugging hints

If you got an exception inside of your own code, then debug that as you would any other code.

If your code finished executing and returns a serialization but it was the wrong serialization, then it is probably dying in the method `TestSerialize` in the file SerializerTests/FakeUnity/Tests.cs. It checks the serialization your code generated against the serialization that my reference implementation generated. Look at the values of the variables `serialization` (what your code generated) and `expectedSerialization` (what it was supposed to generate) to see why the test failed.

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).

When writing a complex object, check if there's already a serial number assigned to the object.

- If so, 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.

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.

Important: If you want to add any additional methods or fields to the Deserializer class, include them inside Deserializer.cs.

Debugging hints

Again, if something isn't working, it either crashed in your code, or else your code finished executing, returned a deserialized object graph, and then that was rejected by the test.

Once again, if it died in your code, you mostly want to just debug as you would debug any other code. However, note that if the test either got a StackOverflowException or, god forbid, ran out of memory (heap overflow), then that almost certainly means you have an infinite recursion and that recursion almost certainly involves going through ReadComplexObject. So set a breakpoint on ReadComplexObject and see what's happening.

If your code didn't crash, but its output was rejected by the testing code, then things are a little more complicated than for the serializer. The output of the serializer was just a string, so it was easy to test: run the serializer, get the string, see if it's the right string. The output of the deserializer is an object graph. So each test method has a whole set of Assert statements checking different properties of the object graph to make sure it's the expected graph. Those are different for each test, so you have to look at the different called `Assert.AreEqual`, `Assert.IsInstanceOfType`, etc. to see why the test isn't working.

Turning it in

When you're finished, save all your files, rerun the tests to make sure they work, and make sure your code compiles without issuing warnings. Once everything works smoothly, make a zip file consisting of just your Serializer.cs and Deserializer.cs files and upload it to Canvas.