School of Science, Computing and Engineering Technologies

Object Oriented Programming

Credit Task 5.3: Drawing Program — Saving and Loading

Overview

Drawing programs have a natural affinity with object oriented design and programming, with easy to see roles and functionality. In this task you will start to create an object oriented drawing program.

Purpose:	See how to use interact with the file system within an inheritance structure.
Task:	Extend the shape drawing program to save and load drawings.
Time:	Aim to complete this task by the start of week 7

Submission Details

You must submit the following files, formatted using formatmytask.com:

- Program source code
- Screenshot of program execution

Note: Once you have made a submission, this task requires you to have a discussion with your tutor in your lab or at the help desk before it can be signed off as Complete.

SWINBURNE UNIVERSITY OF TECHNOLOGY

Instructions

This task continues the Shape Drawer from the previous topic, adding the ability to save and load the drawings.

Text-based File IO is similar to reading and writing values from the Terminal, but involves the extra steps for opening and closing the file. The strategy for saving a Drawing will involve a **Save** method in the Drawing class, which will then use a **SaveTo** method from the Shape class to allow the different shapes to write their values to the file.

1. Open your **Shape Drawing** solution.

The **Stream Reader** and **Stream Writer** classes provide features to read and write lines of text from the Terminal. By default, we cannot easily read integers or floats, and we cannot read or write **SplashKit** colours. So let's fix that. For reading integers and floats we can use the **Convert** class. For handling colours we will read and write the colour's RGB values and use **SplashKit**'s **Color.RGBColor** to convert the RGB values back to a **Color**. To avoid duplicating this code what we really want is for **Stream Reader** to have the ability to **Read Integer**, **Read Single** and **Read Color**, and we want **Stream Writer** to have the ability to **Write Color**. We can use C# **extension methods** to add new features to the **Stream Reader** and **Stream Writer** classes.

2. Create a new **Extension Methods** class using the following code.

```
using System;
using System.IO;
using SplashKitSDK;

namespace MyGame
{
  public static class ExtensionMethods
  {
    public static int ReadInteger(this StreamReader reader)
    {
        return Convert.ToInt32(reader.ReadLine ());
    }
     public static float ReadSingle(this StreamReader reader)
    {
        return Convert.ToSingle(reader.ReadLine ());
    }
     public static Color ReadColor(this StreamReader reader)
    {
        return Color.RGBColor(reader.ReadSingle(), reader.ReadSingle(),
          reader.ReadSingle());
    }
     public static void WriteColor(this StreamWriter writer, Color clr)
    {
        writer.WriteLine("{0}\n{1}\n{2}", clr.R, clr.G, clr.B);
    }
  }
}
```

Note: The **this StreamReader** parameter indicates that this is an **extension method**. It is added to the **Stream Reader.** Note the class is also static.

3. Switch to the **Drawing** class.
4. Add a new **Save** method using the following pseudocode.

```
Method Save
---------------------
Parameters:
 - filename: String path to the file to save
---------------------
Local Variables
 - writer: StreamWriter
 - s: Shape
---------------------
 1: Assign writer, a new StreamWriter passing in filename

 2: Tell writer to WriteColor, passing in Background
 3: Tell writer to WriteLine, passing in ShapeCount

 4: For each Shape s in _shapes
 5:      Tell s to SaveTo writer

 6: Tell writer to Close
```

This code will output the background color as three floating point numbers (RGB), which can then be used to recreate the color using Color's **RGBColor** function when the file is loaded. Following this it outputs the number of shapes, so that when the file is read back in we can use that number to determine how many shapes need to be read in.

Note: You will need to use the System.IO namespace.

5. Switch to the **Shape** class, and add a **virtual SaveTo** method. This method will save the details from Shape to file, but will be overridden by child classes to add extra details.

```
Virtual Method: SaveTo
---------------------
Parameters:
 - writer: The StreamWriter to save to
---------------------
 1: Tell writer to WriteColor, passing in Color
 2: Tell writer to WriteLine, passing in X
 3: Tell writer to WriteLine, passing in Y
```

Note: You could also save the selected property. In this case we have left it out so it will default to false when shapes are loaded.

6. Switch to the **MyRectangle** class, and **override** the **SaveTo** method.

```
Override Method: SaveTo
---------------------
Parameters:
 - writer: The StreamWriter to save to
---------------------
 1: Tell writer to WriteLine, passing in "Rectangle"
 2: Tell base to Save To writer
 3: Tell writer to WriteLine, passing in Width
 4: Tell writer to WriteLine, passing in Height
```

Initially we are writing out "Rectangle" so that when the file is loaded we can work out what kind of class to create. After this we save out the details from the **base** class. This will call **Save To** from the **Shape** class, saving the color, X and Y values. Lastly the Width and Height parameters are written to the file.

7. Switch to the **MyCircle** class, and **override** the **SaveTo** method.
8. Switch to the **MyLine** class, and **override** the **SaveTo** method.

Hint: The pattern will be the same in Circle and Line.

9. Switch to **Program** and add code so that pressing the **S** key will save the Drawing to your Desktop in a file names TestDrawing.txt.

Tip: Hard code the path to your Desktop for the moment - or other folder if you want.

10. Compile and run your program. Add some shapes and then save the file.
11. Open your TestDrawing.txt file and review the contents. The file should appear something like:

```
1
1
1
7
Circle
0
0
1
135
157
50
Rectangle
0.5
0
0.5
326
210
100
100
Rectangle
…
```

Match this with the code in your Save methods. The **1, 1, 1** is the RGB values of the background color, there are **7** shapes, the first is a **Circle**. Notice the Circle saves its color(RGB), x, y, and then radius, whereas Rectangle saves its color(RGB), x, y, width and height.

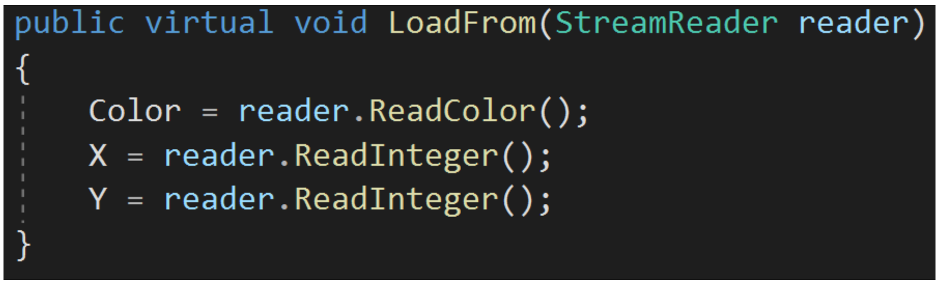
Now we can add code to load our drawing.

12. Switch to the **Drawing** class and create a public **Load** method in Drawing.

```
Method: Load
---------------------
Parameters:
 - filename: The name and path of the file to load
---------------------
Local Variables:
 - reader: The StreamReader to load from
 - count: Integer for the number of shapes
 - s: A reference to a Shape
 - kind: A string
---------------------
 1: Assign reader, a new StreamReader passing in filename

 3: Assign Background = Ask reader to Read Color
 4: Assign count = Ask reader to Read Integer
 5: Ask _shapes to Clear its list

 6: For i looped from 0 while < count
 7:      Assign kind, ask reader to ReadLine

 8:      when kind is "Rectangle":
 9:           Assign s a new MyRectangle
10:      when kind is "Circle":
11:           Assign s a new MyCircle
12:      if none of the above, then continue

13:      Tell s to Load From reader
14:      Add Shape s

15: Tell reader to Close
```

This is a little more complex than Saving as it needs to work out what kind of shape to create. Notice how it uses the data that we saved to the file, and how this data enables us to reconstruct appropriate objects. Like Save, Load asks the Shapes to load themselves from the file. They will know how this data is formatted, that is their responsibility.

13. Switch to **Shape** and add the following method:



```csharp
public virtual void LoadFrom(StreamReader reader)
{
    Color = reader.ReadColor();
    X = reader.ReadInteger();
    Y = reader.ReadInteger();
}
```

Like SaveTo, LoadFrom is virtual so that child classes can override this behaviour.

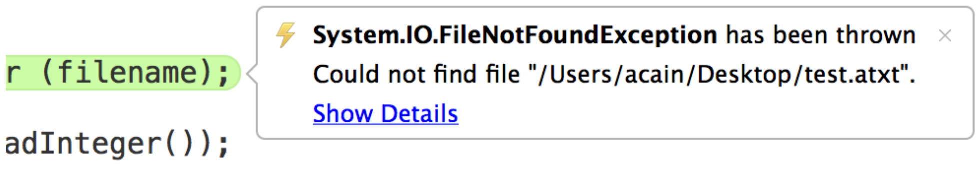
14. Switch to **MyRectangle** and override **Load From**.

```
Override Method: Load From
---------------------
Parameters:
 - reader: The StreamReader to load from
---------------------
 1: Tell base to Load From reader
 2: Assign Width, the result of asking reader to ReadInteger
 3: Assign Height, the result of asking reader to ReadInteger
```

15. Override **Load From** in the **MyCircle** and **MyLine** classes.
16. Switch to **Program** and add the code so that pressing the O key will open (load) the test drawing file from your desktop.
17. Compile and run the program. Load your drawing… you should see your Shape objects reappear!

Now that the basic features are working, what happens when there are problems?

18. Try changing the code so that it asks to load a file that does not exist.
19. Compile and run the program - it should crash and the debugger will break at the point where the error occurred!
20. Press the **continue execution** button and you will see the program just ends… not a great user experience!



```
r (filename);
adInteger());
```

System.IO.FileNotFoundException has been thrown
Could not find file "/Users/acain/Desktop/test.atxt".

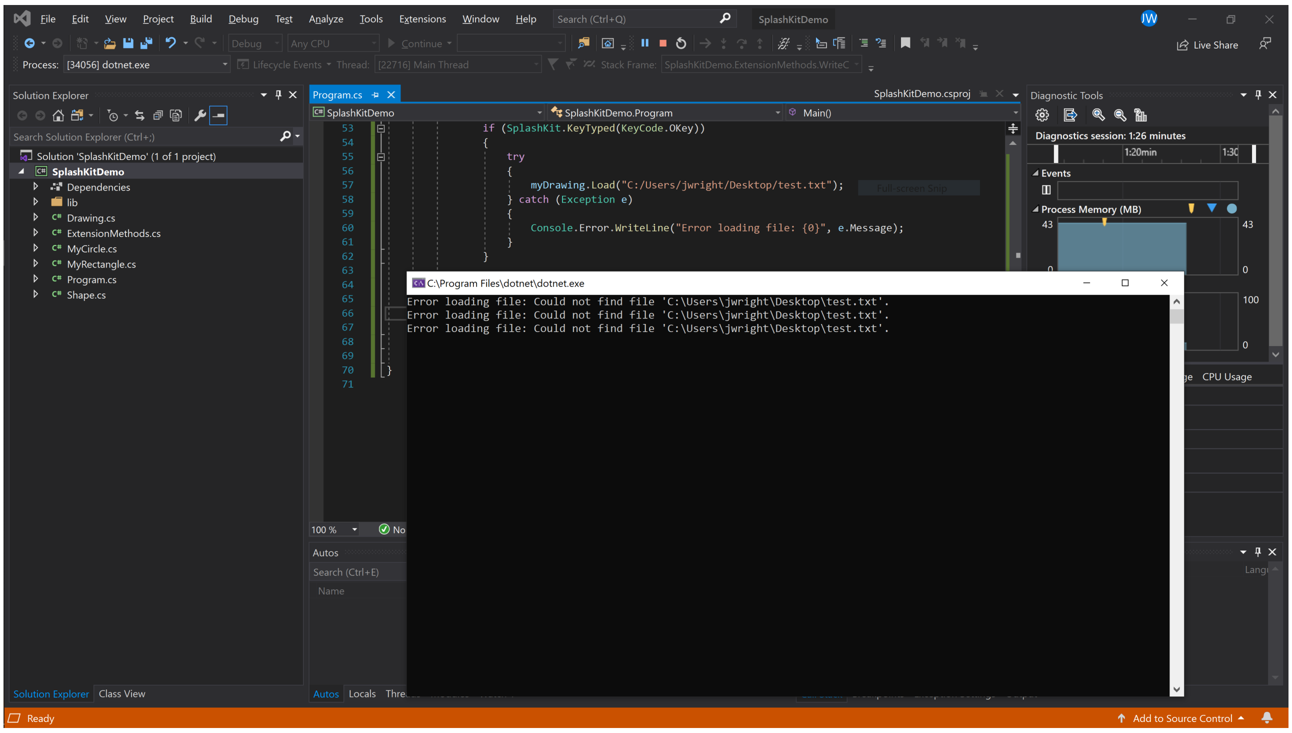
This demonstrates the use of **Exceptions**. Exceptions are a structured means of dealing with errors. When an unexpected error occurs your code can **throw** an Exception that will then discontinue the standard sequence of execution. Instead it will abort this method, and all previous methods as it looks to **goto** a point that can **catch** this exception. If there are no handlers, then the exception will also terminate the Main method and the program will end.

Lets handle this error in **Main** so that any exception in loading the file will not end the program.

21. Switch to **Program**. Add the following code around your call to Load in Drawing.

```
if (SplashKit.KeyTyped(KeyCode.OKey))
{
    try
    {
        myDrawing.Load("C:/Users/jwright/Desktop/test.txt");
    } catch (Exception e)
    {
        Console.Error.WriteLine("Error loading file: {0}", e.Message);
    }
}
```

22. Run the program again, and notice that you get an error message in the **Application Output** area of the IDE.



```
if (SplashKit.KeyTyped(KeyCode.OKey))
{
    try
    {
        myDrawing.Load("C:/Users/jwright/Desktop/test.txt");
    } catch (Exception e)
    {
        Console.Error.WriteLine("Error loading file: {0}", e.Message);
    }
}
```

```
C:\Program Files\dotnet\dotnet.exe
Error loading file: Could not find file 'C:\Users\jwright\Desktop\test.txt'.
Error loading file: Could not find file 'C:\Users\jwright\Desktop\test.txt'.
Error loading file: Could not find file 'C:\Users\jwright\Desktop\test.txt'.
```

Note: Ideally we should display the message to the user, but for the purpose of understanding exceptions this will be sufficient.

23. Change the code to load the previous drawing file, and run it to verify that it still works as expected.

Now what happens when the file contains invalid data?

24. Change the first line of your drawing text file to "Fred" - or something else that is not a number.

25. Run the program again. Notice that the Convert class also throws exceptions when it gets invalid data. Our catch block is catching all exceptions, so it handles this exception as well.

26. What about if you change a "Rectangle" (or one of the other markers that indicate which type to create) to "Something else"? Change the file and try to load it.

Notice that there are no exceptions or errors here. Our program just continues to try to load the file. What we could do is throw an exception so that there is an error when an unknown shape kind is located.

27. Switch to the **Drawing** class and locate the **Load** method.

28. Change the switch statement so that you **throw** a new **Invalid Data Exception**.

```
for (int i = 0; i < count; i++)
{
    kind = reader.ReadLine();
    switch(kind)
    {
        case "Rectangle":
            s = new MyRectangle();
            break;
        case "Circle":
            s = new MyCircle();
            break;
        default:
            throw new InvalidDataException("Uknown shape kind: " + kind);
    }

    s.LoadFrom(reader);
    AddShape(s);
}
```

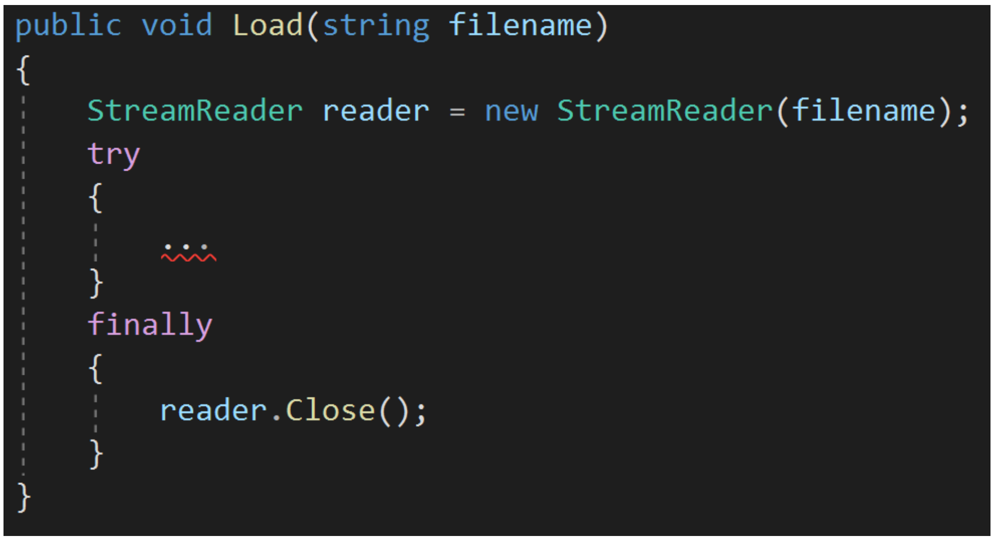
Note: C# has a family of related Exception types (i.e. lots of classes that inherit from Exception). By catching **Exception** the catch block is able to handle any exception. If you want to only handle certain kinds of exceptions you can catch the more explicit types.

When throwing exceptions, you should throw an appropriate exception object. In this case the InvalidDataException seems the most appropriate. See MSDN's Exception Hierarchy for details..

Now, one problem with exceptions is that they can skip code that you don't want them to. For example, at the end of the **Save** and **Load** methods we **close** the reader/writer. This is important as keeping it open will mean we cant change or delete it. We **always** want to close the file no matter what happens — close on success and close on exceptions.

To fix this we can use a **finally** block which ensures that the code within it is always run.

29. Switch to **Drawing** and locate the **Load** method.
30. Add a **try** … **finally** blocks around the body of this method so that when the reader is loaded it is always closed. This should appear as shown:



```
public void Load(string filename)
{
    StreamReader reader = new StreamReader(filename);
    try
    {
        ...
    }
    finally
    {
        reader.Close();
    }
}
```

Note: The one try block can be followed by many catch blocks (as long as each catches a different Exception class), and one finally block.

31. Add a **try … finally** to your **Save** method.
32. Rerun the program and make sure you can still load and save.

There's one last problem you may have noticed — your program isn't able to save and load lines.

33. Using what you have learned through this task so far, update your code so that lines correctly save and load.
34. Rerun the program and make sure you can still load and save.

Once your program is working correctly you can prepare it for your portfolio. To get it signed off as complete you will need to demonstrate it working to your tutor either during your lab or at the help desk.

Assessment Criteria

Make sure that your task has the following in your submission:

- The program must work with Lines, Circles, and Rectangles.
- The "Universal Task Requirements" (see Canvas) have been met.