

Introduction

[S#](#) is a .NET scripting language. It is good for creating domain specific languages and embedding scripts into .NET application. Few days ago a great article revealing integration between [S# and XAML in Silverlight](#) was posted.

In this post I am going to show another useful application of S# language as an extensible markup for creating PDFs.

Obviously S# will not generate PDFs itself. There is a whole range of different components for producing PDF files. They are powerful in their functionality and may be used in .NET languages such as C# or S#.

I am going to use [iText](#) library. This is a library that allows you to generate PDF files on the fly. Here is a quote from official iText site: "Typically you won't use it on your Desktop as you would use Acrobat or any other PDF application. Rather, you'll build iText into your own applications so that you can automate the PDF creation and manipulation process."

Let's make a step further and wrap iText into S# scripting language to produce a markup language for generating PDF documents.

Goal

My aim is to get following script working and producing correct document like given on the picture below the script:

```
//Create document at given location
BeginDocument('c:\\output.pdf');
    //Change title in document's meta data
    ActiveDocument.AddTitle('Sample document');

    //Create paragraphs with different alignment and color options
    Paragraph('Hello World', ALIGN_CENTER);
    Paragraph('This is demo', ALIGN_RIGHT, BLUE);
    Paragraph('This pdf was generated by S#', GREEN);

    //Create a list of string items
    BeginList();
        ListItem('One');
        ListItem('Two');
        ListItem('Three');
    EndList();

    //Create a table with tree columns
    BeginTable(3);
        //Create cells for the first row
        Cell('1');
        Cell('One');
        Cell(Paragraph('Description of One', RED));

        //Create cells for second row
        Cell('2');
        Cell('Two');
```

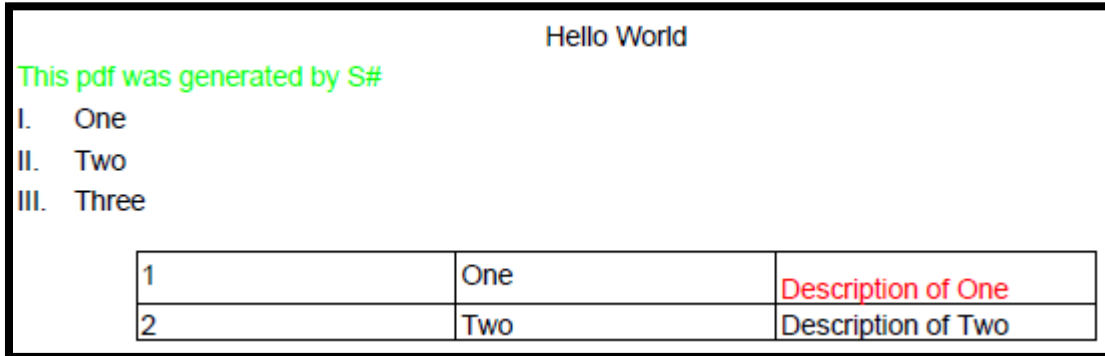
```

        Cell('Description of Two');
    EndTable();

//Flush and close document
EndDocument();

```

Target PDF document:



Implementation

S# has many points for extensibility and customization. For the task purpose we will be using context customization via constants and custom functions.

Preferable pattern for such customization is to introduce a new façade over standard RuntimeHost class from S#:

```

1. public static class PdfRuntimeHost
2. {
3.     public static void Initialize()
4.     {
5.         RuntimeHost.Initialize();
6.
7.         //Make any required customization to RuntimeHost
8.     }
9. }

```

ScriptContext is a part Script instance. ScriptContext store run-time information about variables, functions, scopes, etc which will be used during script execution.

There are two static methods within Script class for compiling and executing code:

```

1. namespace Orbifold.SSharp
2. {
3.     public class Script : IDisposable
4.     {
5.         public static Script Compile(string code);
6.         public static object RunCode(string code);
7.
8.         ...
9.     }

```

Again, preferred pattern for customization is to create a new façade class which customizes ScriptContext for those methods. So, I will introduce two new methods to PdfRuntimeHost class:

```

1. public static class PdfRuntimeHost
2. {
3.     public static void Initialize()
4.     {
5.         RuntimeHost.Initialize();
6.     }
7.
8.     public static Script Compile(string code)
9.     {
10.        Script s = Script.Compile(code);
11.        CustomizeScriptContext(s.Context);
12.        return s;
13.    }
14.
15.    public static object RunCode(string code)
16.    {
17.        IScriptContext context = new ScriptContext();
18.        CustomizeScriptContext(context);
19.        return Script.RunCode(code, context);
20.    }
21.    . . .

```

Context customization is quite simple itself. I am going to introduce couple of constants:

```

1. context.SetItem("ALIGN_CENTER", "CENTER"); //iText alignment values
2. context.SetItem("ALIGN_LEFT", "LEFT");
3. context.SetItem("ALIGN_RIGHT", "RIGHT");
4.
5. context.SetItem("RED", new BaseColor(255, 0, 0)); //iText colors
6. context.SetItem("GREEN", new BaseColor(0, 255, 0));
7. context.SetItem("BLUE", new BaseColor(0, 0, 255));

```

There will be three variables:

```

1. context.SetItem(Functions.ActiveListVariable, null);
2. context.SetItem(Functions.ActiveTableVariable, null);
3. context.SetItem(Functions.ActiveDocumentVariable, null);

```

ActiveDocumentVariable – will store reference to active Document object

ActiveListVariable – will store reference to active list variable

ActiveTableVariable – will store reference to active table variable

Note that it is not possible to have nested objects, like list of lists, or table containing other table in one of cells. However, this limitation may be overcome by introducing stacks of active containers instead of single reference variable.

At the final step I will build functional structure of new language. Let's consider Paragraph function as example (Other functions may be found in Functions.cs file in supplied solution file).

Paragraph function creates new paragraph object and automatically appends it to the document when necessary.

```

1. public static object Paragraph(IScriptContext context, object[] args)
2. {

```

```

3.         //Extract active document from context
4.         Document document =
5.             context.GetItemAs<Document>(ActiveDocumentVariable);
6.
7.         //Create new paragraph object. First argument - is a text
8.         Paragraph pf = new Paragraph((string)args[0]);
9.
10.        //If two arguments provided, second may be either color or
11.        //alignment value
12.        if (args.Length == 2)
13.        {
14.            BaseColor color = args[1] as BaseColor;
15.            if (color == null)
16.                pf.SetAlignment((string)args[1]);
17.            else
18.                pf.Font.Color = color;
19.        }
20.
21.        //If there three arguments, second is alignment, and
22.        //third is a color
23.        if (args.Length == 3)
24.        {
25.            pf.SetAlignment((string)args[1]);
26.            pf.Font.Color = (BaseColor)args[2];
27.        }
28.
29.        //Should we add paragraph to the document,
30.        //or it was generated for other container, i.e. List or Table
31.        if (!IsContainerScope(context))
32.            document.Add(pf);
33.
34.        return pf;
35.    }

```

Now, each new function should be added into ScriptContext like this:

```

context.SetItem("BeginDocument",
    new FunctionWrapper(Functions.BeginDocument));

```

Where FunctionWrapper is class implementing S#'s IInvokable interface. It takes method delegate as a first constructor parameter and executes this delegate as a result of invoking Invoke method with supplied parameters.

Conclusion

This example gives a good example of S# language customization. With a few simple steps you may create your own domain specific language. In this example such language simulates PDF markup.

But it is not only a markup. It is also a program and experienced users may write scripts which benefits from this:

```

BeginList();
    for(i=0; i<10; i++)
        ListItem(i.ToString());
EndList();

```

Downloads

- [Generated PDF file](#)
- [S# PDF Generator \(Sources\)](#)