http://projects.nikhilk.net

# Script#

Version 0.4.2.0
September 11, 2007

# Table of Contents

Introduction ..... 5
What can you build with Script#? ..... 5
Script# 101 ..... 7
The Hello World Application (Using Scriptlets for Script Code-behind) ..... 7
Building a Script Component in a Script# Class Library ..... 18
Summary ..... 22
The Script# Framework ..... 23
Script Type System and Base Class Library ..... 23
Script# Framework ..... 23
Core Programming Model, Networking, and UI Concepts - ssfx.Core.dll ..... 23
Cross-Domain AJAX using JSONP - ssfx.XDAjax.dll ..... 24
UI Controls and Behaviors - ssfx.UI.Forms.dll ..... 24
Reflection Utility - ssfx.Reflection.dll ..... 24
Microsoft Silverlight XAML DOM - ssagctrl.dll ..... 24
Microsoft Virtual Earth APIs - ssve4.dll ..... 24
Windows Vista Sidebar Gadgets - ssgadgets.dll ..... 24
File System APIs - ssfso.dll ..... 24
RSS Feeds - ssfeeds.dll ..... 24
Naming Convention ..... 25
Using Script# with Microsoft ASP.NET AJAX ..... 25
Differences and Limitations ..... 25
Importing Existing Script Libraries and Scriptable APIs ..... 26
Native Scriptable Objects and ActiveX Controls ..... 26
Existing Script Libraries ..... 28
A Deeper Look at the Script# System ..... 29
How Script# Works? ..... 29
Components and Layers ..... 30
Script Runtime Choice ..... 31
Using Script# ..... 31
Script Components and Libraries ..... 31
Scriptlets ..... 32

Limitations ..... 33
C# Limitations ..... 33
JavaScript Limitations ..... 34
How Do I Accomplish a Particular Script Scenario? ..... 35
Using *eval* to Execute Code and Perform JSON Deserialization? ..... 35
Using alert, prompt and Related Methods ..... 35
Performing Late-bound Member Access ..... 36
Deleting a Field from an Object ..... 37
Enumerating Members of an Object ..... 37
Retrieving the Native Script Type of an Object ..... 37
Defining and Implementing Global Methods ..... 38
Invoking Global Methods ..... 38
Defining Nested Functions to Implement Closures ..... 38
Creating and Using Plain Script or JSON Objects ..... 39
Creating Plain JSON Objects ..... 40
Checking for Undefined ..... 41
How is a Particular C# Feature Modeled in Script? ..... 41
How is a Namespace Defined? ..... 41
How is a Class Defined? ..... 41
How is a Derived Class Defined? ..... 42
How is an Interface Defined? ..... 42
How is a Delegate Type Defined? ..... 43
How are Delegates Used? ..... 43
How are Enumerations Defined and Consumed? ..... 44
How are Properties Declared and Accessed? ..... 45
How are Indexers Declared and Accessed? ..... 45
How are Events Declared and Accessed? ..... 46
How are Static Members Declared and Accessed? ..... 47
How is a foreach Statement Implemented? ..... 47
How are Anonymous Delegates Implemented ..... 48
Roadmap ..... 49
Feedback ..... 50

Version History....51
License....57

# Introduction

Script# is a C# compiler that generates JavaScript (instead of MSIL) for use in Web applications or other script-based applications such as Windows Vista Sidebar gadgets. The primary goal of Script# is to provide a more productive scripting environment for developing Ajax applications that are more maintainable over the long term by leveraging various aspects of the C# development model such as:

- Type checking and build errors at compile time
- Natural OOP-style programming
- C# IDE features and gestures, such as intellisense, refactoring, class browsing etc.
- Doc-comments

A key tenet of Script# is to generate JavaScript that is readable, understandable, and closely matches the originating source wherever possible. The goal is not to abstract away the underlying script APIs (such as the DOM), but to instead allow direct access, so the engineering choice of using a more productive tool does not sacrifice functionality or performance. The resulting JavaScript is compatible with the script support available in modern browsers.

Script# works by leveraging the C# build process, and fits in naturally and intuitively, as explained in the How Script# Works section. The compiler requires .NET Framework v2.0 on the development machine. You can deploy the resulting scripts on any server environment. You will need ASP.NET 2.0 on the server if you're using the server-side features (script code-behind for pages).

Script# can be leveraged in existing applications. It provides facilities to import existing script libraries as well as scriptable APIs such as the DOM, and those exposed by browser plugins such as Flash and other ActiveX controls. This is explained in the Importing Existing Script Libraries and Scriptable APIs section.

To enable you to quickly start creating script-based applications and components, Script# provides support for key DOM APIs, as well as other APIs such as WPF/E and Virtual Earth. It provides support for Microsoft ASP.NET AJAX, and also provides its own highly functional framework as an option for you to use. For out-of-browser scenarios, Script# provides a number of APIs that can be used to build Sidebar gadgets such as the Gadget API itself, RSS feeds APIs and the File System APIs. The feature set of these APIs are described in the Script# Framework section.

Script# is an evolving project as presented in the Roadmap section. Please do send feedback and suggestions, so they may be incorporated into the overall roadmap and future versions. The introductory blog post is at http://www.nikhilk.net/ScriptSharpIntro.aspx. The Script# project page is located at http://projects.nikhilk.net/Projects/ScriptSharp.aspx.

## What can you build with Script#?

The Script# install provides a number of project templates that are oriented around the key scenarios enabled by the technology.

- **Script code-behind for Web pages.**
  This is useful for adding little bits of script to a Web page either in the form of event handlers for

UI elements such as buttons or to initialize script functionality implemented in script files referenced in the page.

Script# introduces the concept of a Scriptlet, which can be thought of as the equivalent to the "main" method of a regular client application. The runtime capability is provided by the Script# Framework, which can be used in a vanilla html page. However, this is further simplified by its server control counterpart, <ssfx:Scriptlet> which has a dependency on ASP.NET 2.0. This control also provides a design-time experience including C# editing of the code-behind.

- **Script# Class Library.**
  This is perhaps the most common use of Script#. A class library is essentially a set of APIs and classes that encapsulate some functionality for reuse across pages. The Script# Framework itself is an example of a class library built using Script#.

  A regular C# Class Library project is used along with added Script# build steps to produce both release and debug flavors of script files in addition to the regular .NET assembly. The resulting script files can be packaged into server controls, or imported into pages directly. The original .NET assemblies can be used as references for other class library projects, or references within Scriptlet controls.

- **Script# Gadget.**
  A Script# Gadget is a special type of class library. Gadgets are essentially HTML pages with their functionality and behavior implemented in script, much like a regular Web page. The key differentiator is that a Gadget project refers to the Gadget API assembly, and implements specific classes representing the different parts of the gadget: the main UI, the settings page etc.

- **Microsoft ASP.NET AJAX-based Components, Controls and Behaviors.**
  Microsoft ASP.NET AJAX provides a core framework for implementing components, controls and behaviors. You can use it instead of the Script# framework. This works exactly like the Class Library model. The only difference is that it references the Microsoft ASP.NET AJAX runtime and core APIs instead of the Script# runtime and core APIs.

  The functionality of the compiler is almost identical. A few compiler features have been disabled, and various APIs from core types have been removed as they are not supported by the Microsoft ASP.NET AJAX runtime. The Script# framework is supported alongside the ASP.NET AJAX runtime; however this allows you to use the Script# development methodology and compiler even if you want to constrain your dependencies to the ASP.NET AJAX runtime.

  The usage of the ASP.NET AJAX mode and its limitations are described in the section on Using Script# with Microsoft ASP.NET AJAX.

The next section provides a Script# 101 overview that describes using scriptlets and building script class libraries.

# Script# 101

This section walks you through a HelloWorld-like scenario to demonstrate how you can implement client-side browser-based Web applications and reusable components using Script# and Visual Studio.

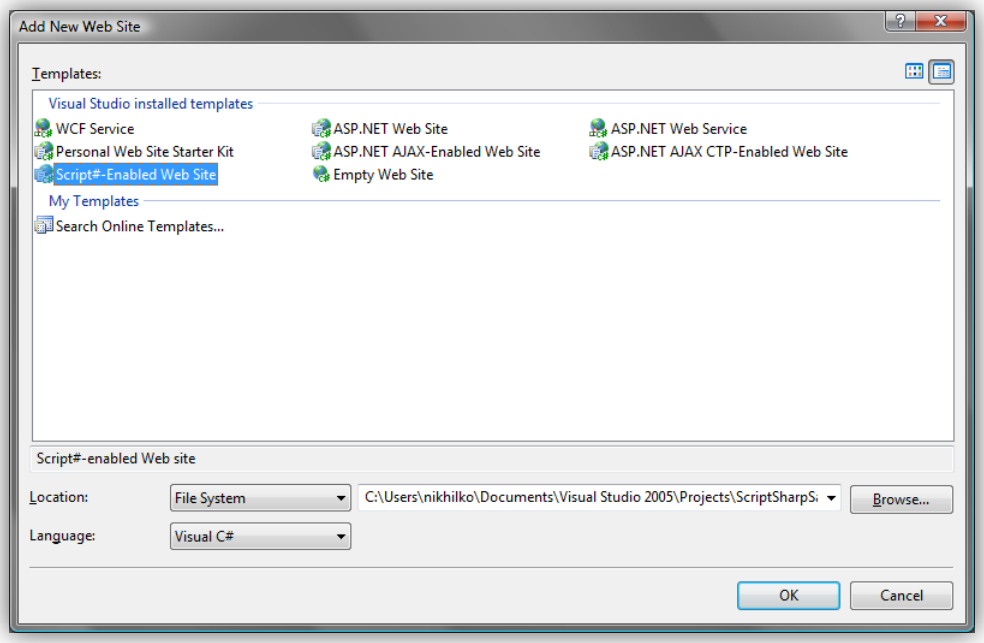
This walkthrough assumes you have installed Script#. While the walkthrough is described using Visual Studio, the steps can be adapted to Visual C# Express and Visual Web Developer just as well.

## The Hello World Application (Using Scriptlets for Script Code-behind)

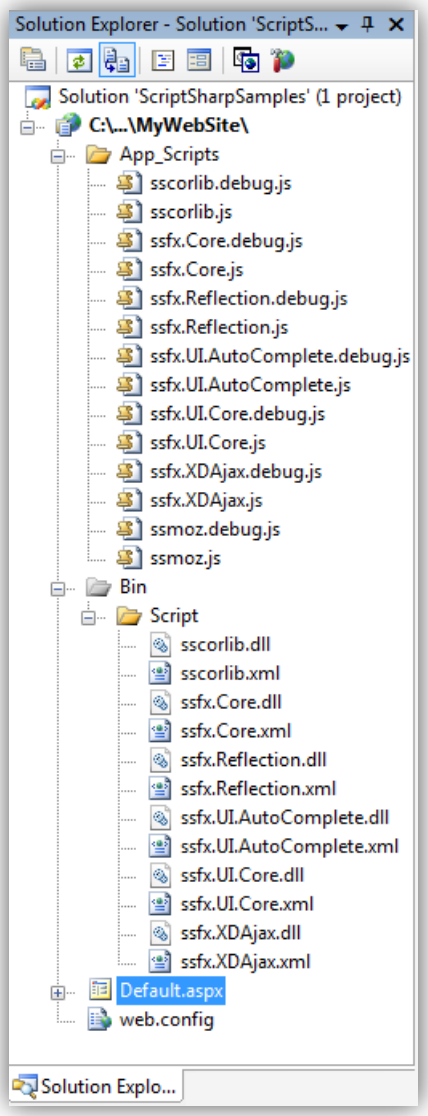
For the first sample, we're going to build the classic Hello World Script# equivalent, and then enhance it with some Ajax characteristics.

### Step 1: Create a new Script#-enabled Web site

Script# provides a Web site template that starts you with various files and configuration so that you are ready to quickly start using Script#.




The resulting Web site structure looks like the following (as shown in the screenshot of the solution explorer below) with Script# assemblies present in the Bin\Script folder and the corresponding debug and release script files within the App_Scripts folder.




**Step 2: Add the HTML to create a "Hello World" page**

The Web site contains a Default.aspx page which contains a reference to the <ssfx:Scriptlet> server control. This is a control that is provided by Script# and we'll see how it helps in a bit.[1]

The content in bold below was added to the default page created from the Web site template to provide a user interface to implement the Hello World scenario.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs" Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Hello World Script# Style</title>
</head>
```

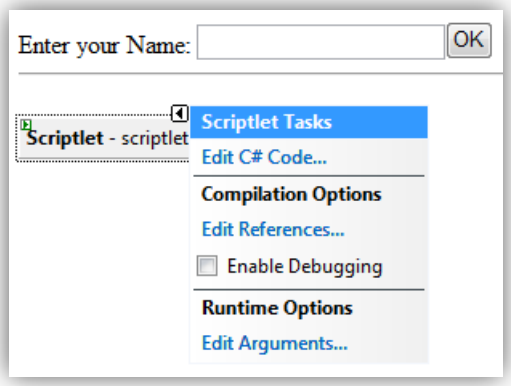
[1] This walkthrough is based on server controls. However, it is certainly possible to use Script# without a dependency on ASP.NET or ASP.NET server controls.

```
<body>
  <form id="form1" runat="server">
    <div>
      <label>Enter your Name:</label>
      <input type="text" id="nameTextBox" />
      <button type="button" id="okButton">OK</button>
      <hr />
      <label id="greetingLabel"></label>
    </div>
  </form>
  <ssfx:Scriptlet runat="server" ID="scriptlet">
    <References>
      <ssfx:AssemblyReference Name="sscorlib" />
      <ssfx:AssemblyReference Name="ssfx.Core" />
    </References>
    <Code>
      using System;
      using ScriptFX;

      public class MyScriptlet {

          public static void Main(ScriptletArguments arguments) {
          }
      }
    </Code>
  </ssfx:Scriptlet>
</body>
</html>
```

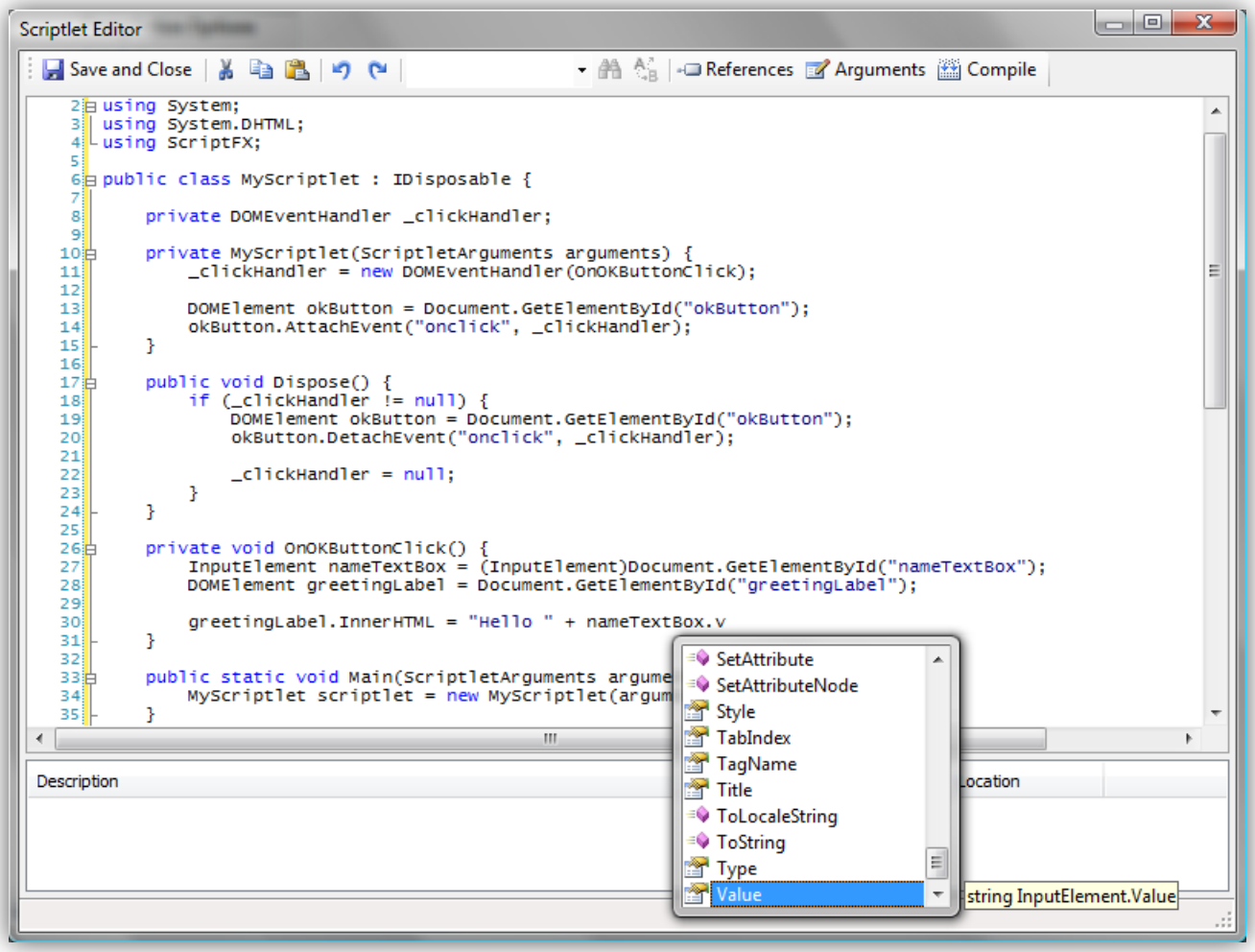
**Step 3: Add the Code to implement the "Hello World" scenario**

At this point, the next step is to add the logic to extract the text entered by the user into the nameTextbox element, process it and generate a greeting to be displayed in the greetingLabel element. Script# enables you to write this code in C#, which then gets translated into JavaScript.

The following is a screenshot of the page inside the IDE when you switch to design view, select the Scriptlet control, and choose to show its associated task panel.



The Scriptlet server control provides a design-time experience to allow you to edit the C# code-behind, add script and assembly references etc. in much the same way as you would for a regular C# client application. Select “Edit C# Code” from the Scriptlet’s associated task panel. The screenshot below illustrates the C# code editing experience provided by the server control designer including color coding, intellisense, general text editor features such as change tracking, line numbers etc.




Here is the code you need to author within the Scriptlet control. As you can see it is regular C# code.

```
using System;
using System.DHTML;
using ScriptFX;

public class MyScriptlet : IDisposable {

    private DOMEventHandler _clickHandler;

    private MyScriptlet(ScriptletArguments arguments) {
        _clickHandler = new DOMEventHandler(OnOKButtonClick);

        DOMElement okButton = Document.GetElementById("okButton");
```

```
        okButton.AttachEvent("onclick", _clickHandler);
    }

    public void Dispose() {
        if (_clickHandler != null) {
            DOMElement okButton = Document.GetElementById("okButton");
            okButton.DetachEvent("onclick", _clickHandler);

            _clickHandler = null;
        }
    }

    private void OnOKButtonClick() {
        InputElement nameTextBox = (InputElement)Document.GetElementById("nameTextBox");
        DOMElement greetingLabel = Document.GetElementById("greetingLabel");

        greetingLabel.InnerHTML = "Hello " + nameTextBox.Value + "!";
    }

    public static void Main(ScriptletArguments arguments) {
        MyScriptlet scriptlet = new MyScriptlet(arguments);
    }
}
```

Essentially execution starts in the Main method of your scriptlet. The code creates an instance of the MyScriptlet class which then uses the DOM to find the <button> with id=”okButton” in the page. It creates a delegate to the button click event handler and associates that with the button.

The click handler finds the <input> with id=”nameTextBox” and the <label> with id=”greetingLabel” to extract the text entered by the user, and to display a formatted greeting in the page.

The MyScriptlet instance implements the IDisposable interface, and performs cleanup, namely disassociates the event handler from the button. This is automatically called by Script# when the page is being unloaded.

The code here uses the DHTML DOM directly. Script# also provides higher level abstractions that simplify UI programming, but the DOM APIs provide sufficient functionality to implement this Hello World scenario.

**Step 4: Run the page**

Right click on the page, and choose View in Browser. In the browser, enter your name in the textbox, and click the OK button. The script in the page should generate an appropriate Hello greeting in the page.

Essentially at runtime, the C# code was compiled against the set of referenced assemblies and imported namespaces, and the equivalent JavaScript was generated for your class. In addition the Scriptlet control also generates some JavaScript to load the scripts corresponding to assemblies you referenced, and start your code once those scripts have been loaded.

If you are curious about the code that was generated, you can View Source in your browser. Here is the generated script:

```
window.main = function main() {
    Type.createNamespace('Scriptlet');

    ////////////////////////////////////////////////////////////////////////////////
    // Scriptlet.MyScriptlet

    Scriptlet.MyScriptlet = function Scriptlet_MyScriptlet(arguments) {
        this._clickHandler = Delegate.create(this, this._onOKButtonClick);
        var okButton = $('okButton');
        okButton.attachEvent('onclick', this._clickHandler);
    }
    Scriptlet.MyScriptlet.main = function Scriptlet_MyScriptlet$main(arguments) {
        var scriptlet = new Scriptlet.MyScriptlet(arguments);
    }
    Scriptlet.MyScriptlet.prototype = {
        _clickHandler: null,

        dispose: function Scriptlet_MyScriptlet$dispose() {
            if (this._clickHandler) {
                var okButton = $('okButton');
                okButton.detachEvent('onclick', this._clickHandler);
                this._clickHandler = null;
            }
        },

        _onOKButtonClick: function Scriptlet_MyScriptlet$_onOKButtonClick() {
            var nameTextBox = $('nameTextBox');
            var greetingLabel = $('greetingLabel');
            greetingLabel.innerHTML = 'Hello ' + nameTextBox.value + '!';
        }
    }

    Scriptlet.MyScriptlet.createClass('Scriptlet.MyScriptlet', null, IDisposable);

    ScriptFX.Application.Current.run(Scriptlet.MyScriptlet);
}
ScriptHost.initialize([
        'App_Scripts/ssfx.Core.js'
    ]);
```

As you will notice, the generated script code looks similar to the originating C# code. This is intentional. The Scriptlet control allows you to change code generation settings to generate release mode code that is minimized to reduce the size of the script.

The Scriptlet control also supports the scenario where the C# code has been precompiled.

Finally, you can use the scriptlet model without necessarily using the Scriptlet server control or ASP.NET (for example, in a vanilla HTML page) by precompiling the C# code, and using some boilerplate template that mimic the bootstrapping code generated by the Scriptlet server control.

**Step 5: Debugging the code**

Ensure that EnableDebugging is set to true on the Scriptlet control. This ensures the generated code is debuggable, and the debug versions of script references are included. The generated debug script code looks almost identical to the original C# code, and all the programmatic identifiers, method names etc.

are preserved. Even though you are not stepping through the original code in the debugger, this enables the debugging experience to still be usable.

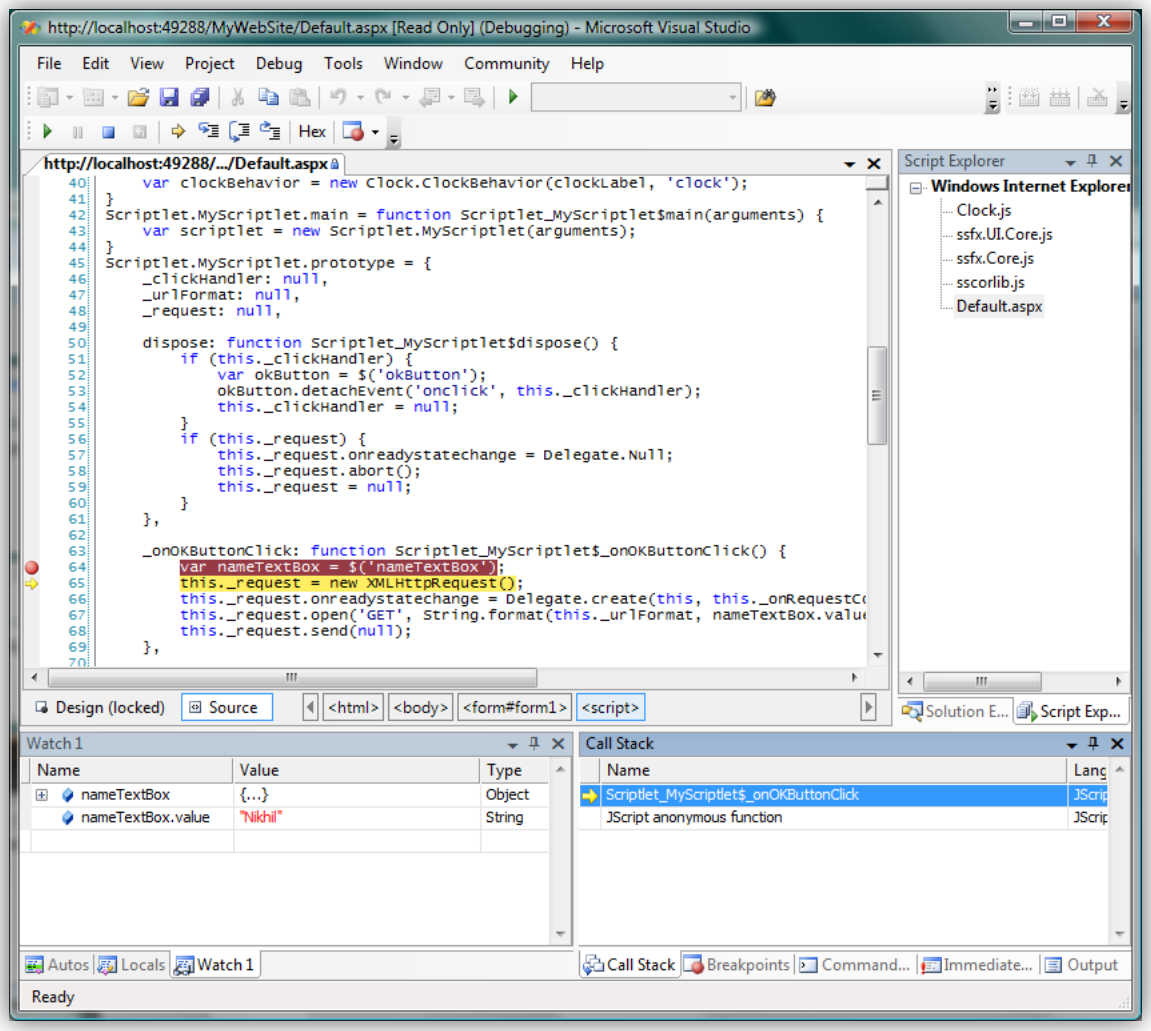
**Debugging in Internet Explorer**

Open a new instance of Visual Studio. Once the browser is running, you can attach to the browser (iexplorer.exe) using the Tools | Attach to Process menu. When you attach to the browser make sure you pick Script as the type of debugging you'd like to perform (instead of Native or Managed).

Open the Script Explorer tool window in Visual Studio, from the Debug | Windows menu. This shows all executing scripts. You can open any script file and place breakpoints.

For the purposes of experimentation, place the breakpoint in Default.aspx within the Scriptlet_MyScriptlet$_onOKButtonClick function. This is the event handler for the button.

Now click the button in the browser. The breakpoint should be hit, and you can now use the debugger, for example, examine variables in the watch window.

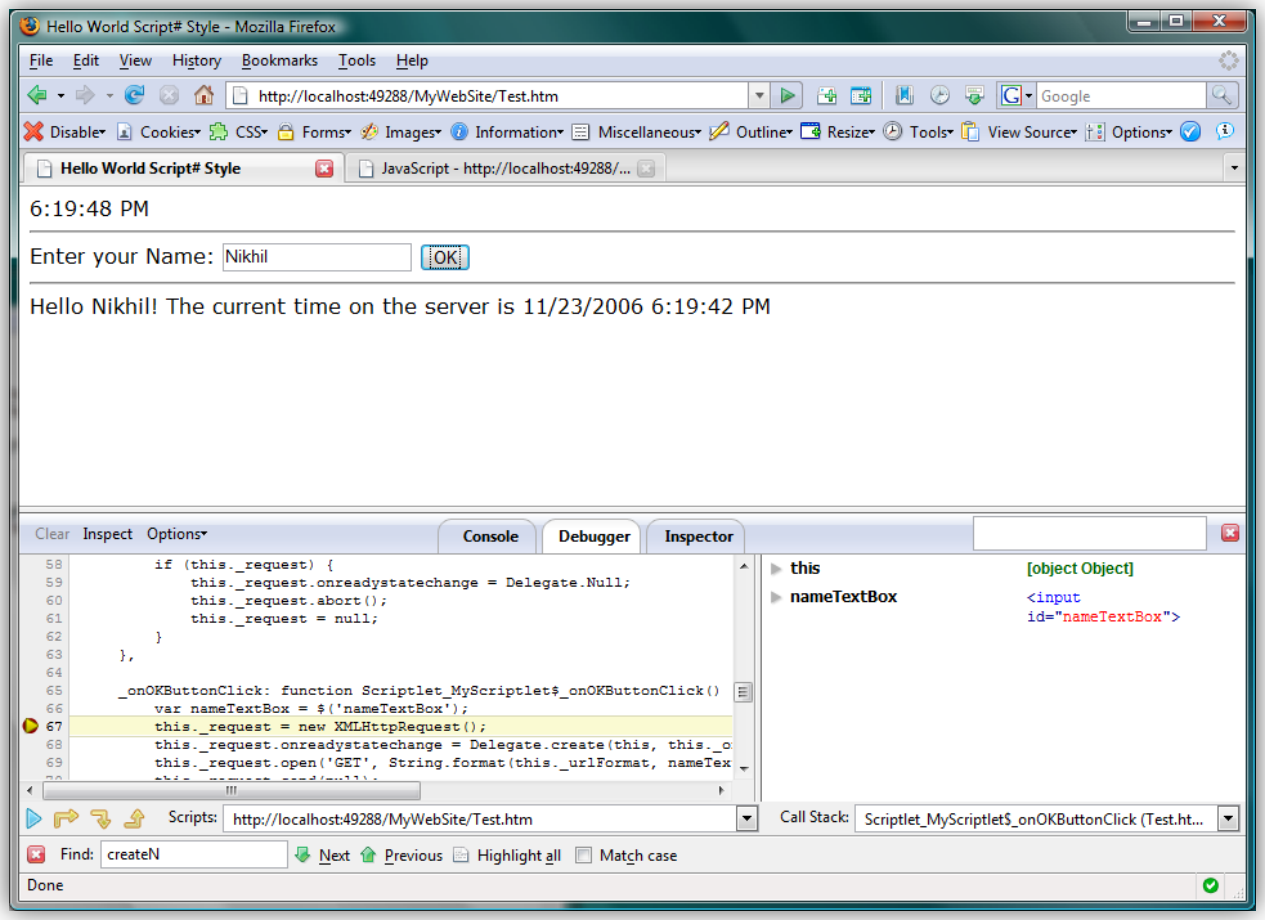


**NOTE:** You need to ensure script debugging is enabled in Internet Explorer. It is not by default. In order to enable it, open the Internet Options dialog via the Tools | Options menu, and open the Advanced tab. In the Browsing category, ensure the following settings:

- Disable script debugging (Internet Explorer): Unchecked
- Disable script debugging (Other): Unchecked
- Display a notification about every script error: Checked

**Debugging in Firefox**

You can get the Firebug extension for Firefox which embeds a script debugger within the Firefox browser.

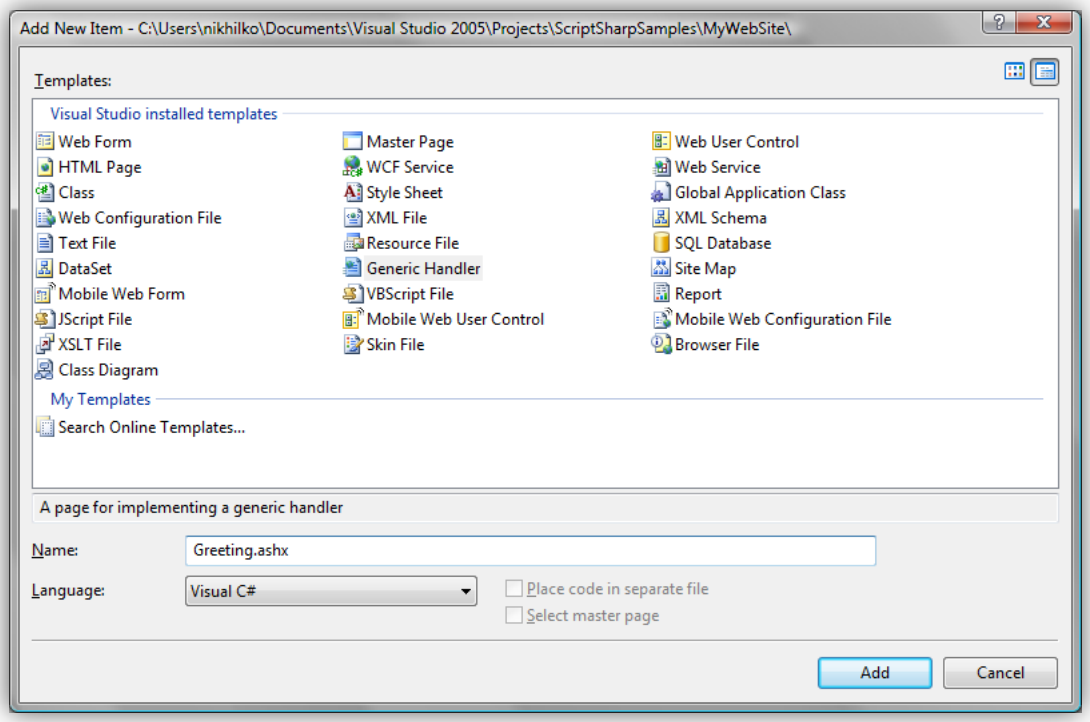



The debugger has the equivalents of the Visual Studio debugger. The dropdown on the bottom in the Debugger tag displays the list of script files similar to the Script Explorer. You can select Default.aspx from that list. The content of the script file is shown. You can place a breakpoint within the same method as before. When you click the button in the page, and the breakpoint is encountered, the right hand side of the debugger window displays a watch window and a call stack dropdown.

**Step 6: Implementing an Ajaxian Hello World page**

The primary goal of Script# is to enable the development of Ajax applications that contain significant amounts of client-side code. Additionally they contain application logic that uses XMLHttp to make HTTP requests to the server.

First we're going to add a Web handler (.ashx file) that will accept requests from the client, create a greeting on the server, and send the resulting greeting back to the client. Right click on the Web site in the solution explorer, and choose Add New Item, and pick the Generic Handler item and name the new file Greeting.ashx.




Here is the code that needs to go into the handler.

```
<%@ WebHandler Language="C#" Class="Greeting" %>

using System;
using System.Web;

public class Greeting : IHttpHandler {

    public void ProcessRequest(HttpContext context) {
        string name = context.Request.QueryString["name"];

        context.Response.ContentType = "text/plain";
        context.Response.Write("Hello " + name + "! The current time on the server is " +
                               DateTime.Now);
    }

    public bool IsReusable {
        get {
```

```
            return false;
        }
    }
}
```

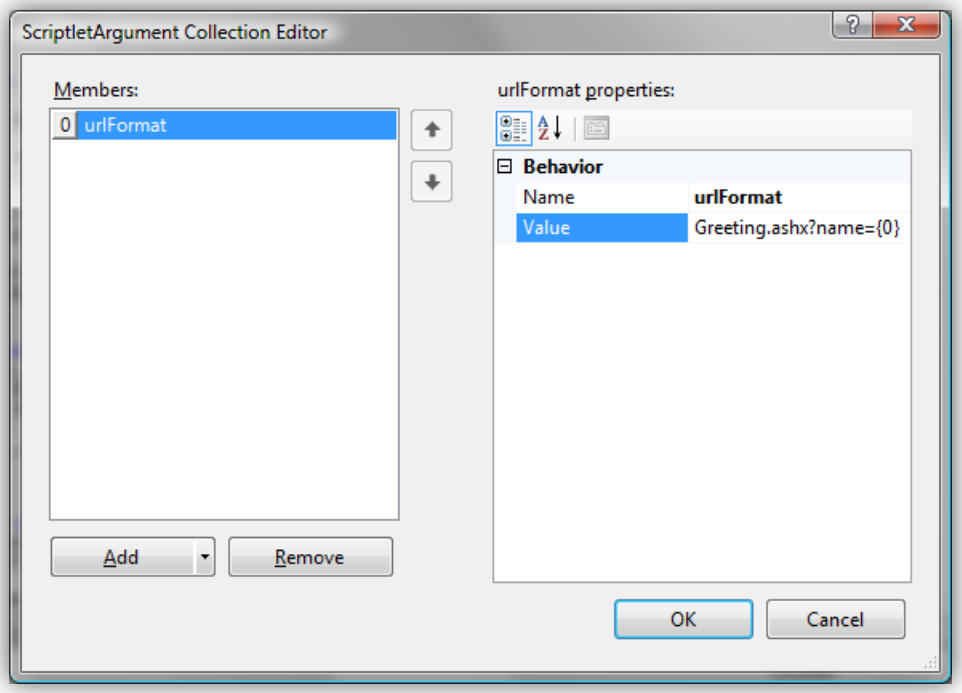
Now we have an HTTP service we can call from the client. The requests to the service take the form of Greeting.ashx?name=<text_entered_by_user>.

Now we need to write some code that uses XMLHttp to make these calls when the OK button is clicked instead of generating the greeting text completely on the client.

Go back to design view and edit the C# code.

Rather than hard-coding the URL format inside the code, lets pass it in as an argument into the Scriptlet. This will introduce the Arguments feature of the Scriptlet control, and the corresponding ScriptletArguments type in the C# code.

From the code editor window, click the Arguments button on the toolbar. This brings up an Argument collection editor, where you can add an string literal argument named urlFormat that contains the format of the URL requests to be used.

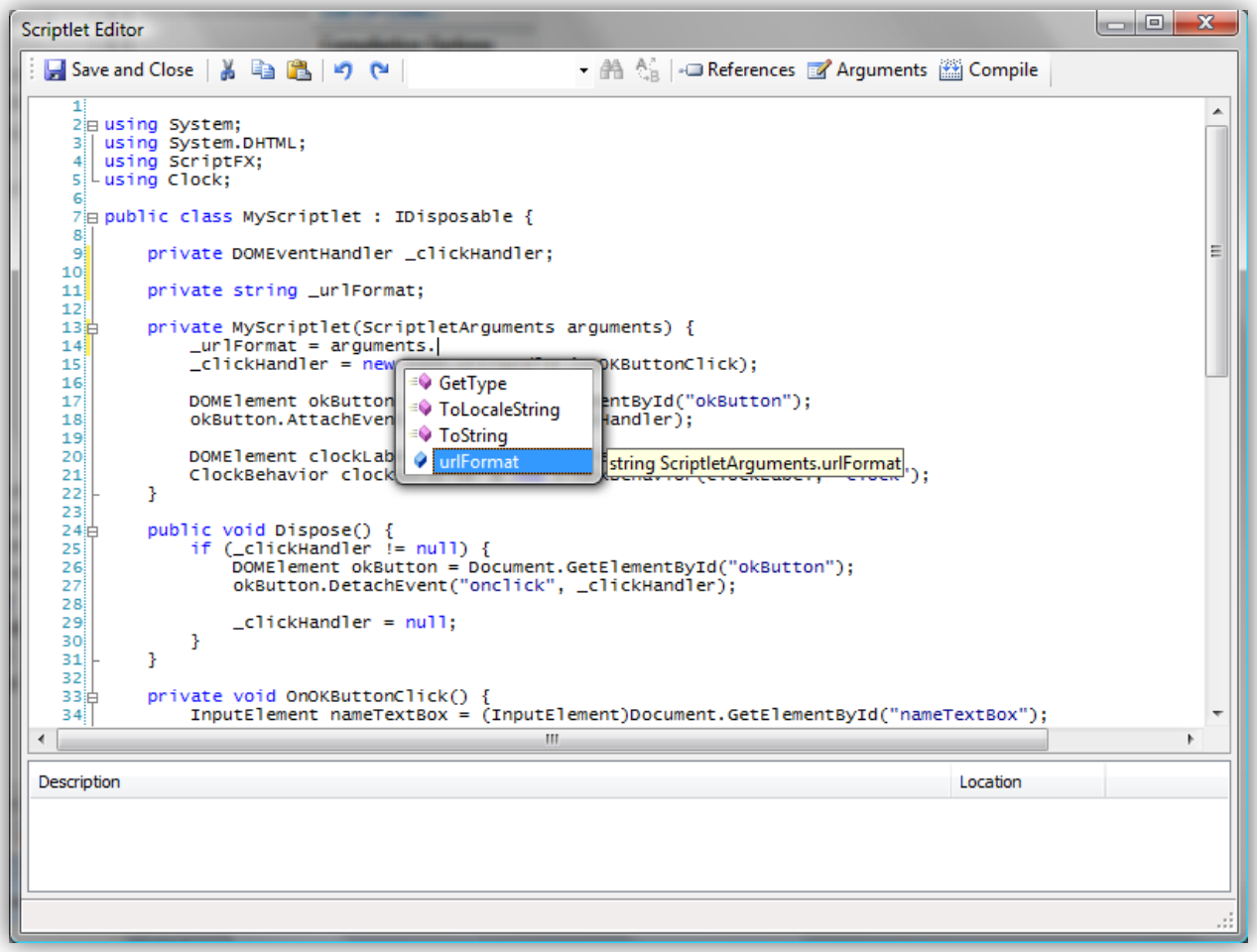



When the Scriptlet is saved, the argument is added as shown below.

```
<ssfx:Scriptlet runat="server" ID="scriptlet">
  <Arguments>
    <ssfx:StringLiteral Name="urlFormat" Value="Greeting.ashx?name={0}"></ssfx:StringLiteral>
  </Arguments>
```

```
  ...
</ssfx:Scriptlet>
```

This argument is now available as a member of the ScriptletArguments instance passed into your scriptlet. In fact, the ScriptletArguments class is a dynamically generated class, so the intellisense in the code editor window also detects the addition of the urlFormat member of the arguments instance as shown:




The code shown in bold needs to be added to MyScriptlet class.

```
public class MyScriptlet : IDisposable {

    ...

    private string _urlFormat;
    private XMLHttpRequest _request;

    private MyScriptlet(ScriptletArguments arguments) {
        _urlFormat = arguments.urlFormat;
        ...
    }
```

```
    public void Dispose() {
        ...
        if (_request != null) {
            _request.Onreadystatechange = (Callback)Delegate.Null;
            _request.Abort();
            _request = null;
        }
    }

    private void OnOKButtonClick() {
        InputElement nameTextBox = (InputElement)Document.GetElementById("nameTextBox");

        _request = new XMLHttpRequest();
        _request.Onreadystatechange = this.OnRequestComplete;
        _request.Open("GET", String.Format(_urlFormat, nameTextBox.Value.Escape()),
                      /* async */ true);
        _request.Send(null);
    }

    private void OnRequestComplete() {
        if (_request.ReadyState == 4) {
            _request.Onreadystatechange = (Callback)Delegate.Null;

            DOMElement greetingLabel = Document.GetElementById("greetingLabel");
            greetingLabel.InnerHTML = _request.ResponseText;

            _request = null;
        }
    }

    ...
}
```

The Script# framework provides higher level HTTPRequest classes to simplify network programming. XMLHttp was used here to minimize the concept count for this walkthrough. Furthermore, the Script# framework provides various other APIs and objects to further increase your productivity as a script developer, and to enable you to create richer Web applications.

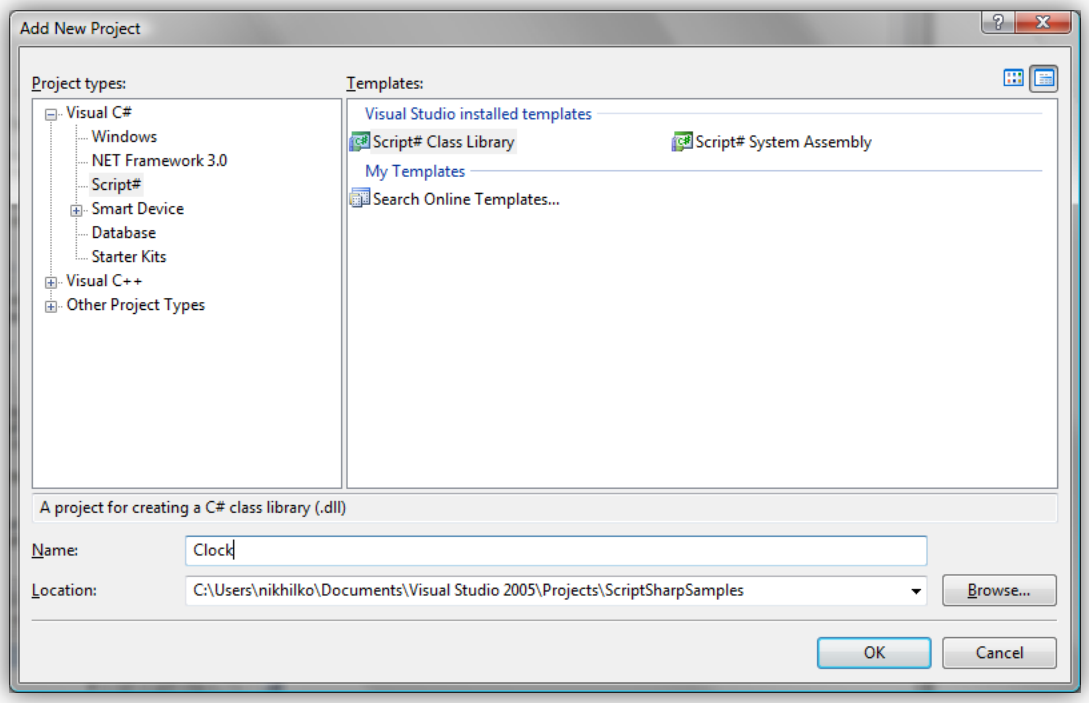
## Building a Script Component in a Script# Class Library

We are going to build a ClockLabel reusable component that can be used to display the current time in a label element on the page.

### Step 1: Add a Script# Class Library project

A Class Library project allows building a reusable component. It generates a .NET assembly (.dll file) and associated doc-comments (.xml file) that can be referenced in subsequent class library projects or by scriptlets. It also generates debug and release script files (.js files) that are included into a Web site and are sent down to the browser.

Add a project named “Clock” to the solution, and choose Script# Class Library as the type of project.




**Step 2: Implement the ClockBehavior component**

We're going to implement a class called ClockBehavior that derives from the ScriptFX.UI.Behavior class. A behavior provides a mechanism to author and encapsulate script functionality that can be attached to an HTML element.

The ClockBehavior is designed to be attached to a <label> element, and encapsulates a timer that ticks once per second. On each tick, the behavior displays the current time in the associated label. Here is the code for the ClockBehavior.

```
// ClockBehavior.cs
//

using System;
using System.DHTML;
using ScriptFX;
using ScriptFX.UI;

namespace Clock {

    public class ClockBehavior : Behavior {

        private int _intervalCookie;

        public ClockBehavior(DOMElement domElement, string id)
            : base(domElement, id) {
            _intervalCookie = Window.SetInterval(OnTimer, 1000);
        }

        public override void Dispose() {
            if (_intervalCookie != 0) {
```

```
                Window.ClearInterval(_intervalCookie);
            }
            base.Dispose();
        }

        private void OnTimer() {
            DateTime dateTime = new DateTime();
            DOMElement.InnerHTML = dateTime.Format("T");
        }
    }
}
```

**Step 3: Build the project**

Build the project. This compiles your code, and displays any compile errors that might result. Once your project successfully builds, the C# and Script# compiler generate the .NET assembly and associated script files respectively.

If you are curious about what was generated you can choose to show all files in the project which displays the hidden “bin” folder. Within the “Debug” folder you should see Clock.dll, Clock.xml, Clock.js and Clock.debug.js.

Open Clock.debug.js which contain script code as shown below.

```
Type.createNamespace('Clock');

////////////////////////////////////////////////////////////////////////////////
// Clock.ClockBehavior

Clock.ClockBehavior = function Clock_ClockBehavior(domElement, id) {
    Clock.ClockBehavior.constructBase(this, [ domElement, id ]);
    this._intervalCookie$1 = window.setInterval(Delegate.create(this, this._onTimer$1), 1000);
}
Clock.ClockBehavior.prototype = {
    _intervalCookie$1: 0,

    dispose: function Clock_ClockBehavior$dispose() {
        if (this._intervalCookie$1) {
            window.clearInterval(this._intervalCookie$1);
        }
        Clock.ClockBehavior.callBase(this, 'dispose');
    },

    _onTimer$1: function Clock_ClockBehavior$_onTimer$1() {
        var dateTime = new Date();
        this.get_domElement().innerHTML = dateTime.format('T');
    }
}

Clock.ClockBehavior.createClass('Clock.ClockBehavior', ScriptFX.UI.Behavior);
```

Again the generated script looks similar to the initial authored C# code, and is structured for ease of debugging.

Clock.js contains a release mode version of the same script in which whitespace has been stripped out along with any comments. Furthermore, private and internal members have been minimized to reduce script file size.

```
Type.createNamespace('Clock');Clock.ClockBehavior=function(domElement,id){Clock.ClockBehavior.con
structBase(this,[domElement,id]);this.$1_0=window.setInterval(Delegate.create(this,this.$1_1),100
0);}
Clock.ClockBehavior.prototype={$1_0:0,dispose:function(){if(this.$1_0){window.clearInterval(this.
$1_0);}Clock.ClockBehavior.callBase(this, 'dispose');},$1_1:function(){var $0=new
Date();this.get_domElement().innerHTML=$0.format('T');}}
Clock.ClockBehavior.createClass('Clock.ClockBehavior',ScriptFX.UI.Behavior);
```

**Step 4: Reference the project and consume the component**

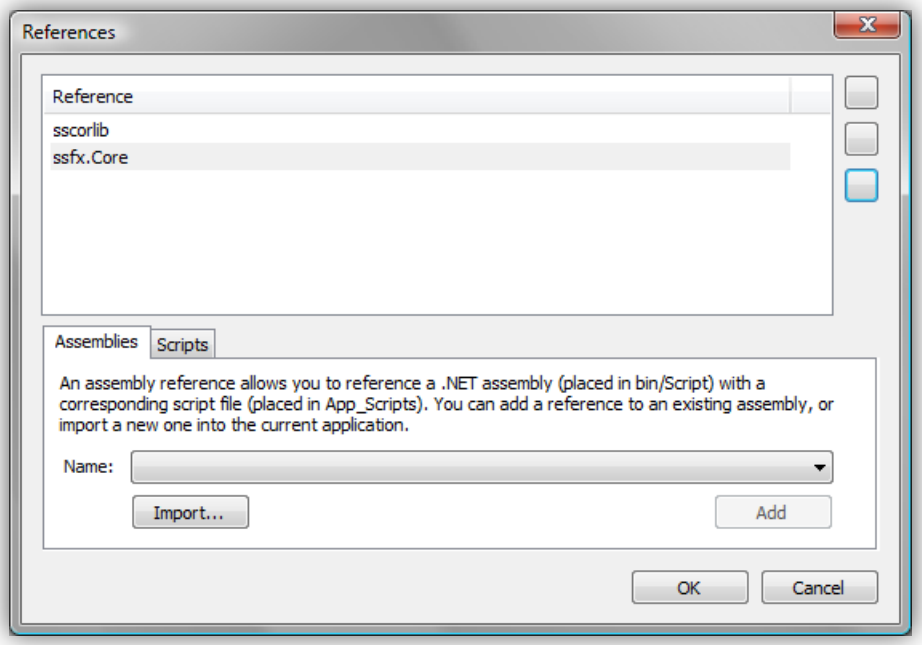
In order to run the class you have just authored, you need to include it in a Web site, and use it on a page by associating it with a label element.

Open Default.aspx and add a <label> with id=”clockLabel” to the page.

```
<label id="clockLabel"></label>
```

Next we need to add a reference to the assembly we just authored, and add some code to instantiate an instance of the ClockBehavior class.

Switch to design view, and select the Scriptlet control. Select the Edit References task of the Scriptlet control’s task panel. The following dialog appears:




In order to reference the Clock assembly, we need to first import this into the Web site. Click Import and select the Clock.dll file from the bin\Debug folder of the Clock project. The import process copies

Clock.dll and Clock.xml files into the Web site's bin\Script folder. It also copies the Clock.debug.js and Clock.js script files into the Web site's App_Scripts folder. Once the import has taken place, click Add to add the assembly to the list of referenced assemblies, and then OK, to save the changes to the References collection of the Scriptlet.

The last part of this step is to write some code that uses the ClockBehavior class. Select the Edit Code task from the scriptlet and add the code shown in bold.

```
using System;
using System.DHTML;
using ScriptFX;
using Clock;

public class MyScriptlet : IDisposable {

    ...

    private MyScriptlet(ScriptletArguments arguments) {
        ...

        DOMElement clockLabel = Document.GetElementById("clockLabel");
        ClockBehavior clockBehavior = new ClockBehavior(clockLabel, "clock");
    }

    ...
}
```

**Step 5: Run the page**

Now you can run the page by selecting View in Browser. When the browser loads the page, the script in the page will load Clock.js, and your scriptlet code will instantiate an instance of ClockBehavior.

The ClockBehavior code that you authored will now start displaying the current time in its associated label element.

## Summary

This walkthrough provided a very basic overview of Script#, and using it to implement application code-behind in the form of scriptlets, as well as reusable components. It showed how you can use C# to implement your client-side logic, without writing any more script, thereby deriving the benefits of intellisense in the IDE, and compile-time checking of your code. You can use the Class Browser feature in Visual Studio to browse the Script# assemblies as well as the custom script components you author, which leads to increased discoverability of framework features beyond what can be achieved by looking at script code directly.

There is a lot more you can do with Script# as well as additional options, and framework components and building blocks to use. Hopefully this walkthrough provided enough context and a general idea of starting your development with Script#!

# The Script# Framework

This section introduces the parts of the Script# Framework. The full reference documentation for the APIs is provided in the form of XML doc-comments that can be used with a tool such as .NET Reflector. Eventually these doc-comments will be compiled into a help file.

## Script Type System and Base Class Library

The type system and base class library are represented by the sscorlib.dll assembly and its corresponding sscorlib.js script file. These must be referenced in all code compiled using Script#. This core script library contains extensions to the standard JavaScript objects such as Function, Array, String, Number, etc. as well as a small number of new types.

The type system simulates key OOP constructs to provide structure to the generated code that mimics the original C# source code. The OOP constructs include namespaces, classes, interfaces, enumerations, and delegates. The implementation allows runtime inspection of types to retrieve type names, base classes, checking for the existence of an interface implementation etc. The type system uses the prototype feature of JavaScript to define classes.

The base class library is partly implemented by extending the prototype of key objects such as Array, String, Number, Date etc. to provide an API that is consistent to the .NET equivalents, as well as API that is useful in the scripting environment that is beyond that available in .NET. The base class library also provides some new types such as StringBuilder, CultureInfo, Debug etc. and capabilities such as string formatting that simplify various aspects of script authoring.

## Script# Framework

The Script# Framework is a set of assemblies that provide a script-based application and programming model. It is possible to implement an alternative framework on top of Script# without depending on the Script# Framework, if you have unique requirements.

### Core Programming Model, Networking, and UI Concepts - ssfx.Core.dll

This assembly provides the core programming model in the ScriptFX namespace, which consists of an Application class and related services such as dispose mechanism, history management, session state, idle task execution, and script loading. It also consists of generally useful utility classes to manage event handlers, perform JSON serialization, detect browser and host information etc.

The ScriptFX.Net namespace provides a higher-level HTTP-based networking stack with HTTPRequest/HTTPResponse classes with useful features such as timeouts, request building, scheduled execution, custom caching, and monitoring hooks via a centralized HTTPRequestManager etc. It provides an extensible model for plugging in different transports (such as XMLHttp) to perform actual invocation of network requests.

The ScriptFX.UI namespace provides the core infrastructure classes to associate script-based logic with DOM elements. Specifically it introduces the notion of Controls and Behaviors. In addition it defines core interface contracts for some UI elements. It provides a high-performance extensible animation core that

can be used to incorporate visual glitz into an application. In the future this will consist of other core UI infrastructure and services such as templating and drag/drop.

### Cross-Domain AJAX using JSONP - ssfx.XDAjax.dll

This assembly provides the implementation of an alternate HTTPTransport in the ScriptFX.Net namespace that can be plugged in into the networking system to enable cross-domain requests performed using script tags and the JSONP protocol.

### UI Controls and Behaviors - ssfx.UI.Forms.dll

This assembly provides the implementation of commonly used controls and behaviors when implementing forms in Web pages. This includes controls like TextBox, Button etc. as well as higher level features such a rich AutoComplete behavior, Watermark behavior that can be attached to input textboxes. In the future, this assembly will provide features such as validation, calendar controls etc.

### Reflection Utility - ssfx.Reflection.dll

This assembly provides higher-level .NET-like reflection functionality in the ScriptFX.Reflection namespace in order to enumerate namespaces, types, and members. This can be used to implement class-browsing scenarios and applications.

### Microsoft Silverlight XAML DOM - ssagctrl.dll[2]

This assembly provides APIs to program against the Microsoft Silverlight control within the browser. Specifically it allows programming against the XAML DOM to enable AJAX applications to make use of scriptable vector graphics, media and animations that are offered by the Silverlight control.

### Microsoft Virtual Earth APIs - ssve4.dll

This assembly provides APIs that map to v4 of the Microsoft Virtual Earth Map Control. The map control is an existing script library, and this assembly simply provides metadata so that it may be used from C# and compilation with Script#.

### Windows Vista Sidebar Gadgets - ssgadgets.dll

This assembly enables programming against the scriptable APIs that can be used to develop gadgets that run within Microsoft Windows Vista Sidebar.

### File System APIs - ssfso.dll

This assembly provides a metadata assembly that enables using the FileSystem Scripting Object available to trusted script applications on Microsoft Windows. Trusted script applications include gadgets. Typically browser-based applications are not trusted. This API allows trusted applications to work against files and folders, as well as read and write local files.

### RSS Feeds - ssfeeds.dll

This assembly provides a metadata assembly that enables programming against the user's RSS store introduced in Internet Explorer 7.0. Like the file system APIs listed above, this API is only available

[2] Microsoft Silverlight APIs and assembly names often contain the "ag" prefix. Ag stems from the periodic table, representing the Silver metal. ☺

outside the Web browser to trusted applications such as gadgets. This API allows subscribing/unsubscribing to RSS feeds, and access the data present in the user's RSS feeds.

### Naming Convention

You may have noticed a naming convention at play. Indeed there is a pattern. All the assemblies named ssfx*.dll represent framework implementations in script (that were compiled using Script# of course). All the other ss*.dll assemblies represent existing script APIs or scriptable APIs that have been imported.

## Using Script# with Microsoft ASP.NET AJAX

While the Script# framework is supported alongside the ASP.NET AJAX runtime in the same page, some developers and applications need to constrain their dependencies to the ASP.NET AJAX runtime. The Script# compiler provides an Atlas[3]-mode to cater to this scenario, thereby enabling developers of these applications to continue to derive the productivity and other benefits of the Script# methodology.

Essentially the compiler is switched into Atlas-mode when the Atlas runtime (aacorlib.dll) is referenced instead of the Script# runtime (sscorlib.dll). Only one runtime can be referenced when invoking the compiler, and this provides a handy way to both switch modes, and at the same time provides the specific APIs available for use with ASP.NET AJAX. In addition to aacorlib.dll

Along with aacorlib.dll, aaagctrl.dll provides the metadata to program against Microsoft Silverlight when using ASP.NET AJAX. None of the functionality implemented in any of the ss*.dll assemblies can be used in this mode, as all the ss*.dll assemblies reference sscorlib.dll.

### Differences and Limitations

ASP.NET AJAX does not provide various APIs when compared to the Script# runtime (which is a logical superset in terms of functionality). The motivation for ASP.NET AJAX support is to provide the Script# constrained to Microsoft ASP.NET AJAX functionality. As such there are a handful of differences and limitations to be aware of:

1. No support for foreach over arrays – In Script# arrays are extended to implement IEnumerable, and they are not in ASP.NET AJAX. Hence foreach over plain arrays does not work. The work around is to use a regular for loop.
2. No support for auto-generated event accessors. Auto-generated event accessors require the existence of a Delegate class with Delegate.Combine/Remove semantics, which are not provided by ASP.NET AJAX. The workaround is to explicitly implement the add/remove accessors for events in your code, rather than have the compiler generate it.
3. Lack of Array, String, Math etc. extensions – ASP.NET AJAX does not provide the full set of extensions that are provided by Script# over the core JavaScript objects. Furthermore, in ASP.NET AJAX, the extensions to the Array object are provided as static methods on the Array type, rather than instance methods on Array instances. These differences manifest themselves

[3] Atlas was the codename for ASP.NET AJAX.

by virtue of the core types in aacorlib.dll having a reduced or different APIs than their counterparts in sscorlib.dll which represents the Script# runtime.

## Importing Existing Script Libraries and Scriptable APIs

An assembly can be used to define types and APIs to define metadata about native scriptable APIs such as the DOM, and ActiveX controls or existing script APIs and libraries, so that they can be referenced from C# code. These types and APIs do not contain an actual C# implementation that gets compiled to script. Instead they exist to simply provide information about the set of available types, and the signatures of the APIs they offer[4].

A type that represents a native object or existing script and simply exists to provide signature information for APIs that can be used application types is marked as such by applying the Imported metadata attribute. This allows other code to reference the type, but indicates to the Script# compiler, that the type does not contain any C# source that needs to be compiled into script.

### Native Scriptable Objects and ActiveX Controls

The following is a rough example of how the Windows Media Player ActiveX control could be exposed to C# code using a system assembly.

```
namespace System.WindowsMediaPlayer {

    // For intrinsic native objects, the namespace is meaningless. However it is
    // useful to partition types as they are exposed to C# code. This attribute
    // tells the Script# compiler to ignore the namespace at script-generation time.
    // Since the type represents an existing native scriptable object, it is marked as
    // Imported.
    [IgnoreNamespace]
    [Imported]
    public sealed class MediaPlayerControls {

        private MediaPlayerControls() {
            // Private to disallow creation
        }

        public void Play() {
            // No code is necessary, since all that is needed is the
            // definition of the Play method and its signature.
        }

        public void Stop() {
        }
    }

    [IgnoreNamespace]
    [Imported]
    public sealed class MediaPlayer : DOMElement {
        // This particular class derives from DOMElement as media player is
        // represented by an <object> tag.
```

[4] Prior builds of Script# had a concept called "System Assemblies". Such assemblies only contained metadata, and no implementation. With newer builds of Script# (starting with 0.3.0.0), there is no longer a special type of assembly. Instead metadata definitions can be included alongside types that do contain C# implementation that gets compiled into script as described in this section.

```
        private MediaPlayer() {
            // Private to disallow creation
            // Application code can use Document.GetElementById and cast
            // the resulting element into a MediaPlayer instance to
            // program against a media player instance on the page.
        }

        [IntrinsicProperty]
        public MediaPlayerControls Controls {
            // Notice the use of the IntrinsicProperty attribute. This
            // ensures the compiler does not add the prefix get_ in calls
            // to this property.
            get { return null; }
        }

        [IntrinsicProperty]
        public string URL {
             get { return null; }
             set { }
        }
    }
}
```

The object model created in the snippet above is representative of the type of object model needed to create a page shown in the media player sample at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wmplay10/mmp_sdk/usingtheplayercontrolinawebpage.asp.

Once the above wrapper has been created, the following sample C# code can be written to consume it. Since the MediaPlayer was written to derive from DOMElement, Document.CreateElement can be used to create an ActiveX <object> tag, and the result can be cast into MediaPlayer. Thereafter you can use the strongly typed OM on your class. This works because the casts only exist to enable the C# code to compile. In script, the casts simply become no-ops.

```
using System;
using System.WindowsMediaPlayer;

namespace MyApp {

    public sealed class PlayerApplication {

        private MediaPlayer _player;

        public PlayerApplication() {
            _player = (MediaPlayer)Document.CreateElement('object');
            // Set other properties of the <object> tag

            Document.Body.AppendChild(_player);
        }

        private void OnPlayButtonClick() {
            _player.Controls.Play();
        }
    }
}
```

Note that the above approach can also be used for plugin objects added to pages using the <embed> tag that is required in browsers other than Internet Explorer.

For ActiveX COM objects that aren't inserted into the DOM, and are instead instantiated by calling new ActiveXObject (allowed on Internet Explorer), you could define the C# wrapper class by deriving from Object directly. In addition you'll need to provide an implicit type conversion operator (operator declarations are only allowed on metadata wrapper classes) from the ActiveXObject type, so that you can successfully cast the resulting ActiveXObject into your object for use in C# code. The following example defines metadata for the FileSystemObject.

```
namespace System.FileSystem {

    [IgnoreNamespace]
    [Imported]
    public sealed class FileSystemObject {

        private FileSystemObject() {
            // Private to disallow direct creation. App code should use new ActiveXObject
            // to create instances.
        }

        // Members specific to the COM object

        // This allows an implicit conversion from ActiveXObject into FileSystemObject.
        public static implicit operator FileSystemObject(ActiveXObject o) {
            return null;
        }
    }
}
```

With this metadata, it is now possible use the FileSystemObject as follows:

```
using System;
using System.FileSystem;

namespace MyApp {

    public sealed class SampleApp {

        public SampleApp() {
            FileSystemObject fso = new ActiveXObject("Scripting.FileSystemObject");
        }
    }
}
```

## Existing Script Libraries

The following is a basic example of how the Virtual Earth script APIs can be exposed to C# code.

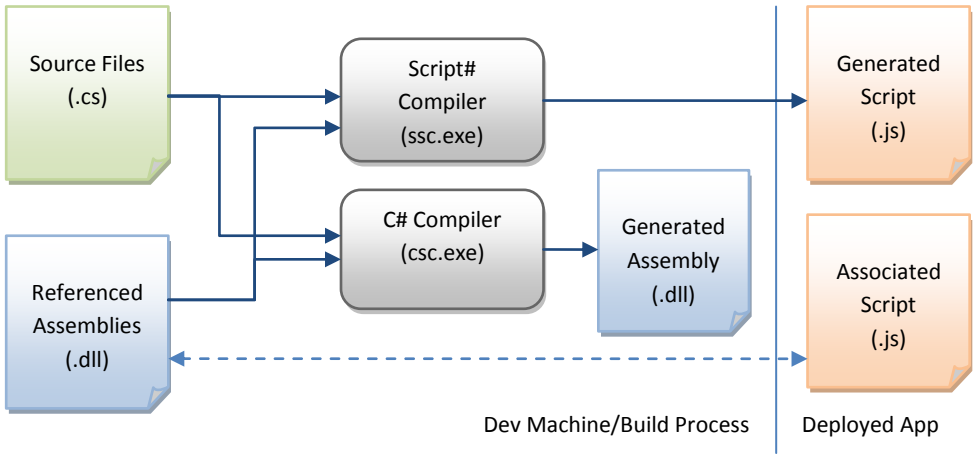
For example, Virtual Earth can be consumed by importing the scripts at described at http://dev.live.com/virtualearth/sdk/. In order to consume them from C#, a system assembly can provide metadata for those APIs to C# code.

```
namespace System.VirtualEarth {
```

```
    [IgnoreNamespace]
    [Imported]
    public sealed class VEMap {

        public VEMap(string elementID) {
        }

        [PreserveCase]
        public void Pan(int deltaX, int deltaY) {
            // Notice the use of the PreserveCase attribute. Unlike most
            // scripting APIs, Virtual Earth uses pascal-cased names. This
            // attribute suppresses conversion to camel-case names that is
            // otherwise performed by the Script# compiler.
        }
    }
}
```

# A Deeper Look at the Script# System

## How Script# Works?




Script# works by converting C# source directly into JavaScript. In order to do so, it parses your C# code like a real C# compiler. In addition to your .cs source files, it consumes a set of reference assemblies that contain metadata about namespaces and types you are importing. The assumption is that these assemblies have corresponding JavaScript files you will include into your deployed Web application, along with your generated script code. In addition to the Script# compiler, you can also run the regular C# compiler to validate the C# code and to generate an assembly that can be used as a reference in a future project. In fact, the Script# compiler assumes that you will compile your code through the C# compiler to ensure it is valid C# code.

When authoring code to be compiled into JavaScript it is important to reference the sscorlib.dll instead of the regular mscorlib.dll that C# projects ordinarily do. sscorlib.dll provides the definition of key .NET types such as System.Object, primitive types, and other core BCL types. A C# project can avoid

depending on mscorlib.dll using the /nostdlib option on the command line call to csc.exe and by setting the NoStdLib property to True in the .csproj project file. The Script# install provides a project template with the right project settings.

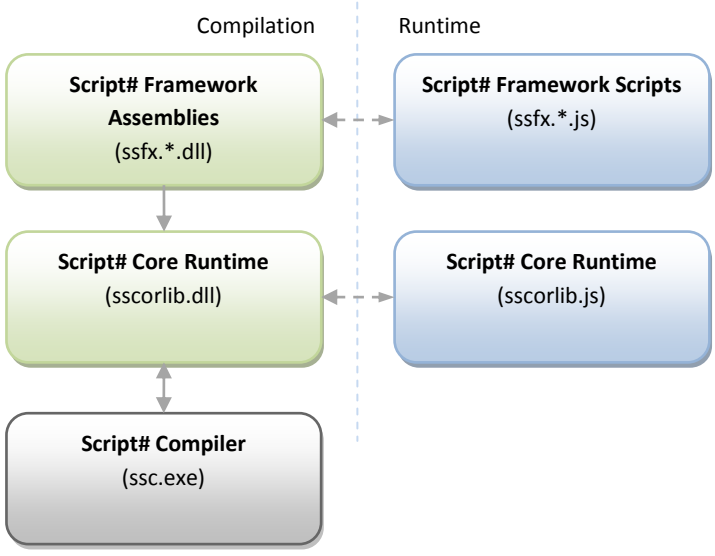
## Components and Layers

Logically speaking there are three components or layers to Script# system: the compiler, the core runtime, and the framework.

The Script# compiler and the Script# core runtime are closely coupled to one another. The compiler compiles the C# code for a target type system and base class library which is defined by the Script# core runtime.

The core runtime is provided in the form of sscorlib.dll which must be referenced by C# projects. Its equivalent script, sscorlib.js must be referenced by Web applications. C# projects that need to be translated to script should not reference mscorlib.dll. Instead they should reference sscorlib.dll, which provides the equivalent for key types in the System namespace such as Object, Int32, String, Array, Type etc. tuned for what is available at runtime in script. The associated script file extends the core set of JavaScript types such as String, Number, Function etc. so that they provide APIs that model what the C# or .NET programmer would expect.

In addition to sscorlib.dll, the Script# system also contains other assemblies such as sswpfe.dll. The latter provides the metadata necessary to program against the WPF/E browser plugin which enables programming against the scriptable media and vector graphics capabilities offered by the plugin.




The Script# Framework provides higher level concepts and abstractions that make application authoring more productive, and provides a way for structuring UI code and attaching functionality to the HTML DOM in more robust manner. The Script# framework itself follows a layered design. It consists of a Core (ssfx.Core.dll) which provides key capabilities such as an Application class, script loading, and a networking stack, JSON serialization and other key features useful generically across a broad set of applications. These key features include common UI concepts such as Controls, Behaviors, animations and drag/drop that are needed for UI components to integrate with one another. On top of this core framework are a number of independent components such as Form controls including AutoComplete functionality

(ssfx.UI.Forms.dll), an implementation of cross-domain Ajax using JSON, JSONP and script elements (ssfx.XDAjax.dll) and Reflection (ssfx.Reflection.dll).

Each assembly is associated with a JavaScript file. The assemblies are used at compilation time. The JavaScript files are sent down to the browser and used at runtime.

The compiler and the core runtime are however completely decoupled from the Script# framework. As a result, it is entirely possible to use Script# without buying into the rest of the framework. This is especially useful if you have an alternative framework design you'd like to implement, or have unique requirements.

## Script Runtime Choice

The Script# compiler can be used with one of two runtimes.

**sscorlib**: This assembly is the native script runtime of choice that is specifically designed for Script#. It provides a type system, BCL-like extensions to native script objects, as well as runtime APIs to support various compiler features. This enables the best C# development experience, and associated framework APIs.

**aacorlib**: This assembly is a trimmed down version of sscorlib that can be used to target the Microsoft ASP.NET Ajax runtime. It is a subset of sscorlib. Some compiler features are disabled when using aacorlib as the runtime, since Microsoft ASP.NET Ajax provides a subset of the functionality available with sscorlib.

## Using Script#

The Script# compiler can be from the command-line either directly by running ssc.exe or indirectly by building a .csproj using msbuild. In the latter case, the .csproj includes a reference to the Script# msbuild .targets file (nStuff.ScriptSharp.targets), which includes references to a Script# msbuild task that is responsible for invoking the Script# compiler programmatically.

It is recommended that you use the msbuild approach rather than using ssc.exe directly, as this provides the full benefits of a C# project in the Visual Studio IDE.

### Script Components and Libraries

You can choose to create a Script# Class Library project when the resulting script will be generically useful in a variety of applications (eg: a Grid component, or a Clock control). In this mode, the resulting script files (release and debug) compiled by the Script# compiler is included into the application, and the resulting assembly compiled by the C# compiler is included as a reference in other Script# projects.

Component developers are encouraged to distribute the generated script files, as well as the .NET assembly and the associated doc-comment file. The .NET assembly provides metadata that can be included as a reference, so users can write and compile their applications and components in Script#. The XML doc-comments are used by the code editor engine in Visual Studio to enhance the intellisense experience.

The generated script code can be referenced by hand-coded JavaScript, from other components compiled using Script#, or included into ASP.NET server controls in order to implement their client-side functionality. Script# can be used in all of these scenarios. This model of using Script# offers the maximum flexibility since the generated script can be used in any manner. In particular it can be used to create frameworks, as well as redistributable components that can then plug into those frameworks.

An assembly meant for use with Script# needs to have a the ScriptAssembly metadata attribute. Script components and libraries should contain the following metadata:

```
using System;
[assembly: ScriptAssembly]
```

The Script# install provides a Script# Class Library project template that can be used to start developing your components and libraries. The project includes a reference to sscorlib.dll and sets the NoStdLib property to true to automatically avoid including a reference to mscorlib.dll. You can add references to other Script# assemblies (both the ones the ship with Script#, or custom ones you've developed or acquired) via the standard Add Reference dialog in Visual Studio. The template also includes the assembly metadata attribute.

The Script# Framework is itself written in C# and compiled into a set of assemblies (ssfx.*.dll) using the C# compiler and the associated set of script files (ssfx.*.js and ssfx.*.debug.js) using the Script# compiler.

### Scriptlets

Scriptlets are the equivalent of executable applications. They are used to implement code-behind for a Web page. Unlike components and script libraries, scriptlets are usually specific to a particular page, and often work directly against the HTML elements present on the particular page they are associated with.

In concrete programmatic terms, a Scriptlet is a class that has a public static Main method. The signature of a Scriptlet is slightly different. Rather than taking in an array of string arguments as a parameter, a Scriptlet enables more advanced scenarios by accepting either a Dictionary of parameters, or an instance of ScriptletArguments which is essentially a dictionary, but offers strong typing.

While a Scriptlet can be easily used in any HTML page, it is especially geared toward ASP.NET scenarios where a Scriptlet server control allows you to implement your client-side code-behind in C# that is converted to JavaScript dynamically, as well as provide a number of other facilities.

For example, here is an example Scriptlet control.

```
<ssfx:Scriptlet runat="server" ID="scriptlet" EnableDebugging="true">
  <Arguments>
    <ssfx:StringLiteral Name="requestURLFormat"
      Value="Hello.ashx?name={0}" />
  </Arguments>
  <References>
    <ssfx:AssemblyReference Name="sscorlib" />
    <ssfx:AssemblyReference Name="ssfx.Core" />
  </References>
  <Code>
```

```
  ...
  </Code>
</ssfx:Scriptlet>
```

The Scriptlet server control offers a design-time experience that enables editing the C #code with intellisense and other aspects of the regular C #code editor, as well as the ability to compile and check the code for errors before running the page.

The Scriptlet control converts your code to script dynamically at runtime. Its PrecompiledScriptlet property can be set instead of specifying the code inline.

The scriptlet also generates some boilerplate code to bootstrap your code by loading in scripts corresponding to the references listed in the References collection. In addition it passes an object containing argument name/value pairs into the Main method of your code.

Scriptlets can be included in any HTML page manually. They can be precompiled into a script assembly and the corresponding generated script can be included into the page. The generated boilerplate bootstrapping script can be included manually into the page.

## Limitations

### C# Limitations

Script# is not intended to take an arbitrary existing C# application, and convert it to script to run within the browser. It does not attempt to provide a script implementation for the full .NET framework (eg. things like Windows Forms or the entire BCL). Doing so would not be practical or scale to the runtime scripting environment. Script# targets a subset of the C# language as implemented in .NET Framework 2.0. The idea is that Script# is very much about script development, but in a manner that benefits from an overall better tooling and authoring support.

The following are the set of unsupported C# constructs:

- All types must belong to a namespace. Nested types are not allowed.
- Nested namespace declarations are not allowed. Instead you must declare the whole namespace in one location.
- The “System” namespace can only be used for imported types representing native scriptable objects or existing script types.
- The set of reserved words that cannot be used in Script# include not only C# reserved words, but also JavaScript’s reserved keywords.
- Struct types may have just a constructor and some fields. Methods and properties are disallowed.
- Pointer types are disallowed.
- Method and constructor overloads are not supported except in imported types.
- Destructors, operators and object conversion are not supported.
- Enumeration fields must have an explicit value.

- Set-only properties are not supported.
- “new” modifiers are not supported on members.
- Throw statements must be associated with an explicit object.
- Struct types are not supported (instead use [Record] metadata attribute on sealed classes).
- Unsupported statements: goto, using scope statement, lock/unlock, and yield.
- Unsupported expressions: sizeof, fixed, stackalloc, and default value.

Some of these limitations listed above do not apply to creating System assemblies. System assemblies were covered in the conceptual overview, and are covered in more depth later on.

In addition there are a known set of limitations in the current implementation that will hopefully be supported in future builds:

- You cannot specify namespaced-qualified type names either. As a workaround you can use aliases (eg. using Foo = SomeNamespace.SomeType;)
- Generics
- Support for ref, out and params modifiers on parameters
- Miscellaneous others (to be documented)

### JavaScript Limitations

Script# provides support for the majority of JavaScript constructs needed to write real world applications and frameworks. It does however have some limitations, and some small differences in the authoring model as a result of being grounded in C# and OOP.

- Limited support for closures
  Closures are used for a few scenarios. They may be used to define classes, and implement encapsulation of member variables and implementation private to the class. They are also used to implement callback methods, which have access to data/variables present in the outer scope. While Script# supports the second scenario using anonymous delegates, the first scenario is not supported. The generated types do not use closures as their implementation. Instead Script# types are implemented using the more natural JavaScript prototype-based model.
- Functional Programming
  Script# provides an OOP-style using C#, and hence does not provide a more functional style of programming that can otherwise be used when working with JavaScript. Script# does support functional programming such as array comprehensions using delegates as a mechanism to represent functions that can be passed as parameters to various APIs.
- Global Methods
  The script engine offers a set of global methods. C# however has no concept of global methods. The interesting set of global methods and associated functionality has been surfaced as statics on various classes such as System.Script. Script# does allow generation of global methods.
- Identifier Characters
  JavaScript allows the use of “$” within an identifier. This is not supported in C#. In fact the

generator makes use of "$" in generated identifier names to ensure they do not conflict with your own identifiers.

As you'll see the limitations have minimal impact. The limitations should not prevent you from implementing any real Ajax scenarios, or from leveraging the full capabilities of the DOM.

## How Do I Accomplish a Particular Script Scenario?

This section provides a few of the common scripting scenarios that are somewhat unique to script relative to C#. It also presents the C# abstractions or mechanisms supported by the compiler to enable you to target those script scenarios.[5]

### Using *eval* to Execute Code and Perform JSON Deserialization?

Script has a global eval function that allows you to execute JavaScript code or deserialize JavaScript objects from their string representation.

However, global methods are not permitted in C#. Therefore, this functionality is exposed to C# developers via the Script.Eval method instead. Script.Eval is however not a runtime abstraction. Instead the compiler transforms this into the native eval method in generated script.

```
// C#
string code = ...;
object result = Script.Eval(code);

// Generated Script
var code = ...;
var result = eval(code);
```

One of the key uses of eval is to parse JSON literals as part of deserialization. The Script.Eval method can be used here as shown.

```
// C#
object data = Script.Eval("[ { name: 'abc', value: 123 } ]");
```

### Using alert, prompt and Related Methods

Some of the common Script global methods such as alert, prompt, and confirm are in reality methods on the DHTML window object. Just like the eval method, the Script class contains these APIs that are transformed by the compiler into their typical script usage.

```
// C#
Script.Alert(msg);

// Generated Script
alert(msg);
```

[5] If there are other interesting script scenarios not covered in this document, please do send feedback.

## Performing Late-bound Member Access

Script allows accessing fields or invoking methods off any object in a late-bound manner. C# on the other hand does not support late-bound code. Hence Script# provides a set of APIs that allow you to author explicit late-bound code that is then turned into implicit late-bound script at generation time.

Script# also introduces the notion of properties which are implemented via get/set accessor methods, and the notion of events which are implemented via add/remove accessor methods. The late-bound access extends to these higher level concepts as well.

As shown in the examples, you can use late-bound access where the member name is a constant at compile time, or is a variable. Both models work, and generate the minimal script making full use of script's dynamic nature.

```
// C#
object o = ...;
string fieldName = ...;
object value = Type.GetField(o, “aaa”);
Type.SetField(o, “aaa”, value);
Type.SetField(o, fieldName, value);

// Generated Script
var o = ...;
var fieldName = ...;
var value = o.aaa;
o.aaa = value;
o[fieldName] = value;

// C#
object o = ...;
string methodName = ...;
object result = Type.InvokeMethod(o, “doFoo”, param1, param2);
Type.InvokeMethod(o, methodName, param1);

// Generated Script
var o = ...;
var result = o.doFoo(param1, param2);
o[methodName](param1);

// C#
object o = ...;
string propName = ...;
object value = Type.GetProperty(o, “foo”);
Type.SetProperty(o, “foo”, value);
Type.SetProperty(o, propName, value);

// Generated Script
var o = ...;
var propName = ...;
var value = o.get_foo();
o.set_foo(value);
o[‘set_’ + propName](value);
```

## Deleting a Field from an Object

The delete operator in script allows deleting a member field off an object. This is different from simply setting the field's value to null.

```
// C#
object o = ...;
Type.DeleteField(o, "xyz");

// Generated Script
var o = ...;
delete o.xyz;
```

## Enumerating Members of an Object

Script allows the use of a "for" statement to enumerate members of an arbitrary object. Script# enables you to do so via a foreach member that enumerates the set of DictionaryEntry objects in a Dictionary. Dictionary itself represents an arbitrary, plain JavaScript object.

```
// C#
Dictionary d = ...;
foreach (DictionaryEntry entry in d) {
    // Use entry.Key and entry.Value
}

// Generated Script
for (var $key in d) {
    var entry = { key: $key, value: d[$key] };
}
```

Script in fact allows you to enumerate the members of any object, and not just plain JavaScript objects. In order to support this scenario, Script# allows you to create a Dictionary from any object in your C# code (this simply becomes a no-op in generated script) which you can then enumerate as described above.

```
// C#
MyClass c = ...;
foreach (DictionaryEntry entry in Dictionary.GetDictionary(c)) {
}

// Generated Script
var c = ...;
for (var $key in c) {
    var entry = { key: $key, value: c[$key] };
}
```

## Retrieving the Native Script Type of an Object

The script type of an object is the type of an object as tracked by the script engine. It may be "Object", "Number", "Boolean", "String", "Function" or "undefined". This is different from the GetType() method on object which returns the specific type of class associated with the instance.

```
// C#
```

```
object o = ...;
if (Type.GetScriptType(o) == “undefined”) { ... }

// Generated Script
var o = ...;
if (typeof(o) == ‘undefined’) { ... }
```

## Defining and Implementing Global Methods

While global methods are strongly discouraged, sometimes you simply need them. The need can arise when you want to define a system assembly and need to represent existing global APIs, or when you must generate a global method that can interop with other existing script, or if you want to generate an event handler you are going to hook up to an element’s event via an attribute within the HTML markup.

You can create a static class and annotate the class with the GlobalMembers metadata attribute. All methods and properties of this class are promoted as top-level global members. Such a class may not contain fields, or events.

```
// C#
[GlobalMethods]
public static class PageImplementation {

    public static void OnBodyLoad() {
    }
}

// Generated Script
function onBodyLoad() {
}
```

## Invoking Global Methods

You can use the technique described above to define a class with static methods to represent global methods, and use those static APIs to invoke them. If you need to invoke global methods in a late-bound manner, where you only know the name of the method, you can use Type.InvokeMethod (which is used to perform late-bound access to member methods as described earlier).

```
// C#
object result = Type.InvokeMethod(null, “doFoo”, param1, param2);

// Generated Script
var result = doFoo(param1, param2);
```

## Defining Nested Functions to Implement Closures

Nested functions and closures are modeled using anonymous delegates in C#.

```
// C#
public delegate void XyzDelegate(int i);

public class MyClass {
    private int _data;
```

```
    public void MyMethod(object o) {
        int localData = 0;

        DoStuff(o, delegate(int i) {
            _data = localData + i + MyClass.GlobalData;
        });
    }

    public void DoStuff(object o, XyzDelegate callback) {
        // Some code, including code that invokes the delegate passed in
    }
}

// Generated Script
MyClass = function() {
}
MyClass.prototype = {
    _data: 0,
    myMethod: function(o) {
        var localData = 0;
        nStuff.ScriptSharp.Tests.MyClass.doStuff(o,
            new Delegate(this, function(i) {
                this._data = localData + i + MyClass.GlobalData;
            }));
    },
    doStuff: function(o, d) {
        // Some code, including code that invokes the delegate passed in
    }
}

MyClass.createClass('MyClass');
```

## Creating and Using Plain Script or JSON Objects

The first thing to clarify is what is a plain script object. A plain script object is essentially an instance of Object, rather than any specific class. The runtime type of such objects (as returned by the typeof script operator) is literally “Object.”

Such objects are modeled via sealed classes with the [Record] metadata attribute in C# as shown below:

```
// C#
namespace UI {
    [Record]
    public sealed class Point {
        public int x;
        public int y;
        public Point(int x, int y) {
            this.x = x;
            this.y = y;
        }
    }
}

Point p = new Point(10, 100);

// Generated Script
Type.createNamespace(‘UI’);

UI.$create_Point = function(x, y) {
```

```
    var $o = { };
    $o.x = x;
    $o.y = y;
    return $o;
}

var p = UI.$create_Point(10, 100);
```

As you can see the Point object instantiated is just a plain-script object. In essence a plain-script object is essentially equivalent to an associative collection of name/value pairs, aka, a Dictionary.

One interesting scenario where plain script objects come into play is objects deserialized from a JSON string. However, deserialized Dictionaries aren't the friendliest thing to program against, esp. in a strongly typed language such as C#, as will be evident from the following example. You can use records to define a shape/type structure to enable friendly programming against raw data.

```
// C#
// Without Structs
string data = '{ x: 10, y: 100 }';
Dictionary d = (Dictionary)ScriptFX.JSON.Deserialize(data);
int x = (int)d["x"];

// With Structs
Point p = (Point)ScriptFX.JSON.Deserialize(data);
int x = p.x;
```

## Creating Plain JSON Objects

Sometimes you need to create a script object with a certain set of name/value pairs. Often this is referred to as a plain script object. While you can use the technique described above when you have a fixed and well-known set of name/value pairs, sometimes you don't know the fields in advance. Such objects are represented as a Dictionary object in C#.

The Dictionary object supports creation of an object which you can add fields to. It also provides a special constructor that takes a sequence of name/value pairs to create an object literal in script.

```
// C#
Dictionary d1 = new Dictionary();
d1["abc"] = 123;
d1["xyz"] = true;
d1["foo"] = "bar";

Dictionary d2 = new Dictionary("abc", 123, "xyz", true);
d2["foo"] = "bar";

// Generated Script
var d1 = {};
d1['abc'] = 123;
d1['xyz'] = true;
d1['foo'] = 'bar'

var d2 = { abc: 123, xyz: true };
d2['foo'] = 'bar';
```

### Checking for Undefined

Often times it is needed to check if a variable has a value, i.e. it is not undefined. Also frequent is to check if the variable is null or undefined, and not incorrectly identify 0, false, or empty strings as null or undefined. Script# provides methods on the Script class for this purpose. These include IsNull, IsNullOrUndefined, and IsUndefined.

```
// C#
void DoSomething(string o) {
    if (Script.IsNullOrUndefined(o)) {
        return “”;
    }
}

// Generated Script
function DoSomething(o) {
    // This test equates to:
    // if ((o === null) || (o === undefined))
    if (isNullOrUndefined(o)) {
        return “”;
    }
}
```

## How is a Particular C# Feature Modeled in Script?

This section provides information about how various C# constructs and patterns are modeled in script. The goal is to minimize the abstraction level, while bringing forward the useful C#-isms[6].

### How is a Namespace Defined?

Namespaces are simulated in script via an object off of the global window object.

```
// C#
namespace Sample.Components {
}

// Generated Script
Type.createNamespace(‘Sample.Components’);
```

### How is a Class Defined?

A script function is used to define types (and their associated constructors), and type members are defined on the prototype associated with the function. The function is registered as a class, which enables the core type system to perform necessary initialization of the type.

```
// C#
namespace Demo {
    public class Person {
         private string _name;
         public Person(string name) { ... }

         public string Name {
```

[6] Please send feedback if there are additional interesting C# constructs and patterns to cover.

```
            get { ... }
        }
    }
}

// Generated Script
Demo.Person = function(name) {
    ...
}
Demo.Person.prototype = {
    _name: null,
    get_Name: function() { ... }
};
Demo.Person.createClass('Demo.Person');
```

Note that various access modifiers like internal, public, sealed, abstract etc. do not manifest in the generated script. These are enforced at the C# source level, by the C# compiler.

## How is a Derived Class Defined?

The inheritance chain for derived classes is specified at class registration time. The type system implementation in sscorlib.js ensures that the derived type actually inherits members from the parent class, and sets up the chain, so that inspecting the class for base type information provides the expected result. The derived class constructor calls the base class constructor.

```
// C#
namespace Demo {
    public class Employee : Person {
        public Employee(string name) : base(name) { ... }
    }
}

// Generated Script
Demo.Employee = function(name) {
    Demo.Employee.initializeBase(this, [ name ]);
}
Demo.Employee.createClass('Demo.Employee', Demo.Person);
```

## How is an Interface Defined?

A script function is used to define interface types.

```
// C#
public interface IDisposable {
    void Dispose();
}

// Generated Script
var IDisposable = function() { }
IDisposable.createInterface('IDisposable');
```

Note that the members of an interface are not defined in generated script. This is because an interface in script is simply a marker on the type, that can be inspected at runtime. The full implementation of all interface members is enforced at the C# source level, by the C# compiler.

```
// C#
public class Control : IDisposable {
}

// Generated Script
var Control = function() {
}
Control.createClass('Control', null, IDisposable);
```

## How is a Delegate Type Defined?

Delegates can be used when authoring C# code, but have no script representation to define them. Only the functions used as methods passed around as delegates exist in the generated script. The signature of the delegate is enforced at the C# source level, by the C# compiler.

## How are Delegates Used?

Delegates can be created by passing in a reference to a function. The created delegated tracks the object instance and the function reference.

```
// C#
public class App {
    private void OnClick(object sender, EventArgs e) { ... }

    private void Main() {
        EventHandler handler = new EventHandler(this.OnClick);
    }
}

// Generated Script
App = function() {
}
App.prototype = {
    _onClick: function(sender, e) { ... },

    _main: function() {
        Var handler = Delegate.create(this, this._onClick);
    }
}
```

Once a delegate has been created, it can be passed off as an event handler or callback to another component. That component can manage a list of delegates. This is done using the Delegate.Combine and Delegate.Remove methods just like you'd expect.

```
// C#
public class Button {
    private EventHandler _clickHandler;

    public event EventHandler Click {
        add {
            _clickHandler = (EventHandler)Delegate.Combine(_clickHandler, value);
        }
        remove {
            _clickHandler = (EventHandler)Delegate.Remove(_clickHandler, value);
        }
    }
```

```
}

// Generated Script
Button.prototype = {
    _clickHandler: null,

    add_click: function(value) {
        this._clickHandler = Delegate.combine(this._clickHandler, value);
    },
    remove_click: function(value) {
        this._clickHandler = Delegate.remove(this._clickHandler, value);
    }
}
```

Finally, a delegate can be invoked, which invokes the sequence of delegates referenced by the delegate instance. Each function is automatically instantiated using the instance of the object bound to a particular delegate instance.

```
// C#
public class Button {
    protected virtual void OnClick(EventArgs e) {
        if (_clickHandler != null) {
            _clickHandler(this, e);
        }
    }
}

// Generated Script
Button.prototype = {
    onClick: function(e) {
        if (this._clickHandler) {
            this._clickHandler(this, e);
        }
    }
}
```

## How are Enumerations Defined and Consumed?

Enumerations are simulated by an object with named members corresponding to each enumeration field. The type system implemented in sscorlib.js provides an API to define enumerations, as well as conversions of enum values to/from their corresponding string representations.

```
// C#
public enum Colors { Red = 0, Green = 1, Blue = 2 }

[Flags]
public enum Options { OptionA = 1, OptionB = 2, OptionC = 4 }

Colors c = Colors.Red;
Options o = Options.OptionA | Options.OptionB;

// Generated Script
var Colors = Type.createEnum(false, 'Colors', { Red: 0, Green: 1, Blue: 2 });
var Options = Type.createEnum(true, 'Options',
                              { OptionA: 1, OptionB: 2, OptionC: 4 });

var c = Colors.Red;
```

```
var o = Options.OptionA | Options.OptionB;
```

Note that in debug and non-minimized builds enumeration fields are referenced explicitly. However in the interest of saving script size, the compiler substitutes field references with their numeric values when minimization is turned on. Consequently, all type information associated with enumeration values is lost, and their runtime type is the same as that of an integer or number. Therefore it is recommended that you do not retrieve the runtime type of enumeration field values. For example, the minimized variant of the above code would be as follows:

```
var Colors = Type.createEnum(false, 'Colors', { Red: 0, Green: 1, Blue: 2 });
var Options = Type.createEnum(true, 'Options',
                              { OptionA: 1, OptionB: 2, OptionC: 4 });

var c = 0;
var o = 1 | 2;
```

## How are Properties Declared and Accessed?

Properties are modeled as a pair of get/set accessor methods. A naming convention using "get_" and "set_" prefixes is used to define and call these accessors.

```
// C#
namespace Demo {
    public class Person {
         private string _name;
         public string Name {
             get { return _name }
             set { _name = value; }
         }
    }
}

Person p;
p.Name = "Nikhil Kothari";

// Generated Script
Demo.Person = function() {
}
Demo.Person.prototype = {
    _name: null,
    get_name: function() { return this._name; }
    set_name: function(value) { this._name = value; }
};
Demo.Person.createClass('Demo.Person');

var p;
p.set_name('Nikhil Kothari');
```

## How are Indexers Declared and Accessed?

Indexers are modeled as a pair of get/set accessor methods like properties, with a special name "item" (which matches the CLR). The index parameters become parameters to the accessors as you might expect.

```
// C#
namespace Demo {
    public class Set {
        public object this[string name] {
            get { ... }
            set { ... }
        }
    }
}

Set s;
s["abc"] = object1;
object o = s["abc"];

// Generated Script
Demo.Set = function() {
}
Demo.Set.prototype = {
    get_item: function(name) { ... }
    set_item: function(name, value) { ... }
};
Demo.Set.createClass('Demo.Set');

var s;
s.set_item('abc', object1);
var o = s.get_item('abc');
```

## How are Events Declared and Accessed?

Events are modeled as a pair of add/remove accessor methods. A naming convention using "add_" and "remove_" prefixes is used to define the pair of accessors.

The accessors can either be explicitly defined, or they can be auto-generated by the compiler for a field event as shown below.

```
// C#
public class Button {
    public event EventHandler Click {
        add { ... }
        remove { ... }
    }
}
public class Timer {
    public event EventHandler Tick;
}

// Generated Script
Button.prototype = {
    add_click: function(value) {
        ...
    },
    remove_click: function(value) {
        ...
    }
}
Timer.prototype = {
    __tick: null,

    add_tick: function(value) {
```

```
        __tick = Delegate.combine(this.__tick, value);
    }
    remove_tick: function(value) {
        __tick = Delegate.remove(this.__tick, value);
    }
}
```

## How are Static Members Declared and Accessed?

Static members are defined on the class instead of on the class prototype. In addition to static methods, Script# also enables you to write static constructors.

```
// C#
public class Application {
    private static Application Current;
    static Application() {
        Current = new Application();
    }

    public static Application GetCurrent() {
        return Current;
    }
}

// Generated Script
Application = function() {
}
Application.prototype = {
    /* instance members */
}
Application.createClass('Application');

/* Static Methods */
Application.getCurrent = function() {
    return Application._current;
}

/* Static Ctors run as the script file loads */
Application._current = new Application();
```

## How is a foreach Statement Implemented?

You can use foreach to enumerate IEnumerable objects such as Arrays, Dictionaries, and custom types implementing this interface. Script# distinguishes models two forms of foreach statements: 1) where the item being enumerated is of type DictionaryEntry and 2) all other enumerations.

```
// C#
int[] items;
Dictionary table;
foreach (int i in items) {
    // Consume i
}
foreach (DictionaryEntry entry in table) {
    // Consume entry, i.e. entry.Key and entry.Value
}

// Generated Script
```

```
var items;
var table;
var $var1 = items.getEnumertor();
while ($var1.moveNext()) {
    var i = $var1.get_current();
    // Consume i
}
var $dict1 = table;
for (var $key in $dict1) {
    var entry = { key: $key, value: $dict1[$key] };
    // Consume entry, i.e. entry.key and entry.value
}
```

You can build support for enumeration into your own classes by implementing IEnumerable just like you would in C#.

## How are Anonymous Delegates Implemented

Anonymous delegates are implemented using JavaScript closures. A closure is represented by a nested function that inherits the context of the outer function.

Anonymous delegates within static members do not have context to the member variables of the class, vs. anonymous delegates defined within an instance member do have access to member variables.

```
// C#
public delegate void XyzDelegate(int i);

public class MyClass {
    private static int GlobalData = 0;
    private int _data;

    public static void StaticMethod(object o) {
        int localData = 0;

        DoStuffStatic(o, delegate(int i) {
            localData = i + MyClass.GlobalData;
        });
    }

    private static void DoStuffStatic(object o, XyzDelegate d) {
        // Some code, including code that invokes the delegate passed in
    }

    public void InstanceMethod(object o) {
        int localData = 0;

        DoStuffInstance(o, delegate(int i) {
            _data = localData + i + MyClass.GlobalData;
        });
    }

    public void DoStuffInstance(object o, XyzDelegate d) {
        // Some code, including code that invokes the delegate passed in
    }
}

// Generated Script
MyClass = function() {
```

```
}
MyClass.staticMethod = function(o) {
    var localData = 0;
    MyClass.doStuffStatic(o, new Delegate(null, function(i) {
        localData = i + MyClass.GlobalData;
    }));
}
MyClass.doStuffStatic = function(o, d) {
    // Some code, including code that invokes the delegate passed in
}
MyClass.prototype = {
    _data: 0,
    instanceMethod: function(o) {
        var localData = 0;
        doStuffInstance(o, new Delegate(this, function(i) {
            this._data = localData + i + MyClass.GlobalData;
        }));
    },
    doStuffInstance: function(o, d) {
        // Some code, including code that invokes the delegate passed in
    }
}

MyClass.createClass('MyClass');
MyClass.GlobalData = 0;
```

The example shows the generation of nested functions, which use data local to the outer function or scope via a script closure. The example also shows how the nested method is packaged as a delegate to match C# semantics, and how the delegates are declared differently for anonymous methods within static and instance methods.

## Roadmap

Script# is an on-going project. There are some unfinished features, as well as various ideas for future development. The current set of builds are considered the v1 generation of the product. The v-next generation will commence once v1.0 of Script# will be completed.

If there is feedback about what you're looking for in any of these areas, please do send comments and suggestions.

### Compiler

The key incomplete features include support for generics, parameter modifiers (ref, out, params), and support for metadata generation into the resulting script. The plan for implementing this is in the v-next generation of the product.

Other plans for the future include generating instrumented code to enable code coverage measurement, profiling, static linking etc.

**Script# Core Runtime**
Some key features missing from the type system include a metadata system that can be used by classes to describe their properties, methods, and events so that higher level frameworks can inspect classes and their object models.

**Script# Framework**
This is where most of the work will take place in terms of both script libraries and system script assemblies.

Some of the functionality in terms of new script framework functionality include support for data-binding, completing the RPC stack, adding drag/drop support, adding a declarative markup model, completing the animation stack, and providing other useful UI infrastructure components such as a Calendar control and a RichTextBox control.

## Feedback

All feedback on Script# as well as on this document, and other help content is welcome. The project page at http://projects.nikhilk.net/projects/ScriptSharp.aspx includes discussion forums and a bug tracking list. In order to send comments, questions, bug reports and other feedback offline, please send it via my contact form at http://www.nikhilk.net/Contact.aspx.

Please make sure to include “Script#” in the subject. Also please include sufficient details about what you are doing, what you are seeing, what you’d expect etc. (information that you think is relevant). Please do not send source code as-is. It will be deleted without being looked at. You may include relevant snippets, if they are relevant.

## Version History

| Version | Date | Notes |
|---|---|---|
| 0.1.0.0 | 5/22/2006 | Initial Release |
| 0.1.1.0 | 6/01/2006 | Following are the key changes based on dogfooding feedback, and some comments to initial blog post:<br>Feature Work<br>• Simplify scriptlet authoring (see the ScriptMain method)<br>• Add Scriptlet arguments and references in <nStuff:Scriptlet>server control<br>• Add EnableScriptDebugging property on Scriplet control. Selecting True sends down unminimized framework scripts (sscorlib.js and ssfxcore.js); setting to false (the default) sends down minimized scripts.<br>• Support Application and Library projects in IDE and command-line compiler<br>• Support for delete, typeof, >>>, and >>>= operators<br>• Support for calling global methods (eval, parseInt, isNaN, alert, etc.)<br>• Support for generating global functions<br>• Support for enumerating members off any object<br>• New methods on Array and String to improve programmability<br>• Added documentation guide<br>Bug Fixes/Incremental Changs<br>• Enum.ToString now works<br>• Enums with non-integer based underlying values now work<br>• Generate $() alias for calls to Document.GetElementById<br>• Fix Exception class properties<br>• Fix enumeration type definitions<br>• Enable running sscorlib.js and ssfxcore.js in Mozilla with 0 warnings in strict mode<br>• Include raw framework scripts in zip download |
| 0.1.2.0 | 6/25/2006 | Bug fixes from dogfooding feedback, comments, as well as some more feature work.<br>Feature Work<br>• Add some APIs to strings (eg. Quote, Unquote, TrimStart, TrimEnd etc.)<br>• Add ability to create types in a late-bound manner (Type.CreateInstance)<br>• Enable usage of global methods as delegates when subscribing to events<br>• Add XML DOM APIs as well as a good chunk of DHTML DOM APIs to sscorlib.<br>• Add JSON class to ScriptFX.Core<br>• Add basic networking stack to ScriptFX.Core.<br>• Add ScriptFX.Reflection.dll for higher level reflection APIs.<br>• Debug.Trace and Debug.Inspect use debugService when available. Web Development Helper provides a debug service to the page.<br>• Added basic JsonReader/JsonWriter for use in server-side code.<br>Bug Fixes<br>• Assemblies involved in project to project references got locked by msbuild/VS preventing further rebuilds – this is no longer the case.<br>• Fix Dictionary.GetDictionary to return a string<br>• Make +=, \|= etc. operators work on properties.<br>• Make structs work<br>• Default values for fields of non-integer and non-boolean types<br>• Fix event fields with auto-generated add/remove accessors actually work<br>• Fix Enum and Delegate implementations in sscorlib<br>• Fix retail build minimization bugs related to generated parameter names, and parameter references in methods.<br><br>One big change worth calling out specifically: The type system in sscorlib.js has been changed to co-exist with the Atlas type system; the two type systems will now not clobber and trample over each other; types defined within one type system however will not be perceived as types in the other type system. |
| 0.1.3.0 | 7/15/2006 | Bug fixes |

| | | |
|---|---|---|
| | | • Enable string concats such as 100 + "Hello"<br>• Enable InstanceOf checks on things like empty strings, 0 numeric values, false Boolean values<br>• Fix generation of structs without ctors<br>• Fix number parsing to use parseFloat conditionally<br><br>New features<br>• Added support for using alias stmts (using Foo = SomeNamespace.SomeType;) – these are useful given namespace-qualified types are not supported in Script#.<br>• Optimize away enum field references with actual runtime value in minimized release builds to reduce script size, and derive some small perf benefit with reduced lookups<br>• Enum creation syntax slightly modified for slightly improved performance<br>• Flesh out more of mozilla compat layer, and DOM objects such as Style<br>• Added InnerText and Children properties to DOMElement<br>• Added CultureInfo, string formatting<br>• Added string APIs such as PadLeft, PadRight, FromChar<br>• Added the ability to create global names for delegate instances (Delegate.CreateExport/DeleteExport)<br>• Optimize usage of global functions as eventhandlers wrapped with a delegate so there is no overhead of a delegate instance.<br>• Add Browser detection logic and capability to Application class<br>• Added support for cross-domain Ajax calls (ScriptFX.XDAjax.dll) using script elements and callbacks (ala JSONP); Added BookmarkScriptlet sample working against del.icio.us' APIs. |
| 0.1.3.1 | 7/16/2006 | Quick bug fix<br>• Introduced DOMDocumentFragment in sscorlib.dll. Document.CreateDocumentFragment now returns a DOMDocumentFragment instead of DOMDocument. |
| 0.1.3.2 | 7/17/2006 | More quick bug fixes<br>• String and number formatting<br>• Syntax errors in release flavor |
| 0.1.4.0 | 7/23/2006 | New Features<br>• Support for conversion of C# anonymous delegates into nested script functions added<br>• Support for browser-based CSS selectors (HTML.IE, HTML.IE7, HTML.Firefox etc.)<br>• Added ScriptFX.UI.DOMEventList<br>• Added Script.IsNull, IsNullOrUndefined, IsUndefined, to allow testing of local variables against null or undefined<br>• Added various APIs on String: IndexOfAny, LastIndexOfAny, Compare,<br>• Added various APIs on Array: Sort(callback), Map, ForEach, Filter, Some, Every<br>• CompareTo, Insert, Remove, Replace, HTMLEncode, HTMLDecode<br>• Allow c#-style exception catch handlers without an explicit exception variable<br>• Add HTTPRequest.SetContentAsForm to simplify simulating form post requests<br>• Add basic OM for filters/transitions (Filters collection on DOMElement, VisualFilter and related classes)<br>• Add ActiveXObject class<br>Bug Fixes<br>• Fixes to JSON serialization related to serializing 0, false, and empty string |
| 0.1.4.1 | 8/3/2006 | Simplified usage of Delegates with intrinsic script objects – no need for calling Delegate.Unwrap anymore<br>APIs with Delegate parameter take a strongly typed delegate instead of untyped Function object.<br>Fixed Delegate.Remove<br>Replaced Function.Empty with Delegate.Null |
| 0.1.5.0 | 8/30/2006 | New Features/Changes<br>• UI Framework start: Behavior, Control<br>• AutoComplete, PopupBehavior<br>• Initial bits of RESTful services and REST-based RPC stack<br>• HTTPRequest.SetContentAsJSON, .CreateURI |

| | | |
|---|---|---|
| | | • Add XMLNodeType and DocumentElementType enums<br>• Changed Debug.trace to Debug.WriteLine for .NET consistency<br>• Core UI framework is now a separate assembly/script file<br>Bug Fixes<br>• Fixed HTTPRequest.SetContentAsForm<br>• Fixed BooleanLiteral rendering<br>• Fixed importing of struct types from another assembly reference<br>• Fixed delegate bugs related to multiple removing/recombining scenarios<br>• Fixed Application unloading/unload events<br>• Fixed JSON serialization of null<br>• Fixed IArray implementation on Array<br>• Fixed DOMEventList to allow multiple calls to Dispose<br>• Fixed script generation and JSON serialization of strings containing Unicode characters<br>• Fixed minimization bugs<br>• Handle similar named private methods in parent and derived class by disambiguating them in generated script |
| 0.1.6.0 | 9/10/2006 | New Features<br>• Added Session, and HistoryManager classes to add support for the back/forward buttons, and logical bookmarking<br>• Added Animation base classes to support atomic animations, repeating animations, sequences etc. as well as an AnimationManager that bases animation progress on frames/second setting.<br>• Updated samples to show usage of history and animation<br>• Added ScriptLoader to load scripts in sequence and parallel (not yet used; just prep work for new scriptlet model)<br>Bug Fixes<br>• Fixes related to script minimization in release builds for interface members<br>• Fixes related to minimization of protected internal members |
| 0.2.0.0 | 11/27/2006 | Support for WPF/E scripting<br><br>New Features and changes<br>• Rearchitected Scriptlet scenario to allow for scriptlet code within .aspx, or as a .scriptlet file within the Web project rather than as a separate class library project.<br>• Changes in Script# class library project to build both release and debug versions of scripts.<br>• Named enums added which allow defining string constants.<br>• Support for including header text (eg. Copyright info) into generated script files. Generated script files now contain Script# information in the footer.<br><br>Bug Fixes and minor changes<br>• Fixes related to minimization of internal delegate types and array types<br>• Fixes related to disposing controls<br>• Fix for Firefox version detection<br>• Support for rgb() color parsing<br>• Support for app unload prompts<br>• Added DateTime.Now and DateTime.Today<br>• Enhancements to AutoComplete to support additional service parameters, client-side events, and ability to return complex objects from the service.<br>• Changes to HTTPRequest.CreateRequest to support URI format with embedded HTTPTransport selection.<br>• Assembly names and script files should now be in sync.<br><br>Type system implementation now compatible with Microsoft ASP.NET Ajax. A type defined within Script# now appears as a type defined in Atlas and vice versa. Furthermore it is possible to derive a type in Script# from an Atlas type (assuming there is a system script assembly providing metadata representing Atlas types). |

| | | |
|---|---|---|
| 0.2.1.0 | 12/20/2006 | Quick incremental release primarily to fix bugs<br>Framework Updates<br>• Fixed implementation of GetXML () method on HTTP response implementation<br>• Added Request property on IHTTPResponse<br>• Added Aggregate, Group and ToDictionary methods on Array<br>• Added String.Replace overload that takes in callback<br>• Fixed number formatting of negative numbers<br>• Added EventManager, IServiceContainer functionality on ScriptFX.Application<br>Compiler Updates<br>• Added Indexer property to ScriptletArguments type, to make it behave as a dictionary in addition to providing strongly typed named parameters<br>• Fixed minimization bug associated with anonymous delegate parameters<br>• Fixed compiler parsing of double literals without leading digits<br>• Added a feature to the compiler to allow quick emitting of JSON objects (described earlier in the document – using new Dictionary(name1, value1, … nameN, valueN) syntax<br>• Detect usage of reserved words in member names correctly based on camel-casing during script generation<br>IDE Integration Updates<br>• Added a Script# class item template that is now available in the Add Item dialog, so that references to .NET assemblies are not added to a Script# project |
| 0.2.1.1 | 12/27/2006 | Support for WPF/E scripting<br><br>Changes<br>• Structs are no longer allowed. Instead use sealed classes annotated with [Record]. This helps preserve script reference type semantics.<br><br>Fixes<br>• Implement fixes to date encoding in JSON, as well as added support for Unicode sequences.<br>• Fixed scriptlets (broken in previous build) |
| 0.2.2.0 | 1/8/2007 | New<br>• Added support for Virtual Earth (ssve4.dll) along with a basic sample (map.aspx)<br>• Add fixed size arrays (new int[5])<br>• Added ArrayList – for variable sized array scenarios<br>• Added Dictionary.Count, Clear, Remove, ContainsKey<br>• When building using msbuild, scripts can now be generated in a different directory than c# assembly (using the ScriptPath property on the ScriptSharpTask) – this enables placing C# projects under bin\Script in Web sites, and having scripts generated into App_Scripts<br><br>Fixes:<br>• Fixed Type.IsAssignableFrom to take in a Type and match .NET signature<br>• Fixed casing of XML property on XMLNode<br>• Fixed crash when trying to define operator methods (now raises an error as expected) |
| 0.2.3.0 | 2/12/2007 | Assembly refactoring:<br>• ssfx.UI.Core.dll is no more; its been merged into ssfx.UI.Core to simplify the assembly layering (there is essentially one core now).<br>• ssfx.UI.AutoComplete.dll is also no more; the functionality has been moved into ssfx.UI.Forms.dll along with other core forms/controls features.<br><br>Changes/Updates:<br>• Use of === and !== instead of == and != in generated script to match C# equality semantics<br>• Added some metadata for some DOM classes (click, select, scrollIntoView, pixel???), some compat APIs<br>• Enable calling global methods using Type.InvokeMethod (using null as the first param) |

| | | |
|---|---|---|
| | | • Added String.Equals, Math.Truncate<br>• Removed obsolete APIs off Array (use ArrayList for those APIs)<br>• Changed DictionaryEntry.Key from Object to String to match script semantics<br><br>New:<br>• OverlayBehavior to create translucent overlays for modal dialog scenarios<br>• ssgadgets.dll – metadata for Sidebar gadget APIs<br>• ssfso.dll – metadata for Scripting File System Object<br>• New Sidebar gadget project template<br><br>Fixes<br>• Fixed the bug preventing passing delegates as ctor parameters<br>• Fixed the bug preventing anonymous delegates from accessing variables defined within a nested block inside a function<br>• Return error codes from ssc.exe on failed compiles<br>• Fixes to debug.inspect |
| 0.2.3.2 | 2/13/2007 | Fix some project template bugs in gadget and web site templates. |
| 0.3.0.0 | 5/21/2007 | Introduction of a Microsoft ASP.NET Ajax compatible mode, by referencing aacorlib.dll instead of sscorlib.dll. This runs the compiler in reduced functionality mode such that the generated script only depends on MicrosoftAjax.js.<br><br>Removed sswpfe.dll and added ssagctrl.dll. This is synchronized with the Microsoft Silverlight 1.0 Beta build. Also added is the aaagctrl.dll, which is the equivalent dll for that works on top of aacorlib.dll.<br><br>Removed the distinction between regular and system assemblies. Any assembly can now contain a mix of application types and imported types. Application types contain code that is converted to script. Imported types are skipped from conversion, but can be used to represent native scriptable APIs and existing script libraries.<br><br>Lots of fixes, and some additional metadata for DHTML objects.<br><br>Addition of ssfeeds.dll to represent the IE7 RSS Feeds API. |
| 0.4.0.0 | 8/29/2007 | Add support for Silverlight 1.0.<br>Add support for strongly typed attached properties for Silverlight objects.<br>Add support for credentials to HTTPRequest and Online property on HTTPRequestManager in the ScriptFX framework<br>Misc. metadata additions and fixed for DHTML DOM.<br><br>Bug fixes:<br>• Added support for ?? operator.<br>• Fixed ^= operator.<br>• Fixed usage of byte types.<br>• Added support for properties named the same as a type.<br>• Match C# semantics when searching for types by auto-including parent namespaces in search order.<br>• Fixed code generation of derived classes with static ctors<br>• Fixed compiler parsing of floats and decimals on non US-english locales.<br>• Fixed JSON parsing of long and float numbers<br>• Fixed JSON serialization of double values<br>• Fixed Sys.UI.DomEvent and ArrayList.RemoveAt to match ASP.NET Ajax signatures |
| 0.4.1.0 | 9/6/2007 | Quick incremental release to fix some key things:<br>• Add Add/RemoveEventListener on Silverlight storyboard<br>• Fix silverlight creation javascript bootstrapper<br>• Add support for setting style on silverlight object/embed tag<br>• Marked Sys.Debug method as [DebugConditional] so calls are stripped out in release |

| | | |
|---|---|---|
| | | builds.<br>• Queue class in ASP.NET Ajax mode now correctly resolves to Array<br>• Fix access to static constants consumed in a derived class<br><br>New feature:<br>• Support for indexer methods in code – compiled into Javascript as parameterized property getters/setters. |
| 0.4.2.0 | 9/11/2007 | Minor Silverlight related bug fixes<br>• OnResize/OnFullScreen events<br>• Workaround for Silverlight bug resulting in invalid size property values during the onLoad event.<br>• Add some missing Silverlight 1.0 APIs |

# License

**End User License Agreement for Script#**

IT IS IMPORTANT THAT YOU CAREFULLY READ THIS NOTICE BEFORE INSTALLING THIS PRODUCT. BY INSTALLING, OR OTHERWISE USING THIS SOFTWARE, YOU AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE AGREEMENT (THE "AGREEMENT") WHICH CONSTITUTES A LEGALLY BINDING CONTRACT BETWEEN THE LICENSOR (PROJECTS.NIKHILK.NET, HEREAFTER 'WE', OR 'US') AND THE LICENSEE (EITHER AN INDIVIDUAL OR ENTITY, HEREAFTER 'YOU').

**THIS AGREEMENT**

**1.1** In this Agreement, the phrase "Software" means any version of the computer programs above and all associated media, printed materials, "online" or electronic documentation and bundled software.
**1.2** The Software is licensed, not sold, to You for use only under the terms of this Agreement. We reserve any rights not expressly granted to You.
**1.3** By installing, copying or otherwise using the Software, You agree to be bound by the terms of this Agreement. If You do not agree to the terms of this Agreement You must not use the Software and must immediately delete any and all copies of the Software in your procession.

**GRANT OF LICENSE**

**2.1** We hereby grant You the following non-exclusive license to use the Software. The rights granted to the Licensee are personal and non-transferable.
**2.2** You may deploy the script files included with the product or those generated from using the product to a Web server.
**2.3** The following are the restrictions placed on the use of the Software. You may not:

- Remove the auto-generated header identifying Script# as the generator or tool used to produce the script files you deploy into your application or component.
- Modify or adapt the Software into another program or product.
- Reverse engineer, disassemble or decompile, or make any attempt to discover the source code of the Software through current or future available technologies.
- Redistribute, publish or deploy the Software on a standalone basis for others to copy without prior acknowledgment from the Licensor.
- Copy or republish any portion of the documentation without prior acknowledgment from the Licensor.
- Sell, re-license, sub-license, rent, lease any part of the Software or create derivative works.
- Use the Software to perform any unauthorized transfer of information or any illegal purpose.

**2.4** We may from time to time create updated versions of the Software and may, at our option, make such updates available to You.
**2.5** The Software is pre-release software. We have the sole right to determine all aspects of future updates, changes, and releases of the Software.
**2.6** You permit the Software to connect and communicate with our servers to send version and usage information for the purposes of improving the Software or sending information about available updates.
**2.7** You agree to indemnify, hold harmless, and defend Us from and against any claims, allegations, lawsuits, losses and costs (including attorney fees), that arise or result from the use, deployment or distribution the software.
**2.8** Any feedback including bug reports, feature suggestions or ideas provided by You to Us through any communication channel are given to Us without any associated charge or implied patent or intellectual rights. Thereafter, We have the full right to use, share and commercialize such feedback in any way and

for any purpose. You will not give feedback that is subject to a license that requires Us to license the Software to third parties because of inclusion of such feedback. These rights survive this Agreement.
**2.9** We do not provide any support services because the software is being made available to You in "as-is" form.
**2.10** We reserve the right to update the Agreement and the terms of the License with newer versions of the Software.

### Intellectual Property Rights

**3.1** The Software is protected by copyright and other intellectual property laws. Title to, ownership of, and all rights and interests in each and every part of the Software (including all copyrights, trademarks, patent rights or other intellectual property rights of whatever nature), and all copies thereof shall remain at all times vested in Us.

### Warranties

**4.1** We expressly disclaim any warranty for the Software. The Software and any associated materials are provided "As Is" without warranty of any kind, either express or implied, including without limitation, the implied warranties or merchantability, fitness for a particular purpose, or non-infringement. The entire risk arising out of use or performance of the Software remains with You.

### Termination

**5.1** This Agreement takes effect upon your use of the Software and remains effective until terminated. You may terminate it at any time by destroying all copies of the Software in possession. It will also automatically terminate if You fail to comply with any term or condition of this Agreement. You agree on termination of this Agreement to destroy all copies of the Software in possession.

### General Terms

**6.1** This written Agreement is the exclusive agreement between You and Us concerning the Software and supersedes any prior agreement, communication, advertising or representation concerning the Software.
**6.2** This Agreement may be modified only by a writing signed by You and Us.
**6.3** In the event of litigation between You and Us concerning the Software, the prevailing party in the litigation will be entitled to recover attorney fees and expenses from the other party.
**6.4** This Agreement is governed by the laws of the State of Washington, USA. Irrespective of the country in which the Software was acquired, the construction, validity and performance of the Agreement shall be governed in all respects by English law. You agree to submit to exclusive jurisdiction of English courts.
**6.5** If any provision of this Agreement is found to be invalid by any court having competent jurisdiction, the invalidity of such provision shall not affect the validity of the remaining provisions of this Agreement, which shall remain in full force and effect.
**6.6** You agree that the Software will not be shipped, transferred or exported into any country or used in any manner prohibited by the United States Export Administration Act or any other export laws, restrictions or regulations.