DLR Hosting Spec

Bill Chiles

WARNING: This is a big document, in case you are thinking of printing it. This document is reasonably stable, but there is ongoing work in some sections. Some sections are not fully fleshed out for types more rarely used or "more obvious" in their usage.

1 Introduction

One of the top DLR features is common hosting support for all languages implemented on the DLR. The primary goal is supporting .NET applications hosting the DLR's ScriptRuntime and engines for the following high-level scenarios:

- SilverLight hosting in browsers
- MerlinWeb on the server
- Interaction consoles where the ScriptRuntime is possibly isolated in another app domain.
- Editing tool with colorization, completion, and parameter tips (may only work on live objects in v1)

- PowerShell, C#, and VB.NET code using dynamic objects and operating on them dynamically in the same app domain

A quick survey of functionality includes:

- Create ScriptRuntimes locally or in remote app domains.
- Execute snippets of code.
- Execute files of code in their own execution context (ScriptScope).
- Explicitly choose language engines to use or just execute files to let the DLR find the right engine.
- Create scopes privately or publicly for executing code in.
- Create scopes, set variables in the scope to provide host object models, and publish the scopes for dynamic languages to import, require, etc.
- Create scopes, set variables to provide object models, and execute code within the scopes.
- Fetch dynamic objects and functions from scopes bound to names or execute expressions that return objects.
- Call dynamic functions as host command implementations or event handlers.
- Get reflection information for object members, parameter information, and documentation.
- Control how files are resolved when dynamic languages import other files of code.

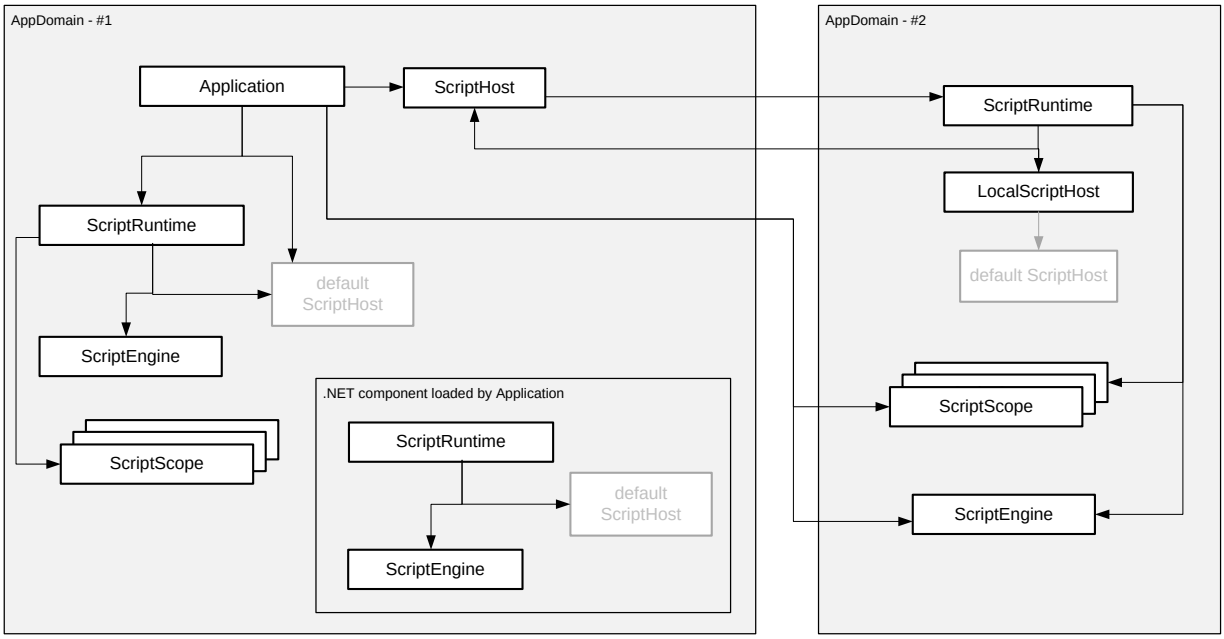
Hosts always start by calling statically on the ScriptRuntime to create a ScriptRuntime. In the simplest case, the host can set globals and execute files that access the globals. In more advanced scenarios, hosts can fully control language engines, get services from them, work with compiled code, explicitly execute code in specific scopes, interact in rich ways with dynamic objects from the ScriptRuntime, and so on.

2 High-level Hosting Model Concepts

The hosting APIs can be grouped by levels of engagement. Level One uses a couple of types for executing code in scopes and working with variable bindings in those scopes. Level Two involves a few more types and supports more control over how code executes, using compiled code in various scopes, and using various sources of code. Level Three opens up to several advanced concepts such as overriding how filenames are resolved, providing custom source content readers, reflecting over objects for design-time tool support, providing late bound variable values from the host, and using remote ScriptRuntimes.

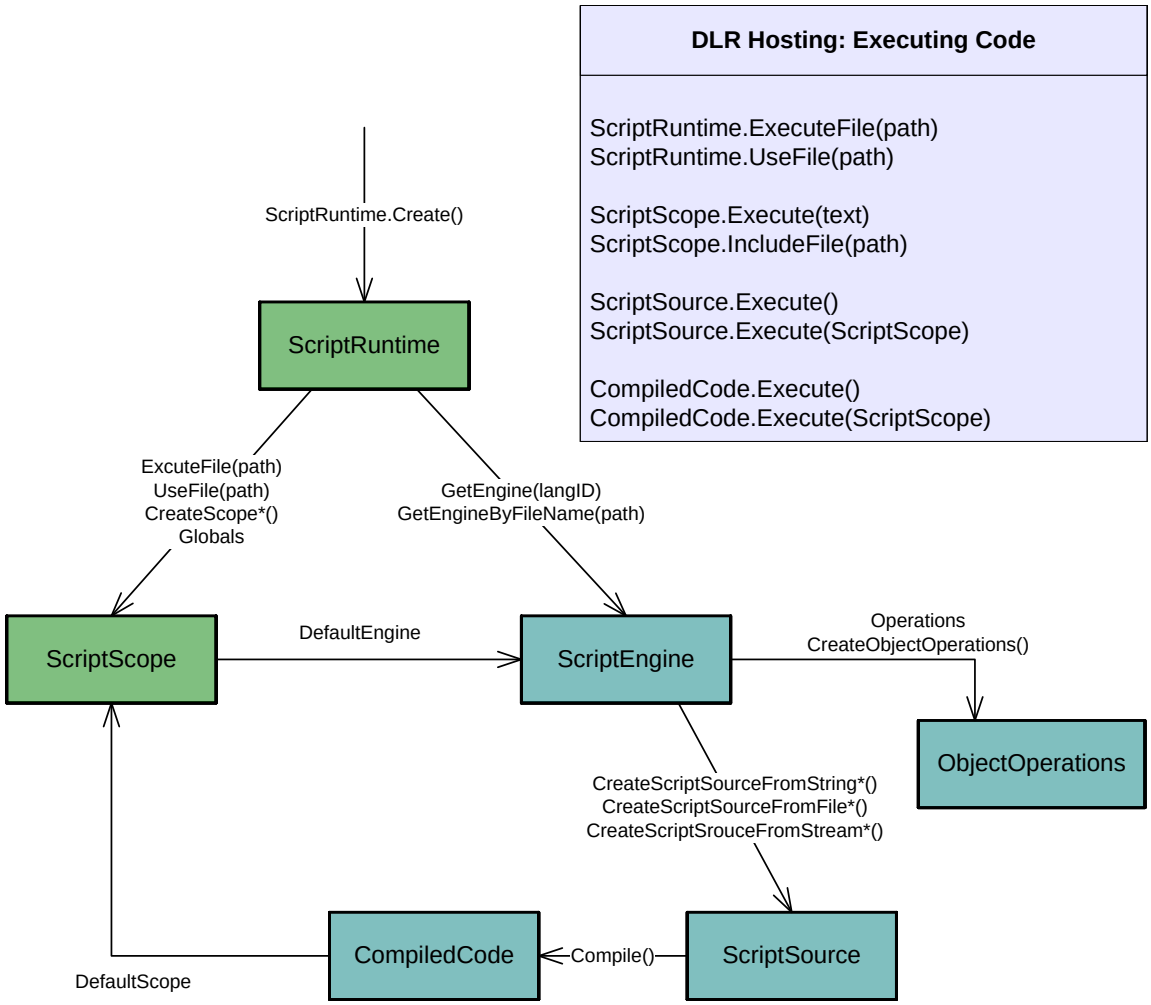
There are three basic mechanisms for partially isolating state for code executions within a process. .NET offers Appdomains, which allows for code to run at different trust levels and to be completely torn down and unloaded. The DLR offers multiple ScriptRuntimes within an AppDomain, each having its own global object of name bindings, distinct references to .NET namespaces from specified assemblies, distinct options, etc. The DLR also provides ScriptScopes which provide variable binding isolation, and you can execute code in different scopes to work with distinct bindings of free variable references.

The following diagram shows conceptually how hosts relate to ScriptRuntimes and other hosting objects:



It is important for the DLR to support distinct ScriptRuntimes within .NET's AppDomains for a couple of reasons. First, key customers betting on us require the lighter-weight isolation than what AppDomains provide. Second, consider two independent .NET components loading as extensions to an application. Each component wants to provide scripting support to end users. The components should be able to do so without having to worry about how they might clash with other script-enabled components. Multiple ScriptRuntimes also makes the main application's job easier since it does not have to provide a model for independent components getting access to and coordinating code executions around a central ScriptRuntime.

There's a rich set of ways to execute code. Our goal is to strike a balance between convenient execution methods on various objects and keeping redundancy across the types to a minimum. The diagram below shows how to navigate to the key objects for running code. The Sections on Levels One and Level Two below talk about types in this diagram (green being Level One types):



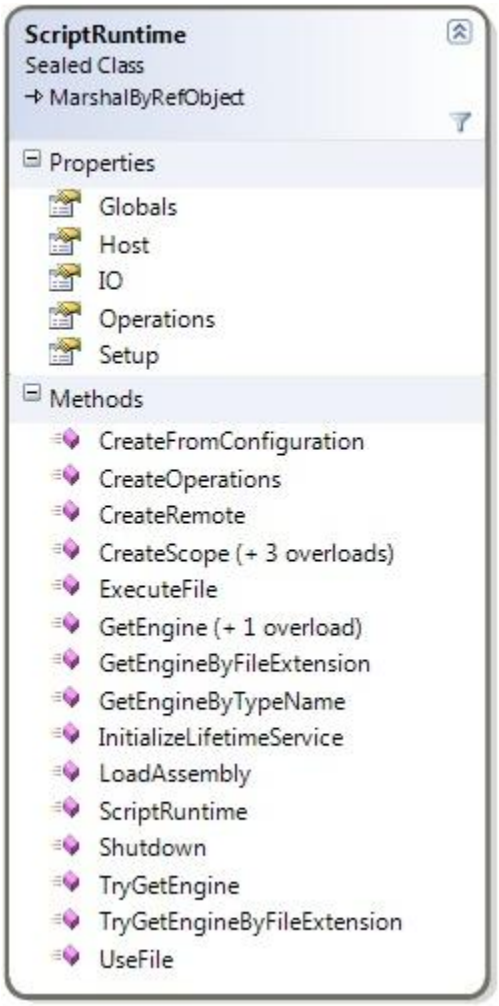
2.1 Level One -- Script Runtimes, Scopes, and Executing Files and Snippets

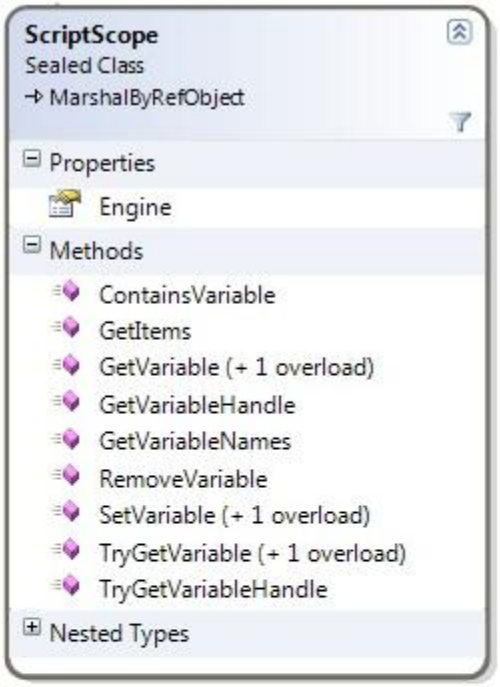
For simple application programmability, you want to provide a host object model that dynamic languages code can use. You then want to execute files of code that consume that object model. You may also want to get the values of variables from the dynamic language code to use dynamic functions as command implementations or event handlers.

There are two types you will use at this level. The ScriptRuntime class is the starting point for hosting. You create a runtime with this class. The ScriptRuntime represents global script state, such as referenced assemblies and a global object (a ScriptScope). The ScriptScope class essentially represents a namespace. Hosts can bind variable names in ScriptScopes, fetch variable values, etc. Hosts can execute code within different scopes to isolate free variable resolutions.

There are a lot of members on these types because they are also used for Level Two and Level Three. For Level One you only need a few members and can ignore the rest. You need to create a ScriptRuntime, from which you might use ExecuteFile or Globals. The Globals object lets you set variables to provide access to a host object model. From ScriptScope, you will likely only use GetVariable and SetVariable.

These types are shown in the diagram:





The ScriptRuntime.GetEngine and ScriptScope.Engine are bridges to more advanced hosting functionality. In Level Two and Level Three scenarios, the other members of ScriptRuntime and ScriptScope will be useful along with ScriptEngine.

2.1.1 Code Sample -- Application Programmability

The following code sample assumes you have a default app .config file (see section 4.13.3.2):

```
public class Level_1 {
    ScriptRuntime env = ScriptRuntime.CreateFromConfiguration();
    MyHostObjectModel hostOM = new MyHostObjectModel();

    /// <summary>
    /// Shows setting Host OM on globals so that dynamic languages
    /// can import, require, etc., to access the host's OM.
    /// </summary>
    /// my_user_script.py:
    /// import HostModule
    /// def foo () ...
    /// HostModule.UserCommands["foo"] = foo
    ///
```

```
    public void RunFile_Isolated_Scope_ImportsFromHost() {
        env.Globals.SetVariable("HostModule", hostOM);
        // Imagine this runs my_user_script.py above.
        env.ExecuteFile(GetFileFromUserOrSettings());
    }

    delegate void Command();

    /// <summary>
    /// Shows getting command implementations from dynamic language.
    /// Assumes menu item text is command name in table.
    /// Builds on previous function.
    /// </summary>
    public void Run_User_Command_from_MenuItem (string menuItemName) {
        // UserCommands is Dictionary<string, Command>.
        hostOM.UserCommands[menuItemName]();
    }


    /// Fetch scope var as Command.
    ///
    /// <summary>
    /// Shows discovering command implementations from globals in
scope.
    /// Above user code explicitly added commands, but this code finds
    /// commands matching a delegate type, Command.
    /// </summary>
    public void Collect_User_Commands_From_File (string menuItemName) {
        env.Globals.SetVariable("HostModule", hostOM);
        ScriptScope scope
            = env.ExecuteFile(GetFileFromUserOrSettings());
        // UserCommands is dictionary from string to Command.
        Command fun;
        foreach (string id in scope.GetVariableNames()) {
            bool got_fun = scope.TryGetVariable<Command>(id, out fun);
            if (got_fun) {
                my_OM.UserCommands[id] = fun;
            }
        }
    }
}
```

2.2 Level Two -- Engines, Compiled Code, Sources, and Object Operations

The next level of engagement involves operating directly with engines and abstractions of source code and compiled code. You can compile code and run it in any scope or its default scope. You also have more control over how you provide sources to the DLR.

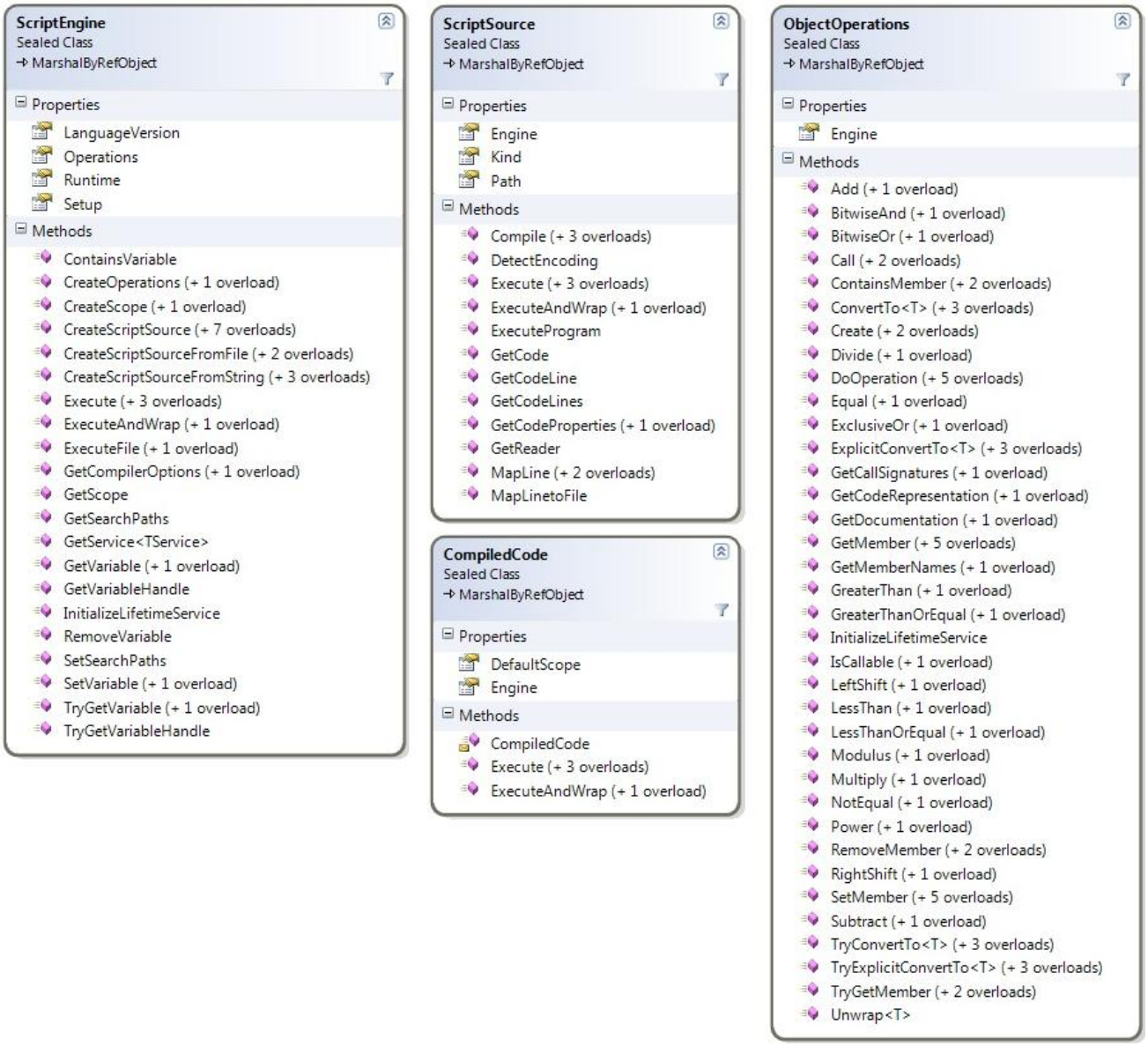
Besides the new types you'll use for Level Two scenarios, you will likely use more of the ScriptRuntime and ScriptScope classes. You'll certainly use their members to get to engines. You'll likely use more flavors of getting and setting variables on scopes as you have richer interactions with dynamic code. You might use the ability of scopes to support executing multiple languages within one scope (using execute methods on engines).

The main new types you'll use in Level Two scenarios are ScriptEngines, ScriptSources, and ObjectOperations. ScriptEngines are the work horse. They offer various ways to execute code and create ScriptScopes and ScriptSources. ScriptSources offer methods for executing code in various ways from different kinds of sources.

You may use ScriptRuntime.LoadAssembly to makes namespaces and types available to script code. The ScriptRuntime coordinates with ScriptEngines, but how script code accesses .NET namespaces and types is language-dependent. The language may require an 'import', 'using', or 'require' construct, or the language may put first class dynamic objects in ScriptRuntime.Globals.

ObjectOperations provide a large catalogue of object operations such as member access, conversions, indexing, and operations like addition. There are several introspection and tool support services that we'll discuss in Level Three scenarios. You get ObjectOperation instances from engines, and they are bound to their engines for the semantics of the operations.

These are the main types of level two:



2.2.1 Code Sample -- REPL and Merlin Web

The following code sample assumes you have a default app .config file (see section 4.13.3.2):

```
public class Level_2 {
    ScriptRuntime env = ScriptRuntime.CreateFromConfiguration();

    public void REPL_fragments {
        ScriptSource input;
        object result;
        // Assume user has chosen a context with the REPL's
        // default scope and the Python language.
        ScriptScope defaultScope = env.CreateScope();
        ScriptEngine curEngine = env.GetEngine("py");

        // Use interactive source units for special REPL variables
        // or syntax, such as IPy's underscore or VB's '?',
        // provided by standard interpreters for the languages.
        ScriptSource input
            = curEngine
                .CreateScriptSourceFromString
                  // E.g., input is "x = 'foo'".
                  (GetInputAsString(),
                   SourceCodeKind.InteractiveCode);
        result = input.Execute(defaultScope);
        REPLOutput.WriteLine(curEngine
                               .Operations
                                 .GetObjectCodeRepresentation(result));

        // Assume user has chosen somehow to switch to Ruby.
        curEngine = env.GetEngine("rb");
        // E.g., input is "puts x"
        input = curEngine
                  .CreateScriptSourceFromString
                    (GetInputAsString(),
                     SourceCodeKind.InteractiveCode);
        result = input.Execute(defaultScope);
        System.Console.WriteLine
                        (curEngine
                           .Operations
                             .GetObjectCodeRepresentation(result));

        // Assume user has chosen to execute a file from another
        // language, and you want to set the REPL's context to that
        // file's scope.  Use scope now for interactions like above,
        // and save it for use with this file later.
        ScriptScope scope = env.ExecuteFile(GetFileFromUserOrEditor());
        curEngine = scope.Engine;
    }


    public delegate string OnLoadDelegate();

    // MerlinWeb:
    public void ReUseCompiledCodeDifferentScopes() {
```

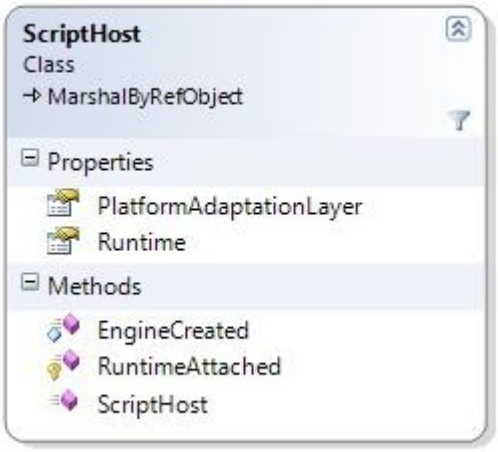
```
        ScriptEngine engine = env.GetEngine("rb");
        CompiledCode compiledCode
          = engine.CreateScriptSourceFromFile("foo.rb").Compile();

        // on each request, create new scope with custom dictionary
        // for latebound look up of elements on page.  This uses a
        // derived type of DynamicObject for convenience.
        DynamicObject pageDynObj = GetNewDynamicObject();
        pageDynObj["Page"] = thisRequestPage;
        ScriptScope scope = engine.CreateScope(pageDynObj);
        compiledCode.Execute(scope);
        // Expect on_Load function name to be defined, or throw error.
        // Could have used scope.GetVariable but it's case-sensitive.
        scope.GetVariable<OnLoadDelegate>("on_Load")();
    }

}
```

2.3 Level Three -- Full Control, Remoting, Tool Support, and More

Level three engagement as a host gives you full control over the ScriptRuntime. There are many things hosts can do at this level from controlling runtime configuration, to handling runtime exceptions, to using remote ScriptRuntimes, to providing full programming tool support with completion, parameter info pop-ups, and colorization.

With level three support, you can create a ScriptRuntimeSetup object to control which languages are available or set options for a ScriptRuntime's behavior. For example, you can limit the ScriptRuntime certain versions of particular languages. You can also use .NET application configuration to allow users to customize what languages are available.

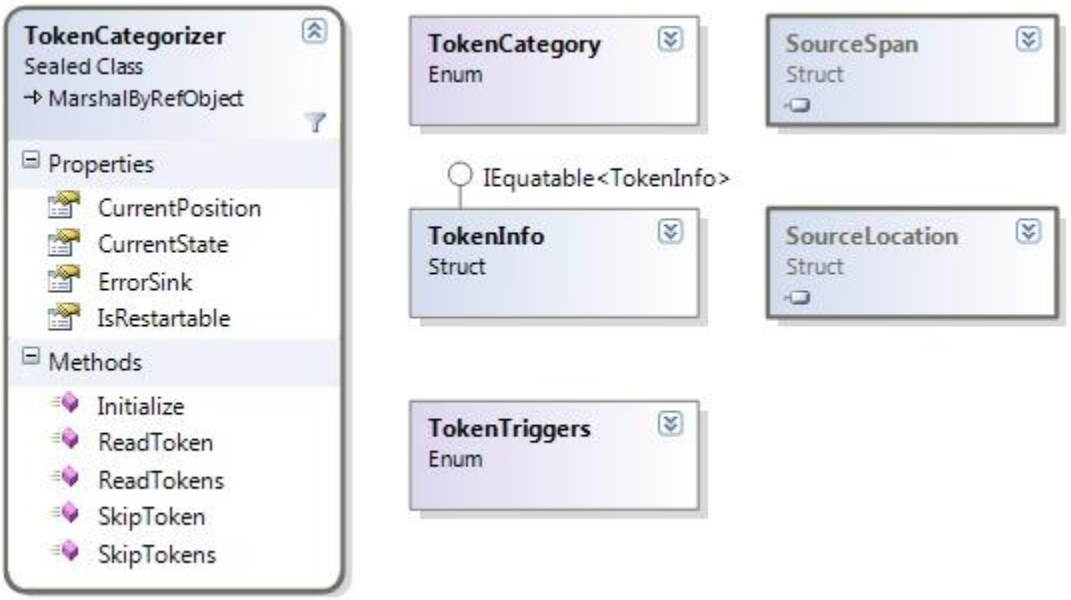
Another simple mechanism in level three is deriving from ScriptHost. This lets you provide a custom PlatformAdaptationLayer object to override file name resolution. For example, you might only load files from a particular directory or go to a web server for files. A host communicates its sub type of ScriptHost to the DLR when it creates a ScriptRuntime. Many hosts can just use the DLR's default ScriptHost. ScriptHost looks like:



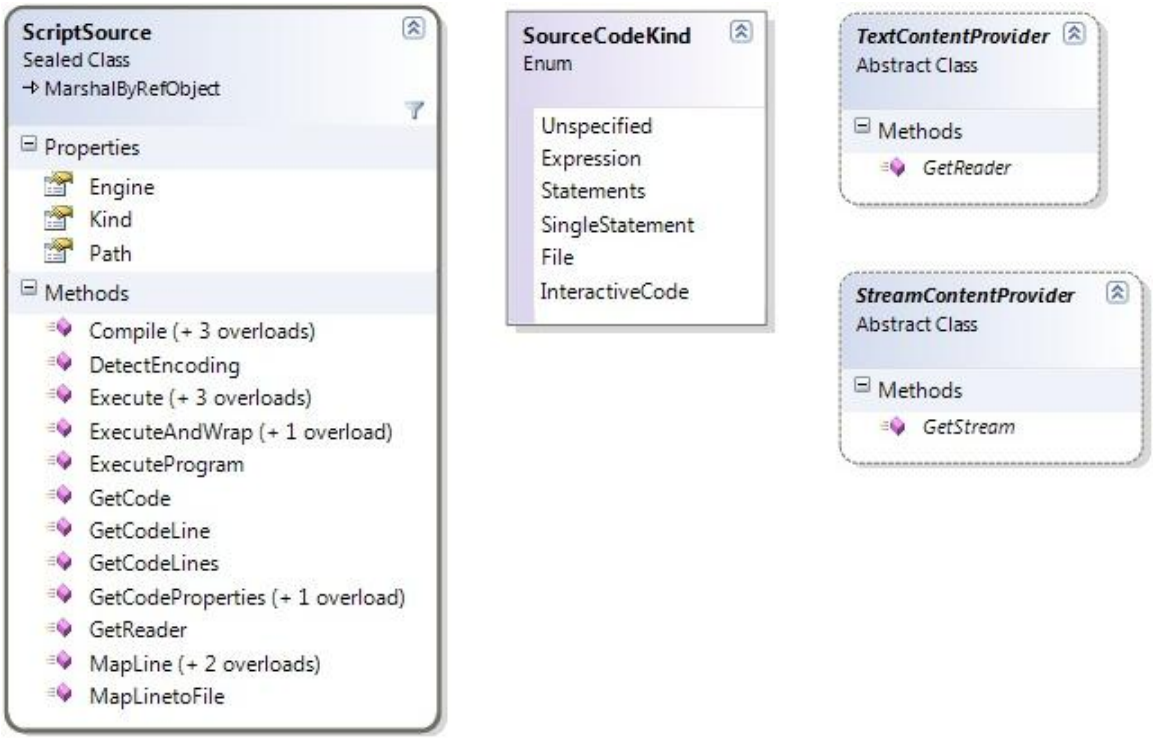
The ObjectOperations class provides language-specific operations on objects, including some tool building support. ObjectOperations includes introspection of objects via members such as GetMemberNames, IsCallable, GetCallSignatures, GetDocumentation, and GetCodeRepresentation. These give you a language-specific view in that you see members of objects and signatures from the flavor of a specific language. For example, you would see the Python meta-programming members of objects that Python manifests on objects. ObjectOperations enables you to build tool support for dynamic languages on the DLR, but you need another set of objects for parsing code.

Hosts can get parsing support for providing colorization in editors and interpreters. The following are the types used for that functionality:

This will definitely change in the future. When we finalize this after CLR 4.0, it will be in another spec and removed from here. For now, it is a place holder and represents some prototyping we did.



Hosts that implement tools for programmers will likely also create ScriptSources and implement TextContentProviders so that tokenizers can read input directly from the host's data structures. For example, an editor with a file in memory (in an editor buffer) could implement a TextContentProviders that reads input directly from the buffer's data structure. The types relating to sources are:



Advanced hosts can also provide late-bound values for variables that dynamic languages access. Each time the variable's value is fetched the host can compute a new value or deliver one that's cached in its data structures. Hosts do this by being the implementer of an IDynamicMetaObjectProvider that the hosts supply when creating a ScriptScope. A simple way to implement this interface is to derive from DynamicObject which implements the interface. Then just override methods such as TryGetMember and TrySetMember. If you don't need to do fancy late binding of names, you could use a ScriptScope or ExpandoObject as a fast property bag.

2.3.1 Code Sample -- Tools Support and Remoting

The following code sample assumes you have a default app .config file (see section 4.13.3.2):

```
public class Level_3 {
    ScriptRuntime env = ScriptRuntime.CreateFromConfiguration();

    /// <summary>
    /// Shows reflecting over object members for tool support
    /// </summary>
    public static void Introspection() {
        ScriptEngine engine = env.GetEngine("py");
        ScriptScope scope = env.CreateScriptScope();
        engine.Execute("import datetime", scope);
        object obj = scope.GetVariable("datetime");
        ObjectOperations objops = engine.Operations;
        string[] members = objops.GetMemberNames(obj);
        // Start with known "date" member. Throws KeyNotFoundException.
        obj = objops.GetMember(obj, "date");
        // Drill down through members for a.b.c style
```

```
        // completion support in a tool.
        obj = objops.GetMember(obj, "replace");
        objops.IsCallable(obj);
        string[] signatures = objops.GetCallSignatures(obj);
    }


    /// <summary>
    /// Shows reflection for tools using a remote ScriptRuntime.
    /// </summary>
    public static void RemoteIntrospection() {
        setup = ScriptRuntimeSetup.ReadConfiguration();
        ScriptRuntime env =
ScriptRuntime.CreateRemote(appDomain,setup);
        ScriptEngine engine = env.GetEngine("py");
        ScriptScope scope = env.CreateScriptScope();
        engine.Execute("import datetime", scope);
        ObjectHandle obj = scope.GetVariableAndWrap("datetime");
        ObjectOperations objops = engine.Operations;
        string[] members = objops.GetMemberNames(obj);
        // Start with "date" member.
        objops.GetMember(obj, "date", out obj);
        // Drill down through members for a.b.c style
        // completion support in a tool.
        objops.GetMember(obj, "replace", out obj);
        objops.IsCallable(obj);
        string[] signatures = objops.GetCallSignatures(obj);
    }


////////////////////////////////////////////////////////////////////
/// <summary>
/// Shows late binding host-supplied variables.
/// </summary>
class Program {
    public static object GetLateBoundValue (string name) {
        // Pretend call back into host to look up the value of name.
        return "yo";
    }
    static void Main(string[] args) {
        var sr = ScriptRuntime.CreateFromConfiguration();
        var eng = sr.GetEngine("ironpython");
        var scope2 = eng.CreateScope(new MyLBHG());
        Console.WriteLine("Engine: " + eng.Setup.DisplayName);
        eng.Execute("print 'hey'");
        eng.Execute("x = 3", scope2);
        eng.Execute("print x", scope2);
        // foo is supplied by host when script code fetches the value.
        eng.Execute("print foo", scope2);
        //eng.Execute("foo = 3", scope2); shouldn't work, host decided
        Console.Read();
    }
}

public class MyLBHG : LateBoundHostGlobals {
    // Instantiate class with names that should be fetched from host
    // each time value is needed, and host must implement properties
```

```
    // in this class for each name in the list.
    public MyLBHG ()
        : base("foo") {
    }

    public object foo {
        get { return Program.GetLateBoundValue("foo"); }
        // Notice no 'set', hence "foo = 3" does not work
    }
}

////////////////////////////////////
// This will become part of the DLR so that you do not need to write
it.
// Will remove from spec when that happens.
// For more info see sites-binder-dynobj-interop.doc on DLR Codeplex.
////////////////////////////////////
public class LateBoundHostGlobals : IDynamicMetaObjectProvider {
    private readonly Dictionary<string, object> _dict =
        new Dictionary<string, object>();
    internal readonly string[] _specialNames;

    public LateBoundHostGlobals (params string[] specialNames) {
        _specialNames = specialNames;
    }

    public DynamicMetaObject GetMetaObject
          (System.Linq.Expressions.Expression parameter) {
        return new Meta(parameter, this);
    }

    public bool TryGetMember (string name, out object value) {
        return _dict.TryGetValue(name, out value);
    }

    public object SetMember (string name, object value) {
        return _dict[name] = value;
    }


    class Meta : DynamicMetaObject {
        private Expression _parameter;

        private string [] SpecialNames {
            get {
                return ((LateBoundHostGlobals)Value)._specialNames;
            }
        }
        public Meta (Expression parameter, LateBoundHostGlobals self)
            : base(parameter, BindingRestrictions.Empty, self) {
            _parameter = parameter;
        }

        public override DynamicMetaObject BindGetMember
               (GetMemberBinder binder) {
            if (SpecialNames.Contains(binder.Name)) {
                return binder.FallbackGetMember(this);
```

```
            }
            var param = Expression.Parameter(typeof(object));
            return new DynamicMetaObject(
                Expression.Block(
                    new[] { param },
                    Expression.Condition(
                        Expression.Call(
                            Expression.Convert
                               (Expression,
                               typeof(LateBoundHostGlobals)),
                            typeof(LateBoundHostGlobals)
                               .GetMethod("TryGetMember"),
                            Expression.Constant(binder.Name),
                            param),
                        param,
                        Expression.Convert(
                           binder.FallbackGetMember(this)
                             .Expression, typeof(object)))),
                BindingRestrictions.GetTypeRestriction(
                    _parameter,
                    Value.GetType()));
        } //BindGetMember

        public override DynamicMetaObject BindSetMember
                (SetMemberBinder binder, DynamicMetaObject value) {
            if (SpecialNames.Contains(binder.Name)) {
                return binder.FallbackSetMember(this, value);
            }
            return new DynamicMetaObject(
                Expression.Call(
                    Expression.Convert
                       (Expression, typeof(LateBoundHostGlobals)),

typeof(LateBoundHostGlobals).GetMethod("SetMember"),
                    Expression.Constant(binder.Name),
                    Expression.Convert(value.Expression,
                                       typeof(object))),
                BindingRestrictions.GetTypeRestriction(
                    _parameter,
                    Value.GetType()));
        }

    } // class Meta
} // class LateBoundHostGlobals
////////////////////////////////////////////////////////////////////


    /// <summary>
    /// executes a file within a remote environment;
    /// note almost transparent remoting (host only needs to care when
    /// converting to a non-remotable object.
    /// Shows late binding host-supplied variables.
    /// </summary>
    public static void Remotable1(System.AppDomain appDomain) {
        setup = ScriptRuntimeSetup.ReadConfiguration();
```

```
        ScriptRuntime env =
ScriptRuntime.CreateRemote(appDomain,setup);
        ScriptScope scope = env.ExecuteFile("foo.py");
        ObjectHandle load_fun = scope.GetVariableHandle("on_load");
        ScriptEngine engine = scope.Engine;
        ObjectHandle result
           = engine.Operations.Call(load_fun, "arg1", "arg2");

        object localResult;
        try {
            localResult = engine.Operations.UnWrap<object>(result);
        }
        catch (System.Runtime.Serialization.SerializationException) {
            // error
        }
    }
```

~~2.4 Implementer's Convenience~~

~~There is a ConsoleHost and a couple of related types for setting options and parsing options on a shell command line. This is just a default parameterized cmd.exe style interpreter that you can quickly have running for your DLR language. It is not very functional in terms of handling input, but it gives you a quick run target to see your language working as soon as you have some parsing working.~~

~~*These types have not been thought through and will likely change over time.*~~

3 Hosts Requirements

While the functional requirements are accurate here, the descriptions are a bit dated and many use old APIs (even PythonEngine APIs from pre 1.0 releases) for examples.

3.1 SilverLight (RIA)

We need to ensure that the DLR can plug into whatever hosting story Telesto has for integrating into web browsers. We want web pages to be able to have one or more DLR script elements on them. Scenarios for such script are the same as if the script were native JavaScript in the browser.

We will work with Telesto to build an object that can be hosted via an object element in the HTML. Anyone using DLR scripting languages will use the same object. This object will derive from Telesto's PageLet class, and when it spins up, it will scan the page for DLR script blocks, load the DLR, and spin up script engines.

3.2 MerlinWeb (Server Side)

MerlinWeb needs to work with various script blocks of different languages in the same execution ScriptRuntime in a language-neutral way (or with minimal language-specific awareness).

3.2.1 Compilation

Represent each page with an EE scope (ipy EngineScopes).

With IPy, all ASP.NET files that are normally managed classes (e.g., aspx/ascx/master/ashx files) are represented by EngineScope objects, created using:

```
engineScope = s_engine.CreateScope(String.Empty /*scopeName*/,
   globals, false/*publishScope*/);
```

Execute script as text in the context of an EE scope.

Currently, the code found in a page's code behind file or in its <script runat="server"> block is fed into the page's IPy EngineScope:

```
s_engine.Execute(scriptCode, engineScope);
```

Enumerate a scope for functions.

Currently, after feeding code into an EngineScope, MerlinWeb looks for top-level functions with:

```
foreach (KeyValuePair<string, object> pair in EngineScope.Globals) {

    PythonFunction f = pair.Value as PythonFunction;
    //...
```

For each PythonFunction, MerlinWeb checks a few properties such as its ArgCount (for a rough signature check) and line number where the function starts:

```
f.ArgCount

((FunctionCode)f.FunctionCode).FirstLineNumber"
```

Invoke functions.

Currently, MerlinWeb invokes function via Ops.Call().

Compile code fragments.

Code snippets come from *<% code %>* and *<%= expr %>* elements in the HTML.

Currently, MerlinWeb uses CreateMethod and CreateLambda, respectively. MerlinWeb likes the way these work now and wants something similar in DLR.

Reference a special directory for access to language scopes.

MerlinWeb apps support an App_Script directory containing script files that any code can reference. This is the logical equivalent of ASP.NET's App_Code directory, but for dynamic script files.

Currently, the code does the following:

```
s_engine.Sys.path.Add(s_scriptFolder);
```

Reload scripts that are in use and have changed.

MerlinWeb detects scripts that have changed, and it needs to direct the engine to reload those scripts, resetting global state and definitions for the scope.

Currently, MerlinWeb does something like the following:

```
foreach (PythonScope scope in s_engine.Sys.scopes.Values) {
    if (scope.Filename.StartsWith(s_scriptFolder))
        Builtin.Reload(scope);
}
```

MerlinWeb is also okay with throwing out the ScriptRuntime and creating a new instance if this is on the order of 1-2s vs. several seconds.

3.2.2 Globals and Member Injection

Provide "globals" and participate in name lookup

MerlinWeb needs to inject global as name->object mappings. Furthermore, MerlinWeb needs to be able to participate in name lookup, so that dynamically when a name's value is fetched MerlinWeb can discover what that name should be bound to and provide an object when asked.

Inject members into pre-defined types with per instance values

MerlinWeb needs to be able to dynamically compute member lookup for objects/types to provide a better programming experience for script on pages for managed objects that were defined before the advent of MerlinWeb. MerlinWeb uses this mechanism to support simpler syntax for things like:

- page.TextBox1 instead of page.FindControl("TextBox1")
- request.Foo instead of request.QuesryString["Foo"]

Currently, MerlinWeb is highly dependent on the new IAttributesInjector and the Ops.RegisterAttributesInjectorForType mechanism.

Allow for injected members on types defined for MerlinWeb

MerlinWeb needs to make .NET objects appear to be dynamic objects for member lookup resolution, resolving names for members of a given object when the names come from different sources. MerlinWeb uses this ability to make objects (for example, control and page objects) and associated script functions appear on a unified object where members appearing on the

object come from both the variables on the page or control as well as come from globals in associated scripts.

Currently, each dynamic page (aspx/ascx/master) is made up of both a regular Page-derived object, and an associated EngineScope. MerlinWeb makes them behave as if the EngineScope adds methods to the Page (the way a partial class would). This works by having the custom Page implement ICustomAttributes. The ICustomAttributes implementation 'combines' everything on the EngineScope (obtained via EngineScope.Globals) with everything on the Page's DynamicType (obtained via Ops.GetDynamicType).

3.2.3 Error handling

Syntax error handling

MerlinWeb needs to get syntax error information for reporting and interacting with tools.

Currently, it catches PythonSyntaxErrorException's to get line number information from them using the FileName and Line properties.

Runtime error handling

When a run-time error occurs, MerlinWeb needs to get an error message, file name, and line number for reporting.

Currently, the code looks like this (not pretty, should be cleaned up):

```
// Though we ignore the return value, this call is needed
Ops.ExtractException(e, s_engine.Sys);

Tuple t = s_engine.Sys.exc_info();
string message = (string)t[1].ToString();
TraceBack tb = (TraceBack)t[2];

// Find the initial exception frame
while (tb.Next != null)
    tb = tb.Next;

// Clear the exception
Ops.ClearException(s_engine.Sys);

int line = tb.Line;
string path = (string)((FunctionCode)
      (((TraceBackFrame)(tb.ScopeScope)).Code)).Filename;
```

Line pragmas for debugging support

MerlinWeb generates code for code found in .aspx files. For debuggers and tools to present the right info to users, MerlinWeb needs to give hints. Also, code in a single scope may come from more than one source location.

Currently, this is not well supported. For example, if you have this on an .aspx page:

```
<%
for i in range(max):
    %>
i=<%=i%> i*i=<%=i*i%><br />
```

MerlinWeb will generate code that looks like this:

```
for i in range(max):
    __param.RenderControl(0)
    __param.Render(i)
    __param.RenderControl(1)
    __param.Render(i*i)
    __param.RenderControl(2)
```

Note that this code contains a mix of user code (in green) and generated code. Furthermore, there is no correlation between the line numbers of the user code in the aspx and in the generated code. By using a line pragma mechanism, we can tell the compiler exactly where each snippet of code came from, which allows the aspx file to be debugged directly.

3.3 Base Tools Support

This section lists tools scenarios and requirements.

3.3.1 General Hosting (ScriptEngineServer)

Inject global name bindings. The tool needs to provide name bindings that are global across a ScriptRuntime, that is, across one or more script engines. The tool needs to inject one or more objects to provide necessary functionality to script engines running the tool's command implementations.

Host script engines remotely or isolated (including language tool support). The tool is running one ScriptRuntime for its commands, and it wants to run one remotely for the program the developer is working on. The remote ScriptRuntime for running users programs may need to be torn down, restarted, or debugged (stopping all threads) without putting the tool's ScriptRuntime at risk. Users should never lose work because the ScriptRuntime for tools commands gets messed up to where users cannot save files. However, if users are interactively developing for the tool's command ScriptRuntime, they might mess up the run time enough to render the editor useless. The language tool support may not run remotely, but it is local, it would need access to its associated script engine for some operations as well as some sort of RPC to it.

Eval code from string to get CLR object back. This supports hosts like future C# language support for interacting with dynamic objects as well as hosts like MerlinWeb that use dynamic languages to perform work within the same app domain and want to point at objects that come back. The eval operation can raise an exception, and if it does, the tool wants to catch the exception and call on the script engine for its (language-specific) reporting or formatting of error to get a string to print.

Get a string representation of eval results. The tool host should either be able to call ToString on an object returned from eval to get a reasonable language-specific print representation, or

the result of an eval operation is a string representing the result that the tool can print. For remote hosting, the tool should NOT have to call ToString on a proxy object, roundtripping with the script engine again to print results.

Interrupt ScriptRuntime to stop all processing in script engines. The ScriptRuntime and script engines server objects run on threads that still respond so that the host can interact with the ScriptRuntime. This action either throws to top level or drops into the debugger depending on what we can do (debugger is preferable, at least getting a stack trace) and what the host chooses to do. This needs to work for aborting an editor command written in a script language (just throws to the editor's top-level input loop). It also needs to work for the interaction window where you're likely to want to land in the debugger for you program.

Ask for reflection information language-independently. Tool hosts want to get the following:

- members lists from an identifier and context (scope or finer-grain parsing context)
- doc strings for types, members, and objects
- parameter names and types (if available)

Regardless of whether objects are from static .NET languages/assemblies or any given script engine, tool hosts call general hosting methods. Hosts do not need to know how to express such queries in each language and then evaluate expressions to get this information.

Reset scope or ScriptRuntime. Want to be able to reset a scope (definitions, global bindings, etc.) and reload it. Want to be able to reset a default scope with no code or file associated with it so that you can start fresh interactions. Want to be able to reset the entire ScriptRuntime (all engines), tearing down all engines, so that all the execution state you're working with is reset for a clean program execution. Not sure if we also need to be able to reset an engine without affecting other running engines, but we think this case collapses together with the reset the whole ScriptRuntime. A scenario for resetting the whole ScriptRuntime is if the tool has a primary script buffer designated so that F5 in any window means reset the entire ScriptRuntime and start again running the primary script.

*ISSUE: Unless we're forced to support it, resetting a scope does NOT address aliases to old values. For example, I might have script A.py and B.vbx, and I've used "from B import *" in A's scope. If I reload B.vbx, what happens to the bindings created in A that alias B's old definitions and values? My instinct, and the Python expectation is that A's bindings to B's defs and globals do not change with the reload. We need to think about whether there should be a requirement to support resetting those bindings on B's reload. Note, if A.py only refers to the object that is the scope for B.vbx, then reloading B and later "dotting" into its scope will reveal new values.*

3.3.2 Language Tool Support (IScriptEngine or ILanguageToolService)

Fetch runtime banner per script engine. When an interactive window comes up, we want users to see a banner that is consistent with what they expect from using interpreters for their language on other community tools. We may not show the banner in some situations, but certainly if the user launches the tool as an interpreter window (vs. as an editor first), then we want to show a familiar introductory banner.

Fetch standard language interpreter prompt. Again, we want to provide an interaction window experience that is similar to what a language community expects, so we want to be able to ask a script engine what its prompt looks like.

Fetch standard language debugger prompt. Again, we want to provide an interaction window experience that is similar to what a language community expects, so we want to be able to ask a script engine what its prompt looks like when it is in a debugging context.

Update the prompt. For languages where the prompt changes over time, or with different modes (for example, a number representing the stack frame to support frame_goto commands), we should have something in our API for allowing the script engine to push "next prompt" text to the host. This could be as simple as all script engines have a property that can be fetched each time a prompt needs to be printed. Note, this does not mean a tool should be in a special input mode and have a multi-line input prompt like IronPython does today. We should auto-indent and just detect on enter whether input is ready for evaluation or not, allowing editing of previous lines and so on.

Get token/coloring info for a language.

Tokenizer can act on "whole buffers/files" or expression fragments. Fragments can come from an editor file buffer or the interaction window.

Tokenizer is restartable at a designated location in a buffer (stream?). Tools need to incrementally request parsing and lexical analysis information at any point in the buffer. It's fine if the tool has to record cookies for each line so that the parser has some state to leverage to restart parsing.

Parse for possibly complete expression. Can pass a string to find out tokenizing info as well as determine if it is a complete expression/statement suitable for evaluation.

Fetch line-oriented comment start sequence. This is needed so that we can do comment and uncomment region.

Accept redefinitions regardless of live instances or active stack frames. We should be able to redefine functions even if there are pending calls to the function on the stack. Pending calls should continue execution of the old definition, and new calls should call the new definition. We should be able to redefine types even if there are existing instances, at least during interactive development. Declarations or compiler switches could harden definitions for optimizations if languages support that. Fetching members of old instances could fault to updated dual instances from the new definition, possible raising an exception if a member is uninitialized.

3.3.3 Static Analysis (Post MIX 07)

Get language info regardless of where object comes from. If a language has a variable that it knows is (will be) bound to a type from another language's script/scope, there should be a way to get member completion for that type if the other language can provide it. Maybe the other file needs to be opened, or maybe the file needs to be loaded into a script engine. The API the host calls on should be language agnostic for this support.

Get member completion of symbol at a buffer location. If a language has enough declared type info or if it has enough static analysis of variable lifetimes and known result types of expressions, a tool would want to get completion, doc strings, etc., for those variables. The tool should have an API to ask for it in case some languages do have this.

3.4 What Languages and DLR Need to Support

This section captures characteristics of a hostable language via DLR APIs. These characteristics are a blend of requirements and high-level "work items". The requirements are from hosts, such as application programmability hosts, that could have been listed in previous sub sections. The work items are functionality a language plugging into the DLR would need to support in its parser and binder (or perhaps in their compiler if they compiled all the way to IL on their own).

Discoverability: Language is discoverable from DLR hosting registration and loadable via hosting API.

Globals: Language can execute code that can access the runtime's global scope (global object). The language may resolve free references to variables in the global scope, or have its imports/require/using mechanism provide access to names bound on the global scope object. The language could provide library or builtin functions for accessing the variables, but this could also be cumbersome to programmers.

Scopes: Language can execute code in a host supplied scope, resolving free references to variables in the scope or some other linguistic mechanism. The language could provide library or builtin functions for accessing the variables, but this could also be cumbersome to programmers.

LanguageContext: Language implements a LanguageContext which supports many hosting APIs (such as ScriptEngine), compiling, creating scopes, working with ScriptSource objects, etc.

ScriptHost: Language defers to ScriptHost for getting/resolving sources for import/require/using features.

Interrupt Execution: Language supports DLR mechanisms for host to abort execution cleanly so that host can interrupt runaway code. If we allow languages to avoid this work, then those langs cannot be used in the host's process, cannot support in-situ REPLs for interactive development, etc. If the language uses DLR compilation or interpretation, they should not need to do anything. This is important because if the host has to stop execution by calling Thread.Abort or other rude abort, the thread could have transitioned into an unsafe portion of the host's code. This could happen via the script code calling back on the host OM which in turn calls an internal API not coded for rude aborts.

Limit Stack: Language supports DLR mechanisms for host to control maximum stack depth. If we allow languages to avoid this work, then those langs cannot be used in the host process, cannot support in-situ REPLs for interactive development, etc. If the language uses DLR compilation or interpretation, they should not need to do anything. This requirement is motivated by similar concerns to the execution interrupt requirement.

Debugging: Language supports DLR mechanisms for cooperative debugging or interpreted debugging so that hosts can use the DLR in the same process and app domain while not running all of its managed code at the speed of debug mode. If the language uses DLR compilation or interpretation, it should not need to do anything.

Exceptions Filter: Language supports an exception filter call back to the host. A scenario is that hosts may provide access to components (WPF control) that could execute hosted code outside of the host's entry points (cmd invocation, host OM event handler, etc.). The host would like to be able to catch all hosted code exceptions to cleanly present the issue to the end user or

cleanup the host's OM state as appropriate. If the language uses DLR compilation or interpretation, it should not need to do anything.

P2 Memory Limit: Language supports a limit on memory allocation so that the host can somehow nearly drive to zero the chance that running some hosted code will use up too much memory for the host to cleanly shutdown or clean up the hosted language runtime.

P2 Disabled Methods: language supports a list of methods that cannot be invoked, and if the hosted language tries to do so, the host call back for exception filtering is invoked. For example, on a shared public server that runs user code, allowing code to invoke Environment.FailFast is probably a bad idea.

Degree of Interop support:

- Level 1 -- Hosts can consume the language (as above), and language can consume globals/locals and .NET objects. Nothing more needed. Programmers would need to cast globals to known .NET types, or language would need to provide its own reflection-based late binding support.
- Level 2 -- Hosts can consume language, and language interops with DLR languages and dynamic objects. Language then needs to compile operations to DynamicSites (or generate DLR ASTs with ActionExpr nodes). Language may not need to produce any ASTs still since the default binder may be sufficient fall back when DynSites encounter IDynObjs that do not produce a rule for the operation at the site. If the default binder's semantics aren't good enough, the language would have to produce ASTs for tests and targets when asked for a rule in the DynSite.
- Level 3 -- Language is built on DLR or can produce full ASTs for interpreted mode, complete integration, etc. NOT SURE THIS IS A LEVEL, but could impact whether language is debuggable and can interact with host while debugging (depends on debugging story).

Aspects of tool support (optionally supporting each):

- Colorization -- lang produces TokenCategorizer service from LanguageContext
- Completion, Param info -- if we have static analysis service for tool support, then lang produces service. Probably nothing to do here if completion is all about live objects (other than support get doc and get sigs members/ops on LanguageContext).
- REPLs -- lang parser can return code properties at end of parse and support interactive execution (for example, is this snippet complete enough to execute or recognizing special REPL syntax or variables)
- Debugging -- story still under development here, may be nothing language needs to do, may be about AST
- NOT saying anything here about VSTA, VSA, VBA, etc., style hosting (project model, code model events, UI, host embedded UI, extension distribution, project persistence and discovery, etc.)

3.5 Configuration Notes

These are somewhat raw notes from various conversations on hosting configurations requirements, goals, and scenarios.

3.5.1 Requirements

Hosts need to have full control of config. (what langs are available, what versions, how code executes, etc.).

Config and options representation has to be very pliable and extensible between host, DLR, and languages. (likely means Dict<str,obj>).

There needs to be two levels of config (one for engine IDs, names, and LCs, and then one for per engine opts).

We want as simple a model as possible without rolling our own mechanisms.

Assuming two language versions work on same CLR version, should be able to use both.

Conflicting langIDs (simple names) are host's job to rectify (or to provide affordances for end-user to rectify).

LangIDs map to an assembly-qualified type that is the LanguageCtx that the hosting API implementations need to instantiate and hook up.

LangIDs are compared case-insensitively.

Languages need to be able to merge options from host that come through DLR (they know their defaults, but hosts shouldn't have to supply a full set of options to change a few).

There is some affordance for file exts mapping to engines, but this could break down, leaving hosts to use langIDs.

There is a display name property that we identify for how we pass into from host to DLR, but we don't have to standardize on how hosts get that or languages provide it.

GAC'ing is NOT required (that is, can be avoided in all end-to-end scenarios).

3.5.2 Goals

These are characteristics we'd like to satisfy, but we may not satisfy them all. In fact, some are mutually exclusive.

- Hosts should be able to discover what languages are available for the default DLR usage on the machine.
- Xcopy installation should be possible.
- An application using the DLR should be able to run from a USB drive.

3.5.3 Scenarios

End user uses one or more applications that utilize DLR scripting solution with no special actions on behalf of the end-user, he can use VB and C# as hosted languages (once they're hostable).

End-user uses one or more applications that support end-user options for scripting language choice.

- End-user installs IronPython and IronRuby on his machine.
- End-user may have to update each app (which could have different user options models) to identify appropriate DLLs, .NET types for language, and default options.

Barring language implementer's decisions, end-users use v1 and v2 of same language (that is, the DLR is not the gating factor), even in the same ScriptRuntime.

Host wants to tag sources with test/python type (or a web page with "language=python"), then just grab that string to map to an engine where it processes the page. It uses langIDs that map to an engine's assembly-qualified type name to avoid having to update pages all over a web site (and for more readable code, simple name vs. AQTN).

Internal test frameworks and code submission tools work for testing DLR and configurations.

Hosts may need to associate scripts with specific engine implementations

- It is okay (and consistent with other guidance) that they have to manage that association and config data themselves, then reconstitute it.
- Languages must do the work to be able to be SxS, if there's anything they'd have to do.

3.6 Remote Construction Design Notes

These are rough notes on making remote construction easier for consumers.

We finally decided to go with constructors over factory methods for a couple of reasons. The first is that constructors are more natural when factories aren't needed. The second is that having factories only made the remote construction more tedious and hard to get right.

We kept the static CreateRemote factory for discoverability, but it was much simpler to implement with the constructors. Going to constructors from factory methods would allow you to write these two "lines" instead of the helper class et al needed with only factory methods.

```
AppDomain.CreateInstanceAndUnwrap
             (typeof(ScriptRuntime).Assembly.FullName,
              typeof(ScriptRuntime).FullName);
```

And if you want to pass ScriptRuntimeSetup (as value of setup variable below):

```
AppDomain.CreateInstanceAndUnwrap
              (typeof(ScriptRuntime).Assembly.FullName,
               typeof(ScriptRuntime).FullName, false,
               BindingFlags.Default, null, new object[] { setup
},
               null, null, null);
```

If we only had factory methods, and we did not provide this factory for users:

```
runtime = ScriptRuntime
             .Create(AppDomain.CreateDomain("foo"), setup);
```

Then they would have to write:

```
runtime = RemoteRuntimeFactory
            .CreateRuntime(AppDomain.CreateDomain("foo"), setup);

private sealed class RemoteRuntimeFactory : MarshalByRefObject {
    public readonly ScriptRuntime Runtime;

    public RemoteRuntimeFactory(ScriptRuntimeSetup setup) {
```

```
            Runtime = ScriptRuntime.Create(setup);
        }

        public static ScriptRuntime CreateRuntime
                                    (AppDomain domain,
                                     ScriptRuntimeSetup setup) {
            RemoteRuntimeFactory rd
                = (RemoteRuntimeFactory)domain
                      .CreateInstanceAndUnwrap
                          (typeof(RemoteRuntimeFactory)
                               .Assembly.FullName,
                           typeof(RemoteRuntimeFactory).FullName,
                           false, BindingFlags.Default, null,
                           new object[] { setup }, null, null, null);
            return rd.Runtime;
        }
    }
```

4 API References

Need to dope in what exceptions are thrown where I say we throw one, and need to add more details about argument checking and exceptions we throw.

4.1 ScriptRuntime Class

This class is the starting point for hosting. For Level One scenarios, you just create ScriptScopes, use Globals, and use ExecuteFile. For Level Two scenarios, you can get to ScriptEngines and so on.

ScriptRuntime represents global script state. This includes referenced assemblies, a "global object" (ScriptRuntime.Globals), "published" scopes (scopes bound to a name on Globals), available language engines, etc.

ScriptRuntime has a single constructor and two convenience factory methods. You can create multiple instances of a ScriptRuntime in a single AppDomain. For information on configuring a ScriptRuntime for what languages it allows, global settings, language settings, etc., see ScriptRuntimeSetup.

4.1.1 Class Summary

```
public class ScriptRuntime : MarshalByRefObject
    public ScriptRuntime(ScriptRuntimeSetup setup)
    public static ScriptRuntime CreateFromConfiguration()
    public static ScriptRuntime
        CreateRemote(AppDomain domain, ScriptRuntimeSetup setup)

    public ScriptScope ExecuteFile(string path)
    public ScriptScope Globals { get; set;}
    public ScriptScope CreateScope()
    public ScriptScope CreateScope(IDynamicMetaObjectProvider storage)

    public ScriptEngine GetEngine(string languageId)
```

```
public ScriptEngine GetEngineByFileExtension(string extension)
public string[] GetRegisteredFileExtensions()
public string[] GetRegisteredLanguageIdentifiers()

public void LoadAssembly(Assembly assm)

public ObjectOperations Operations { get; }
public ObjectOperations CreateOperations()

public ScriptRuntimeSetup Setup { get; }
public ScriptHost Host { get; }
public ScriptIO IO { get; }
public void Shutdown()
```

4.1.2 Constructor

The constructor requires a ScriptRuntimeSetup, which gives the host full control of the languages allowed in the ScriptRuntime, their options, and the global runtime options.

This method ensures the list of languages in the setup object has no duplicate elements based on the LanguageSetup.TypeName property (just comparing them as strings at this point). Later, when engines fault in, the DLR also ensures that none of the assembly-qualified types actually identify the same type.

This method ensures the list of languages in the setup have no conflicting LangaugeSetup.Names elements or Language.FileExtensions elements.

After calling this method, modifying the ScriptRuntimeSetup object throws an exception.

Signature:

```
public ScriptRuntime(ScriptRuntimeSetup setup)
```

4.1.3 Create* Methods

These factory methods construct and return ScriptRuntimes. They primarily are for convenience and discoverability via editors that complete members on types.

CreateFromConfiguration is just a convenience for:

```
new ScriptRuntime(ScriptRuntimeSetup.ReadConfiguration())
```

CreateRemote creates the ScriptRuntime in the specified domain, instantiates the ScriptRuntimeSetup.HostType in that domain, and returns the ScriptRuntime. Any arguments specified in ScriptRuntimeSetup.HostArguments must derive from MBRO or serialize across app domain boundaries. The same holds for any values in ScriptRuntimeSetup.Options and any LanguageSetup.Options.

Signatures:

```
public static ScriptRuntime CreateFromConfiguration()
public static ScriptRuntime
    CreateRemote(AppDomain domain, ScriptRuntimeSetup setup)
```

4.1.4 ExecuteFile Method

This method executes the source identified in the path argument and returns the new ScriptScope in which the source executed. This method calls on the ScriptRuntime.Host to get the PlatformAdaptationLayer and then calls on it to resolve and open the path. ExecuteFile determines the language engine to use from the path's extension and the ScriptRuntime's configuration, comparing extensions case-insensitively.

This convenience method exists for Level 1 hosting scenarios where the path is likely an absolute pathname or a filename that naturally resolves with standard .NET BCL file open calls.

Signature:

```
public ScriptScope ExecuteFile(string path)
```

Each time this method is called it create a fresh ScriptScope in which to run the source. Calling Engine.GetScope returns the last ScriptScope created for repeated invocations of ExecuteFile on the same path.

This method adds variable name bindings within the ScriptRuntime.Globals object as appropriate for the language. Dynamic language code can then access and drill into objects bound to those names. For example, the IronPython loader adds the base file name to the ScriptRuntime.Globals as a Python module, and when IronPython is importing names, it looks in ScriptRuntime.Globals to find names to import. IronRuby's loader adds constants and modules to the ScriptRuntime.Globals object. DLR JScript adds all globals there.

In Globals, each language decides its own semantics for name conflicts, but the expected model is last-writer-wins. Languages do have the ability to add names to Globals so that only code executing in that language can see the global names. In this case, other languages would not have the ability to clobber the name bindings. For example, Python might do this for its special built-in modules. However, most names should be added so that all languages can see the bindings and interoperate with the objects bound to the names.

4.1.5 UseFile Method

This method executes the source identified in the path argument and returns the new ScriptScope in which the source executed. If the identified file was already executed, this method does NOT execute the file again, but instead the method just returns the ScriptScope. The path must have a language file extension registered with the ScriptRuntime. UseFile affects ScriptRuntime.Globals the same way ExecuteFile does.

This method is like ExecuteFile except in two ways. ExecuteFile always executes the file each time you call it, but UseFile executes the file at most once. UseFile resolves the path argument against the language engines search paths to find the file, but ExecuteFile just calls .NET open functions to find the file.

Essentially, this method finds the engine for the file extension and calls GetScope on the results of joining the path argument with each of the items in the engine's search paths. If it finds no scope, then it calls ExecuteFile on the first existing file found by joining the argument path with the items in the engine's search paths.

This convenience method exists for Level 1 hosting scenarios where the host wants to load a file of script code in the same manner a language would (for example, Python's import statement or Ruby's require function).

Signature:

```
public ScriptScope UseFile(string path)
```

4.1.6 Globals Property

This property returns the "global object" or name bindings of the ScriptRuntime as a ScriptScope. You can set the globals scope, which you might do if you created a ScriptScope with an IDynamicMetaObjectProvider so that your host could late bind names. The easiest way to provide an IDynamicMetaObjectProvider is to use ExpandObject or derive from DynamicObject.

Signature:

```
public ScriptScope Globals { get; set; }
```

4.1.7 CreateScope Method

This method returns a new ScriptScope.

Signatures:

```
public ScriptScope CreateScope()
public ScriptScope CreateScope(IDynamicMetaObjectProvider storage)
```

The storage parameter lets you supply the dictionary of the scope so that you can provide late bound values for some name lookups. If storage is null, this method throws an ArgumentNullException.

4.1.8 GetEngine Method

This method returns the one engine associated with this ScriptRuntime that matches the languageId argument, compared case-insensitively. This loads the engine and initializes it if needed.

Signature:

```
public ScriptEngine GetEngine(string languageId)
```

If languageId is null, or it does not map to an engine in the ScriptRuntime's configuration, then this method throws an exception.

4.1.9 GetEngineByFileExtension Method

This method takes a file extension and returns the one engine associated with this ScriptRuntime that matches the extension argument. This strips one leading period if extension starts with a period.

This loads the engine and initializes it if needed. The file extension associations are determined by the ScriptRuntime configuration (see configuration section above). This method compares extensions case-insensitively.

Signature:

```
public ScriptEngine GetEngineByFileExtension(string extension)
```

If extension is null, or it does not map to an engine in the ScriptRuntime's configuration, then this method throws an exception.

~~4.1.10 GetEngineByMimeType Method~~

~~This method takes a MIME type and returns the one engine associated with this ScriptRuntime that matches the argument.~~

~~This loads the engine and initializes it if needed. The MIME type associations are determined by the ScriptRuntime configuration (see configuration section above).~~

~~Signature:~~

```
public ScriptEngine GetEngineByMimeType(string mimetype);
```

~~If mimetype is null, or it does not map to an engine in the ScriptRuntime's configuration, then this method throws an exception.~~

4.1.11 GetRegisteredFileExtensions Method

This method returns an array of strings (without periods) where each element is a registered file extension for this ScriptRuntime. Each file extension maps to a language engine based on the ScriptRuntime configuration (see configuration section above). If there are none, this returns an empty array.

Signature:

```
public string[] GetRegisteredFileExtensions()
```

4.1.12 GetRegisteredLanguageIdentifiers Method

This method returns an array of strings where each element is a registered language identifier for this ScriptRuntime. Each language identifier maps to a language engine based on the ScriptRuntime configuration (see configuration section above). Typically all registered file extensions are also language identifiers. If there are no language identifiers, this returns an empty array.

Signature:

```
public string[] GetRegisteredLanguageIdentifiers()
```

4.1.13 LoadAssembly Method

This method calls on language engines to inform them of DLLs whose namespaces and types should be available to code the engines execute. Language engines can make the names available however they see fit. They may resolved free identifiers first to ScriptRuntime.Globals and then to names provided by loaded DLLs. They may make the names available to 'using',

'import', or 'require' expressions. They may add names to ScriptRuntime.Globals that are bound to dynamic objects for accessing sub namespaces and types by drilling in from root namespaces stored in Globals.

By default, the DLR seeds the ScriptRuntime with Mscorlib and System assemblies. You can avoid this by setting the ScriptRuntimeSetup option "NoDefaultReferences" to true. When new language engines load, the ScriptRuntime passes the list of loaded assemblies.

Signature:

```
public void LoadAssembly(Assembly assm)
```

The following is the old behavior that was cut in lieu of the more flexible, langauge-specific support above (kept here should it come back before adding these APIs to .NET):

~~walks the assembly's namespaces and adds name bindings in ScriptRuntime.Globals to represent namespaces available in the assembly. Each top-level namespace name becomes a name in Globals, bound to a dynamic object representing the namespace. Within each top-level namespace object, the DLR binds names to dynamic objects representing each sub namespace or type.~~

~~*There is a bug today that instead of dynamic objects representing namespaces and types, the DLR stores NamespaceTrackers and TypeTrackers, which only implement IAttributesCollection. We are considering cutting these types, perhaps even leaving all reflection with dynamic objects to the langauge. See the Sympl language example for what it does. Production-quality languages may want something as sophisticated as the tracker objects, which IronPython will continue to use from an IronPython DLL.*~~

~~By default, the DLR seeds the ScriptRuntime with Mscorlib and System assemblies. You can avoid this by setting the ScriptRuntimeSetup option "NoDefaultReferences" to true.~~

~~When this method encounters the same fully namespace-qualified type name, it merges names together objects representing the namespaces. If you called LoadAssembly on two different assemblies, each contributing to System.Foo.Bar namespace, then all names within System.Foo.Bar from both assemblies will be present in the resulting object representing Bar.~~

4.1.14 Operations Property

Likely cutting this from the hosting APIs soon. We've shifted our thinking away from having a language-invariant static helper since it can't do very much without baking in language choices, like implicit conversions and whatnot just to invoke objects.

This property returns a default, language-neutral ObjectOperations. ObjectOperations lets you perform various operations on objects. When the objects do not provide their own behaviors for performing the operations, this ObjectOperations uses general .NET semantics. Because there are many situations when general .NET semantics are insufficient due to dynamic objects often not using straight .NET BCL types, this ObjectOperations will throw exceptions when one produced by a ScriptEngine would succeed.

Because an ObjectOperations object caches rules for the types of objects and operations it processes, using the default ObjectOperations for many objects could degrade the caching

benefits. Eventually the cache for some operations could degrade to a point where ObjectOperations stops caching and does a full search for an implementation of the requested operation for the given objects. For simple hosting situations, this is sufficient behavior.

See CreateOperations for alternatives.

Signature:

```
public ObjectOperations Operations { get; }
```

4.1.15 CreateOperations Methods

Likely cutting this from the hosting APIs soon. We've shifted our thinking away from having a language-invariant static helper since it can't do very much without baking in language choices, like implicit conversions and whatnot just to invoke objects.

These methods return a new ObjectOperations object. See the Operations property for why you might want to call this and for limitations of ObjectOperations provided by a ScriptRuntime instead of one obtained from a ScriptEngine.

There currently is little guidance on how to choose when to create new ObjectOperations objects. However, there is a simple heuristic. If you were to perform some operations over and over on the same few types of objects, it would be advantageous to create an ObjectOperations just for use with those few cases. If you perform different operations with many types of objects just once or twice, you can use the default instance provided by the ObjectOperations property.

Signature:

```
public ObjectOperations CreateOperations()
```

4.1.16 Setup Property

This property returns a read-only ScriptRuntimeSetup object describing the configuration information used to create the ScriptRuntime.

Signature:

```
public ScriptRuntimeSetup Setup { get; }
```

4.1.17 Host Property

This property returns the ScriptHost associated with the ScriptRuntime. This is not settable because the ScriptRuntime must create the host from a supplied type to support remote ScriptRuntime creation. Setting it would also be bizarre because it would be similar to changing the owner of the ScriptRuntime.

Signature:

```
public ScriptHost Host { get; }
```

4.1.18 IO Property

This property returns the ScriptIO associated with the ScriptRuntime. The ScriptIO lets you control the standard input and output streams for code executing in the ScriptRuntime.

Signature:

```
public ScriptIO IO { get; }
```

4.1.19 Shutdown Method

This method announces to the language engines that are loaded that the host is done using the ScriptRuntime. Languages that have a shutdown hook or mechanism for code to release system resources on shutdown will invoke their shutdown protocols.

There are no other guarantees from this method. For example, It is undefined when code executing (possibly on other threads) will stop running. Also, any calls on the ScriptRuntime, hosting API objects associated with the runtime, or dynamic objects extracted from the runtime have undefined behavior.

Signature:

```
public void Shutdown()
```

4.2 ScriptScope Class

This class represents a namespace essentially. Hosts can bind variable names in ScriptScopes, fetch variable values, etc. Hosts can execute code within scopes for distinct name bindings.

ScriptScopes also have some convenience members and an optional language affinity. Scopes use the language to look up names and convert values. If the ScriptScope has no default language, then these convenience methods throw an exception, and the Engine property returns null.

Hosts can store ScriptScopes as the values of names on ScriptRuntime.Globals or in other scopes. When dynamic language code encounters a ScriptScope as an object, the DLR manifests the scope as a dynamic object. This means that normal object member access sees the variables stored in the ScriptScope first. Languages executing code that is doing the object member access get a chance to find members if the members are not variables in the ScriptScope. The language might bind to the .NET static type members documented here. They might detect the .NET type is ScriptScope and throw a missing member exception, use meta-programming hooks to give dynamic language code a chance to produce the member, or return sentinel objects according to the language's semantics.

Hosts can use ScriptScopes (regardless of whether they have a language affinity) to execute any kind of code within their namespace context. ScriptEngine methods that execute code take a ScriptScope argument. There are parallel methods on engines for getting and setting variables so that hosts can request a name lookup with any specific language's semantics in any ScriptScope.

You create instances of ScriptScopes using the CreateScope and ExecuteFile methods on ScriptRuntimes or CreateScope on ScriptEngine.

Note, members that take or return ObjectHandles are not present on Silverlight.

4.2.1 Class Summary

```
public class ScriptScope : MarshalByRefObject {
```

```
public object GetVariable(string name)
public ObjectHandle GetVariableHandle(string name)
public bool RemoveVariable(string name)
public void SetVariable(string name, object value)
public void SetVariable(string name, ObjectHandle handle)
public bool TryGetVariable(string name, out object value)
public bool TryGetVariableHandle(string name,
                                 out ObjectHandle handle)
public T GetVariable<T>(string name)
public bool TryGetVariable<T>(string name, out T value)
public bool ContainsVariable(string name)

public IEnumerable<string> GetVariableNames()
public IEnumerable<KeyValuePair<string, object>> GetItems()

public ScriptEngine Engine { get;}
```

4.2.2 GetVariable* Methods

These methods fetch the value of a variable stored in the scope.

If there is no engine associated with the scope (see ScriptRuntime.CreateScope), then the name lookup is a case-sensitive, literal lookup of the name in the scope's dictionary. If there is a default engine, then the name lookup uses that language's semantics.

Signatures:

```
public object GetVariable(string name)
public ObjectHandle GetVariableHandle(string name)
public T GetVariable<T>(string name)
```

GetVariableHandle is useful when the ScriptScope is remote so that you get back an ObjectHandle referring to the value.

GetVariable<T> uses language-specific (based on the default language in the Engine property) conversions. These may be implicit only, or include explicit conversions too. This method throws a NotSupportedException if the engine cannot perform the requested type conversion. If there is no associated engine, this method essentially just casts to T, which could throw an ArgumentException.

If you need an explicit conversion to T, you can use scope.Engine.Operations.ExplicitConvertTo<T>.

4.2.3 SetVariable Methods

These methods assign a value to a variable in the scope, overwriting any previous value.

If there is no engine associated with the scope (see ScriptRuntime.CreateScope), then the name mapping is a case-sensitive, literal mapping of the name in the scope's dictionary. If there is a default engine, then the name lookup uses that language's semantics.

Signatures:

```
public void SetVariable(string name, object value)
public void SetVariable(string name, ObjectHandle handle)
```

4.2.4 TryGetVariable* Methods

These methods fetch the value of a variable stored in the scope and return a Boolean indicating success of the lookup. When the method's result is false, then it assigns null to value.

If there is no engine associated with the scope (see ScriptRuntime.CreateScope), then the name lookup is a case-sensitive, literal lookup of the name in the scope's dictionary. If there is a default engine, then the name lookup uses that language's semantics.

Signatures:

```
public bool TryGetVariable(string name, out object value)
public bool TryGetVariableHandle(string name,
                                 out ObjectHandle handle)
public bool TryGetVariable<T>(string name, out T value)
```

TryGetVariableHandle is useful when the ScriptScope is remote so that you get back an ObjectHandle referring to the value.

TryGetVariable<T> uses language-specific (based on the default language in the Engine property) conversions. These may be implicit only, or include explicit conversions too. It throws a NotSupportedException if the engine cannot perform the requested type conversion. If there is no associated engine, this method uses standard .NET conversion, which could throw an ArgumentException.

If you need an explicit conversion to T, you can use scope.Engine.Operations.TryExplicitConvertTo<T>.

4.2.5 ContainsVariable Method

This method returns whether the variable is exists in this scope and has a value.

If there is no engine associated with the scope (see ScriptRuntime.CreateScope), then the name lookup is a literal lookup of the name in the scope's dictionary. Therefore, it is case-sensitive for example. If there is a default engine, then the name lookup uses that language's semantics.

Signature:

```
public bool ContainsVariable(string name)
```

4.2.6 GetVariableNames Method

This method returns an enumeration of strings, one string for each variable name in this scope. If there are no names, then it returns an empty array. Modifying the array has no impact on the ScriptScope. This method returns a new instance for the result of each call.

Signature:

```
public IEnumerable<string> GetVariableNames()
```

4.2.7 GetItems Method

This method returns an IEnumerable of variable name/value pairs, one for each variable name in this scope. If there are no names, then the enumeration is empty. Modifying the array has no

impact on the ScriptScope. This method returns a new instance for the result of each call, and modifying the scope while using the enumeration has undefined behavior.

Signature:

```
public IEnumerable<KeyValuePair<string, object>> GetItems ()
```

4.2.8 RemoveVariable Method

This method removes the variable name and returns whether the variable existed and had a value in the scope when you called this method.

If there is no engine associated with the scope (see ScriptRuntime.CreateScope), then the name lookup is a literal lookup of the name in the scope's dictionary. Therefore, it is case-sensitive for example. If there is a default engine, then the name lookup uses that language's semantics.

Some languages may refuse to remove some variables. If the scope has an associated language that has variables that cannot be removed, and name identifies such a variable, it is undefined what happens. Languages vary on whether this is a no-op or exceptional.

Signature:

```
public bool RemoveVariable(string name)
```

4.2.9 Engine Property

This property returns the engine associated with this scope. If the scope was created without a language affinity, then this property returns null.

Signature:

```
public ScriptEngine Engine { get;}
```

4.3 ScriptEngine Class

ScriptEngines represent a language implementation in the DLR, and they are the work horse for intermediate and advanced hosting scenarios. ScriptEngines offer various ways to execute code and create ScriptScopes and ScriptSources. ScriptSources offer methods for executing code in various ways from different kinds of sources. ScriptEngines offer the more common or convenience methods for executing code.

There is only one instance of a ScriptEngine for a given language in a given ScriptRuntime. You get to engines with ScriptRuntime's methods or the Engine property of ScriptScope.

Note, members that take or return ObjectHandles are not present on Silverlight.

4.3.1 Class Summary

```
public class ScriptEngine : MarshalByRefObject {
    internal ScriptEngine()
    public ScriptRuntime Runtime { get; }
    public string LanguageDisplayName { get; }
    public string[] GetRegisteredIdentifiers()
    public string[] GetRegisteredExtensions()

    public object Execute(string expression)
```

```
public object Execute(string expression, ScriptScope scope)
public T Execute<T>(string code)
public T Execute<T>(string expression, ScriptScope scope)
public ObjectHandle ExecuteAndWrap(string expression)
public ObjectHandle ExecuteAndWrap(string expression,
                                   ScriptScope scope)
public ScriptScope ExecuteFile(string path)
public ScriptScope ExecuteFile(string path, ScriptScope scope)

public ScriptScope GetScope(string path)

public ObjectOperations Operations { get; }
public ObjectOperations CreateOperations()
public ObjectOperations CreateOperations(ScriptScope Scope)

public ScriptSource CreateScriptSourceFromString
    (string expression)
public ScriptSource CreateScriptSourceFromString
    (string expression, string path)
public ScriptSource CreateScriptSourceFromString
    (string code, SourceCodeKind kind)
public ScriptSource CreateScriptSourceFromString
    (string code, string path, SourceCodeKind kind)

public ScriptSource CreateScriptSourceFromFile(string path)
public ScriptSource CreateScriptSourceFromFile
    (string path, System.Text.Encoding encoding)
public ScriptSource CreateScriptSourceFromFile
    (string path, System.Text.Encoding encoding,
     SourceCodeKind kind)

public ScriptSource CreateScriptSource
    (StreamContentProvider contentProvider, string path)
public ScriptSource CreateScriptSource
   (StreamContentProvider contentProvider, string path,
    System.Text.Encoding encoding)
public ScriptSource CreateScriptSource
    (StreamContentProvider contentProvider, string path,
     System.Text.Encoding encoding, SourceCodeKind kind)
public ScriptSource CreateScriptSource
    (TextContentProvider contentProvider, string path,
     SourceCodeKind kind)

public ScriptSource CreateScriptSource(CodeObject content)
public ScriptSource CreateScriptSource(CodeObject content,
                                       string path)
public ScriptSource CreateScriptSource(CodeObject content,
                                       SourceCodeKind kind)
public ScriptSource CreateScriptSource
                    (System.CodeDom.CodeObject code,
                     string path, SourceCodeKind kind)

public ScriptScope CreateScope()
public ScriptScope CreateScope(IDynamicMetaObjectProvider globals)

public ServiceType GetService<ServiceType>(params object[] args)
                              where ServiceType : class
```

```
public LanguageSetup Setup { get; }
public CompilerOptions GetCompilerOptions()
public CompilerOptions GetCompilerOptions(ScriptScope scope)

public ICollection<string> GetSourceSearchPaths()
public void SetSearchPaths (ICollection<string> paths)
public System.Version LanguageVersion { get; }
```

4.3.2 Runtime Property

This property returns the ScriptRuntime for the context in which this engine executes.

Signature:

```
public ScriptRuntime Runtime { get; }
```

4.3.3 LanguageDisplayName Property

This property returns a display name for the engine or language that is suitable for UI.

Signature:

```
public string LanguageDisplayName { get; }
```

4.3.4 GetRegistered* Methods

These methods return unique identifiers for this engine and file extensions that map to this engine and its language. This information comes from configuration data passed to ScriptRuntime.Create.

Modifying the results of these methods has no effect on configuration of this engine.

Signatures:

```
public string[] GetRegisteredIdentifiers()
public string[] GetRegisteredExtensions()
```

4.3.5 Execute* Methods

These methods execute the strings as expressions and return a result in various ways. There are complementary overloads that take a ScriptScope. The overloads that do not take scopes create a new scope for each execution. These methods throw the scope away and use it for side effects only, returning the result in the same way the complementary overload does.

Execute<T> returns the result as the specified type, using the engine's Operations.ConvertTo<T> method. If this method cannot convert to the specified type, then it throws a NotSupportedException.

ExecuteAndWrap returns an ObjectHandle for use when the engine and/or scope are remote.

Signatures:

```
public object Execute(string expression)
public object Execute(string expression, ScriptScope scope)
public T Execute<T>(string expression)
public T Execute<T>(string expression, ScriptScope scope)
```

```
public ObjectHandle ExecuteAndWrap(string expression)
public ObjectHandle ExecuteAndWrap(string expression,
                                   ScriptScope scope)
```

4.3.6 ExecuteFile Methods

These methods execute the strings the contents of files and return the scope in which the string executed. The overload that does not take a ScriptScope creates a new one each time it is called.

Signatures:

```
public ScriptScope ExecuteFile(string path)
public ScriptScope ExecuteFile(string path, ScriptScope scope)
```

4.3.7 GetScope Method

This method returns the ScriptScope in which the specified path/source executed. This method works in conjunction with LoadFile and language implementer APIs for loading dynamic language libraries (see LoadFile's side note). The path argument needs to match a ScriptSource's Path property because it is the key to finding the ScriptScope. Hosts need to make sure they create ScriptSources (see ScriptHost as well as methods on ScriptEngine) with their Path properties set appropriately (for example, resolving relative paths to canonical full pathnames, FileInfo.FullPath for standard .NET resolved paths).

GetScope is primarily useful for tools that need to map files to their execution scopes when the tool did not create the scope. For example, an editor and interpreter tool might execute a file, Foo, that imports or requires a file, Bar. The editor end user might later open the Bar and want to execute expressions in its context. The tool would need to find Bar's ScriptScope for setting the appropriate context in its interpreter window. This method helps with this scenario.

Languages may return null. For example, Ruby's require expression executes a file's contents in the calling scope. Since Ruby does not have a distinct scope in which the file executed in this case, they return null for such files.

Signature:

```
public ScriptScope GetScope(string path)
```

4.3.8 Operations Property

This property returns a default ObjectOperations for the engine. ObjectOperations lets you perform various operations on objects. Because an ObjectOperations object caches rules for the types of objects and operations it processes, using the default ObjectOperations for many objects could degrade the caching benefits. Eventually the cache for some operations could degrade to a point where ObjectOperations stops caching and does a full search for an implementation of the requested operation for the given objects. For simple hosting situations, this is sufficient behavior.

See CreateOperations for alternatives.

Signature:

```
public ObjectOperations Operations { get; }
```

4.3.9 CreateOperations Methods

These methods return a new ObjectOperations object. See the Operations property for why you might want to call this.

There currently is little guidance on how to choose when to create new ObjectOperations objects. However, there is a simple heuristic. If you were to perform some operations over and over on the same few types of objects, it would be advantageous to create an ObjectOperations just for use with those few cases. If you perform different operations with many types of objects just once or twice, you can use the default instance provided by the ObjectOperations property.

Signature:

```
public ObjectOperations CreateOperations()
public ObjectOperations CreateOperations(ScriptScope Scope)
```

The overload that takes a ScriptScope supports pretty advanced or subtle scenarios. It allows you to get an ObjectOperations that uses the execution context built up in a ScriptScope. For example, the engine affiliated with the scope could be IronPython, and you could execute code that did an "import clr" or "from __future__ import true_division". These change execution behaviors within that ScriptScope. If you obtained objects from that scope or executing expressions in that scope, you may want to operate on those objects with the same execution behaviors; however, you generally do not need to worry about these subtleties for typical object interactions.

4.3.10 CreateScriptSourceFromString Methods

These methods return ScriptSource objects from string contents. These are factory methods for creating ScriptSources with this language binding.

The default SourceCodeKind is AutoDetect.

The ScriptSource's Path property defaults to null. When path is non-null, if executing the resulting ScriptSource would create a ScriptScope, then path should map to the ScriptScope via GetScope.

Signatures:

```
public ScriptSource CreateScriptSourceFromString
    (string expression)
public ScriptSource CreateScriptSourceFromString
    (string expression, string path)
public ScriptSource CreateScriptSourceFromString
    (string code, SourceCodeKind kind)
public ScriptSource CreateScriptSourceFromString
    (string code, string path, SourceCodeKind kind)
```

4.3.11 CreateScriptSourceFromFile Methods

These methods return ScriptSource objects from file contents. These are factory methods for creating ScriptSources with this language binding. The path's extension does NOT have to be registered or valid for the engine. This method does NOT go through the PlatformAdaptationLayer to open the file; it goes directly to the file system via .NET.

The default SourceCodeKind is File.

The ScriptSource's Path property will be the path argument, which needs to be in some canonical form according to the host if the host is using GetScope to find the source's execution context later.

Creating the ScriptSource does not open the file. Any exceptions that will be thrown on opening or reading the file happen when you use the ScriptSource to execute or compile the source.

The encoding defaults to the platform encoding.

Signatures:

```
public ScriptSource CreateScriptSourceFromFile(string path)
public ScriptSource CreateScriptSourceFromFile
    (string path, System.Text.Encoding encoding)
public ScriptSource CreateScriptSourceFromFile
    (string path, System.Text.Encoding encoding,
     SourceCodeKind kind)
```

4.3.12 CreateScriptSource Methods

These methods returns a ScriptSource based on a CodeDom object or content providers. This is a factory method for creating a ScriptSources with this language binding.

```
public ScriptSource CreateScriptSource
    (StreamContentProvider contentProvider, string path)
public ScriptSource CreateScriptSource
   (StreamContentProvider contentProvider, string path,
    System.Text.Encoding encoding)
public ScriptSource CreateScriptSource
    (StreamContentProvider contentProvider, string path,
     System.Text.Encoding encoding, SourceCodeKind kind)
public ScriptSource CreateScriptSource
    (TextContentProvider contentProvider, string path,
     SourceCodeKind kind)

public ScriptSource CreateScriptSource
                    (System.CodeDom.CodeObject code,
                     string path, SourceCodeKind kind)
public ScriptSource CreateScriptSource(CodeObject content)
public ScriptSource CreateScriptSource(CodeObject content,
                                       string path)
public ScriptSource CreateScriptSource(CodeObject content,
                                       SourceCodeKind kind)
public ScriptSource CreateScriptSource
                    (System.CodeDom.CodeObject code,
                     string path, SourceCodeKind kind)
```

The method taking a TextContentProvider lets you supply input from Unicode strings or stream readers. This could be useful for implementing a TextReader over internal host data structures, such as an editor's text representation.

The method taking a StreamContentProvider lets you supply binary (sequence of bytes) stream input. This is useful when opening files that may contain language-specific encodings that are marked in the first few bytes of the file's contents. There is a default StreamContentProvider

used internally if you call CreateScriptSourceFromFile. The encoding defaults to the platform encoding if the language doesn't recognize some other encoding (for example, one marked in the file's first few bytes).

The method taking a System.CodeDom.CodeObject, and the expected CodeDom support is extremely minimal for syntax-independent expression of semantics. Languages may do more, but hosts should only expect CodeMemberMethod support, and only sub nodes consisting of the following:

- CodeSnippetStatement
- CodeSnippetExpression
- CodePrimitiveExpression
- CodeMethodInvokeExpression
- CodeExpressionStatement (for holding MethodInvoke)

This support exists primarily for ASP.NET pages that contain snippets of DLR languages, and these requirements were very limited. When the CodeObject argument does not match this specification, you will get a type cast error, but if the language supports more options, you could get different errors per engine.

The path argument in all cases is a unique ID that the host may use to retrieve the scope in which the source executes via Engine.GetScope.

4.3.13 CreateScope Method

This method returns a new ScriptScope with this engine as the default language for the scope.

Signatures:

```
public ScriptScope CreateScope()
public ScriptScope CreateScope(IDynamicMetaObjectProvider globals)
```

The globals parameter lets you supply the dictionary of the scope so that you can provide late bound values for some name lookups. The easiest way to supply your own dictionary is to use ExpandoObject or derive from DynamicObject .

4.3.14 GetService Method

This method returns a language-specific service. It provides a point of extensibility for a language implementation to offer more functionality than the standard engine members discussed here. If the specified service is not available, this returns null.

Signature:

```
public ServiceType GetService<ServiceType>(params object[] args)
                                where ServiceType : class
```

The following are services expected to be supported:

ExceptionOperations	This duplicates some members of Exception and can return a string in the style of this engine's language to describe the exception argument.
TokenCategorizer	This is for building tools that want to scan languages and

	get token info, such as colorization categories. *This type will change and be spec'ed external to this document eventually, see the section below for this type.*
`OptionsParser`	This can parse a command shell (cmd.exe) style command line string. Hosts that are trying to be an interactive console or incorporate standard command line switches of a language's console can get the engine's command line parser. *This is a place holder for DLR v2. Its design will definitely change. We have a big open issue to redesign language and DLR support for building interactive UI, interpreters, tools, etc., with some common support around command lines and consoles.*
`CommandLine`	is a helper object for parsing and processing interactive console input, maintaining a history of input, etc. *This is a place holder for DLR v2. Its design will definitely change. We have a big open issue to redesign language and DLR support for building interactive UI, interpreters, tools, etc., with some common support around command lines and consoles.*
`ScriptConsole`	This is a helper object for the UI of an interpreter, how output is displayed and how we get input. If the language does not implement a ScriptConsole, there is a default Console object they can return. *This is a place holder for DLR v2. Its design will definitely change. We have a big open issue to redesign language and DLR support for building interactive UI, interpreters, tools, etc., with some common support around command lines and consoles. Need to distinguish this and CommandLine.*

4.3.15 Setup Property

This property returns a read-only LanguageSetup describing the configuration used to instantiate this engine.

Signature:

```
public LanguageSetup Setup { get; }
```

4.3.16 GetCompilerOptions Method

This method returns the compiler options object for the engine's language. The overload that takes a ScriptScope returns options that represent any accrued imperative options state from the scope (for example, "from futures import truedivision" in python). To operate on the

options before passing them to ScriptSource.Compile, for example, you may need to cast the result to the documented subtype of CompilerOptions for the language you're manipulating.

If scope is null, this throws an ArgumentNullException.

Signatures:

```
public CompilerOptions GetCompilerOptions()
public CompilerOptions GetCompilerOptions(ScriptScope scope)
```

CompilerOptions type will likely change by the time the DLR Hosting APIs move into the .NET libraries, possibly becoming Dictionary<str,obj>.

4.3.17 GetSearchPaths Method

This method returns the search paths used by the engine for loading files when a script wants to import or require another file of code. These are also the paths used by ScriptRuntime.UseFile.

These paths do not affect ScriptRuntime.ExecuteFile. The ScriptHost's PlatformAdaptationLayer (or the default's direct use of .NET file APIs) controls partial file name resolution for ExecuteFile.

Signature:

```
public ICollection<string> GetSearchPaths ()
```

4.3.18 SetSearchPaths Method

This method sets the search paths used by the engine for loading files when a script wants to import or require another file of code. Setting these paths affects ScriptRuntime.UseFile.

These paths do not affect ScriptRuntime.ExecuteFile. The ScriptHost's PlatformAdaptationLayer (or the default's direct use of .NET file APIs) controls partial file name resolution for ExecuteFile.

Signature:

```
public void SetSearchPaths (ICollection<string> paths)
```

4.3.19 LanguageVersion Property

This property returns the language's version.

Signature:

```
public System.Version LanguageVersion { get; }
```

4.4 ScriptSource Class

ScriptSource represents source code and offer a variety of ways to execute or compile the source. You can get ScriptSources from factory methods on ScriptEngine, and ScriptSources are tied to the engine that created them. The associated engine provides the execution and compilation semantics for the source.

ScriptSources have properties that direct the parsing of and report aspects of the source. For example, the source could be marked as being an expression or a statement, for language that need to distinguish expressions and statements semantically for how to parse them. The code

could be marked as being interactive, which means the language's parser should handle standard interpreter affordances the language might support (for example, Python's "_" variable or VB's "?" syntax).

ScriptSources also have a Path property. This is mostly useful for those marked as being a file. The Path is the key for engines recognizing ScriptSources they have seen before so that they do not repeatedly load files when load-once semantics should apply (see ScriptEngine.LoadFile). The Path also helps the engine find the ScriptScope the file executed in (see ScriptEngine.GetScope), which is useful for some tool host scenarios. The host defines what a canonical representation of a path is. The host needs to set the path to the same string when it intends for ScriptSources to match for the purposes of the above functions on ScriptEngine.

You can create ScriptSource objects with factory methods on ScriptEngine.

Note, members that take or return ObjectHandles are not present on Silverlight.

4.4.1 Class Summary

```
public sealed class ScriptSource : MarshalByRefObject {
  internal ScriptSource()
  public string Path { get; }
  public SourceCodeKind Kind { get;}

  public ScriptCodeParseResult GetCodeProperties ()
  public ScriptCodeParseResult GetCodeProperties
                               (CompilerOptions options)

  public ScriptEngine Engine { get; }

  public CompiledCode Compile()
  public CompiledCode Compile(ErrorListener sink)
  public CompiledCode Compile(CompilerOptions options)
  public CompiledCode Compile(CompilerOptions options,
                              ErrorListener sink)

  public object Execute()
  public object Execute(ScriptScope scope)
  public ObjectHandle ExecuteAndWrap ()
  public ObjectHandle ExecuteAndWrap (ScriptScope scope)
  public T Execute<T>()
  public T Execute<T>(ScriptScope scope)
  public int ExecuteProgram()

  public ScriptCodeReader GetReader()
  public Encoding DetectEncoding()

  // line/file mapping:
  public string GetCode()
  public string GetCodeLine(int line)
  public string[] GetCodeLines(int start, int count)
  public SourceSpan MapLine(SourceSpan span)
  public SourceLocation MapLine(SourceLocation loc)
  public int MapLine(int line)
  public string MapLineToFile(int line)
```

4.4.2 Path Property

This property returns the identifier for this script source. In many cases the Path doesn't matter. It is mostly useful for file ScriptSources. The Path is the key for engines to recognize ScriptSources they have seen before so that they do not repeatedly load files when load-once semantics should apply. The Path also helps the engine find the ScriptScope the file executed in, which is useful for some tool host scenarios (see ScriptEngine.GetScope).

The Path is null if not set explicitly on construction. The path has the value the ScriptSource was created with. In the case of relative file paths, for example, the DLR does not convert them to absolute or canonical representations.

Signature:

```
public string Path { get; }
```

4.4.3 Kind Property

This property returns the kind of source this ScriptSource represents. This property is a hint to the ScriptEngine how to parse the code ScriptSource (as an expression, statement, whole file, etc.).

If you're unsure, File can be used to direct the language to generally parse the code. For languages that are expression-based, they should interpret Statement as Expression.

Signature:

```
public SourceCodeKind Kind { get;}
```

4.4.4 GetCodeProperties Methods

This method returns the properties of the code to support tools. The values indicate the state of parsing the source relative to completeness, or whether the source is complete enough to execute.

Signature:

```
public ScriptCodeParseResult GetCodeProperties ()
public ScriptCodeParseResult GetCodeProperties
                             (CompilerOptions options)
```

CompilerOptions type will likely change by the time the DLR Hosting APIs move into the .NET libraries, possibly becoming Dictionary<str,obj>.

4.4.5 Engine Property

This property returns the language engine associated with this ScriptSource. There is always a language tied to the source for convenience. Also, we do not think it is useful to support having a piece of code that could perhaps be parsed by multiple languages.

Signature:

```
public ScriptEngine Engine { get; }
```

4.4.6 Compile Methods

These methods compile the source and return a CompileCode object that can be executed repeatedly in its default scope or in other scopes without having to recompile the code.

Each call to Compile returns a new CompiledCode object. Each call to Compile always calls on its content provider to get sources, and the default file content provider always re-opens the file and reads its contents.

Signatures

```
public CompiledCode Compile()
public CompiledCode Compile(ErrorListener sink)
public CompiledCode Compile(CompilerOptions options)
public CompiledCode Compile(CompilerOptions options,
                            ErrorListener sink)
```

If any arguments are null, these throw ArgumentNullExceptions.

If you supply an error listener, and there were errors, these methods return null. Otherwise, it leaves any raised exceptions unhandled.

These methods do not take a ScriptScope to compile against. That would prevent compilation from choosing optimized scope implementations. You can always execute compiled code against any scope (see Execute* methods).

CompilerOptions type will likely change by the time the DLR Hosting APIs move into the .NET libraries, possibly becoming Dictionary<str,obj>.

4.4.7 Execute* Methods

These methods execute the source code and return a result in various ways. There are complementary overloads that take a ScriptScope and those that do not. The overloads with no arguments create a new scope for each execution. These methods throw the scope away and use it for side effects only, returning the result in the same way the complementary overload does.

These methods always execute the ScriptSource. Even when the source is a file, and the associated engine's language has an execute-at-most-once mechanism, these methods always execute the source contents.

Each call to Execute always calls on its content provider to get sources, and the default file content provider always re-opens the file and reads its contents.

Signatures:

```
public object Execute()
public object Execute(ScriptScope scope)
public ObjectHandle ExecuteAndWrap ()
public ObjectHandle ExecuteAndWrap (ScriptScope scope)
public T Execute<T>()
public T Execute<T>(ScriptScope scope)
public int ExecuteProgram()
```

Execute returns an object that is the resulting value of running the code. When the ScriptSource is a file or statement, the language decides what is an appropriate value to return. Some languages return the value produced by the last expression or statement, but languages that are not expression based may return null.

ExecuteAndWrap returns an ObjectHandle for use when the engine and/or scope are remote.

Execute<T> returns the result as the specified type, using the associated engine's Operations.ConvertTo<T> method. If this method cannot convert to the specified type, then it throws an exception.

ExecuteProgram runs the source as though it were launched from an OS command shell and returns a process exit code indicating the success or error condition of executing the code. Each time this method is called it creates a fresh ScriptScope in which to run the source, and if you were to use ScriptEngine.GetScope, you'd get whatever last ScriptScope the engine created for the source.

4.4.8 GetReader Method

This method returns a derived type of TextReader that is bound to this ScriptSource. Every time you call this method you get a new ScriptCodeReader reset to beginning parsing state, and no two instances interfere with each other.

Signature:

```
public ScriptCodeReader GetReader()
```

4.4.9 DetectEncoding Method

This method returns the encoding for the source. The language associated with the source has the chance to read the beginning of the file if it has any special handling for encodings based on the first few bytes of the file. This method could return an encoding different than what the source was created with.

Signature:

```
public Encoding DetectEncoding()
```

4.4.10 GetCode Method

This method returns all the source code contents as a string. The result may share storage with the string passed to create the ScriptSource.

Each call to GetCode always calls on its content provider to get sources, and the default file content provider always re-opens the file and reads its contents.

Signature:

```
public string GetCode()
```

4.4.11 GetCodeLine* Methods

These methods return a string (or strings) for the line (or lines) indexed. Count is one-based. The count argument can be greater than the number of lines. The start argument cannot be zero or negative.

The line and count arguments can cause indexing to go beyond the end of the source. GetCodeLine returns null in that case. GetCodeLines returns strings only for existing lines and does not throw an exception or include nulls in the array. If start is beyond the end, the result is an empty array.

Signatures:

```
public string GetCodeLine(int line)
public string[] GetCodeLines(int start, int count)
```

4.4.12 MapLine Methods

These methods map physical line numbers to virtual line numbers for reporting errors or other information to users. These are useful for languages that support line number directives for their parsers and error reporting.

Signatures:

```
public SourceSpan MapLine(SourceSpan span)
public SourceLocation MapLine(SourceLocation loc)
public int MapLine(int line)
```

4.4.13 MapLineToFile Method

This method maps a physical line number to a .NET CLR pdb or file with symbol information in it. The result is an absolute path or relative path that resolves in a standard .NET way to the appropriate file.

Signature:

```
public string MapLineToFile(int line)
```

4.5 CompiledCode Class

CompiledCode represents code that has been compiled to execute repeatedly without having to compile it each time, and it represents the default ScriptScope the code runs in. The default scope may have optimized variable storage and lookup for the code. You can always execute the code in any ScriptScope if you need it to execute in a clean scope each time, or you want to accumulate side effects from the code in another scope.

You can get CompiledCode from Compile methods on ScriptSource. CompiledCode objects have an internal reference to the engine that produced them. Because they have a default scope in which to execute, and the use for CompiledCode objects is to execute them, they have several execute methods.

Note, members that take or return ObjectHandles are not present on Silverlight.

4.5.1 Class Summary

```
public class CompiledCode : MarshalByRefObject {
    internal CompiledCode()
    public ScriptScope DefaultScope { get; }
    public ScriptEngine Engine { get; }
    public object Execute()
    public object Execute(ScriptScope scope)
```

```
public ObjectHandle ExecuteAndWrap()
public ObjectHandle ExecuteAndWrap(ScriptScope scope)
public T Execute<T>()
public T Execute<T>(ScriptScope scope)
```

4.5.2 DefaultScope Property

This property returns the default ScriptScope in which the code executes. This allows you to extract variable values after executing the code or insert variable bindings before executing the code.

Signature:

```
public ScriptScope DefaultScope { get; }
```

4.5.3 Engine Property

This property returns the engine that produced the compiled code.

Signature:

```
public ScriptEngine Engine { get; }
```

4.5.4 Execute* Methods

These methods execute the compiled code in a variety of ways. Half of the overloads do the same thing as their complement, one executes in the default scope while the other takes a ScriptScope in which to execute the code. If invoked on null, this throws an ArgumentNullException.

Signatures:

```
public object Execute()
public object Execute(ScriptScope scope)
public ObjectHandle ExecuteAndWrap()
public ObjectHandle ExecuteAndWrap(ScriptScope scope)
public T Execute<T>()
public T Execute<T>(ScriptScope scope)
```

ExecuteAndWrap returns an ObjectHandle for use when the engine and/or scope are remote.

Execute<T> returns the result as the specified type, using the engine's Operations.ConvertTo<T> method. If this method cannot convert to the specified type, then it throws an exception.

4.6 ObjectOperations Class

This utility class provides operations on objects. The operations work on objects emanating from a ScriptRuntime or straight up .NET static objects. The behaviors of this class are language-specific, depending on which language owns the instance you're using.

You get ObjectOperations objects from ScriptEngines. The operations have a language-specific behavior determined by the engine from which you got the ObjectOperations object. For example, calling GetMember on most objects to get the "__dict__" member using an ObjectOperations obtained from an IronPython ScriptEngine will return the object's dictionary

of members. However, using an ObjectOperations obtained from an IronRuby engine, would raise a member missing exception.

The reason ObjectOperations is a utility class that is not static is that the instances provide a context of caching for performing the operations. If you were to perform several operations over and over on the same few objects, it would be advantageous to create a special ObjectOperations just for use with those few objects. If you perform different operations with many objects just once or twice, you can use the default instance provided by the ScriptEngine.

Half of the methods do the same thing as their complement, one works with objects of type Object while the other works with ObjectHandles. We need the overloads for clear method selection and to allow for an ObjectHandle to be treated as Object should that be interesting.

You obtain ObjectOperation objects from ScriptEngines' Operations property and CreateOperations method.

Note, members that take or return ObjectHandles are not present on Silverlight.

4.6.1 Class Summary

```
public sealed class ObjectOperations : MarshalByRefObject {
    public ScriptEngine Engine { get; }
    public ObjectHandle Add(ObjectHandle self, ObjectHandle other)
    public Object Add(Object self, Object other)
    public Object BitwiseAnd(Object self, Object other)
    public ObjectHandle BitwiseAnd(ObjectHandle self,
                                   ObjectHandle other)
    public ObjectHandle BitwiseOr(ObjectHandle self,
                                  ObjectHandle other)
    public Object BitwiseOr(Object self, Object other)
    public Boolean ContainsMember(ObjectHandle obj, String name)
    public Boolean ContainsMember(Object obj, String name,
                                  Boolean ignoreCase)
    public Boolean ContainsMember(Object obj, String name)
    public T ConvertTo<T>(Object obj)
    public ObjectHandle ConvertTo<T>(ObjectHandle obj)
    public Object ConvertTo(Object obj, Type type)
    public ObjectHandle ConvertTo(ObjectHandle obj, Type type)
    public ObjectHandle CreateInstance(ObjectHandle obj,
                                       params ObjectHandle[] parameters)
    public ObjectHandle CreateInstance(ObjectHandle obj,
                                        params Object[] parameters)
    public Object CreateInstance(Object obj,
                                 params Object[] parameters)
    public Object Divide(Object self, Object other)
    public ObjectHandle Divide(ObjectHandle self,
                               ObjectHandle other)
    public Object DoOperation(ExpressionType operation,
                              Object target)
    public TResult DoOperation<TTarget, TResult>
       (ExpressionType operation, TTarget target)
    public Object DoOperation
       (ExpressionType operation, Object target, Object other)
    public TResult DoOperation<TTarget, TOther, TResult>
       (ExpressionType operation, TTarget target, TOther other)
    public Object DoOperation(ExpressionType op,
```

```
                          ObjectHandle target)
public ObjectHandle DoOperation
   (ExpressionType op, ObjectHandle target, ObjectHandle other)
public Boolean Equal(Object self, Object other)
public Boolean Equal(ObjectHandle self, ObjectHandle other)
public Object ExclusiveOr(Object self, Object other)
public ObjectHandle ExclusiveOr(ObjectHandle self,
                                ObjectHandle other)
public T ExplicitConvertTo<T>(Object obj)
public Object ExplicitConvertTo(Object obj, Type type)
public ObjectHandle ExplicitConvertTo(ObjectHandle obj, Type type)
public ObjectHandle ExplicitConvertTo<T>(ObjectHandle obj)
public T ImplicitConvertTo<T>(Object obj)
public Object ImplicitConvertTo(Object obj, Type type)
public ObjectHandle ImplicitConvertTo(ObjectHandle obj, Type type)
public ObjectHandle ImplicitConvertTo<T>(ObjectHandle obj)
public String Format(Object obj)
public String Format(ObjectHandle obj)
public IList<System.String> GetCallSignatures(ObjectHandle obj)
public IList<System.String> GetCallSignatures(Object obj)
public String GetDocumentation(Object obj)
public String GetDocumentation(ObjectHandle obj)
public T GetMember<T>(ObjectHandle obj, String name)
public T GetMember<T>(Object obj, String name, Boolean ignoreCase)
public Object GetMember(Object obj, String name)
public Object GetMember(Object obj, String name,
                        Boolean ignoreCase)
public ObjectHandle GetMember(ObjectHandle obj, String name)
public T GetMember<T>(Object obj, String name)
public IList<System.String> GetMemberNames(ObjectHandle obj)
public IList<System.String> GetMemberNames(Object obj)
public Boolean GreaterThan(Object self, Object other)
public Boolean GreaterThan(ObjectHandle self, ObjectHandle other)
public Boolean GreaterThanOrEqual(Object self, Object other)
public Boolean GreaterThanOrEqual(ObjectHandle self,
                                  ObjectHandle other)
public ObjectHandle Invoke(ObjectHandle obj,
                           params ObjectHandle[] parameters)
public ObjectHandle Invoke(ObjectHandle obj,
                           params Object[] parameters)
public Object Invoke(Object obj, params Object[] parameters)
public Object InvokeMember(Object obj, String memberName,
                           params Object[] parameters)
public Boolean IsCallable(Object obj)
public Boolean IsCallable(ObjectHandle obj)
public ObjectHandle LeftShift(ObjectHandle self,
                              ObjectHandle other)
public Object LeftShift(Object self, Object other)
public Boolean LessThan(Object self, Object other)
public Boolean LessThan(ObjectHandle self, ObjectHandle other)
public Boolean LessThanOrEqual(ObjectHandle self,
                               ObjectHandle other)
public Boolean LessThanOrEqual(Object self, Object other)
public ObjectHandle Modulo(ObjectHandle self, ObjectHandle other)
public Object Modulo(Object self, Object other)
public ObjectHandle Multiply(ObjectHandle self, ObjectHandle other)
public Object Multiply(Object self, Object other)
```

```
public Boolean NotEqual(Object self, Object other)
public Boolean NotEqual(ObjectHandle self, ObjectHandle other)
public Object Power(Object self, Object other)
public ObjectHandle Power(ObjectHandle self, ObjectHandle other)
public Boolean RemoveMember(Object obj, String name)
public Boolean RemoveMember(ObjectHandle obj, String name)
public Boolean RemoveMember(Object obj, String name,
                            Boolean ignoreCase)
public ObjectHandle RightShift(ObjectHandle self,
                               ObjectHandle other)
public Object RightShift(Object self, Object other)
public void SetMember(Object obj, String name, Object value,
                      Boolean ignoreCase)
public void SetMember(ObjectHandle obj, String name,
                      ObjectHandle value)
public void SetMember<T>(Object obj, String name, T value,
                         Boolean ignoreCase)
public void SetMember<T>(Object obj, String name, T value)
public void SetMember<T>(ObjectHandle obj, String name, T value)
public void SetMember(Object obj, String name, Object value)
public ObjectHandle Subtract(ObjectHandle self, ObjectHandle other)
public Object Subtract(Object self, Object other)
public Boolean TryConvertTo<T>(ObjectHandle obj,
                               out ObjectHandle result)
public Boolean TryConvertTo(Object obj, Type type,
                            out Object result)
public Boolean TryConvertTo<T>(Object obj, out T result)
public Boolean TryConvertTo(ObjectHandle obj, Type type,
                            out ObjectHandle result)
public Boolean TryExplicitConvertTo<T>(Object obj, out T result)
public Boolean TryExplicitConvertTo<T>(ObjectHandle obj,
                                       out ObjectHandle result)
public Boolean TryExplicitConvertTo(ObjectHandle obj, Type type,
                                    out ObjectHandle result)
public Boolean TryExplicitConvertTo(Object obj, Type type,
                                    out Object result)
public Boolean TryImplicitConvertTo<T>(Object obj, out T result)
public Boolean TryImplicitConvertTo<T>(ObjectHandle obj,
                                       out ObjectHandle result)
public Boolean TryImplicitConvertTo(ObjectHandle obj, Type type,
                                    out ObjectHandle result)
public Boolean TryImplicitConvertTo(Object obj, Type type,
                                    out Object result)
public Boolean TryGetMember(Object obj, String name,
                            Boolean ignoreCase, out Object value)
public Boolean TryGetMember(ObjectHandle obj, String name,
                            out ObjectHandle value)
public Boolean TryGetMember(Object obj, String name,
                            out Object value)
public T Unwrap<T>(ObjectHandle obj)
```

4.6.2 Engine Property

This property returns the engine bound to this ObjectOperations. The engine binding provides the language context or semantics applied to each requested operation.

Signature:

```
public ScriptEngine Engine { get; }
```

4.6.3 IsCallable Methods

These methods returns whether the object is callable. Languages should return delegates when fetching the value of variables or executing expressions that result in callable objects. However, sometimes you'll get objects that are callable, but they are not wrapped in a delegate. Note, even if this method returns true, a call may fail due to incorrect number of arguments or incorrect types of arguments.

Signatures:

```
public bool IsCallable(object obj)
public bool IsCallable(ObjectHandle obj)
```

4.6.4 Invoke Methods

These methods invoke objects that are callable. In general you should not need to call these methods. Languages should return delegates when fetching the value of variables or executing expressions that result in callable objects. However, sometimes you'll get objects that are callable, but they are not wrapped in a delegate. If you're calling an object multiple times, you can use ConvertTo to get a strongly typed delegate that you can call more efficiently. You'll also need to use Invoke for objects that are remote.

If any obj arguments are null, then these throw an ArgumentNullException.

Signatures:

```
public ObjectHandle Invoke(ObjectHandle obj,
                           params ObjectHandle[] parameters)
public ObjectHandle Invoke(ObjectHandle obj,
                           params Object[] parameters)
public Object Invoke(Object obj, params Object[] parameters)
```

4.6.5 InvokeMember Method

This method invokes callable members from objects.

If the obj argument is null, then this throws an ArgumentNullException.

Signatures:

```
public Object InvokeMember(Object obj, String memberName,
                           params Object[] parameters)
```

4.6.6 CreateInstance Methods

These methods create objects when the input object can be instantiated.

If any obj arguments are null, then these throw an ArgumentNullException.

Signatures:

```
public ObjectHandle CreateInstance(ObjectHandle obj,
                                   params ObjectHandle[] parameters)
```

```
public ObjectHandle CreateInstance(ObjectHandle obj,
                                   params Object[] parameters)
public Object CreateInstance(Object obj,
                             params Object[] parameters)
```

4.6.7 GetMember* Methods

These methods return a named member of an object.

The generic overloads do not modify obj to convert to the requested type. If they cannot perform the requested conversion to the concrete type, then they throw a NotSupportedException. You can use Unwrap<T> after ConvertTo<T> on ObjectHandle to get a local T for the result. The generic overloads use language-specific conversions (based on the default language in the Engine property), like ConvertTo<T>.

If the specified member does not exist, or if it is write-only, then these throw exceptions.

Signatures:

```
public T GetMember<T>(ObjectHandle obj, String name)
public T GetMember<T>(Object obj, String name, Boolean ignoreCase)
public Object GetMember(Object obj, String name)
public Object GetMember(Object obj, String name,
                        Boolean ignoreCase)
public ObjectHandle GetMember(ObjectHandle obj, String name)
public T GetMember<T>(Object obj, String name)
```

4.6.8 TryGetMember Methods

These methods try to get a named member of an object. They return whether name was a member of obj and set the out value to name's value. If the name was not a member of obj, then this method sets value to null.

If obj or name is null, then these throw an ArgumentNullException.

Signatures:

```
public Boolean TryGetMember(Object obj, String name,
                            Boolean ignoreCase, out Object value)
public Boolean TryGetMember(ObjectHandle obj, String name,
                            out ObjectHandle value)
public Boolean TryGetMember(Object obj, String name,
                            out Object value)
```

4.6.9 ContainsMember Methods

These methods return whether the name is a member of obj.

Signatures:

```
public Boolean ContainsMember(ObjectHandle obj, String name)
public Boolean ContainsMember(Object obj, String name,
                              Boolean ignoreCase)
public Boolean ContainsMember(Object obj, String name)
```

4.6.10 RemoveMember Methods

These methods remove name from obj so that it is no longer a member of obj. If the object or the language binding of this ObjectOperations allows read-only or non-removable members, and name identifies such a member, then it is undefined what happens. Languages vary on whether this is a no-op or exceptional.

If any arguments are null, then these throw an ArgumentNullException.

Signatures:

```
public Boolean RemoveMember(Object obj, String name)
public Boolean RemoveMember(ObjectHandle obj, String name)
public Boolean RemoveMember(Object obj, String name,
                            Boolean ignoreCase)
```

4.6.11 SetMember Methods

These members set the value of a named member of an object. There are generic overloads that can be used to avoid boxing values and casting of strongly typed members.

If the object or the language binding of this ObjectOperations supports read-only members, and name identifies such a member, then these methods throw a NotSupportedException.

If any arguments are null, then these throw an ArgumentNullException.

Signatures:

```
public void SetMember(Object obj, String name, Object value,
                      Boolean ignoreCase)
public void SetMember(ObjectHandle obj, String name,
                      ObjectHandle value)
public void SetMember<T>(Object obj, String name, T value,
                         Boolean ignoreCase)
public void SetMember<T>(Object obj, String name, T value)
public void SetMember<T>(ObjectHandle obj, String name, T value)
public void SetMember(Object obj, String name, Object value)
```

4.6.12 ConvertTo* Methods

These methods convert an object to the requested type using language-specific (based on the default language in the Engine property) conversions. These may be implicit only, or include explicit conversion too. The conversions do not modify obj. Obj may be returned if it is already the requested type. You can use Unwrap<T> after ConvertTo<T> on ObjectHandle to get a local T for the result.

If any of the arguments is null, then these throw an ArgumentNullException.

If these methods cannot perform the requested conversion, then they throw a NotSupportedException.

Signatures:

```
public T ConvertTo<T>(Object obj)
public ObjectHandle ConvertTo<T>(ObjectHandle obj)
public Object ConvertTo(Object obj, Type type)
public ObjectHandle ConvertTo(ObjectHandle obj, Type type)
```

4.6.13 TryConvertTo* Methods

These methods try to convert an object to the requested type language-specific (based on the default language in the Engine property) conversions. These may be implicit only, or include explicit conversion too. The conversions do not modify obj. They return whether they could perform the conversion and set the out result parameter. If the methods could not perform the conversion, then they set result to null.

You can use Unwrap<T> after calling overloads on ObjectHandle to get a local T for the result.

If they cannot perform the conversion to the requested type, then they throw a NotSupportedException.

If obj is null, then these throw an ArgumentNullException.

Signatures:

```
public Boolean TryConvertTo<T>(ObjectHandle obj,
                               out ObjectHandle result)
public Boolean TryConvertTo(Object obj, Type type,
                            out Object result)
public Boolean TryConvertTo<T>(Object obj, out T result)
public Boolean TryConvertTo(ObjectHandle obj, Type type,
                            out ObjectHandle result)
```

4.6.14 ExplicitConvertTo* Methods

These methods convert an object to the requested type using explicit conversions, which may be lossy. Otherwise these methods are the same as the ConvertTo* methods.

```
public T ExplicitConvertTo<T>(Object obj)
public Object ExplicitConvertTo(Object obj, Type type)
public ObjectHandle ExplicitConvertTo(ObjectHandle obj, Type type)
public ObjectHandle ExplicitConvertTo<T>(ObjectHandle obj)
```

4.6.15 TryExplicitConvertTo* Methods

These methods try to convert an object to the request type using explicit conversions, which may be lossy. Otherwise these methods are the same as TryConvertTo* methods.

```
public Boolean TryExplicitConvertTo<T>(Object obj, out T result)
public Boolean TryExplicitConvertTo<T>(ObjectHandle obj,
                                       out ObjectHandle result)
public Boolean TryExplicitConvertTo(ObjectHandle obj, Type type,
                                    out ObjectHandle result)
public Boolean TryExplicitConvertTo(Object obj, Type type,
                                    out Object result)
```

4.6.16 ImplicitConvertTo* Methods

These methods convert an object to the requested type using implicit conversions, which may be lossy. Otherwise these methods are the same as the ConvertTo* methods.

```
public T ImplicitConvertTo<T>(Object obj)
public Object ImplicitConvertTo(Object obj, Type type)
public ObjectHandle ImplicitConvertTo(ObjectHandle obj, Type type)
public ObjectHandle ImplicitConvertTo<T>(ObjectHandle obj)
```

4.6.17 TryimplicitConvertTo* Methods

These methods try to convert an object to the request type using implicit conversions, which may be lossy. Otherwise these methods are the same as TryConvertTo* methods.

```
public Boolean TryImplicitConvertTo<T>(Object obj, out T result)
public Boolean TryImplicitConvertTo<T>(ObjectHandle obj,
                                       out ObjectHandle result)
public Boolean TryImplicitConvertTo(ObjectHandle obj, Type type,
                                    out ObjectHandle result)
public Boolean TryImplicitConvertTo(Object obj, Type type,
                                    out Object result)
```

4.6.18 Unwrap<T> Method

This method unwraps the remote object reference, converting it to the specified type before returning it. If this method cannot perform the requested conversion to the concrete type, then it throws a NotSupportedException. If the requested T does not serialize back to the calling app domain, the CLR throws an exception.

Signature:

```
public T Unwrap<T>(ObjectHandle obj)
```

4.6.19 Format Methods

These methods return a string representation of obj that is parse-able by the language. ConvertTo operations that request a string return a display string for the object that is not necessarily parse-able as input for evaluation.

Signatures:

```
public string Format(object obj)
public string Format(ObjectHandle obj)
```

4.6.20 GetMemberNames Methods

These methods return an array of all the member names that obj has explicitly, determined by the language associated with this ObjectOperations. Computed or late bound member names may not be in the result.

Signatures:

```
public IList<string> GetMemberNames(object obj)
public IList<string> GetMemberNames(ObjectHandle obj)
```

4.6.21 GetDocumentation Methods

These methods return the documentation for obj. When obj is a static .NET object, this returns xml documentation comment information associated with the DLL containing obj's type. If there is no available documentation for the object, these return the empty string. Some languages do not have documentation hooks for objects, in which case they return the empty string.

Signatures:

```
public string GetDocumentation(object obj)
```

```
public string GetDocumentation(ObjectHandle obj)
```

4.6.22 GetCallSignatures Methods

These methods return arrays of stings, each one describing a call signature that obj supports. If the object is not callable, these throw a NotSupportedException.

Signatures:

```
public IList<string> GetCallSignatures(object obj)
public IList<string> GetCallSignatures(ObjectHandle obj)
```

4.6.23 DoOperation* Methods

These methods perform the specified unary and binary operations on the supplied target and other objects, returning the results. If the specified operator cannot be performed on the object or objects supplied, then these throw an exception. See the Expression Tree spec for information on the expected semantics of the operators.

The Hosting APIs share the ExpressionType enum with Expression Trees and the dynamic object interop protocol to specify what operation to perform. Most values overlap making a distinct enum just another concept to learn, but this enum contains values for operations used in Expression Trees that do not make sense when passed to this method (for example, Block, Try, and Throw). These methods pass the operation to the language that created the ObjectOperations object, and the language handles the ExpressionType as it sees fit. For example, IronPython only supports the following ExpressionType values:

Add	Subtract	SubtractAssign
And	AddAssign	Equal
Divide	AndAssign	GreaterThan
ExclusiveOr	DivideAssign	GreaterThanOrEqual
Modulo	ExclusiveOrAssign	LessThan
Multiply	MultiplyAssign	LessThanOrEqual
Or	OrAssign	NotEqual
Power	PowerAssign	
RightShift	RightShfitAssign	
LeftShift	LeftShiftAssign	

Signatures:

```
public Object DoOperation(ExpressionType operation,
                          Object target)
public TResult DoOperation<TTarget, TResult>
   (ExpressionType operation, TTarget target)
public Object DoOperation
   (ExpressionType operation, Object target, Object other)
public TResult DoOperation<TTarget, TOther, TResult>
   (ExpressionType operation, TTarget target, TOther other)
public Object DoOperation(ExpressionType op,
                          ObjectHandle target)
public ObjectHandle DoOperation
   (ExpressionType op, ObjectHandle target, ObjectHandle other)
```

4.6.24 Add Methods

These methods are convenience members that are equivalent to:

```
DoOperation(ExpressionType.Add, self, other)
```

Signatures:

```
public object Add(object self, object other)
public ObjectHandle Add(ObjectHandle self, ObjectHandle other)
```

4.6.25 Subtract Methods

These methods are convenience members that are equivalent to:

```
DoOperation(ExpressionType.Subtract, self, other)
```

Signatures:

```
public object Subtract(object self, object other)
public ObjectHandle Subtract(ObjectHandle self, ObjectHandle other)
```

4.6.26 Power Methods

These methods are convenience members that are equivalent to:

```
DoOperation(ExpressionType.Power, self, other)
```

Signatures:

```
public object Power(object self, object other)
public ObjectHandle Power(ObjectHandle self, ObjectHandle other)
```

4.6.27 Multiply Methods

These methods are convenience members that are equivalent to:

```
DoOperation(ExpressionType.Multiply, self, other)
```

Signatures:

```
public object Multiply(object self, object other)
public ObjectHandle Multiply(ObjectHandle self, ObjectHandle other)
```

4.6.28 Divide Methods

These methods are convenience members that are equivalent to:

```
DoOperation(ExpressionType.Divide, self, other)
```

Signatures:

```
public object Divide(object self, object other)
public ObjectHandle Divide(ObjectHandle self, ObjectHandle other)
```

4.6.29 Modulo Methods

These methods are convenience members that are equivalent to:

```
DoOperation(ExpressionType.Modulo, self, other)
```

Signatures:

```
public ObjectHandle Modulo(ObjectHandle self, ObjectHandle other)
public Object Modulo(Object self, Object other)
```

4.6.30 LeftShift Methods

These methods are convenience members that are equivalent to:

```
DoOperation(ExpressionType.LeftShift, self, other)
```

Signatures:

```
public object LeftShift(object self, object other)
public ObjectHandle LeftShift(ObjectHandle self, ObjectHandle
other)
```

4.6.31 RightShift Methods

These methods are convenience members that are equivalent to:

```
DoOperation(ExpressionType.RightShift, self, other)
```

Signatures:

```
public object RightShift(object self, object other) {
public ObjectHandle RightShift(ObjectHandle self,
                               ObjectHandle other)
```

4.6.32 BitwiseAnd Methods

These methods are convenience members that are equivalent to:

```
DoOperation(ExpressionType.BitwiseAnd, self, other)
```

Signatures:

```
public object BitwiseAnd(object self, object other) {
public ObjectHandle BitwiseAnd(ObjectHandle self,
                               ObjectHandle other)
```

4.6.33 BitwiseOr Methods

These methods are convenience members that are equivalent to:

```
DoOperation(ExpressionType.BitwiseOr, self, other)
```

Signatures:

```
public object BitwiseOr(object self, object other)
public ObjectHandle BitwiseOr(ObjectHandle self,
                              ObjectHandle other)
```

4.6.34 ExclusiveOr Methods

These methods are convenience members that are equivalent to:

```
DoOperation(ExpressionType.ExclusiveOr, self, other)
```

Signatures:

```
public object ExclusiveOr(object self, object other)
public ObjectHandle ExclusiveOr(ObjectHandle self,
                                ObjectHandle other)
```

4.6.35 Equal Methods

These methods are convenience members that are equivalent to:

```
DoOperation(ExpressionType.Equal, self, other)
```

Signatures:

```
public bool Equal(object self, object other)
public bool Equal(ObjectHandle self, ObjectHandle other)
```

4.6.36 NotEqual Methods

These methods are convenience members that are equivalent to:

```
DoOperation(ExpressionType.Equal, self, other)
```

Signatures:

```
public Boolean NotEqual(Object self, Object other)
public Boolean NotEqual(ObjectHandle self, ObjectHandle other)
```

4.6.37 LessThan Methods

These methods are convenience members that are equivalent to:

```
DoOperation(ExpressionType.LessThan, self, other)
```

Signatures:

```
public bool LessThan(object self, object other)
public bool LessThan(ObjectHandle self, ObjectHandle other)
```

4.6.38 LessThanOrEqual Methods

These methods are convenience members that are equivalent to:

```
DoOperation(ExpressionType.LessThanOrEqual, self, other)
```

Signatures:

```
public Boolean LessThanOrEqual(ObjectHandle self,
                               ObjectHandle other)
public Boolean LessThanOrEqual(Object self, Object other)
```

4.6.39 GreaterThan Methods

These methods are convenience members that are equivalent to:

```
DoOperation(ExpressionType.GreaterThan, self, other)
```

Signatures:

```
public bool GreaterThan(object self, object other)
public bool GreaterThan(ObjectHandle self, ObjectHandle other)
```

4.6.40 GreaterThanOrEqual Methods

These methods are convenience members that are equivalent to:

```
DoOperation(ExpressionType.GreaterThanOrEqual, self, other)
```

Signatures:

```
public bool GreaterThanOrEqual(object self, object other)
public bool GreaterThanOrEqual(ObjectHandle self,
                               ObjectHandle other)
```

4.7 SourceCodeKind Enum

This enum identifies parsing hints to languages for ScriptSource objects. For example, some languages need to know if they are parsing a Statement or an Expression, or they may allow special syntax or variables for InteractiveCode.

4.7.1 Type Summary

```
public enum SourceCodeKind {
    Unspecified,
    Expression,
    Statements,
    SingleStatement,
    File,
    InteractiveCode,
    AutoDetect
```

4.7.2 Members

The Summary section shows the type as it is defined (to indicate values of members), and this section documents the intent of the members.

`Unspecified`	Should not be used.
`Expression`	Start parsing an expression.
`Statements`	Start parsing one or more statements if there's special syntax for multiple statements.
`SingleStatement`	Start parsing a single statement, guaranteeing there's only one if that is significant to the language.
`File`	Start parsing at the beginning of a file.
`InteractiveCode`	Start parsing at a legal input to a REPL. This kind also means the language should wrap the

	source to do language-specific output of the evaluation result.
`AutoDetect`	The language best determines how to parse the input. It may choose Interactive (supporting special syntax or variables), Expression, Statement, or File. This is the most liberal choice for the language to do whatever it determines is best. The goal of this kind is to help uses hosting with embedded code that has extraneous whitespace, several statements or expressions, etc. The language determines how to evaluate and what to return as the result in addition to shaping the extra whitespace to make sense if whitespace is significant in the language. The language may ignore initial lines with only whitespace if that would confuse the parsing or evaluation.

4.8 ScriptCodeParseResult Enum

This enum identifies final parsing state for a ScriptSource objects. It helps with interactive tool support.

May need to rename to ScriptCodeParseResult since .NET naming conventions only like plural enum names for those with flag values. None should probably be Success or something more descriptive?

4.8.1 Type Summary

```
public enum ScriptCodeParseResult {
    Complete,
    Invalid,
    IncompleteToken,
    IncompleteStatement,
    Empty
```

4.8.2 Members

The Summary section shows the type as it is defined (to indicate values of members), and this section documents the intent of the members.

`Complete`	There is no reportable state after parsing.
`Invalid`	The source is syntactically invalid and cannot be parsed.
`IncompleteToken`	The source ended on an incomplete token that aborted parsing.

`IncompleteStatement`	The source ended on an incomplete statement that aborted parsing.
`Empty`	The source is either empty, all whitespace, or all comments.

4.9 TextContentProvider Abstract Class

This class provides a means for hosts to provide TextReaders for a source that is already encoded as a Unicode string. A host only needs to implement one of these if it needs to provide a reader over source that is in its data structures (that is, not from a file or string). For example, an editor host could support reading from its buffer or text representation.

4.9.1 Class Summary

```
[Serializable]
public abstract class TextContentProvider {
    public abstract TextReader GetReader()
```

4.9.2 GetReader Method

This method returns a new TextReader each time you call it. The reader is positioned at the start of the input. It is undefined whether fetching the stream fetches fresh source contents.

Signature:

```
    public abstract TextReader GetReader()
```

4.10 StreamContentProvider Abstract Class

This class provides a mean for hosts to provide multiple Streams for a source of content that is binary data (has no encoding). Languages have an opportunity to decode the binary data directly if their language defines a way to supply encoding information within the source code.

4.10.1 Class Summary

```
[Serializable]
public abstract class StreamContentProvider {
    public abstract Stream GetStream()
```

4.10.2 GetStream Method

This method returns a new Stream each time you call it. The Stream is positioned at the start of the input. It is undefined whether fetching the stream fetches fresh source contents.

Signature:

```
    public abstract Stream GetStream()
```

4.11 ScriptCodeReader Sealed Class

This class simply holds a ScriptSource and the TextReader associated with the ScriptSource. You get to these objects with ScriptSource.GetReader, which returns a new reader each time positioned at the start of the source.

Other than a property that returns the ScriptSource, this class just has overrides for TextReader and a virtual SeekLine. These changes allow the reader to interact with the language engine associated with the source for any special newline handling the language might perform.

4.11.1 Class Summary

```
public sealed class ScriptCodeReader : TextReader {
    public ScriptSource ScriptSource
    internal ScriptCodeReader(SourceUnit sourceUnit,
                              TextReader textReader)
    public override string ReadLine()
    public virtual bool SeekLine(int line)
    public override string ReadToEnd()
    public override int Read(char[] buffer, int index, int count)
    public override int Peek()
    public override int Read()
```

4.12 ScriptIO Class

This class let's you control input and output by default for dynamic code running via DLR hosting. You can access the instance of this class from the IO property on ScriptRuntime.

4.12.1 Class Summary

```
public sealed class ScriptIO : MarshalByRefObject
    internal ScriptIO(...)

    /// Used for binary IO.
    public Stream InputStream { get; }
    public Stream OutputStream { get; }
    public Stream ErrorStream { get; }
    /// Used for pure unicode IO.
    public TextReader InputReader { get; }
    public TextWriter OutputWriter { get; }
    public TextWriter ErrorWriter { get; }
    /// What encoding are the unicode reader/writers using.
    public Encoding InputEncoding { get; }
    public Encoding OutputEncoding { get;  }
    public Encoding ErrorEncoding { get; }

    public void SetOutput(Stream stream, Encoding encoding)
    public void SetOutput(Stream stream, TextWriter writer)

    public void SetErrorOutput(Stream stream, Encoding encoding)
    public void SetErrorOutput(Stream stream, TextWriter writer)

    public void SetInput(Stream stream, Encoding encoding)
    public void SetInput(Stream stream, TextReader reader,
```

```
                                Encoding encoding)
    public void RedirectToConsole()
```

4.12.2 OutputStream Property

This property returns the standard output stream for the ScriptRuntime. This is a binary stream. All code and engines should output binary data here for this ScriptRuntime. Of course, if a language has a mechanism to programmatically direct output to a file or stream, then that language's output would go there as directed by the code.

Signature:

```
    public Stream OutputStream { get;}
```

4.12.3 InputStream Property

This property returns the standard input stream for the ScriptRuntime. This is a binary stream. All code and engines should read binary data from here for this ScriptRuntime. Of course, if a language has a mechanism to programmatically direct input from a file or stream, then that language's input would come from there as directed by the code.

Signature:

```
    public Stream InputStream { get;}
```

4.12.4 ErrorStream Property

This property returns the standard erroroutput stream for the ScriptRuntime. This is a binary stream. All code and engines should send error binary output here for this ScriptRuntime. Of course, if a language has a mechanism to programmatically direct error output to a file or stream, then that language's error output would go there as directed by the code.

Signature:

```
    public Stream ErrorStream { get;}
```

4.12.5 InputReader Property

This property returns the standard input reader for the ScriptRuntime. This is a unicode reader. All code and engines should read text from here for this ScriptRuntime. Of course, if a language has a mechanism to programmatically direct input from a file or stream, then that language's input would come from there as directed by the code.

Signature:

```
    public TextReader InputReader { get; }
```

4.12.6 OutputWriter Property

This property returns the standard output writer for the ScriptRuntime. All code and engines should send text output here for this ScriptRuntime. Of course, if a language has a mechanism

to programmatically direct output to a file or stream, then that language's output would go there as directed by the code.

Signature:

```
public TextWriter OutputWriter { get; }
```

4.12.7 ErrorWriter Property

This property returns the standard error output writer for the ScriptRuntime. All code and engines should send text error output here for this ScriptRuntime. Of course, if a language has a mechanism to programmatically direct error output to a file or stream, then that language's output would go there as directed by the code.

Signature:

```
public TextWriter ErrorWriter { get; }
```

4.12.8 InputEncoding Property

This property returns the encoding used by the TextReader returned from InputReader.

Signature:

```
public Encoding InputEncoding { get; }
```

4.12.9 OutputEncoding Property

This property returns the encoding used by the TextWriters returned from the OutputWriter property.

Signature:

```
public Encoding OutputEncoding { get; }
```

4.12.10 ErrorEncoding Property

This property returns the encoding used by the TextWriters returned from the ErrorWriter property.

Signature:

```
public Encoding ErrorEncoding { get; }
```

4.12.11 SetOutput Method

This method sets the standard output stream for the ScriptRuntime. All code and engines should send output to the specified stream for this ScriptRuntime. Of course, if a language has a mechanism to programmatically direct output to a file or stream, then that language's output would go there as directed by the code.

Signatures:

```
public void SetOutput(Stream stream, Encoding encoding)
public void SetOutput(Stream stream, TextWriter writer)
```

The first method is useful if the host just captures binary stream output. The second method is useful if the host captures unicode text and binary output. Note, if you pass just a stream and an encoding, the this method creates a default StreamWriter, which writes a BOM on first usage. To avoid this, you'll need to pass your own TextWriter.

If any argument to these methods is null, they throw an ArgumentException.

4.12.12 SetErrorOutput Method

This method sets the standard error output stream for the ScriptRuntime. All code and engines should send error output to the specified stream for this ScriptRuntime. Of course, if a language has a mechanism to programmatically direct error output to a file or stream, then that language's output would go there as directed by the code.

Signatures:

```
public void SetErrorOutput(Stream stream, Encoding encoding)
public void SetErrorOutput(Stream stream, TextWriter writer)
```

The first method is useful if the host just captures binary stream output. The second method is useful if the host captures unicode text and binary output.

If any argument to these methods is null, they throw an ArgumentException.

4.12.13 SetInput Method

This method sets the standard input stream for the ScriptRuntime. All code and engines should read input here for this ScriptRuntime. Of course, if a language has a mechanism to programmatically direct input from a file or stream, then that language's input would come from there as directed by the code.

Signature:

```
public void SetInput(Stream stream, Encoding encoding)
public void SetInput(Stream stream, TextReader reader,
                     Encoding encoding)
```

4.12.14 RedirectToConsole Method

This method makes all the standard IO for the ScriptRuntime go to System.Console. Of course, if a language has a mechanism to programmatically direct output to a file or stream, then that language's output would go there as directed by the code.

Signature:

```
public void RedirectToConsole()
```

4.13 ScriptRuntimeSetup Class

This class gives hosts full control over how a ScriptRuntime gets configured. You can instantiate this class, fill in the setup information, and then instantiate a ScriptRuntime with the setup instance. Once you pass the setup object to create a ScriptRuntime, attempts to modify its contents throws an exception.

There is also a static method as a helper to hosts for reading .NET application configuration. Hosts that want to be able to use multiple DLR-hostable languages, allow users to change what languages are available, and not have to rebuild can use the DLR's default application configuration model. See ReadConfiguration for the XML details.

You can also get these objects from ScriptRuntime.Setup. These instances provide access to the configuration information used to create the ScriptRuntime. These instances will be read-only and throws exceptions if you attempt to modify them. Hosts may not have created a ScriptRuntimeSetup object and may not have configuration information without the Setup property.

4.13.1 Class Summary

```
public sealed class ScriptRuntimeSetup {
    public ScriptRuntimeSetup()
    public IList<LanguageSetup> LanguageSetups { get;  }
    public bool DebugMode { get; set; }
    public bool PrivateBinding { get; set; }
    public Type HostType { get; set; }
    public Dictionary<string, object> Options { get; }
    public object[] HostArguments {get; set;  }
    public static ScriptRuntimeSetup ReadConfiguration()
    public static ScriptRuntimeSetup
                 ReadConfiguration(Stream configFileStream)
```

4.13.2 Constructor

The constructor returns an empty ScriptRuntimeSetup object, with no languages preconfigured.

Signature:

```
    public ScriptRuntimeSetup()
```

4.13.3 ReadConfiguration Methods

These methods read application configuration and return a ScriptRuntimeSetup initialized from the application configuration data. Hosts can modify the result before using the ScriptRuntimeSetup object to instantiate a ScriptRuntime.

Signatures:

```
    public static ScriptRuntimeSetup ReadConfiguration()
    public static ScriptRuntimeSetup
                 ReadConfiguration(Stream configFileStream)
```

4.13.3.1 Configuration Structure

These lines must be included in the .config file as the first element under the <configuration> element for the DLR's default reader to work:

```
<configSections>
    <section name="microsoft.scripting"
       type="Microsoft.Scripting.Hosting.Configuration.Section,
Microsoft.Scripting, Version=1.0.0.5000, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
</configSections>
```

The structure of the configuration section is the following (with some notes below):

```
<microsoft.scripting [debugMode="{bool}"]?
                     [privateBinding="{bool}"]?>
    <languages>
      <!-- BasicMap with type attribute as key.  Inherits
language
           nodes, overwrites previous nodes based on key -->
      <language names="{semicolon-separated}"
                extensions="{semicolon-separated, optional-dot}"
                type="{assembly-qualified type name}"
                [displayName="{string}"]? />
    </languages>

    <options>
      <!-- AddRemoveClearMap with option as key.  If language
           Attribute is present, the key is option cross
language.
       -->
      <set option="{string}" value="{string}"
           [language="{langauge-name}"]? />
      <clear />
      <remove option="{string}" [language="{langauge-name}"]? />
    </options>

  </microsoft.scripting>
```

Attributes enclosed in […]? are optional.

{bool} is whatever Convert.ToBoolean(string) works for (“true”, “False”, “TRUE”, “1”, “0”).

<languages> tag inherits content from parent .config files. You cannot remove a language in a child .config file once it is defined in a parent .config file. You can redefine a language if the value of the “type” attribute is the same as a defined in a parent .config file (last writer wins). If the displayName attribute is missing, ReadConfiguration sets it to the first name in the names attribute. If names is the empty string, then ReadConfiguration sets the display name to the type attribute. The names and extensions attributes support semi-colon and comma as separators.

<options> tag inherits options from parent .config files. You can set, remove, and clear options (removes them all). The key in the options dictionary is a pair of option and language attributes. Language attribute is optional. If specified, the option applies to the language whose simple name is stated; otherwise, it applies to all languages. <remove option=”foo”/> removes the option from common options dictionary, not from all language dictionaries. <remove option=”foo” language=”rb”/> removes the option from Ruby language options.

4.13.3.2 Default DLR Configuration

The default application configuration section for using the DLR languages we ship for the desktop is (of course, you need correct type names from your current version):

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="microsoft.scripting"
```

```
       type="Microsoft.Scripting.Hosting.Configuration.Section,
Microsoft.Scripting, Version=1.0.0.5000, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
  </configSections>

  <microsoft.scripting>
    <languages>
      <language names="IronPython;Python;py" extensions=".py"
                displayName="IronPython v2.0"
           type="IronPython.Runtime.PythonContext, IronPython,
Version=2.0.0.5000, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
      <language names="IronRuby;Ruby;rb" extensions=".rb"
                displayName="IronRuby v1.0"
                type="IronRuby.Runtime.RubyContext, IronRuby,
Version=1.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35"
       />

      <!-- If for experimentation you want ToyScript ... -->
      <language names="ToyScript;ts" extensions=".ts"
                type="ToyScript.ToyLanguageContext, ToyScript,
Version=1.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35"
       />
    </languages>
  </microsoft.scripting>
</configuration>
```

4.13.4 LanguageSetups Property

This property returns a list of LanguageSetup objects, each describing one language the ScriptRuntime will allow. When you instantiate the ScriptRuntime, it will ensure there is only one element in the list with a given LanguageSetup.TypeName value.

Signature:

```
public IList<LanguageSetup> LanguageSetups { get;  }
```

4.13.5 HostType Property

This property gets and sets the ScriptHost type that the DLR should instantiate when it creates the ScriptRuntime. The DLR instantiates the host type in the app domain where it creates the ScriptRuntime object. See ScriptHost for more information.

```
public Type HostType { get; set; }
```

4.13.6 HostArguments Property

This property gets and sets an array of argument values that should be passed to the HostType's constructor. The objects must be MBRO or serializable when creating a remote ScriptRuntime.

Here's an example:

```
class MyHost : ScriptHost {
   public MyHost(string foo, int bar)
}
```

```
setup = new ScriptRuntimeSetup()
setup.HostType = typeof(MyHost)
setup.HostArguments = new object[] { "some foo", 123 }
ScriptRuntime.CreateRemote(otherAppDomain, setup)
```

Signature:

```
public object[] HostArguments {get; set;  }
```

4.13.7 Options Property

This property returns a dictionary of global options for the ScriptRuntime. There are two options explicit on the ScriptRuntimeSetup type, DebugMode and PrivateBinding. The Options property is an flexibility point for adding options later. Names are case-sensitive.

There is one specially named global option, "SearchPaths". If this value is present, languages should add these paths to their default search paths. If your intent is to replace an engine's default paths, then you can use Engine.SetSearchPaths (perhaps on the ScriptHost.EngineCreated callback).

Signature:

```
public Dictionary<string, object> Options { get; }
```

4.13.8 DebugMode Property

This property controls whether the ScriptRuntime instance and engines compiles code for debuggability.

Signature.

```
public bool DebugMode { get; set; }
```

4.13.9 PrivateBinding Property

This property controls whether the ScriptRuntime instance and engines will use reflection to access private members of types when binding object members in dynamic operations. Setting this to true only works in app domains running in full trust.

```
public bool PrivateBinding { get; set; }
```

4.14 LanguageSetup Class

This class represents a language configuration for use in a ScriptRuntimeSetup when instantiating a ScriptRuntime. Once you pass the setup object to create a ScriptRuntime, attempts to modify its contents throws an exception.

You can also get these objects from ScriptRuntime.Setup and ScriptEngine.Setup. These instances provide access to the configuration information used to create the ScriptRuntime. These instances will be read-only and throws exceptions if you attempt to modify them. Hosts may not have created a ScriptRuntimeSetup object and may not have language setup information without the Setup properties.

4.14.1 Class Summary

```
public sealed class LanguageSetup {
    public LanguageSetup(string typeName, string displayName)
    public LanguageSetup(string typeName, string displayName,
                         IEnumerable<string> names,
                         IEnumerable<string> fileExtensions)
    public string TypeName {get; set; }
    public string DisplayName {get; set; }
    public IList<string> Names {get; }
    public IList<string> FileExtensions {get; }
    public Dictionary<string, object> Options {get; }

    public bool InterpretedMode {get; set; }
    public bool ExceptionDetail {get; set; }
    public bool PerfStats {get; set; }
    public T GetOption<T>(string name, T defaultValue)
```

4.14.2 Constructors

The minimal construction requires an assembly-qualified type name for the language and a display name. You can set other properties after instantiating the setup object.

These ensure typeName and displayName are not null or empty. The collections can be empty but not null so that you can fill them in after instantiating this type.

Signatures:

```
public LanguageSetup(string typeName, string displayName)
public LanguageSetup(string typeName, string displayName,
                     IEnumerable<string> names,
                     IEnumerable<string> fileExtensions)
```

4.14.3 TypeName Property

This property gets or sets the assembly-qualified type name of the language. This is the type the DLR loads when, for example, it needs to execute files with the specified file extensions.

Signature:

```
public string TypeName {get; set; }
```

4.14.4 DisplayName Property

This property gets or sets a suitably descriptive name for displaying in UI or for debugging. It often includes the version number in case different versions of the same language are configured.

Signature:

```
public string DisplayName {get; set; }
```

4.14.5 Names Property

This property returns a list of names for the language. These can be nicknames or simple names used programmatically (for example, language=python on a web page or in a user's options UI).

Signature:

```
public IList<string> Names {get; }
```

4.14.6 FileExtensions Property

This property gets the list of file extensions that map to this language in the ScriptRuntime.

Signature:

```
public IList<string> FileExtensions {get; }
```

4.14.7 InterpretedMode Property

This property gets or sets whether the language engine interprets sources or compiles and executes them. Not all languages respond to this option.

This method pulls the value from Options in case it is set there via application .config instead of via the property setter. It defaults to false. If the host or reading .config set this option, then it will be in Options with the key "InterpretedMode".

Signature:

```
public bool InterpretedMode {get; set; }
```

4.14.8 ExceptionDetail Property

This property gets or sets whether the language engine should print exception details (for example, a call stack) when it catches exceptions. Not all languages respond to this option.

This method pulls the value from Options in case it is set there via application .config instead of via the property setter. It defaults to false. If the host or reading .config set this option, then it will be in Options with the key "ExceptionDetail".

Signature:

```
public bool ExceptionDetail {get; set; }
```

4.14.9 PerfStats Property

This property gets or sets whether the language engine gathers performance statistics. Not all languages respond to this option. Typically the languages dump the information when the application shuts down.

This method pulls the value from Options in case it is set there via application .config instead of via the property setter. It defaults to false. If the host or reading .config set this option, then it will be in Options with the key "ExceptionDetail".

Signature:

```
public bool PerfStats {get; set; }
```

4.14.10 Options Property

This property returns the list dictionary of options for the language. Option names are case-sensitive. The list of valid options for a given language must be found in its documentation.

Signature:

```
public Dictionary<string, object> Options {get; }
```

4.14.11 GetOption Method

This method looks up name in the Options dictionary and returns the value associated with name, converting it to type T. If the name is not present, this method return defaultValue.

Signature:

```
public T GetOption<T>(string name, T defaultValue)
```

4.15 ScriptHost Class

ScriptHost represents the host to the ScriptRuntime. Hosts can derive from this type and overload behaviors by returning a custom PlatformAdaptationLayer. Hosts can also handle callbacks for some events such as when engines get created.

The ScriptHost object lives in the same app domain as the ScriptRuntime in remote scenarios.

Derived types from ScriptHost can have arguments passed to them via ScriptRuntimeSetup's HostArguments property. For example,

```
class MyHost : ScriptHost {
   public MyHost(string foo, int bar)
}
setup = new ScriptRuntimeSetup()
setup.HostType = typeof(MyHost)
setup.HostArguments = new object[] { "some foo", 123 }
ScriptRuntime.CreateRemote(otherAppDomain, setup)
```

The DLR instantiates the ScriptHost when the DLR initializes a ScriptRuntime. The host can get at the instance with ScriptRuntime.Host.

4.15.1 Class Summary

```
public class ScriptHost : MarshalByRefObject {
    public ScriptHost()
    public ScriptRuntime Runtime { get; }
    public virtual PlatformAdaptationLayer
                   PlatformAdaptationLayer {get; }
    protected virtual void RuntimeAttached()
    internal protected virtual void
        EngineCreated(ScriptEngine engine)
```

4.15.2 Runtime Property

This property returns the ScriptRuntime to which this ScriptHost is attached.

Signature:

```
public ScriptRuntime Runtime { get; }
```

4.15.3 PlatformAdaptationLayer Property

This property returns the PlatformAdaptationLayer associated with the ScriptRuntime. This object adapts the runtime to the system by implementing various file operations, for example. The Silverlight DLR host and PAL might go to the server for some operations or throw an exception for others, depending on the behavior of the operation.

Signature:

```
public virtual PlatformAdaptationLayer
               PlatformAdaptationLayer {get; }
```

4.15.4 RuntimeAttached Method

This method gets called when initializing a ScriptRuntime is finished. The host can override this method to do additional initialization such as calling ScriptRuntime.LoadAssembly.

Signature:

```
protected virtual void RuntimeAttached()
```

4.15.5 EngineCreated Method

This method is a call back from the ScriptRuntime whenever it causes a language engine to be loaded and initialized. Hosts can derive from ScriptHost to override this method, which by default does nothing. An example usage would be for a host to load some standard scripts per language or to load per language init files for end users.

Signature:

```
internal protected virtual void
       EngineCreated(ScriptEngine engine)
```

~~4.16 ScriptRuntimeConfig Class~~

~~This class provides access to ScriptRuntime configuration information provided when it was constructed. The host may not have created a ScriptRuntimeSetup object and may not have this information available otherwise. This object does not report on all options the runtime or language may have; it reports only those supplied via ScriptRuntimeSetup (or the app .config file).~~

~~See ScriptRuntimeSetup for more info on the properties. This type is different only in that it is read-only.~~

~~4.16.1 Class Summary~~

```
public sealed class ScriptRuntimeConfig {
    public IList<LanguageConfig> Languages { get { } }
    public bool DebugMode { get { } }
    public bool PrivateBinding { get { } }
    public IDictionary<string, object> Options { get { } }
```

~~4.17 LanguageConfig Class~~

~~This class provides access to language configuration information provided when creating the ScriptRuntime. The host may not have created a ScriptRuntimeSetup object and may not have this information available otherwise. This object does not report on all options the language may have; it reports only those supplied via ScriptRuntimeSetup (or the app .config file).~~

~~See LanguageSetup for more info on the properties. This type is different only in that it is read-only.~~

~~4.17.1 Class Summary~~

```
public sealed class LanguageConfig {
    public string TypeName {get { } }
    public string DisplayName {get { } }
    public IList<string> Names {get { } }
    public IList<string> FileExtensions {get { } }
```

4.18 PlatformAdaptationLayer Class

This class abstracts system operations used by the DLR that could possibly be platform specific. Hosts can derive from this class and implement operations, such as opening a file. For example, the Silverlight PAL could go to the server to fetch a file.

To use a custom PAL, you derive from this type and implement the members important to you. You also need to derive a custom ScriptHost that returns the custom PAL instance. Then when you create your ScriptRuntime, you explicitly create a ScriptRuntimeSetup and set the HostType property to your custome ScriptHost.

4.18.1 Class Summary

```
public class PlatformAdaptationLayer {
    public static readonly PlatformAdaptationLayer Default

    public virtual Assembly LoadAssembly(string name)
    public virtual Assembly LoadAssemblyFromPath(string path)

    public virtual void TerminateScriptExecution(int exitCode)

    public StringComparer PathComparer { get;}
    public virtual bool FileExists(string path)
    public virtual bool DirectoryExists(string path)
    public virtual Stream
        OpenInputFileStream(string path, FileMode mode,
                            FileAccess access, FileShare share)
    public virtual Stream
        OpenInputFileStream(string path, FileMode mode,
                            FileAccess access, FileShare share,
                            int bufferSize)
    public virtual Stream OpenInputFileStream(string path)
    public virtual Stream OpenOutputFileStream(string path)
    public virtual string[] GetFiles(string path,
                                     string searchPattern)
    public virtual string GetFullPath(string path)
```

```
public virtual string CurrentDirectory {get;}
public virtual string[]
    GetDirectories(string path, string searchPattern)
public virtual bool IsAbsolutePath(string path)
```

4.19 SyntaxErrorException Class

4.20 ScriptExecutionException Class

This class and its subtypes represent errors that occurred while executing code within the hosting API. The hosting API wraps any error that occurs while dynamic language code executes in a ScriptExecutionException object and rethrows that. The DLR does NOT wrap parsing errors since we have other exceptions for those. The DLR does not wrap other API errors in this exception either.

4.21 ErrorListener Class

This is an abstract class that hosts can implement and supply to ScriptSource.Compile methods. Instead of raising exceptions for compilation errors, the compile methods report errors by calling on the ErrorListener.

4.21.1 Class Summary

```
public abstract class ErrorListener : MarshalByRefObject
    protected ErrorListener()

    public abstract void ErrorReported
          (ScriptSource source, string message, SourceSpan span,
           int errorCode, Severity severity)
```

4.22 Severity Enum

This enum identifies compiler error kinds when calling ErrorListener.ErrorReported.

4.22.1 Type Summary

```
public enum Severity {
    Ignore,
    Warning,
    Error,
    FatalError
```

4.23 SourceLocation Struct

4.24 SourceSpan Struct

4.25 ExceptionOperations Class

This class provides language-specific utilities for working with exceptions coming from executing code. You access instances of this type from Engine.GetService.

4.25.1 Class Summary

```
public sealed class ExceptionOperations : MarshalByRefObject {
    public string FormatException(Exception exception)
    public void GetExceptionMessage
                (Exception exception, out string message,
                 out string errorTypeName)
```

4.26 DocumentOperations Class

This class provides language-specific utilities for getting documentation and call signature information for objects coming from executing code. You access instances of this type from Engine.GetService.

4.26.1 Class Summary

```
public sealed class DocumentationOperations : MarshalByRefObject {
    public ICollection<MemberDoc> GetMembers(object value)
    public ICollection<OverloadDoc> GetOverloads(object value)
    public ICollection<MemberDoc> GetMembers(ObjectHandle value)
    public ICollection<OverloadDoc> GetOverloads(ObjectHandle value)
```

4.26.2 GetMembers Method

This method returns the collection of MemberDocs which in turn represent the name and kind of member for each. If there are no members, the collection is empty.

Signatures:

```
    public ICollection<MemberDoc> GetMembers(object value)
    public ICollection<MemberDoc> GetMembers(ObjectHandle value)
```

4.26.3 GetOverloads

This method returns the collection of OverloadDocs which in turn provide signature info. If the object is not invocable, then the collection is empty.

Signature:

```
    public ICollection<OverloadDoc> GetOverloads(object value)
    public ICollection<OverloadDoc> GetOverloads(ObjectHandle value)
```

4.27 MemberDoc Class

This class provides language-specific, basic information about members of an object. You access instances of this type from DocumentationOperations objects.

4.27.1 Class Summary

```
public class MemberDoc {
    public MemberDoc(string name, MemberKind kind)
    public string Name { get {} }
    public MemberKind Kind { get {} }
```

4.27.2 Name Property

4.27.3 Kind Property

4.28 MemberKind Enum

4.28.1 Type Summary

```
public enum MemberKind {
    None,
    Class,
    Delegate,
    Enum,
    Event,
    Field,
    Function,
    Module,
    Property,
    Constant,
    EnumMember,
    Instance,
    Method,
    Namespace
```

4.28.2 Members

The Summary section shows the type as it is defined (to indicate values of members), and this section documents the intent of the members.

None	Unsure of kind.
Class	
Delegate	
Enum	
Event	
Field	
Function	
Module	
Property	
Constant	

EnumMember	
Instance	
Method	
Namespace	

4.29 OverloadDoc Class

This class provides language-specific information about all the overloads when an object is invocable. You access instances of this type from DocumentationOperations objects.

4.29.1 Class Summary

```
public class OverloadDoc {
        public OverloadDoc(string name, string documentation,
                           ICollection<ParameterDoc> parameters)
        public OverloadDoc(string name, string documentation,
                           ICollection<ParameterDoc> parameters,
                           ParameterDoc returnParameter)
        public string Name { get {} }
        public string Documentation { get {} }
        public ICollection<ParameterDoc> Parameters { get {} }
        public ParameterDoc ReturnParameter { get {} }
```

4.29.2 Name Property

4.29.3 Documenation Property

This property returns any doc comments or documentation strings the language allows programmers to embed in code.

Signature:

```
        public string Documentation { get {} }
```

4.29.4 Parameters Property

This property returns a collection of ParameterDocs representing information such as name, type, doc comments, etc., associated with the parameter. If there are no parameters, the collection is empty.

Signature:

```
        public ICollection<ParameterDoc> Parameters { get {} }
```

4.29.5 ReturnParameter

This property returns information about the return value as a ParameterDoc object.

Signature:

```
        public ParameterDoc ReturnParameter { get {} }
```

4.30 ParameterDoc Class

4.30.1 Class Summary

```
public class ParameterDoc {
    public ParameterDoc(string name)
    public ParameterDoc(string name, ParameterFlags paramFlags)
    public ParameterDoc(string name, string typeName)
    public ParameterDoc(string name, string typeName, string documentation)
    public ParameterDoc(string name, string typeName, string documentation,
                        ParameterFlags paramFlags)

    public string Name { get {} }
    public string TypeName { get {} }
    public ParameterFlags Flags { get {} }
    public string Documentation { get {} }
```

4.30.2 Name Property

4.30.3 TypeName Property

This property returns the type of the parameter as a name. If the type has a fully qualified form, this only returns the last token of the name. The purpose of this information is for tool presentation, not crossing over to the .NET reflection model of types.

Signature:

```
    public ParameterFlags Flags { get {} }
```

4.30.4 Documentation Property

This property returns any doc comments or strings associated with the parameter. If there is no such documentation, this property returns null.

Signature:

```
    public string Documentation { get {} }
```

4.31 ParameterFlags Enum

This enum identifies extra information about a parameter, such as whether it is caught as a rest argument or dictionary argument.

4.31.1 Type Summary

```
public enum ParameterFlags {
    None,
    ParamsArray,
    ParamsDict
```

4.31.2 Members

The Summary section shows the type as it is defined (to indicate values of members), and this section documents the intent of the members.

None	Just a positional parameter.
ParamsArray	Indicates the parameter is a rest parameter or collection of all the arguments after any supplied positional arguments.
ParamsDict	Indicated the parameter is a parameter that maps names of parameter to their values. It is like a rest parameter, but the rest of the arguments must be supplied as named arguments.

4.32 POST CLR 4.0 -- TokenCategorizer Abstract Class

Will spec colorization support after reconsidering best API for simple tooling and common support for VS plugins. This needs to be in another spec when fleshed out, but it is a placeholder for consideration.

4.32.1 Class Summary

```
public abstract class TokenCategorizer: MarshalByRefObject {
    void Initialize(object state, ScriptCodeReader sourceReader,
                    SourceLocation initialLocation)
    public abstract bool IsRestartable { get; }
    public abstract TokenInfo ReadToken()
    public abstract bool SkipToken()
    public abstract IEnumerable<TokenInfo> ReadTokens
                                          (int countOfChars)
    public abstract bool SkipTokens(int countOfChars)
    public abstract SourceLocation CurrentPosition { get; }
    public abstract object CurrentState { get; }
    public abstract ErrorListener ErrorListener { get; set; }
```

4.33 POST CLR 4.0 -- TokenCategory Enum

Will spec colorization support after reconsidering best API for simple tooling and common support for VS plugins. This needs to be in another spec when fleshed out, but it is a placeholder for consideration.

4.34 POST CLR 4.0 -- TokenInfo Struct

Will spec colorization support after reconsidering best API for simple tooling and common support for VS plugins. This needs to be in another spec when fleshed out, but it is a placeholder for consideration.

4.35 POST CLR 4.0 -- TokenTriggers Enum

Will spec colorization support after reconsidering best API for simple tooling and common support for VS plugins. This needs to be in another spec when fleshed out, but it is a placeholder for consideration.

4.36 CUT -- ConsoleHost Abstract Class ???

4.37 CUT -- ConsoleHostOptions Class

4.38 CUT -- ConsoleHostOptionsParser Class

5 Current Issues

Host aborts. Need to provide a way for hosts to signal the DLR to abort execution so that we do not do a rude thread abort while the host is potentially executing deep in its code. The scenario is that the host has provided an OM layer that it protects all over and understands dyn code could abort, but the code internal (lower down) that the OM layer calls is not coded for rude aborts. The host executes dyn code, the dyn code calls back into the hosts OM, and the host then calls internal functions that shouldn't have to be protected everywhere against rude thread aborts. These internal functions do not worry about thread aborts because if that's happening, then the app is going down. The host would like to be able to support user actions to abort dyn code executions (for example, spinning cassette in VSMacros or ctrl-c in Nessie) by setting a ScriptRuntime.AbortExecutions flag. The DLR code periodically checks this, does whatever internal throwing it wants, but then just returns from whatever functions. Is the model that the host has to clear the flag, or can the DLR know to stop all code on all threads, then clear the flag?

Stack depth control. Need to add support for the host to provide a delegate or some control on handling stack overflows. It could set a limit, and when we detect it, we call the host's delegate. It could set the abort flag, kill the thread, perhaps return a value to abort, or whatever.

Engine recycling. Need to design model for engines that can recycle themselves and how to shutdown or reset ScriptRuntimes for use in asp.net like server situations.

Doc exceptions. Doc for each member what exceptions can be thrown, should we define catcher exceptions for lower-level calls (e.g., wrapper for loadassm to hide its nine exceptions).

Global value src loc. Consider host inspecting global value and asking for src line where the value came from for good error msgs to users.

MWeb and error sinking. Error sink'ing and MWeb requirements need to be factored into hosting API.

DLL loading filters. Hosts can control dynamic language file resolution, but they don't get called on for DLL resolution. This means they can't redirect or limit which DLLs can load. Not sure this is meaningful since the code could just use .NET calls from standard BCL libs to load assemblies (eh?).

MBRO/remoting lifetime lease timeout. Currently if you do not use a host for 15 min, it dies, and you lose your data. We've added InitializeLifetimeService on objects to cause them to stay alive forever, but we need to think through the design and the host controls here.

Remote AppDomain DLR exceptions need to remote. An exception may be thrown in the remote AppDomain, and the host application needs to be able to catch this error. Currently there are some DLR exceptions that don't serialize across the AppDomain boundary or contain data that doesn't serialize across the AppDomain boundary. Does it wrap all local exceptions with a remotable exception that refers back to the local exception? Or are all exceptions required to be serializable across AppDomain boundaries.

VB hosting and static info for compiling. One team asked us to support hosts that want to pass static info in for globals or scope variables. We think this makes sense so that when the DLR supports optional explicit typing, inferencing, etc., then hosts could supply optional explicit type info.

The rough idea for how to do this (given current VB and C# namespace and scope chains) is to take a chain of SourceMetadata objects with names and types declared in them. Then, when the host executes the code, it would reproduce the shape of the chain with ScriptScope objects. When hosts construct these ScriptScope chains, they probably should be able to both use SetVar to set add members, or create a ScriptScope on a static .NET object (where the members of the object match the SourceMetadata names and types).

Add a SourceMetadata type with:

- Names -> Dictionary<string, Type> (ok to use Type since it would have to be loaded in both app domains if doing remote hosting)
- ReferenceDlls -> string[]
- ImportedNamespaces -> string[]
- Parent -> SourceMetadata

Add to ScriptSource:

- .Compile(CompilerOptions, ErrorSink, SourceMetadata) where first two can be null for defaults.

Add to CompiledCode (if we follow through on supporting this, we might still add this only to a lower-level API like LangCtx):

- .GetExpressionTree(), but I suspect we also need ...

Add to ScriptScope

- .Parent -> ScriptScope (returns null by default, must be set in constructor)

GetMembersVerbose. This would return something like IList<Tuple<string,flags>>, but it would support poor man tooling for objects or languages that wanted to report value namespaces or categorize names somehow.

Compiler options between REPL inputs. Ipy.exe compiles the input "from future import truedivision", and then on the next line compile the input "3/2". How does the scope/compiler context flow from the first to the second? Maybe we can unify the VB SrcMetadata request with this to capture compiler options and let them be updated across compilation.

Need to revisit ScriptIO.SetInput. Do we need encoding arg?

```
public void SetInput(Stream stream, TextReader reader,
                     Encoding encoding)
```

Consider expanding LoadAssembly and simplifying name. Probably need to expand to type libs for COM interop (check with BizApps folks):

ScriptRuntime.LoadTypeLibrary(TypeLibrary library)

ScriptRuntime.LoadTypeLibrary(TypeLibraryName libraryName)

TypeLibrary and TypeLibraryName types don't exist yet. Also consider consolidating:

ScriptRuntime.Load(Assembly assembly)

ScriptRuntime.Load(AssemblyName assembly)

ScriptRuntime.Load(TypeLibrary library)

ScriptRuntime.Load(TypeLibraryName libraryName)

Add a convention for a .NET exception representing failed name look up that is common across all languages. Otherwise, hosts have to handle N different exceptions from all the languages. Can we use the python exception folding trick here?

ObjOps.Equal and Operators.Equals should match in name.

Spec HostingHelpers and think about how it fits the general model.

ScriptScope doesn't offer a case insensitive variable lookup even if languages support it. Since we never fully build the case-insensitive symbol design, and since we intend to go to dict<str,obj> over IAttrColl, we have no model for langs that want to do case-insensitive lookups without O(n) searching the table.