Using .NET Classes in IDEAScript

The purpose of this guide is to demonstrate how to communicate with .NET objects in IDEAScript. This can be helpful in extending the capabilities of IDEAScript and perform processing that is not supported in IDEAScript.

Creating the .NET Project

Since IDEA Version Nine uses the .NET 4.0 Framework this document will be using Visual Studio 2010. The process will be similar in other versions of Visual Studio, including the Express versions, but the screens seen may be slightly different depending on the version. Examples will also be using C# as the programming language but VB.NET could be used just as easily.

To start you will want to create a new Visual C#/VB.NET Class library. First, you will need to start Visual Studio by right clicking its icon and selecting Run as Administrator. Visual Studio needs to register the DLL each time the project is built.

This new library will contain the .NET code that you want IDEAScript to access. Create a new project by clicking File -> New -> Project. Give your project the name IDEAdotNETInterop. After the project is created right click the Class1.cs and select Delete.

The first problem that needs to be addressed is that .NET is managed code and IDEAScript is unmanaged code. In order for the two to work together IDEAScript must be able to find the managed code. You do that by exposing your project to COM. To do that from the Project menu select the properties for the project. On the Application tab click the Assembly Information… button. On the dialog that comes up select the Make assembly COM-Visible check box and click OK.

Now when the project is compiled certain portions of the assembly will be visible to COM, and thus IDEAScript. This will allow the use of the CreateObject function in IDEAScript to create an instance of a .NET class in that assembly and access its properties, methods, and so forth which are visible to COM.

In order for IDEAScript to be able to find the assembly it must be registered. Otherwise Windows will not be able to find the assembly. For development purposes you can setup the project so that Visual Studio will register the assembly. Do so by selecting the Build tab of the Properties window and checking the Register for COM interop check box. In order to use this assembly on a machine other than the development machine the assembly will need to be registered manually. That will be covered in a later section.

Creating the Class

The next step will be to create a class and expose its public members to COM. There is one rule that you will need to follow. The class must either have no constructor or a constructor that takes no parameters. The reason for this is that, when the CreateObject function in IDEAScript is used, no parameters can be passed to the constructor of the class.

Add a new class to the project by right clicking the project in the Solution Explorer, selecting Add and then New Class. Name this new class Calculator. Update the code for the class as follows.

```
public class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }

    public int Multiply(int a, int b)
    {
        return a * b;
    }
}
```

The next step is to make these two methods so that they are visible to COM and in turn IDEAScript. In order to do so, first create an interface with all of the public members and then create an interface for them. This is done by selecting all of those members, right clicking the selection, selecting Refactor and then Extract Interface. Use the default options when extracting the interface. This interface defines what members of the class COM will be able to interact with. Open the interface and add the following attribute to it as follows.

```
[System.Runtime.InteropServices.ComVisible(true)]
interface ICalculator
{
    int Add(int a, int b);
    int Multiply(int a, int b);
}
```

This makes the members in this interface COM visible so IDEAScript can interact with them. Similar attributes need to be added to the Calculator class and to its public members that you want to access in IDEAScript. To save a bit of typing add a using statement for the System.Runtime.InteropServices namespace. The new code for the Calculator class is next.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Runtime.InteropServices;

namespace IDEAdotNETInterop
{
    [ComVisible(true)]
    [ClassInterface(ClassInterfaceType.None)]
    public class Calculator : IDEAdotNETInterop.ICalculator
```

```
    {
        [ComVisible(true)]
        public int Add([MarshalAs(UnmanagedType.I4)]int a, [MarshalAs(UnmanagedType.I4)]int b)
        {
            return a + b;
        }

        [ComVisible(true)]
        public int Multiply([MarshalAs(UnmanagedType.I4)]int a,
[MarshalAs(UnmanagedType.I4)]int b)
        {
            return a * b;
        }
    }
}
```

The class is now visible to COM and in turn the two public methods. The parameters for the methods have been marshalled using the MarshalAs attribute with the type UnmanagedType.I4. This makes it clear what the types are for the parameters for the methods, to others using the class. The value UnmanagedType.I4 represents a 4 byte integer and is equivalent to the Long variable type in IDEAScript. The last step is to build the project by pressing F6 or selecting Build -> Build Solution.

Accessing the Library from IDEAScript

The library has successfully been built and is visible to COM. Now it can be used in IDEAScript the same way as other objects. First, create a new macro that will be used to test that the library can be accessed via IDEAScript. Create a new macro in your current project and update the code for Sub Main as follows.

```
Sub Main
    Dim calc As Object

    calc = CreateObject("IDEAdotNETInterop.Calculator")

    MsgBox(calc.Add(10, 10))
    MsgBox(calc.Multiply(10, 10))
    calc = Nothing
End Sub
```

Saving and running the macro will create two message boxes. The first message box will display 20, which is the sum of 10 and 10. The second will display 100, which is the product of 10 and 10.

Working with Strings

The sample in this article works with a simple data type, integers. It can also work with other basic data types, such as doubles or characters. Even though strings aren't basic data types it is straight forward to work with them. That is easily done once they are marshalled correctly, in the same way as integer types were previously. For use in IDEA string parameters and return values should be marshalled using UnmanagedType.LPStr. Below is some updated code for the interface and class with examples on how to correctly marshal parameters and return types to use strings in IDEAScript.

```
using System;
namespace IDEAdotNETInterop
{
    [System.Runtime.InteropServices.ComVisible(true)]
    interface ICalculator
    {
        int Add(int a, int b);
        int Multiply(int a, int b);
        int StrLen(string value);
        string Replace(string value, string toBeReplaced, string replaceWith);
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;

namespace IDEAdotNETInterop
{
    [ComVisible(true)]
    [ClassInterface(ClassInterfaceType.None)]
    public class Calculator : IDEAdotNETInterop.ICalculator
    {
        [ComVisible(true)]
        public int Add(
            [MarshalAs(UnmanagedType.I4)]int a,
            [MarshalAs(UnmanagedType.I4)]int b)
        {
            return a + b;
        }

        [ComVisible(true)]
        public int Multiply(
            [MarshalAs(UnmanagedType.I4)]int a,
            [MarshalAs(UnmanagedType.I4)]int b)
        {
            return a * b;
        }

        [ComVisible(true)]
        public int StrLen([MarshalAs(UnmanagedType.LPStr)]string value)
        {
            return value.Length;
        }
```

```
        [ComVisible(true)]
        [return: MarshalAs(UnmanagedType.LPStr)]
        public string Replace([
            MarshalAs(UnmanagedType.LPStr)]string value,
            [MarshalAs(UnmanagedType.LPStr)]string toBeReplaced,
            [MarshalAs(UnmanagedType.LPStr)]string replaceWith)
        {
            return value.Replace(toBeReplaced, replaceWith);
        }
    }
}
```

The following is an updated IDEAScript macro that calls the two new functions added.

```
Sub Main()
    Dim calc As Object
    Dim replaced As String

    calc = CreateObject("IDEAdotNETInterop.Calculator")

    MsgBox(calc.Add(10, 10))
    MsgBox(calc.Multiply(10, 10))
    MsgBox(calc.StrLen("What is the length of this string."))

    replaced = calc.Replace("Going to replace cat with dog.", "cat", "dog")
    MsgBox(replaced)

    calc = Nothing
End Sub
```

As can be seen working with strings is relatively straightforward as well. The trick is that the types need to be marshalled correctly.

Registering the Library

There is an extra step that is required if the .NET class will need to be accessed from outside of the development machine. That is of course registering the DLL so that IDEAScript can find it. If you look in the output folder for the project there will be three files with the extensions DLL, PBD and TLB. The ones required are the DLL and TLB. These files will need to be copied to each computer which needs to access the .NET objects. Once they are on that computer they need to be registered using RegAsm.exe.

The simplest deployment strategy is to place all of the required files in one folder, including a batch file, and then run the batch file for either an elevated command prompt or by right clicking and selecting Run as Administrator.

This is a sample batch script that will copy the required files used in this guide to a folder and then register the library. Note that I used the .NET version that IDEA uses in registering. In theory any version of .NET could be targeted. However, it is best if you stick with the version that IDEA uses as there is no guarantee that a different version will be available on another computer.

```
copy IDEAdotNETInterop.* "C:\Program Files (x86)\IDEAdotNETInterop\"

cd "\Program Files (x86)\IDEAdotNETInterop\"
"%systemroot%\Microsoft.NET\Framework\v4.0.30319\RegAsm.exe" IDEAdotNETInterop.dll
/tlb:IDEAdotNETInterop.tlb
```

Pitfalls

There are a few pitfalls to keep in mind. The first is related to memory management. It is important that .NET objects are disposed of correctly and set to null or nothing, depending on the language used. Therefore it is a good idea to include a Dispose method in the class and exposed it so that when the user is done and any objects that need to be disposed can be and then can be set to null or nothing to remove the references.

As with simple data types, it is possible to work with objects as well. However, there are several major issues here and it is recommended to avoid using objects directly. The main issue is that in most cases the object will not be disposed correctly and will remain in memory after the IDEAScript has completed. If you must use a class, it is best to create a wrapper which can be used to call members and set members rather than returning the object itself.

When you are testing the .NET object in IDEAScript you must close out of IDEA completely in order to rebuild the application. IDEA will be holding a reference to the object and any attempts to register the new version will fail because there is still an open reference.

Conclusion

It is possible to use .NET code in an IDEAScript using this guide as a reference but only what you expose to COM. It is best that you use primitive data types where ever possible and do not return objects directly but create wrappers around them.