CHAPTER 4

Using the Scripting API

So far, all the C# code you've seen hasn't been any different from what you've been able to do in C# since version 1.0. That is, you still write C# code, you compile it, and an assembly is generated. Although having the inner workings of the compiler available for public consumption via the Compiler API empowers developers to analyze and transform their code, nothing has substantially altered the flow of the compilation process. However, that changes with the Update 1 release of Visual Studio 2015, because within the Compiler API is a brand-new Scripting API. With the Scripting API, C# can be treated as a scripting language. In this chapter, I'll show you how to use the Scripting API to provide a dynamic way to augment applications. But before we do that, let me briefly define what a scripting language really is.

What Is a Scripting Language?

Before we get into the details of the Scripting API, let's spend some time on scripting languages in general. What makes a language a "scripting" language? What are its characteristics? How does it work? Are scripting languages substantially different from other languages? Knowing the realm that you're entering in this chapter will help you understand the Scripting API better and how C# fits in the domain of scripting languages.

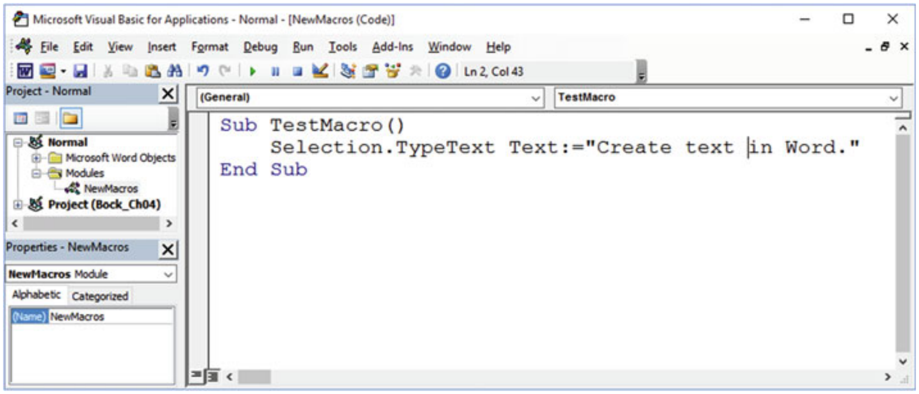
Orchestrating an Environment

Traditionally, scripting languages have been viewed as "glue" languages. They're usually not as powerful as other popular programming languages if their feature sets are compared and contrasted. However, their power lies in their simplicity. They're not designed to build complex domain layers or implement web servers; rather, they work with a given system and extend it in ways that the original designers may not have intended. They'll tie different parts and members together to create new functionality without having to go through a typical compile, test, and deploy scenario that most applications will partake in. Essentially, they *orchestrate* different pieces available to them.

Well-known scripting languages that developers have used are Bash, Python, and Lua. Another that developers in the .NET arena may have heard of and used in their applications is Visual Basic for Applications (VBA). VBA allows developers to control Office applications in a programmic way. The object model for an Office application may

J. Bock, *.NET Development Using the Compiler API*, DOI 10.1007/978-1-4842-2111-2_4

seem quite complex at first, but you can record macros in an Office application to see the pieces of the object model in action. Figure 4-1 shows the VBA macro editor with a snippet of code that was created by me just typing into Word.



```
Sub TestMacro()
    Selection.TypeText Text:="Create text in Word."
End Sub
```

Figure 4-1. *Creating macros in Word*

By using the Macros ➤ Record Macros feature in Word (which is on the View menu), I was able to figure out that by using TypeText on the Selection object, I could insert whatever text I wanted to. Of course, there are far more objects and methods available to you to control whatever part of Word you want to, but you don't have to remember every one. All you have to do is record your interaction and let Word generate the code for you.

Dynamic Capabilities

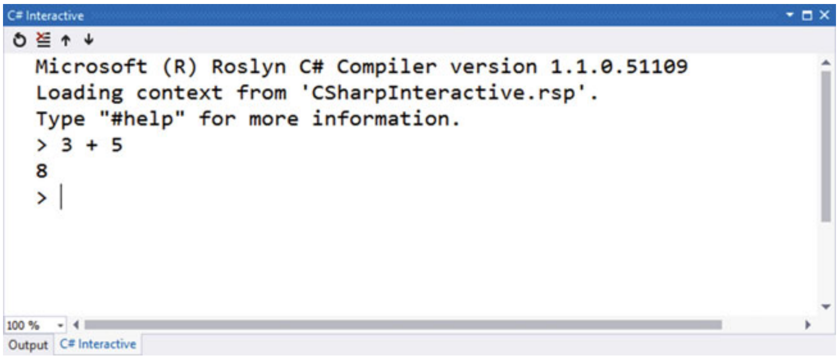
Another common aspect of a scripting language is its dynamic nature. Dynamic languages are those where the notion of types is a loose, or even nonexistent, one. Types can also be changed as the code executes. Examples of languages like this are Ruby and JavaScript. These languages have the notion of classes, but class definitions can change dramatically as code executes. Keep in mind that a scripting language can also be statically typed. There's no hard-and-fast rule when it comes to the dynamic capabilities of a language and whether that qualifies it as a scripting language.

In essence, any language can be considered a scripting language if an environment exists to provide the user with a dynamic experience. This is typically done with something called a Read, Evaluate, Print, Loop (REPL). Developers will use a REPL to try different ways to run their scripts and to immediately execute functionality available to the REPL. Lots of languages have this capability, but C# has always lacked this within the APIs provided by the .NET Framework. However, now with the Scripting API, you can treat C# as a scripting language. Let's start our investigation of the Scripting API by looking at a tool that uses this API in Visual Studio: the C# REPL.

Using the C# REPL

Shipping with Update 1 of Visual Studio 2015 is the C# Interactive window. It's a REPL that uses the Scripting API to allow developers to quickly experiment with snippets of C#. You won't see Scripting API usage just yet in this section, but keep in mind that this Visual Studio feature is powered by the Scripting API. By seeing how this window works, you'll better understand the capabilities available in the Scripting API to power dynamic programming experiences.

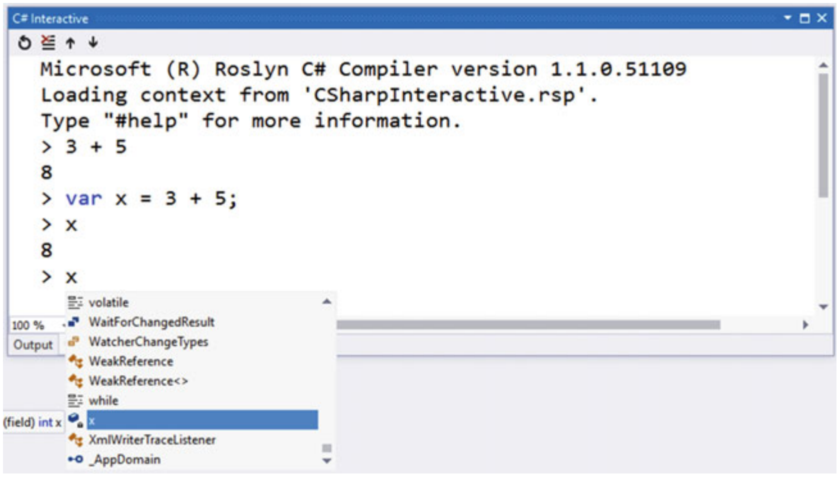
To start working with the C# Interactive window, open Visual Studio, and go to View ➤ Other Windows ➤ C# Interactive window. Note that you don't have to open or create a project to start working with this window. Type "3 + 5" in the window and press the Enter key. Figure 4-2 shows what you should see.



```
Microsoft (R) Roslyn C# Compiler version 1.1.0.51109
Loading context from 'CSharpInteractive.rsp'.
Type "#help" for more information.
> 3 + 5
8
> |
```

Figure 4-2. *Performing simple calculations within the C# Interactive window*

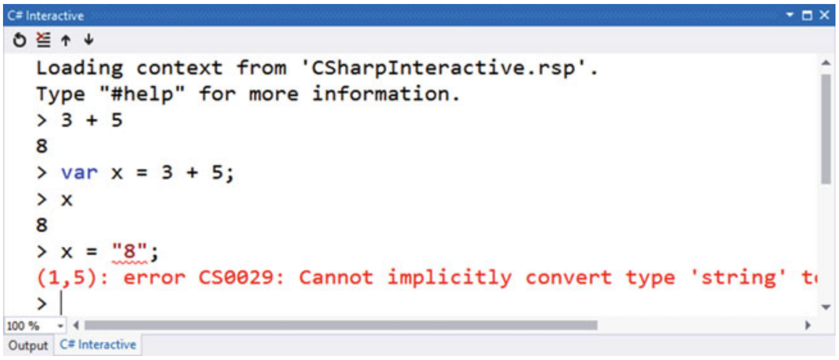
As a developer would expect, executing simple arithmetic calculations works. Let's set the value of that calculation to a variable called x. Once we do that, we can print out its value, as shown in Figure 4-3.



```
Microsoft (R) Roslyn C# Compiler version 1.1.0.51109
Loading context from 'CSharpInteractive.rsp'.
Type "#help" for more information.
> 3 + 5
8
> var x = 3 + 5;
> x
8
> x
```

Figure 4-3. *Printing the value of a variable*

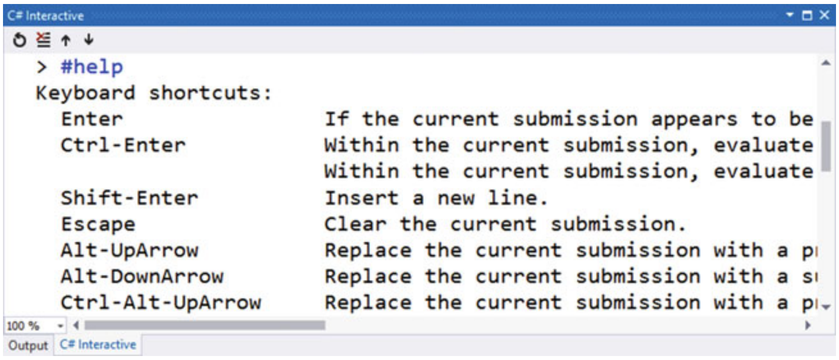
What's interesting to note in Figure 4-3 is that you get Intellisense within the interactive window. It knows there's a variable called x within the scope of this interactive session. It also knows that the variable is typed as an `int`. Scripting languages have a tendency to have very loose typing semantics, but even though the Interactive window is a scripting environment, C# retains its strong typing semantic. Figure 4-4 shows that assigning x to a `string` after it was initially assigned to an `int` won't work.



```
Loading context from 'CSharpInteractive.rsp'.
Type "#help" for more information.
> 3 + 5
8
> var x = 3 + 5;
> x
8
> x = "8";
(1,5): error CS0029: Cannot implicitly convert type 'string' t
> 
```

Figure 4-4. *Strong typing in the C# Interactive window*

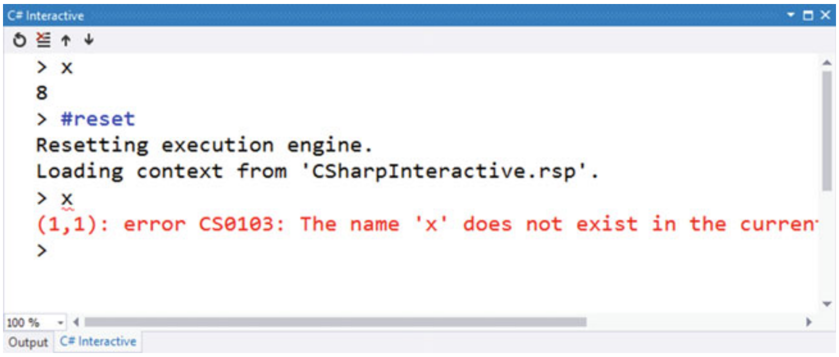
At any time in the window you can type #help to learn different commands that are available during the session. A sample of the #help output is shown in Figure 4-5.



```
> #help
Keyboard shortcuts:
  Enter                 If the current submission appears to be
  Ctrl-Enter            Within the current submission, evaluate
                        Within the current submission, evaluate
  Shift-Enter           Insert a new line.
  Escape                Clear the current submission.
  Alt-UpArrow           Replace the current submission with a pı
  Alt-DownArrow         Replace the current submission with a sı
  Ctrl-Alt-UpArrow      Replace the current submission with a pı
```

Figure 4-5. *Using help to display various commands*

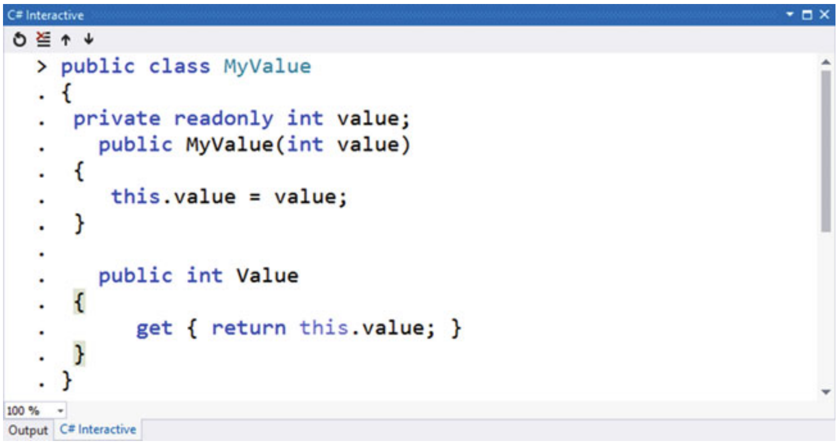
For example, you can type #cls to clear the screen. You can also use #reset to clear any current script state. Figure 4-6 shows what happens after you type #reset and then look at the value for x.



```
> x
8
> #reset
Resetting execution engine.
Loading context from 'CSharpInteractive.rsp'.
> x
(1,1): error CS0103: The name 'x' does not exist in the curren
>
```

Figure 4-6. *Resetting the interactive session*

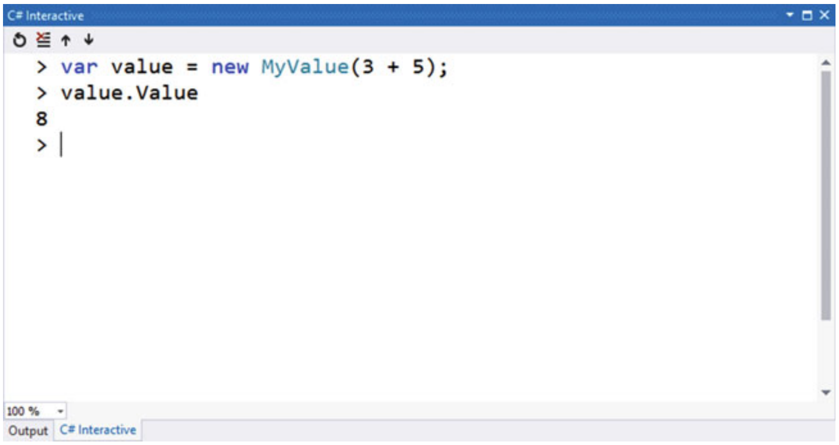
You can also define types within the session. To do this, you start typing the definition of a class, and then press Enter. The Interactive window will go into a multiline edit mode, so you can add members to the class, like fields, properties, and constructors. Figure 4-7 shows what the window looks like when you define a class.



```
> public class MyValue
. {
.  private readonly int value;
.    public MyValue(int value)
.  {
.     this.value = value;
.  }
.
.    public int Value
.  {
.       get { return this.value; }
.  }
. }
```

Figure 4-7. *Creating a class in the Interactive window*

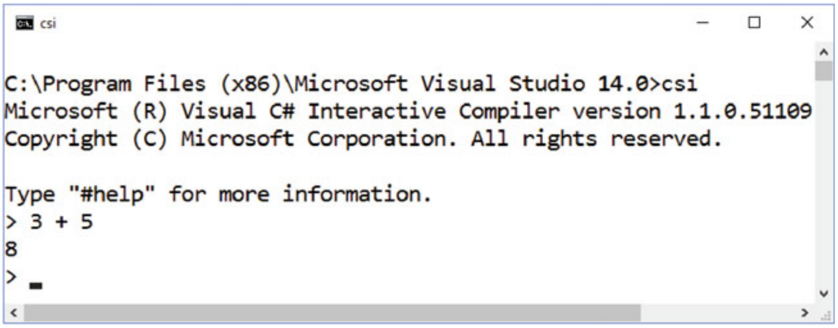
Once the class is defined, you can use it in your session. Figure 4-8 demonstrates code that creates an instance of the class.



```
> var value = new MyValue(3 + 5);
> value.Value
8
> 
```

Figure 4-8. *Using classes defined in the Interactive window*

Note that the Interactive window experience isn't limited to Visual Studio. If you bring up the Developer Command Prompt for VS2015 (from the Windows Start menu), you can type "csi" and get the same experience from a command window (without Intellisense). Figure 4-9 shows what that looks like.



```
C:\Program Files (x86)\Microsoft Visual Studio 14.0>csi
Microsoft (R) Visual C# Interactive Compiler version 1.1.0.51109
Copyright (C) Microsoft Corporation. All rights reserved.

Type "#help" for more information.
> 3 + 5
8
> _
```

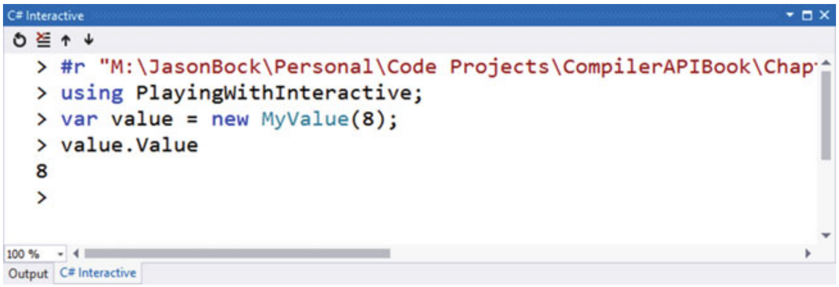
Figure 4-9. *Getting an interactive C# experience from the command line*

That's the basics of the Interactive window in Visual Studio. Now let's look at how to use code assets within the Interactive window.

Loading Code in Script

Creating code in the Interactive window is a great way to try different implementations without needing to create a Visual Studio solution. However, you may want to load references to other assemblies or previous code snippets in the Interactive window. Let's tackle the assembly loading issue first. To do this, you use the `#r` directive, which requires a full path to the location of the assembly file. Once its loaded, you can reference types from that assembly as you normally would. You can even include using statements in your session.

To see assembly referencing in action, create a Class Library project in Visual Studio. Add one class called `MyValue` that is structured the same way as the code in Figure 4-7 earlier in the chapter. After you've compiled the code, figure out the path where the assembly file exists. When you know where that is, you can type in the code you see in Figure 4-10 (notice that your path will be different than the one shown in the figure).



```
> #r "M:\JasonBock\Personal\Code Projects\CompilerAPIBook\Chap
> using PlayingWithInteractive;
> var value = new MyValue(8);
> value.Value
8
>
```

Figure 4-10. *Loading assemblies within the Interactive window*

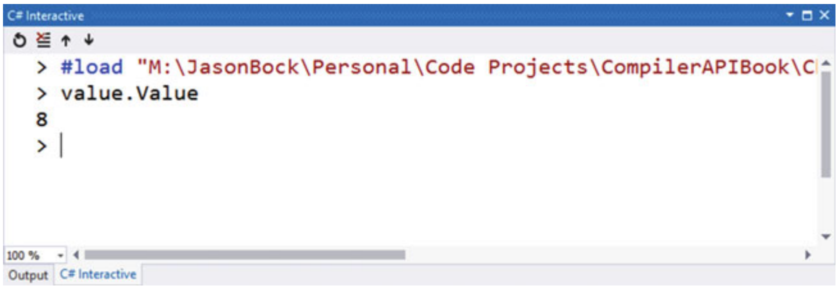
After the assembly is loaded, you can reference namespaces within that assembly via the using statement.

At this point, there is no easy way to save all the code entered in the session to a file. The only way to do this is to manually navigate to every line in the session buffer via the Alt + Arrow Up keystroke and then copy each line of code to a text file. But once you have your code in a file, you can load it at any time via the `#load` directive. Let's say you captured code in Listing 4-1 to a text file.

Listing 4-1. Creating a simple C# script file

```
#r "C:\YourCodePath\PlayingWithInteractive.dll"
using PlayingWithInteractive;
var value = new MyValue(8);
```

Notice that the path to load the assembly in Listing 4-1 would have to change based on where your assembly is located. Once you have that file, you can load it and examine variables that were created from the script, as shown in Figure 4-11.



```
> #load "M:\JasonBock\Personal\Code Projects\CompilerAPIBook\C
> value.Value
8
> |
```

Figure 4-11. *Loading C# script in the Interactive window*

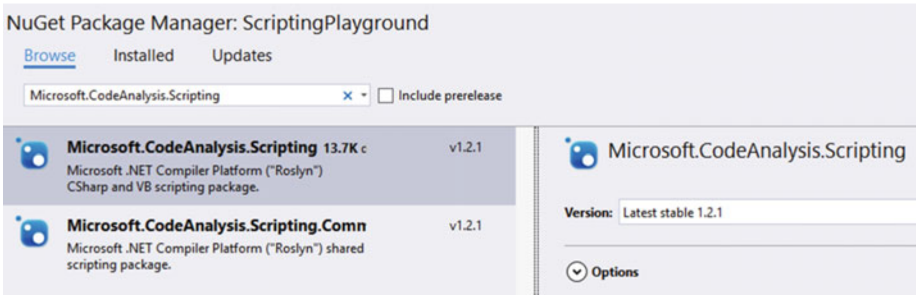
At this point, you should have a good understanding of how the Interactive window works in Visual Studio. Let's move our focus away from the Interactive window and direct it toward the code that powers its implementation: the Scripting API.

Making C# Interactive

You've already seen how to use C# as a scripting language via the Interactive window in Visual Studio and the csi.exe command line tool. Now we'll look at the Scripting API so you can use it to create extensible applications within .NET. You'll execute C# code as it's entered, preserve state from script execution to script execution, and even analyze the structure of C# script code.

Referencing the Scripting NuGet Package

The first activity you need to do is get the right NuGet package installed into your project. This is pretty simple to do. Let's say you create a simple console application called `ScriptingPlayground`. All you need to do is reference the `Microsoft.CodeAnalysis.Scripting` package, as shown in Figure 4-12.


NuGet Package Manager: ScriptingPlayground

Browse | Installed | Updates

Microsoft.CodeAnalysis.Scripting | Include prerelease

Microsoft.CodeAnalysis.Scripting 13.7K c v1.2.1
Microsoft .NET Compiler Platform ("Roslyn") CSharp and VB scripting package.

Microsoft.CodeAnalysis.Scripting.Comn v1.2.1
Microsoft .NET Compiler Platform ("Roslyn") shared scripting package.

Microsoft.CodeAnalysis.Scripting

Version: Latest stable 1.2.1

Options

Figure 4-12. Referencing the NuGet package for the Scripting API

Once NuGet is done, you should see two referenced assemblies with "Scripting" in their name, as shown in Figure 4-13.

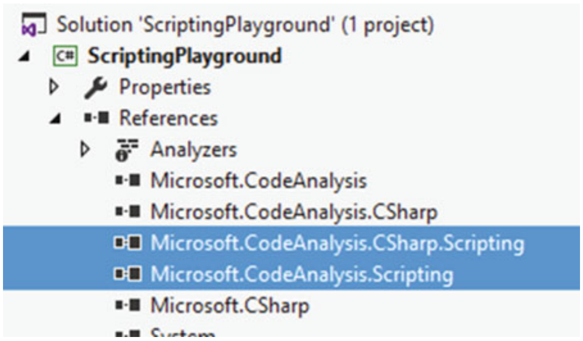

```
Solution 'ScriptingPlayground' (1 project)
  ScriptingPlayground
    Properties
    References
      Analyzers
      Microsoft.CodeAnalysis
      Microsoft.CodeAnalysis.CSharp
      Microsoft.CodeAnalysis.CSharp.Scripting
      Microsoft.CodeAnalysis.Scripting
      Microsoft.CSharp
```

Figure 4-13. Scripting API assemblies referenced in a project

Now that the project has the right references in place, let's start using members from the Scripting API.

Evaluating Scripts

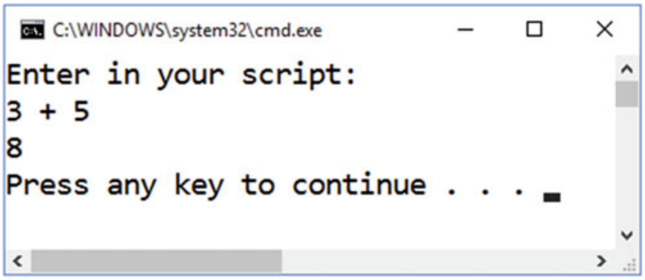
The main class you'll use for scripting is called `CSharpScript`. Its API surface is pretty small, meaning that it doesn't have a lot of methods, but within those methods is all of the power to make C# scriptable. Let's start by creating a simple class that will evaluate any code given to it in a console window. This is shown in Listing 4-2.

Listing 4-2. Evaluating code via EvalulateAsync()

```
using Microsoft.CodeAnalysis.CSharp.Scripting;
using Nito.AsyncEx;
using System;
using System.Threading.Tasks;

namespace ScriptingPlayground
{
  class Program
  {
    static void Main(string[] args)
    {
      AsyncContext.Run(() => Program.MainAsync(args));
    }

    private static async Task MainAsync(string[] args)
    {
      await Program.EvaluateCodeAsync();
    }

    private static async Task EvaluateCodeAsync()
    {
      Console.Out.WriteLine("Enter in your script:");
      var code = Console.In.ReadLine();
      Console.Out.WriteLine
        (await CSharpScript.EvaluateAsync(code));
    }
  }
}
```

■ **Note** The `AsyncContext` class comes from a NuGet package called Nito.AsyncEx. Currently in .NET you can't create an async version of the `Main()` method in a console application. But using `AsyncContext` makes this possible. Hopefully in a future version of .NET console applications will have this capability without needing a helper class. It's currently a feature request on the Roslyn GitHub site (`https://github.com/dotnet/roslyn/issues/1695`), but it's unclear if it will be included in a future C# release.

Executing code via `CSharpScript` is as simple as calling `EvaluateAsync()`. Figure 4-14 shows what the console window looks like when the application evaluates code.



```
Enter in your script:
3 + 5
8
Press any key to continue . . .
```

Figure 4-14. *Evaluating code via the Scripting API*

If you pass code that contains errors to `EvaluateAsync()`, you'll get a `CompilationErrorException`. This exception has a `Diagnostics` property on it that you can use to identify what the errors are with the code that was evaluated.

Although `EvaluateAsync()` lets you run simple pieces of C# code, there's more that you can do with scripting than just code evaluation. You can allow the script to use types and members for other assemblies. For example, let's say you created this class in an assembly called `ScriptingContext`:

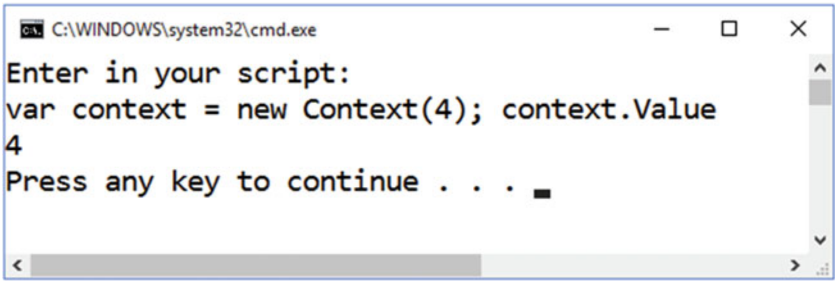
```
public sealed class Context
{
  public Context(int value)
  {
    this.Value = value;
  }

  public int Value { get; }
}
```

Assuming that your console application has a reference to the `ScriptingContext` assembly, you can allow script code to use the `Context` class by passing a `ScriptOptions` object to `EvaluateAsync()`. Listing 4-3 shows how this works.

Listing 4-3. Passing in assembly references to script evaluation

```
private static async Task EvaluateCodeWithContextAsync()
{
  Console.Out.WriteLine("Enter in your script:");
  var code = Console.In.ReadLine();
  Console.Out.WriteLine(
    await CSharpScript.EvaluateAsync(code,
      options: ScriptOptions.Default
        .AddReferences(typeof(Context).Assembly)
        .AddImports(typeof(Context).Namespace)));
}
```

All you need to do is add a reference to the assembly that houses the `Context` class via `AddReferences()`. The `AddImports()` call essentially adds a `using` statement with the namespace of the `Context` class to the script context. Therefore, a developer doesn't have to provide the full type name of the class. Once you change the console application to call `EvaluateCodeWithContextAsync()` on startup, you can reference the `Context` class in your script. Figure 4-15 shows what this looks like.



```
Enter in your script:
var context = new Context(4); context.Value
4
Press any key to continue . . .
```

Figure 4-15. *Using custom types in script*

As you can see in Figure 4-15, the code can use the `Context` class without any issues.

You can also provide an instance of an object to the script, allowing the script to use members on that object. For example, you can create a class called `CustomContext` that exposes a `Context` object and a `TextWriter`:

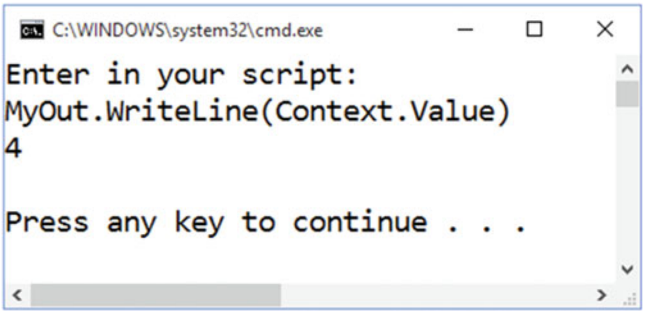
```
using System.IO;

namespace ScriptingContext
{
  public class CustomContext
  {
    public CustomContext(Context context, TextWriter myOut)
    {
      this.Context = context;
      this.MyOut = myOut;
    }

    public Context Context { get; }
    public TextWriter MyOut { get; }
  }
}
```

Then you can create an instance of CustomContext and set the globals argument to EvaluateAsync() to that CustomContext instance, as shown in Listing 4-4.

Listing 4-4. Using a global context object

```
private static async Task EvaluateCodeWithGlobalContextAsync()
{
  Console.Out.WriteLine("Enter in your script:");
  var code = Console.In.ReadLine();
  Console.Out.WriteLine(
    await CSharpScript.EvaluateAsync(code,
      globals: new CustomContext(
        new Context(4), Console.Out)));
}
```

Note that the Out property of the Console class is given to the CustomContext instance, letting the script print out information to the context. Figure 4-16 shows how you can use a script to print the Value property of the Context instance to the console window if you call EvaluateCodeWithGlobalContextAsync() from the async Main() method.



```
Enter in your script:
MyOut.WriteLine(Context.Value)
4

Press any key to continue . . .
```

Figure 4-16. *Using a global object in a script*

So far, you've seen how to use EvaluateAsync() to immediately execute a piece of valid C# code. In the next section, I'll discuss how you can analyze the script before you execute it.

Analyzing Scripts

Running code via EvaluateAsync() requires a bit more care than what I've shown so far. For example, if there's syntax errors, you'll get a CompilationErrorException. Rather than adding an exception handler to code, you can use Create() on the CSharpScript class to be a bit more defensive in your script execution implementation. Furthermore, these methods expose syntax trees and semantic models, so you can query the submitted script for details on what it intends to do. Listing 4-5 demonstrates how you can perform this script analysis (note: assume this method is part of the Program class defined in the "Evaluating Scripts" section).

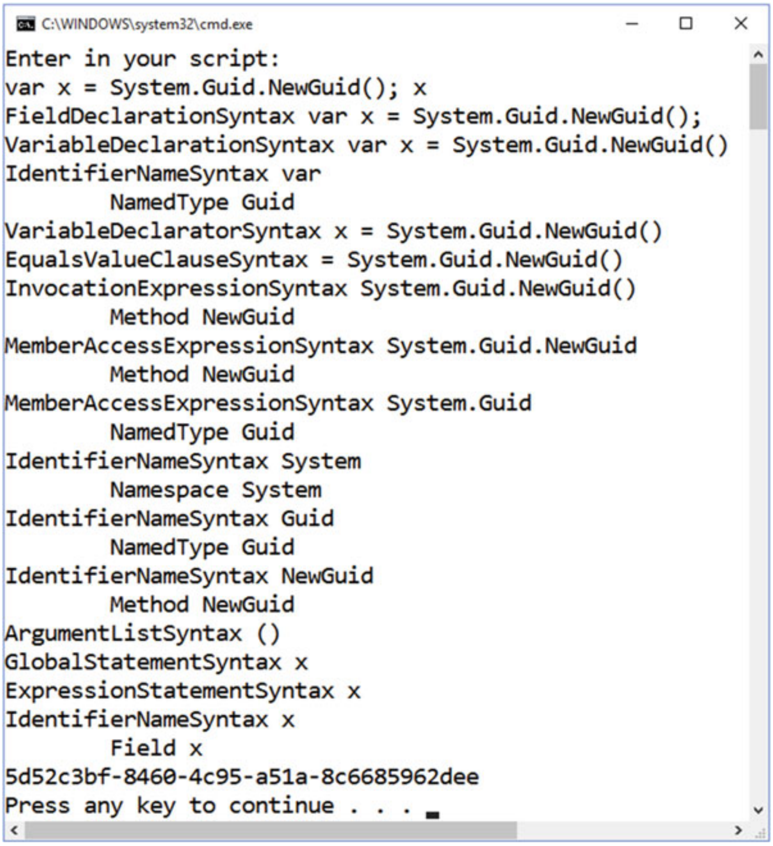
Listing 4-5. Analyzing a script's content

```
private static async Task CompileScriptAsync()
{
  Console.Out.WriteLine("Enter in your script:");
  var code = Console.In.ReadLine();
  var script = CSharpScript.Create(code);
  var compilation = script.GetCompilation();
  var diagnostics = compilation.GetDiagnostics();

  if(diagnostics.Length > 0)
  {
    foreach (var diagnostic in diagnostics)
    {
      Console.Out.WriteLine(diagnostic);
    }
  }
  else
  {
    foreach (var tree in compilation.SyntaxTrees)
    {
      var model = compilation.GetSemanticModel(tree);
      foreach (var node in tree.GetRoot().DescendantNodes(
        _ => true))
      {
        var symbol = model.GetSymbolInfo(node).Symbol;
        Console.Out.WriteLine(
          $"{node.GetType().Name} {node.GetText().ToString()}");

        if (symbol != null)
        {
          var symbolKind = Enum.GetName(
            typeof(SymbolKind), symbol.Kind);
          Console.Out.WriteLine(
            $"\t{symbolKind} {symbol.Name}");
        }
      }
    }

    Console.Out.WriteLine((await script.RunAsync()).ReturnValue);
  }
}
```

The return value of `Create()` is based on a `Script<T>` type, with `T` specified as an `object` (there's also a generic version of `Create()` you can use if you know what the script's return value will be in advance). From this `Script<T>` class, you can get compilation information with `GetCompilation()`, which returns a `Compilation` object. The `Compilation` class is the base class for the `CSharpCompilation` class you saw in

Chapter 1 in Listing 1-1. Therefore, you can look at diagnostic information, syntax trees, semantic models—everything that you learned about in Chapter 1 can be reused here to query the structure of the given script. In this example, if we have diagnostic information, we don't run the script; instead, we print out the error information. Otherwise, we display syntax and semantic information, and then run the script via `RunAsync()`.

Let's see what the code in Listing 4-5 does with a valid script. Figure 4-17 shows the results of a successful script analysis.

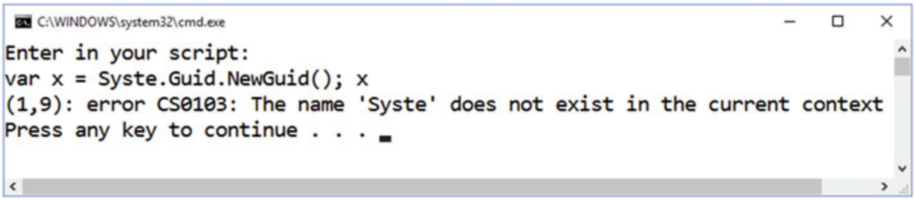


C:\WINDOWS\system32\cmd.exe

```
Enter in your script:
var x = System.Guid.NewGuid(); x
FieldDeclarationSyntax var x = System.Guid.NewGuid();
VariableDeclarationSyntax var x = System.Guid.NewGuid()
IdentifierNameSyntax var
        NamedType Guid
VariableDeclaratorSyntax x = System.Guid.NewGuid()
EqualsValueClauseSyntax = System.Guid.NewGuid()
InvocationExpressionSyntax System.Guid.NewGuid()
        Method NewGuid
MemberAccessExpressionSyntax System.Guid.NewGuid
        Method NewGuid
MemberAccessExpressionSyntax System.Guid
        NamedType Guid
IdentifierNameSyntax System
        Namespace System
IdentifierNameSyntax Guid
        NamedType Guid
IdentifierNameSyntax NewGuid
        Method NewGuid
ArgumentListSyntax ()
GlobalStatementSyntax x
ExpressionStatementSyntax x
IdentifierNameSyntax x
        Field x
5d52c3bf-8460-4c95-a51a-8c6685962dee
Press any key to continue . . .
```

Figure 4-17. *Analyzing a valid script*

Figure 4-18 shows what happens when the submitted script contains errors.



```
C:\WINDOWS\system32\cmd.exe
Enter in your script:
var x = Syste.Guid.NewGuid(); x
(1,9): error CS0103: The name 'Syste' does not exist in the current context
Press any key to continue . . .
```

Figure 4-18. *Analyzing an invalid script*

There's one more aspect to scripts that I should mention: storing state. Let's look at how state works with scripts in the next section.

State Management in Scripts

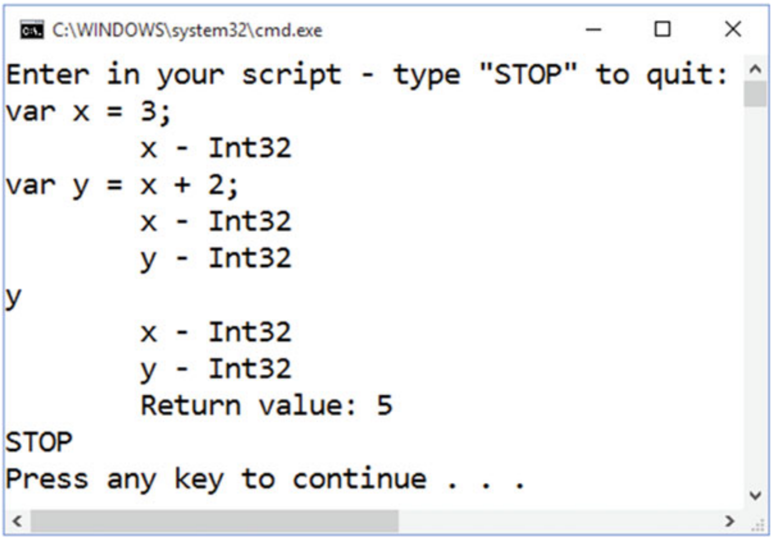
So far the script examples within the "Making C# Interactive" section have all been done via a single execution of script. That is, we get a line of code from the user, execute that script, and then the program is done. As you saw with the C# REPL in the "Using the C# REPL" section, you can type numerous lines of script code and refer to variables and classes issued earlier in the code. Fortunately, we don't have to do a lot to manage state information with the Scripting API. There's a `ScriptState` class that is returned from `RunAsync()` that you can use to retain information from one script execution to another. Listing 4-6 shows you how to use `ScriptState` to manage a script session (assume that this method is part of the `Program` class from the "Evaluating Scripts" section).

Listing 4-6. Using state management for scripts

```
private static async Task ExecuteScriptsWithStateAsync()
{
  Console.Out.WriteLine(
    "Enter in your script - type \"STOP\" to quit:");

  ScriptState<object> state = null;

  while (true)
  {
    var code = Console.In.ReadLine();

    if (code == "STOP")
    {
      break;
    }
```

```
      else
      {
        state = state == null ?
          await CSharpScript.RunAsync(code) :
          await state.ContinueWithAsync(code);

        foreach(var variable in state.Variables)
        {
          Console.Out.WriteLine(
            $"\t{variable.Name} - {variable.Type.Name}");
       }

        if (state.ReturnValue != null)
        {
          Console.Out.WriteLine(
            $"\tReturn value: {state.ReturnValue}");
        }
      }
    }
  }
```

As long as the given text doesn't equal "STOP", the code will continue running the script. Note that we capture the return value of RunAsync() (or ContinueWithAsync() if the state already exists). This return value will contain all of the code that was parsed in previous script executions. For example, we can print out the variables that have been created from each execution. Figure 4-19 shows how variables are retained as more script is entered.



```
Enter in your script - type "STOP" to quit:
var x = 3;
        x - Int32
var y = x + 2;
        x - Int32
        y - Int32
y
        x - Int32
        y - Int32
        Return value: 5
STOP
Press any key to continue . . .
```

Figure 4-19. *Using state to retain script execution information*

As Figure 4-19 shows, the first line of code creates a variable called x. The next line creates a new variable y, but because we're using ScriptState, we can reference the x variable.

Keep in mind that you can always use a global context object as you saw in the code in Listing 4-4. You can also reuse that context object across different script executions. Let's say we defined a class called DictionaryContext:

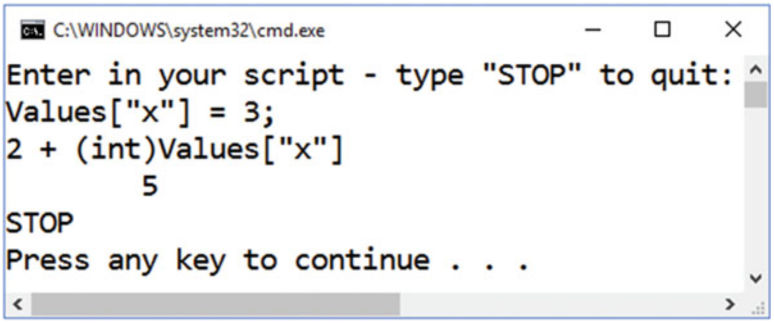
```
using System.Collections.Generic;

namespace ScriptingContext
{
  public sealed class DictionaryContext
  {
    public DictionaryContext()
    {
      this.Values = new Dictionary<string, object>();
    }

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

Listing 4-7 shows how you can manage state with a DictionaryContext instance (again, this code is part of the Program class from the "Evaluating Scripts" section).

Listing 4-7. Using a shared global object to store state

```
private static async Task ExecuteScriptsWithGlobalContextAsync()
{
  Console.Out.WriteLine(
    "Enter in your script - type \"STOP\" to quit:");

  var session = new DictionaryContext();

  while (true)
  {
    var code = Console.In.ReadLine();

    if (code == "STOP")
    {
      break;
    }
    else
    {
      var result = await CSharpScript.RunAsync(code,
        globals: session);
      if(result.ReturnValue != null)
      {
        Console.Out.WriteLine(
          $"\t{result.ReturnValue}");
      }
    }
  }
}
```

Figure 4-20 demonstrates how shared state, stored in `DictionaryContext`, can be used.



```
Enter in your script - type "STOP" to quit:
Values["x"] = 3;
2 + (int)Values["x"]
        5
STOP
Press any key to continue . . .
```

Figure 4-20. *Using a global object to share state*

We don't preserve variables between script executions, but we can store and load values from the shared context. Notice that because all of the values are stored as an `object`, we have to cast the value back to what we think it should be if we try to retrieve it from the dictionary.

Although we've spent a fair amount of time in this chapter looking at the cool features of the Scripting API, there are a couple of aspects of this API that you should be aware of if you decide to include its features in your applications. These are performance, memory usage, and security. Before I close out this chapter, let's take a look at these concerns in detail.

Concerns with the Scripting API

Being able to use C# as a scripting language is a welcome addition to the language's capability. However, there are a couple of areas where care should be taken to minimize potential problems from becoming actual issues. We'll discuss security later in the "Scripts and Security" section, but first we'll start with performance concerns and memory usage in scripts.

Scripts, Performance, and Memory Usage

When you see the Scripting API for the first time, you may start thinking about adding the ability to extend applications with dynamic C# code execution. As you saw with VBA in the "Orchestrating an Environment" section, exposing an object model for an application allows users to add features that aren't included within the application. However, keep in mind that there's there a cost involved with using scripts, both in performance and memory usage.

Let's create a small piece of code in a console application that will continually generate a simplistic, random C# mathematical statement and run it with the Scripting API. Once 1000 scripts are generated, it will generate the working set of the application along with how long it took to execute those scripts. Listing 4-8 shows how this works.

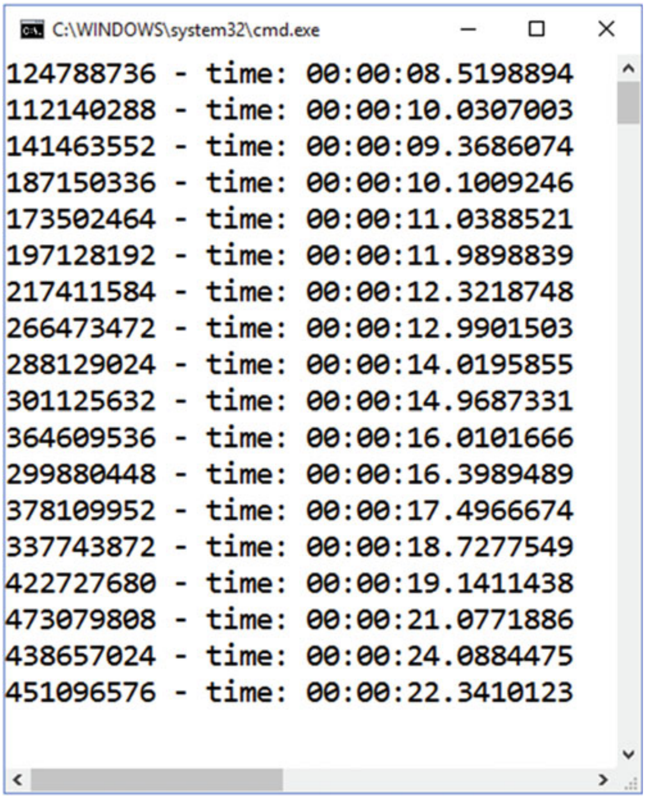
Listing 4-8. Executing random code via the Scripting API

```
using System.Diagnostics;

private static async Task EvaluateRandomScriptsAsync()
{
  var random = new Random();
  var iterations = 0;
  var stopWatch = Stopwatch.StartNew();

  while (true)
  {
    var script = $@"({random.Next(1000)} + {random.Next(1000)}) *
      {random.Next(10000)}";
    await CSharpScript.EvaluateAsync(script);
    iterations++;

    if (iterations == 1000)
    {
      stopWatch.Stop();
      Console.Out.WriteLine(
        $"{Environment.WorkingSet} - time: {stopWatch.Elapsed}");
      stopWatch = Stopwatch.StartNew();
     iterations = 0;
    }
  }
}
```

The code in Listing 4-8 generates code that looks like this: `(452 + 112) * 34`. To run this method, we'll put it into a `Program` class:

```
using Microsoft.CodeAnalysis.CSharp.Scripting;
using Nito.AsyncEx;
using System;
using System.Diagnostics;
using System.Linq.Expressions;
using System.Threading.Tasks;

namespace ScriptingAndMemory
{
  class Program
  {
    static void Main(string[] args)
    {
      AsyncContext.Run(
        () => Program.MainAsync(args));
    }

    private static async Task MainAsync(string[] args)
    {
      await EvaluateRandomScriptsAsync();
    }

    private static async Task EvaluateRandomScriptsAsync()
    {
      /* ... */
    }
  }
}
```

Figure 4-21 shows what happens when you run this code.



```
124788736 - time: 00:00:08.5198894
112140288 - time: 00:00:10.0307003
141463552 - time: 00:00:09.3686074
187150336 - time: 00:00:10.1009246
173502464 - time: 00:00:11.0388521
197128192 - time: 00:00:11.9898839
217411584 - time: 00:00:12.3218748
266473472 - time: 00:00:12.9901503
288129024 - time: 00:00:14.0195855
301125632 - time: 00:00:14.9687331
364609536 - time: 00:00:16.0101666
299880448 - time: 00:00:16.3989489
378109952 - time: 00:00:17.4966674
337743872 - time: 00:00:18.7277549
422727680 - time: 00:00:19.1411438
473079808 - time: 00:00:21.0771886
438657024 - time: 00:00:24.0884475
451096576 - time: 00:00:22.3410123
```

Figure 4-21. *Memory and performance characteristics of script execution*

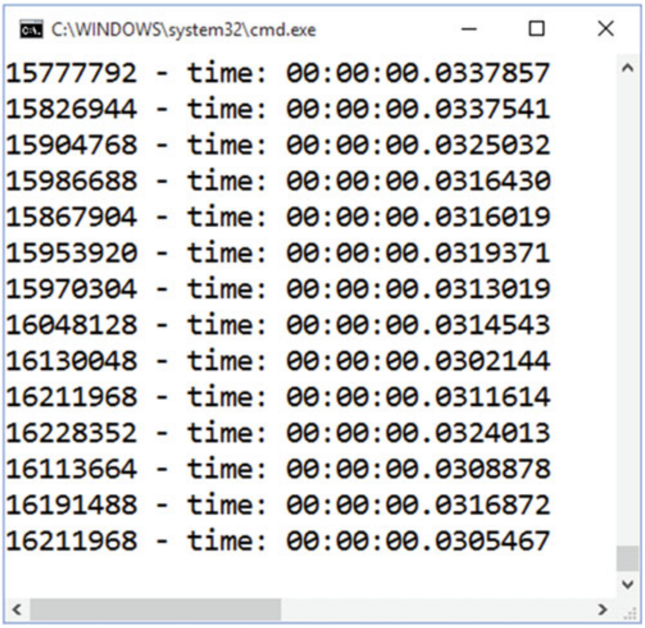
Notice that the size of the working set slowly, but surely, increases over time. Also, the time to execute 1000 scripts slowly increases as well.

Let's compare this approach of generating and executing dynamically generated code using the Scripting API with another technique: expressions. The `System.Linq.Expressions` namespace has types that allow you to create methods that are compiled to IL, just like C# code. Listing 4-9 shows how the Expressions API is used to create methods that are functionally the same as the script code generated in Listing 4-9 (note that this method exists in our `Program` class).

Listing 4-9. Executing random code via the Expressions API

```
private static void EvaluateRandomExpressions()
{
  var random = new Random();
  var iterations = 0;
  var stopWatch = Stopwatch.StartNew();
  while (true)
  {
    var lambda = Expression.Lambda(
      Expression.Multiply(
        Expression.Add(
          Expression.Constant(random.Next(1000)),
          Expression.Constant(random.Next(1000))),
        Expression.Constant(random.Next(10000))));
    (lambda.Compile() as Func<int>)();
    iterations++;

    if (iterations == 1000)
    {
      stopWatch.Stop();
      Console.Out.WriteLine(
        $"{Environment.WorkingSet} - time: {stopWatch.Elapsed}");
      stopWatch = Stopwatch.StartNew();
      iterations = 0;
    }
  }
}
```

Figure 4-22 shows the performance characteristics of the Expressions API approach by calling `EvaluateRandomExpressions()` from the Program's `Main()` method.



```
15777792 - time: 00:00:00.0337857
15826944 - time: 00:00:00.0337541
15904768 - time: 00:00:00.0325032
15986688 - time: 00:00:00.0316430
15867904 - time: 00:00:00.0316019
15953920 - time: 00:00:00.0319371
15970304 - time: 00:00:00.0313019
16048128 - time: 00:00:00.0314543
16130048 - time: 00:00:00.0302144
16211968 - time: 00:00:00.0311614
16228352 - time: 00:00:00.0324013
16113664 - time: 00:00:00.0308878
16191488 - time: 00:00:00.0316872
16211968 - time: 00:00:00.0305467
```

***Figure 4-22.** Memory and performance characteristics of expression execution*

In this case, the working set size and performance values are stable. The working set size is also smaller than the Scripting API approach. Furthermore, based on the values generated by this code, the Expressions API approach is three orders of magnitude faster than the Scripting API. For example, a script takes about 0.015 seconds to run. With an expression, it takes 0.00003 seconds.

The benefits of using the Expressions API doesn't mean that you should avoid using the Scripting API; far from it! Keep in mind that this test is literally creating thousands of scripts, and that's typically not how scripts are executed. Scripts are used to orchestrate other pieces of code in an application in a way that the developers didn't initially anticipate. Running a script 1000 times a second continuously within this context isn't common. Scripts are also exploratory, especially with a REPL. Once a developer has done enough C# scripting experimentation, that code can potentially be moved into a more typical compilation pipeline where the result is an assembly that can be optimized in numerous ways. Finally, a script can allow you to create new classes; the Expressions API is limited to a method implementation.

Another area where a developer should be cautious with scripts is with security. Let's investigate this issue next.

Scripts and Security

It's tempting to give users the ability to interact with an application's features in ways that were not originally codified as pieces of accepted functionality. For example, I've seen a number of applications at clients that I've consulted for where users can create reports based on the information contained within the application's database. Usually, this means they can submit SQL statements and save the data into an Excel spreadsheet. Initially, this sounds like a great idea, because the application empowers uses to go beyond what the application can provide. Unfortunately, this can also be a source of unexpected problems as well, such as:

- Performance. Queries that are submitted may cause significant delays due to unexpected fields being seached, where those fields do not have any indexes in place. This can affect the performance of other areas of the application.
- Resource use. If the users enter queries that start with `"SELECT *"`, they may retrieve a large amount of data that will tax the system's resources.
- Security. Entering queries that take advantage of SQL injection techniques may cause significant damage to the data contained within the database.

Security is the issue we'll focus on. If you want to use the Scripting API, you have to keep in mind what functionality the user will have available and ensure they can only use certain .NET members in their script code, or prevent them from using potentially harmful APIs. Let's look at an example.

This demonstration uses a console application that provides an object model for a user to interact with. The first step is to create a simple `Person` class defined as follows:

```
public sealed class Person
{
  public Person(string name, uint age)
  {
    this.Name = name;
    this.Age = age;
  }

  public void Save() { }

  public uint Age { get; set; }
  public string Name { get; set; }
}
```

We'll also create a script context that exposes a list of people:

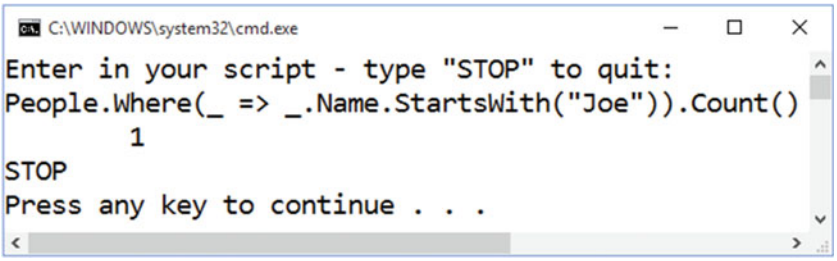
```
public sealed class ScriptingContext
{
  public ScriptingContext()
  {
    this.People = ImmutableArray.Create(
      new Person("Joe Smith", 30),
      new Person("Daniel Davis", 20),
      new Person("Sofia Wright", 25));
  }

  public ImmutableArray<Person> People { get; }
}
```

This is the object model that we'll pass to the `CSharpScript` class so scripts can query the list and find people that match a set of criteria the user defines. Listing 4-10 shows the asynchronous `MainAsync()` method that is created to handle this scenario.

Listing 4-10. Running scripts with an accesssible application object model

```
using System.IO;
using System.Linq;

private static async Task MainAsync(string[] args)
{
  File.WriteAllLines("secrets.txt",
    new[] { "Secret password: 12345" });

  Console.Out.WriteLine(
    "Enter in your script - type \"STOP\" to quit:");
  var context = new ScriptingContext();
  var options = ScriptOptions.Default
    .AddImports(
      typeof(ImmutableArrayExtensions).Namespace)
    .AddReferences(
      typeof(ImmutableArrayExtensions).Assembly);

  while (true)
  {
    var code = Console.In.ReadLine();

    if (code == "STOP")
    {
      break;
    }
    else
    {
      var script = CSharpScript.Create(code,
        globalsType: typeof(ScriptingContext),
        options: options);
      var compilation = script.GetCompilation();
      var diagnostics = compilation.GetDiagnostics();

      if (diagnostics.Length > 0)
      {
        foreach (var diagnostic in diagnostics)
        {
          Console.Out.WriteLine(diagnostic);
        }
      }
      else
      {
        var result = await CSharpScript.RunAsync(code,
          globals: context,
          options: options);

        if(result.ReturnValue != null)
        {
          Console.Out.WriteLine($"\t{result.ReturnValue}");
        }
      }
    }
  }
}
```

The code is similar to code you saw in Listing 4-5. We create a global context object so scripts can use members on that context. If there are no errors, we run the script. Figure 4-23 shows what happens when a script is entered that uses LINQ to query the `People` array.



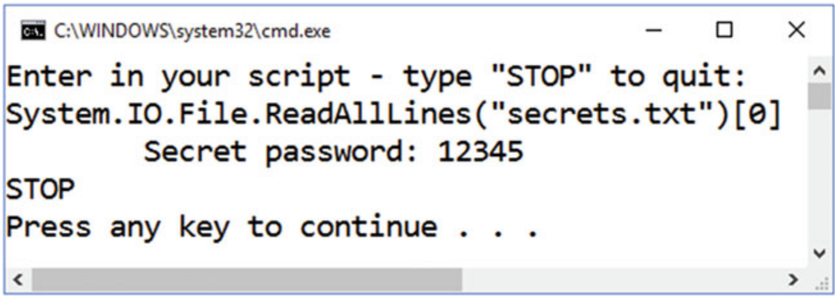
```
Enter in your script - type "STOP" to quit:
People.Where(_ => _.Name.StartsWith("Joe")).Count()
        1
STOP
Press any key to continue . . .
```

Figure 4-23. *Running a script to find specific people*

As expected, looking for people with a name that starts with "Joe" returns one person. But notice that `MainAsync()` creates a file with secret information when it starts. If a script user started using members from the `System.IO` namespace like this:

```
System.IO.Directory.EnumerateFiles(".").ToList()
  .ForEach(_ => System.Console.Out.WriteLine(_));
```

They'll see "secrets.txt" in the resulting list printed out to the `Console` window. Figure 4-24 shows the harm that can be done with this information.



```
Enter in your script - type "STOP" to quit:
System.IO.File.ReadAllLines("secrets.txt")[0]
        Secret password: 12345
STOP
Press any key to continue . . .
```

Figure 4-24. *Finding secrets in a scripting session*

This isn't good! With a small amount of code, a user can find files of interest, and then read that file's contents. Or, a malicious user could use `System.IO` types to delete numerous files on the hard drive.

We definitely do not want users to have access to the file system in this application. Therefore, we should prevent usage of anything from the `System.IO` namespace. However, that's not sufficient. Consider this line of code:

```
System.Type.GetType("System.IO.File").GetMethod(
  "ReadAllLines", new[] { typeof(string) }).Invoke(null, new[] { "secrets.txt" });
```

This code never uses any `System.IO` type or member directly. Rather, it uses the Reflection API to make a method call that will read file contents. If we were specifically looking for `System.IO` usage, this would circumvent it. Therefore, we definitely want to stop any Reflection API usage.

But there's potentially one more issue at hand. Take a look at this script snippet:

```
var person = People.Where(_ => _.Name.StartsWith("Joe"))
  .ToArray()[0]; person.Name = "Changed Name"; person.Save();
```

Do we want users to have the ability to change a person's name, along with calling `Save()`? Granted, `Save()` doesn't do anything in our example, but it's easy to imagine a real-world example where `Save()` may try to access a database and persist any changes. Maybe the user won't have that kind of authority with their account's permissions, but we can also prevent scripts that try to persist changes on a `Person` instance from being executed in the first place.

To implement all the security restrictions we just discussed, we'll create a `VerifyCompilation()` method that will traverse nodes in the syntax tree from the script and examine whether any undesirable members are being used in that code. Listing 4-11 shows how `VerifyCompilation()` is defined.

Listing 4-11. Analyzing scripts for invalid member usage

```
private static ImmutableArray<Diagnostic> VerifyCompilation(
  Compilation compilation)
{
  var diagnostics = new List<Diagnostic>();

  foreach (var tree in compilation.SyntaxTrees)
  {
    var model = compilation.GetSemanticModel(tree);
    foreach (var node in tree.GetRoot().DescendantNodes(
      _ => true))
    {
      var symbol = model.GetSymbolInfo(node).Symbol;

      if (symbol != null)
      {
        var symbolNamespace = Program.GetFullNamespace(symbol);
```

```
          if(symbol.Kind == SymbolKind.Method ||
            symbol.Kind == SymbolKind.Property ||
            symbol.Kind == SymbolKind.NamedType)
          {
            if(symbol.Kind == SymbolKind.Method)
            {
              if (symbolNamespace == typeof(Person).Namespace &&
                symbol.ContainingType.Name == nameof(Person) &&
                symbol.Name == nameof(Person.Save))
              {
                diagnostics.Add(Diagnostic.Create(
                  new DiagnosticDescriptor("SCRIPT02",
                    "Persistence Error", "Cannot save a person",
                    "Usage", DiagnosticSeverity.Error, false),
                  node.GetLocation()));
              }
            }

            if (symbolNamespace == "System.IO" ||
              symbolNamespace == "System.Reflection")
            {
              diagnostics.Add(Diagnostic.Create(
                new DiagnosticDescriptor("SCRIPT01",
                  "Inaccessable Member",
                  "Cannot allow a member from namespace {0} to be used",
                  "Usage", DiagnosticSeverity.Error, false),
                node.GetLocation(), symbolNamespace));
            }
          }
        }
      }
    }

  return diagnostics.ToImmutableArray();
}
```

We attempt to get an ISymbol reference for every node in the syntax tree. If we come across a method, property, or type, we look for two conditions. The first one is when the symbol reference is actually a call to Save() on a Person object. The other one is when the symbol exists within either the System.IO or the System.Reflection namespace. The GetFullNamespace() method gets us the namespace of the symbol; here's how GetFullNamespace() is implemented:

```
private static string GetFullNamespace(ISymbol symbol)
{
  var namespaces = new List<string>();
  var @namespace = symbol.ContainingNamespace;
  while(@namespace != null)
  {
    if(!string.IsNullOrWhiteSpace(@namespace.Name))
    {
      namespaces.Add(@namespace.Name);
    }

    @namespace = @namespace.ContainingNamespace;
  }

  namespaces.Reverse();

  return string.Join(".", namespaces);
}
```

With `VerifyImplementation()` in place, we need to change only one line of code in `MainAsync()`. Change this line:

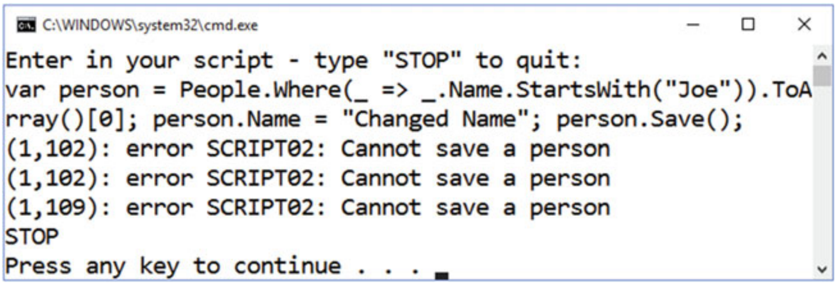
```
var diagnostics = compilation.GetDiagnostics();
```

To this:

```
var diagnostics = compilation.GetDiagnostics()
  .Union(Program.VerifyCompilation(
    compilation))
  .ToImmutableArray();
```

This code combines the diagnostic results from the compilation of the script with any custom diagnostics generated from our analysis.

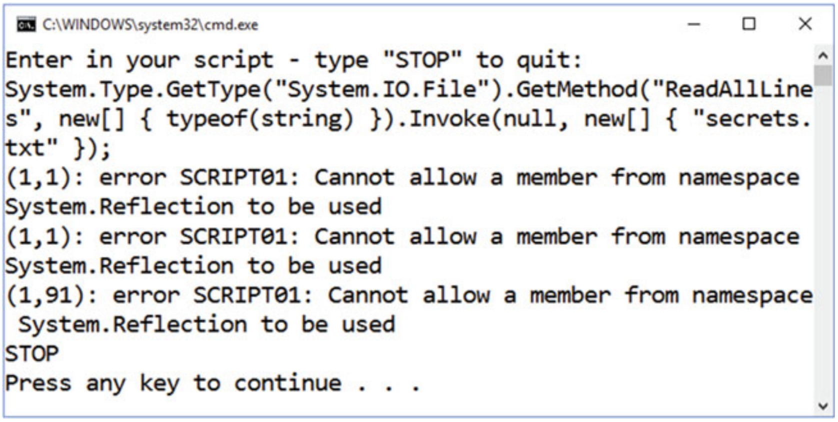
Now, let's run a couple of tests against this new implementation. First, let's run a script that tries to modify and save a `Person` instance. Figure 4-25 shows you what happens.



```
Enter in your script - type "STOP" to quit:
var person = People.Where(_ => _.Name.StartsWith("Joe")).ToA
rray()[0]; person.Name = "Changed Name"; person.Save();
(1,102): error SCRIPT02: Cannot save a person
(1,102): error SCRIPT02: Cannot save a person
(1,109): error SCRIPT02: Cannot save a person
STOP
Press any key to continue . . .
```

Figure 4-25. *Preventing persistence on a Person object*

As expected, the script isn't executed and we get an error. Figure 4-26 shows similar behavior when we try to use Reflection to read secret information in a file.



```
Enter in your script - type "STOP" to quit:
System.Type.GetType("System.IO.File").GetMethod("ReadAllLine
s", new[] { typeof(string) }).Invoke(null, new[] { "secrets.
txt" });
(1,1): error SCRIPT01: Cannot allow a member from namespace
System.Reflection to be used
(1,1): error SCRIPT01: Cannot allow a member from namespace
System.Reflection to be used
(1,91): error SCRIPT01: Cannot allow a member from namespace
 System.Reflection to be used
STOP
Press any key to continue . . .
```

Figure 4-26. *Preventing Reflection calls in a script*

These security-based techniques should be kept in mind if you let users write C# scripts in your applications. However, realize that these security measures don't cover all of the bases. Here are some other thoughts to consider:

- API Exclusion. There may be more APIs that you'll need to blacklist to prevent malicious activites from being executed. For example, we don't prevent members from `System.Reflection.Emit` from being used here. You'd definitely want to include those members because a user could write script that literally creates a new assembly on the fly.
- Restricted UIs. Our example used a simple console application. Real-world applications that users interact with are typically web-, mobile-, or desktop-based. You can create a UI that allows users to interact with the object model but in a restricted way. For example, you can provide a drop down that allows the user to query the `People` list with standardized actions, like "Starts with" for the name, and "less than" for the age. The user doesn't enter code; they interact with UI elements whose values are used to generate script. However, this may limit the ability for the user to interact with the application's object model in ways you can't anticipate.
- Use Restricted User Accounts. Ensure that the identity that is used to execute the script is highly limited. For example, you can create a user account that cannot interact with files on the machine where the script is executed. This would prevent the script from being able to use files even if malicious users figured out how to write script to get around the prevention techniques demonstrated in this section.

Trying to limit what a script can do is not a trivial endeavour. With flexibility and extensibility comes responsibilty and governance. You must ensure that exposing script execution in an application does not reveal any security holes for users to take advantage of.

Conclusion

In this chapter, you saw how you can treat C# as a scripting language with the Scripting API. This included using the Interactive window in Visual Studio and using the `CSharpScript` object to compile and execute script. Performance and security considerations with C# as a scripting language were also investigated. In the next and final chapter, you'll learn how the Compiler API is already being used by open source packages and how C# may change in the future using the Compiler API's features.