IN THIS CHAPTER

- **Communicating with COM**
- **Deploying without primary Interop assemblies**
- **Doing without `ref`**

Online Chapter 1

Helping Out with Interop

The Component Object Model, usually called COM, is a standard for the interface of software bits at the binary level. Because it is binary, it is language-neutral, which was Microsoft's goal when the company introduced COM in 1993. COM is a language-neutral way to implement objects in a lot of different environments.

COM is an umbrella term for a lot of different technologies in the Microsoft world. OLE, OLE2, ActiveX, COM+, and DCOM are all versions of the same idea — just implemented in different ways.

The problem with COM is networking. Although a thorough explanation is outside the scope of this book, it is important to understand that Microsoft's answer to broadly distributed applications in the 1990s was less than good. DCOM, or Distributed COM, was fraught with problems.

When XML web services entered the scene in the late 1990s with SOAP, Microsoft just put a wrapper around COM that translated to and from SOAP. In the background, however, Microsoft was planning a much more sophisticated messaging system for Windows. That system eventually became ASP.NET Web services, and then WCF in its latest iteration.

Applications that are sewn to the desktop don't really use DCOM, though, so they have been slow to move to services and therefore slow to move to .NET. These applications still live in COM, so you still need to interact with COM — even in this service-oriented, .NET world you live in.

What applications could be so Neanderthal? How about Microsoft Office. Yup — Microsoft Office is the biggie, and it is why C# 4.0 included a bunch of COM interoperability features, collectively called Interop Improvements.

Principally, the optional parameters discussed in the book in Book 3, Chapter 2 are implemented for COM Interop. This chapter covers three other major improvements: using Dynamic Import, deploying without primary Interop assemblies (PIAs), and skipping the ref statement.

Using Dynamic Import

Many COM methods accept and return variant types, which are represented in the primary Interop assemblies as objects. In most cases, a programmer calling these methods already knows the static type of a returned object from context, but explicitly has to perform a cast on the returned value to make use of that knowledge. These casts are so common in day-to-day development that they constitute a major nuisance.

To create a smoother experience, you can import these COM APIs in such a way that variants are represented using the type dynamic. In other words, from your point of view, COM signatures have occurrences of dynamic instead of object in them.

This means that you can easily access members directly off a returned object, or you can assign an object to a strongly typed local variable without having to cast. To illustrate, you can now say

```
excel.Cells[1, 1].Value = "Hello";
```

instead of

```
((Excel.Range)excel.Cells[1, 1]).Value2 = "Hello";
```

and

```
Excel.Range range = excel.Cells[1, 1];
```

instead of

```
Excel.Range range = (Excel.Range)excel.Cells[1, 1];
```

```
Why is this a big deal? One reason is that it simplifies the programmer's work.
   Code from Microsoft Office is tremendously difficult to read. Look at this
   code block from an Office application written for the Office.CommandBars
   commandBars = default(Office.CommandBars);
Office.CommandBar commandBar = default(Office.CommandBar);
Office.CommandBarButton runStoreReport = default(Office.CommandBarButton);
commandBars = (Microsoft.Office.Core.CommandBars)Application.CommandBars;
commandBar = commandBars.Add("VSTOAddinToolbar", Office.MsoBarPosition.
   msoBarTop, , true);
commandBar.Context = Visio.VisUIObjSets.visUIObjSetDrawing + "*";
runStoreReport =
   (Microsoft.Office.Core.CommandBarButton)commandBar.Controls.Add(
      Office.MsoControlType.msoControlButton);
runStoreReport.Tag = "Store Report";
runStoreReport.Click += VisualizeSales;
```

Here's what the code block looks like in C# 4.0 and later:

```
Office.CommandBars commandBars = Office.CommandBars;
Office.CommandBar commandBar = Office.CommandBar;
Office.CommandBarButton runStoreReport = Office.CommandBarButton;
commandBars = Application.CommandBars;
commandBar = commandBars.Add("VSTOAddinToolbar", msoBarTop, , true);
commandBar.Context = Visio.VisUIObjSets.visUIObjSetDrawing + "*";
runStoreReport =commandBar.Controls.Add(msoControlButton);
runStoreReport.Tag = "Store Report";
runStoreReport.Click += VisualizeSales;
```

It's a lot simpler to read. Keep in mind, though, that all those casts still exist — they are just handled by the compiler. Microsoft didn't redo the Office components into .NET; the company just made the compiler communicate better. The compiler still builds code that speaks to the primary Interop assemblies as they are.

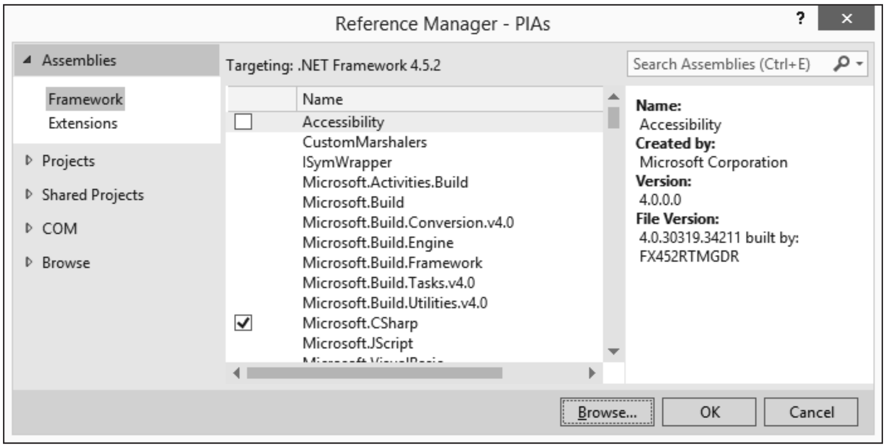
Working without Primary Interop Assemblies

Speaking of PIAs, they are handled a lot better in .NET 4.0 and later in general. PIAs are large .NET assemblies generated from COM interfaces to facilitate strongly typed interoperability. They provide great support at design time, where your experience of the Interop is as good as if the types were really defined in .NET. However, at runtime, these large assemblies can easily bloat your program and also cause versioning issues because they are distributed independently of your application.

The no-PIA feature allows you to continue to use PIAs at design time without having them around at runtime. Instead, the C# 4.0 and later compiler bakes the small part of the PIA that a program actually uses directly into its assembly. The PIA doesn't have to be loaded at runtime. To see how this works, try these steps in any version of Visual Studio 2010 and later:

1. **Create a new console application by choosing File ➪ New ➪ Project and picking C# ➪ Console application (you may need to open the Visual C#/Windows Classic Desktop folder).**
2. **Name your project** PIAs**.**
3. **After the project loads, right-click References in Solution Explorer.**
4. **Click Add Reference.**

   You see the Reference Manager dialog box, shown in Figure OC1-1. Depending on how your system is configured, you may need to perform some additional steps to gain access to the interop.



Targeting: .NET Framework 4.5.2

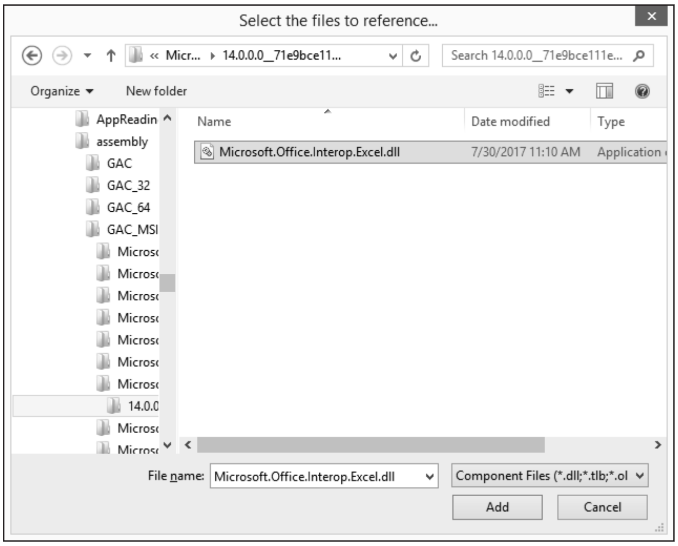
	Name
☐	Accessibility
	CustomMarshalers
	ISymWrapper
	Microsoft.Activities.Build
	Microsoft.Build
	Microsoft.Build.Conversion.v4.0
	Microsoft.Build.Engine
	Microsoft.Build.Framework
	Microsoft.Build.Tasks.v4.0
	Microsoft.Build.Utilities.v4.0
☑	Microsoft.CSharp
	Microsoft.JScript

Name:
Accessibility
Created by:
Microsoft Corporation
Version:
4.0.0.0
File Version:
4.0.30319.34211 built by: FX452RTMGDR

FIGURE OC1-1: Open the Reference Manager to add an Interop assembly reference to your project.

5. **Open the Assemblies/Framework folder.**
6. **Select Microsoft.Office.Interop.Excel, version 14, if you have it, or the latest version you have loaded.**

If you don't see the PIA listed, you need to locate it in the Global Assembly Cache located in the `C:\Windows\assembly\GAC_MSIL\Microsoft.Office.Interop.Excel` folder. You may have to drill down into a subfolder, such as `\14.0.0.0__71e9bce111e9429c`, to locate the `Microsoft.Office.Interop.Excel.dll` file shown in Figure OC1-2. Click Add to add the assembly to your project.



Microsoft.Office.Interop.Excel.dll 7/30/2017 11:10 AM Application

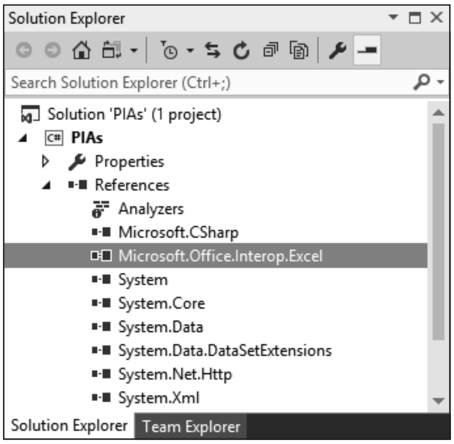
File name: Microsoft.Office.Interop.Excel.dll

Component Files (*.dll;*.tlb;*.ol

FIGURE OC1-2: Locate the `Microsoft.Office.Interop.Excel.dll` file.

7. **Click OK.**

   Visual Studio adds the new reference to your project, as shown in Figure OC1-3.



Solution 'PIAs' (1 project)
- PIAs
  - Properties
  - References
    - Analyzers
    - Microsoft.CSharp
    - Microsoft.Office.Interop.Excel
    - System
    - System.Core
    - System.Data
    - System.Data.DataSetExtensions
    - System.Net.Http
    - System.Xml

FIGURE OC1-3: Solution Explorer displays the new reference.

8. **Add** using Microsoft.Office.Interop.Excel; **to the header.**

9. **Change the Main procedure to the following (just enough to get Excel rolling):**

```
static void Main(string[] args)
{
    // Create a reference to Excel and start the application.
    Microsoft.Office.Interop.Excel.Application xl =
        new Application();

    // Make it possible to see the application.
    xl.Visible = true;

    // Wait to see the window.
    Console.WriteLine("Press Any Key When Ready...");
    Console.Read();

    // Close Excel.
    xl.Quit();
}
```

10. **Run the application.**

    You see the Excel application start. Nothing will be loaded, but you can use the application just as you would normally.

11. **Select the console window and press any key to end the application.**

    The Excel application window disappears and the application stops.

Skipping the Ref Statement

Because of a different programming model, many COM APIs contain a lot of reference parameters. Contrary to refs in C#, these are typically not meant to mutate a passed-in argument for the subsequent benefit of the caller but are simply another way of passing value parameters.

It therefore seems unreasonable that a C# programmer should have to create temporary variables for all such ref parameters and pass these by reference. So go ahead and delete them.

Instead, specifically for COM methods, the C# compiler allows you to pass arguments by values to such a method and automatically generates temporary variables to hold the passed-in values, subsequently discarding these when the call returns. In this way, the caller sees value semantics and doesn't experience any side effects, but the called method still gets a reference.

You can see this in action in the canonical optional parameter example from the book, in Book 3, Chapter 2. In the usual `SaveAs` from Microsoft Word, everything is a reference parameter.

```
object filename = "test.docx";
object missing = System.Reflection.Missing.Value;

doc.SaveAs(ref filename,
           ref missing, ref missing, ref missing,
           ref missing, ref missing, ref missing,
           ref missing, ref missing, ref missing,
           ref missing, ref missing, ref missing,
           ref missing, ref missing, ref missing);
```

In C# 4.0 and later, you can skip both the "missing" optional parameters and the ref statement.

```
object filename = "test.docx";
doc.SaveAs(filename);
```