Contents

- Interoperability
  - Interoperability Overview
  - How to: Access Office Interop Objects by Using Visual C# Features
  - How to: Use Indexed Properties in COM Interop Programming
  - How to: Use Platform Invoke to Play a Wave File
  - Walkthrough: Office Programming (C# and Visual Basic)
  - Example COM Class

Interoperability (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Interoperability enables you to preserve and take advantage of existing investments in unmanaged code. Code that runs under the control of the common language runtime (CLR) is called *managed code*, and code that runs outside the CLR is called *unmanaged code*. COM, COM+, C++ components, ActiveX components, and Microsoft Win32 API are examples of unmanaged code.

The .NET Framework enables interoperability with unmanaged code through platform invoke services, the System.Runtime.InteropServices namespace, C++ interoperability, and COM interoperability (COM interop).

In This Section

Interoperability Overview
Describes methods to interoperate between C# managed code and unmanaged code.

How to: Access Office Interop Objects by Using Visual C# Features
Describes features that are introduced in Visual C# to facilitate Office programming.

How to: Use Indexed Properties in COM Interop Programming
Describes how to use indexed properties to access COM properties that have parameters.

How to: Use Platform Invoke to Play a Wave File
Describes how to use platform invoke services to play a .wav sound file on the Windows operating system.

Walkthrough: Office Programming
Shows how to create an Excel workbook and a Word document that contains a link to the workbook.

Example COM Class
Demonstrates how to expose a C# class as a COM object.

C# Language Specification

For more information, see Basic concepts in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- Marshal.ReleaseComObject
- C# Programming Guide
- Interoperating with Unmanaged Code
- Walkthrough: Office Programming

Interoperability Overview (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

The topic describes methods to enable interoperability between C# managed code and unmanaged code.

Platform Invoke

Platform invoke is a service that enables managed code to call unmanaged functions that are implemented in dynamic link libraries (DLLs), such as those in the Microsoft Win32 API. It locates and invokes an exported function and marshals its arguments (integers, strings, arrays, structures, and so on) across the interoperation boundary as needed.

For more information, see Consuming Unmanaged DLL Functions and How to: Use Platform Invoke to Play a Wave File.

> **NOTE**
>
> The Common Language Runtime (CLR) manages access to system resources. Calling unmanaged code that is outside the CLR bypasses this security mechanism, and therefore presents a security risk. For example, unmanaged code might call resources in unmanaged code directly, bypassing CLR security mechanisms. For more information, see Security in .NET.

C++ Interop

You can use C++ interop, also known as It Just Works (IJW), to wrap a native C++ class so that it can be consumed by code that is authored in C# or another .NET Framework language. To do this, you write C++ code to wrap a native DLL or COM component. Unlike other .NET Framework languages, Visual C++ has interoperability support that enables managed and unmanaged code to be located in the same application and even in the same file. You then build the C++ code by using the **/clr** compiler switch to produce a managed assembly. Finally, you add a reference to the assembly in your C# project and use the wrapped objects just as you would use other managed classes.

Exposing COM Components to C#

You can consume a COM component from a C# project. The general steps are as follows:

1. Locate a COM component to use and register it. Use regsvr32.exe to register or un–register a COM DLL.
2. Add to the project a reference to the COM component or type library.

   When you add the reference, Visual Studio uses the Tlbimp.exe (Type Library Importer), which takes a type library as input, to output a .NET Framework interop assembly. The assembly, also named a runtime callable wrapper (RCW), contains managed classes and interfaces that wrap the COM classes and interfaces that are in the type library. Visual Studio adds to the project a reference to the generated assembly.
3. Create an instance of a class that is defined in the RCW. This, in turn, creates an instance of the COM object.
4. Use the object just as you use other managed objects. When the object is reclaimed by garbage collection, the instance of the COM object is also released from memory.

For more information, see Exposing COM Components to the .NET Framework.

Exposing C# to COM

COM clients can consume C# types that have been correctly exposed. The basic steps to expose C# types are as follows:

1. Add interop attributes in the C# project.

   You can make an assembly COM visible by modifying Visual C# project properties. For more information, see Assembly Information Dialog Box.

2. Generate a COM type library and register it for COM usage.

   You can modify Visual C# project properties to automatically register the C# assembly for COM interop. Visual Studio uses the Regasm.exe (Assembly Registration Tool), using the `/tlb` command-line switch, which takes a managed assembly as input, to generate a type library. This type library describes the `public` types in the assembly and adds registry entries so that COM clients can create managed classes.

For more information, see Exposing .NET Framework Components to COM and Example COM Class.

See also

- Improving Interop Performance
- Introduction to Interoperability between COM and .NET
- Introduction to COM Interop in Visual Basic
- Marshaling between Managed and Unmanaged Code
- Interoperating with Unmanaged Code
- C# Programming Guide

How to: Access Office Interop Objects by Using Visual C# Features (C# Programming Guide)

1/23/2019 • 12 minutes to read • Edit Online

Visual C# has features that simplify access to Office API objects. The new features include named and optional arguments, a new type called `dynamic`, and the ability to pass arguments to reference parameters in COM methods as if they were value parameters.

In this topic you will use the new features to write code that creates and displays a Microsoft Office Excel worksheet. You will then write code to add an Office Word document that contains an icon that is linked to the Excel worksheet.

To complete this walkthrough, you must have Microsoft Office Excel 2007 and Microsoft Office Word 2007, or later versions, installed on your computer.

If you are using an operating system that is older than Windows Vista, make sure that .NET Framework 2.0 is installed.

> **NOTE**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

To create a new console application

1. Start Visual Studio.
2. On the **File** menu, point to **New**, and then click **Project**. The **New Project** dialog box appears.
3. In the **Installed Templates** pane, expand **Visual C#**, and then click **Windows**.
4. Look at the top of the **New Project** dialog box to make sure that **.NET Framework 4** (or later version) is selected as a target framework.
5. In the **Templates** pane, click **Console Application**.
6. Type a name for your project in the **Name** field.
7. Click **OK**.

   The new project appears in **Solution Explorer**.

To add references

1. In **Solution Explorer**, right-click your project's name and then click **Add Reference**. The **Add Reference** dialog box appears.
2. On the **Assemblies** page, select **Microsoft.Office.Interop.Word** in the **Component Name** list, and then hold down the CTRL key and select **Microsoft.Office.Interop.Excel**. If you do not see the assemblies, you may need to ensure they are installed and displayed (see How to: Install Office Primary Interop Assemblies)
3. Click **OK**.

To add necessary using directives

1. In **Solution Explorer**, right-click the **Program.cs** file and then click **View Code**.
2. Add the following `using` directives to the top of the code file.

```
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

To create a list of bank accounts

1. Paste the following class definition into **Program.cs**, under the `Program` class.

```
public class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

2. Add the following code to the `Main` method to create a `bankAccounts` list that contains two accounts.

```
// Create a list of accounts.
var bankAccounts = new List<Account> {
    new Account {
                  ID = 345678,
                  Balance = 541.27
                },
    new Account {
                  ID = 1230221,
                  Balance = -127.44
                }
};
```

To declare a method that exports account information to Excel

1. Add the following method to the `Program` class to set up an Excel worksheet.

   Method Microsoft.Office.Interop.Excel.Workbooks.Add* has an optional parameter for specifying a particular template. Optional parameters, new in C# 4, enable you to omit the argument for that parameter if you want to use the parameter's default value. Because no argument is sent in the following code, `Add` uses the default template and creates a new workbook. The equivalent statement in earlier versions of C# requires a placeholder argument: `ExcelApp.Workbooks.Add(Type.Missing)`.

```
static void DisplayInExcel(IEnumerable<Account> accounts)
{
    var excelApp = new Excel.Application();
    // Make the object visible.
    excelApp.Visible = true;

    // Create a new, empty workbook and add it to the collection returned
    // by property Workbooks. The new workbook becomes the active workbook.
    // Add has an optional parameter for specifying a praticular template.
    // Because no argument is sent in this example, Add creates a new workbook.
    excelApp.Workbooks.Add();

    // This example uses a single workSheet. The explicit type casting is
    // removed in a later procedure.
    Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;
}
```

2. Add the following code at the end of `DisplayInExcel`. The code inserts values into the first two columns of the first row of the worksheet.

```
// Establish column headings in cells A1 and B1.
workSheet.Cells[1, "A"] = "ID Number";
workSheet.Cells[1, "B"] = "Current Balance";
```

3. Add the following code at the end of `DisplayInExcel`. The `foreach` loop puts the information from the list of accounts into the first two columns of successive rows of the worksheet.

```
var row = 1;
foreach (var acct in accounts)
{
    row++;
    workSheet.Cells[row, "A"] = acct.ID;
    workSheet.Cells[row, "B"] = acct.Balance;
}
```

4. Add the following code at the end of `DisplayInExcel` to adjust the column widths to fit the content.

```
workSheet.Columns[1].AutoFit();
workSheet.Columns[2].AutoFit();
```

   Earlier versions of C# require explicit casting for these operations because `ExcelApp.Columns[1]` returns an `Object`, and `AutoFit` is an Excel Range method. The following lines show the casting.

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

   C# 4, and later versions, converts the returned `Object` to `dynamic` automatically if the assembly is referenced by the /link compiler option or, equivalently, if the Excel **Embed Interop Types** property is set to true. True is the default value for this property.

To run the project

1. Add the following line at the end of `Main`.

```
// Display the list in an Excel spreadsheet.
DisplayInExcel(bankAccounts);
```

2. Press CTRL+F5.

   An Excel worksheet appears that contains the data from the two accounts.

To add a Word document

1. To illustrate additional ways in which C# 4, and later versions, enhances Office programming, the following code opens a Word application and creates an icon that links to the Excel worksheet.

   Paste method `CreateIconInWordDoc`, provided later in this step, into the `Program` class. `CreateIconInWordDoc` uses named and optional arguments to reduce the complexity of the method calls to Microsoft.Office.Interop.Word.Documents.Add* and PasteSpecial. These calls incorporate two other new features introduced in C# 4 that simplify calls to COM methods that have reference parameters. First, you can send arguments to the reference parameters as if they were value parameters. That is, you can send values directly, without creating a variable for each reference parameter. The compiler generates temporary variables to hold the argument values, and discards the variables when you return from the call. Second, you can omit the `ref` keyword in the argument list.

   The `Add` method has four reference parameters, all of which are optional. In C# 4, or later versions, you can omit arguments for any or all of the parameters if you want to use their default values. In Visual C# 2008 and earlier versions, an argument must be provided for each parameter, and the argument must be a variable because the parameters are reference parameters.

   The `PasteSpecial` method inserts the contents of the Clipboard. The method has seven reference parameters, all of which are optional. The following code specifies arguments for two of them: `Link`, to create a link to the source of the Clipboard contents, and `DisplayAsIcon`, to display the link as an icon. In C# 4, you can use named arguments for those two and omit the others. Although these are reference parameters, you do not have to use the `ref` keyword, or to create variables to send in as arguments. You can send the values directly. In Visual C# 2008 and earlier versions, you must send a variable argument for each reference parameter.

```
static void CreateIconInWordDoc()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four reference parameters, all of which are
    // optional. Visual C# allows you to omit arguments for them if
    // the default values are what you want.
    wordApp.Documents.Add();

    // PasteSpecial has seven reference parameters, all of which are
    // optional. This example uses named arguments to specify values
    // for two of the parameters. Although these are reference
    // parameters, you do not need to use the ref keyword, or to create
    // variables to send in as arguments. You can send the values directly.
    wordApp.Selection.PasteSpecial( Link: true, DisplayAsIcon: true);
}
```

   In Visual C# 2008 or earlier versions of the language, the following more complex code is required.

```
static void CreateIconInWordDoc2008()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four parameters, all of which are optional.
    // In Visual C# 2008 and earlier versions, an argument has to be sent
    // for every parameter. Because the parameters are reference
    // parameters of type object, you have to create an object variable
    // for the arguments that represents 'no value'.

    object useDefaultValue = Type.Missing;

    wordApp.Documents.Add(ref useDefaultValue, ref useDefaultValue,
        ref useDefaultValue, ref useDefaultValue);

    // PasteSpecial has seven reference parameters, all of which are
    // optional. In this example, only two of the parameters require
    // specified values, but in Visual C# 2008 an argument must be sent
    // for each parameter. Because the parameters are reference parameters,
    // you have to contruct variables for the arguments.
    object link = true;
    object displayAsIcon = true;

    wordApp.Selection.PasteSpecial( ref useDefaultValue,
                                    ref link,
                                    ref useDefaultValue,
                                    ref displayAsIcon,
                                    ref useDefaultValue,
                                    ref useDefaultValue,
                                    ref useDefaultValue);
}
```

2. Add the following statement at the end of `Main`.

```
// Create a Word document that contains an icon that links to
// the spreadsheet.
CreateIconInWordDoc();
```

3. Add the following statement at the end of `DisplayInExcel`. The `Copy` method adds the worksheet to the Clipboard.

```
// Put the spreadsheet contents on the clipboard. The Copy method has one
// optional parameter for specifying a destination. Because no argument
// is sent, the destination is the Clipboard.
workSheet.Range["A1:B3"].Copy();
```

4. Press CTRL+F5.

   A Word document appears that contains an icon. Double-click the icon to bring the worksheet to the foreground.

To set the Embed Interop Types property

1. Additional enhancements are possible when you call a COM type that does not require a primary interop assembly (PIA) at run time. Removing the dependency on PIAs results in version independence and easier deployment. For more information about the advantages of programming without PIAs, see Walkthrough: Embedding Types from Managed Assemblies.

   In addition, programming is easier because the types that are required and returned by COM methods can

be represented by using the type `dynamic` instead of `Object`. Variables that have type `dynamic` are not evaluated until run time, which eliminates the need for explicit casting. For more information, see Using Type dynamic.

In C# 4, embedding type information instead of using PIAs is default behavior. Because of that default, several of the previous examples are simplified because explicit casting is not required. For example, the declaration of `worksheet` in `DisplayInExcel` is written as `Excel._Worksheet workSheet = excelApp.ActiveSheet` rather than `Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet`. The calls to `AutoFit` in the same method also would require explicit casting without the default, because `ExcelApp.Columns[1]` returns an `Object`, and `AutoFit` is an Excel method. The following code shows the casting.

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

2. To change the default and use PIAs instead of embedding type information, expand the **References** node in **Solution Explorer** and then select **Microsoft.Office.Interop.Excel** or **Microsoft.Office.Interop.Word**.

3. If you cannot see the **Properties** window, press **F4**.

4. Find **Embed Interop Types** in the list of properties, and change its value to **False**. Equivalently, you can compile by using the /reference compiler option instead of /link at a command prompt.

To add additional formatting to the table

1. Replace the two calls to `AutoFit` in `DisplayInExcel` with the following statement.

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

The AutoFormat method has seven value parameters, all of which are optional. Named and optional arguments enable you to provide arguments for none, some, or all of them. In the previous statement, an argument is supplied for only one of the parameters, `Format`. Because `Format` is the first parameter in the parameter list, you do not have to provide the parameter name. However, the statement might be easier to understand if the parameter name is included, as is shown in the following code.

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(Format:
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

2. Press CTRL+F5 to see the result. Other formats are listed in the XlRangeAutoFormat enumeration.

3. Compare the statement in step 1 with the following code, which shows the arguments that are required in Visual C# 2008 or earlier versions.

```
// The AutoFormat method has seven optional value parameters. The
// following call specifies a value for the first parameter, and uses
// the default values for the other six.

// Call to AutoFormat in Visual C# 2008. This code is not part of the
// current solution.
excelApp.get_Range("A1", "B4").AutoFormat(Excel.XlRangeAutoFormat.xlRangeAutoFormatTable3,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing,
    Type.Missing);
```

Example

The following code shows the complete example.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;


namespace OfficeProgramminWalkthruComplete
{
    class Walkthrough
    {
        static void Main(string[] args)
        {
            // Create a list of accounts.
            var bankAccounts = new List<Account>
            {
                new Account {
                              ID = 345678,
                              Balance = 541.27
                            },
                new Account {
                              ID = 1230221,
                              Balance = -127.44
                            }
            };

            // Display the list in an Excel spreadsheet.
            DisplayInExcel(bankAccounts);

            // Create a Word document that contains an icon that links to
            // the spreadsheet.
            CreateIconInWordDoc();
        }

        static void DisplayInExcel(IEnumerable<Account> accounts)
        {
            var excelApp = new Excel.Application();
            // Make the object visible.
            excelApp.Visible = true;

            // Create a new, empty workbook and add it to the collection returned
            // by property Workbooks. The new workbook becomes the active workbook.
            // Add has an optional parameter for specifying a praticular template.
            // Because no argument is sent in this example, Add creates a new workbook.
            excelApp.Workbooks.Add();

            // This example uses a single workSheet.
            Excel._Worksheet workSheet = excelApp.ActiveSheet;

            // Earlier versions of C# require explicit casting.
            //Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;

            // Establish column headings in cells A1 and B1.
            workSheet.Cells[1, "A"] = "ID Number";
            workSheet.Cells[1, "B"] = "Current Balance";

            var row = 1;
            foreach (var acct in accounts)
            {
                row++;
                workSheet.Cells[row, "A"] = acct.ID;
                workSheet.Cells[row, "B"] = acct.Balance;
            }
```

```
            workSheet.Columns[1].AutoFit();
            workSheet.Columns[2].AutoFit();

            // Call to AutoFormat in Visual C#. This statement replaces the
            // two calls to AutoFit.
            workSheet.Range["A1", "B3"].AutoFormat(
                Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);

            // Put the spreadsheet contents on the clipboard. The Copy method has one
            // optional parameter for specifying a destination. Because no argument
            // is sent, the destination is the Clipboard.
            workSheet.Range["A1:B3"].Copy();
        }

        static void CreateIconInWordDoc()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;

            // The Add method has four reference parameters, all of which are
            // optional. Visual C# allows you to omit arguments for them if
            // the default values are what you want.
            wordApp.Documents.Add();

            // PasteSpecial has seven reference parameters, all of which are
            // optional. This example uses named arguments to specify values
            // for two of the parameters. Although these are reference
            // parameters, you do not need to use the ref keyword, or to create
            // variables to send in as arguments. You can send the values directly.
            wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
        }
    }

    public class Account
    {
        public int ID { get; set; }
        public double Balance { get; set; }
    }
}
```

See also

- Type.Missing
- dynamic
- Using Type dynamic
- Named and Optional Arguments
- How to: Use Named and Optional Arguments in Office Programming

How to: Use Indexed Properties in COM Interop Programming (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Indexed properties improve the way in which COM properties that have parameters are consumed in C# programming. Indexed properties work together with other features in Visual C#, such as named and optional arguments, a new type (dynamic), and embedded type information, to enhance Microsoft Office programming.

In earlier versions of C#, methods are accessible as properties only if the `get` method has no parameters and the `set` method has one and only one value parameter. However, not all COM properties meet those restrictions. For example, the Excel Range property has a `get` accessor that requires a parameter for the name of the range. In the past, because you could not access the `Range` property directly, you had to use the `get_Range` method instead, as shown in the following example.

```
// Visual C# 2008 and earlier.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
```

Indexed properties enable you to write the following instead:

```
// Visual C# 2010.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.Range["A1"];
```

NOTE

The previous example also uses the optional arguments feature, which enables you to omit `Type.Missing`.

Similarly to set the value of the `Value` property of a Range object in Visual C# 2008 and earlier, two arguments are required. One supplies an argument for an optional parameter that specifies the type of the range value. The other supplies the value for the `Value` property. The following examples illustrate these techniques. Both set the value of the A1 cell to `Name`.

```
// Visual C# 2008.
targetRange.set_Value(Type.Missing, "Name");
// Or
targetRange.Value2 = "Name";
```

Indexed properties enable you to write the following code instead.

```
// Visual C# 2010.
targetRange.Value = "Name";
```

You cannot create indexed properties of your own. The feature only supports consumption of existing indexed properties.

Example

The following code shows a complete example. For more information about how to set up a project that accesses the Office API, see How to: Access Office Interop Objects by Using Visual C# Features.

```
// You must add a reference to Microsoft.Office.Interop.Excel to run
// this example.
using System;
using Excel = Microsoft.Office.Interop.Excel;

namespace IndexedProperties
{
    class Program
    {
        static void Main(string[] args)
        {
            CSharp2010();
            //CSharp2008();
        }

        static void CSharp2010()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add();
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.Range["A1"];
            targetRange.Value = "Name";
        }

        static void CSharp2008()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add(Type.Missing);
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
            targetRange.set_Value(Type.Missing, "Name");
            // Or
            //targetRange.Value2 = "Name";

        }
    }
}
```

See also

- Named and Optional Arguments
- dynamic
- Using Type dynamic
- How to: Use Named and Optional Arguments in Office Programming
- How to: Access Office Interop Objects by Using Visual C# Features
- Walkthrough: Office Programming

How to: Use Platform Invoke to Play a Wave File (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

The following C# code example illustrates how to use platform invoke services to play a wave sound file on the Windows operating system.

Example

This example code uses `DllImport` to import `winmm.dll`'s `PlaySound` method entry point as `Form1 PlaySound()`. The example has a simple Windows Form with a button. Clicking the button opens a standard windows OpenFileDialog dialog box so that you can open a file to play. When a wave file is selected, it is played by using the `PlaySound()` method of the `winmm.dll` library. For more information about this method, see Using the PlaySound function with Waveform-Audio Files. Browse and select a file that has a .wav extension, and then click **Open** to play the wave file by using platform invoke. A text box shows the full path of the file selected.

The **Open Files** dialog box is filtered to show only files that have a .wav extension through the filter settings:

```
dialog1.Filter = "Wav Files (*.wav)|*.wav";
```

```
using System.Windows.Forms;
using System.Runtime.InteropServices;

namespace WinSound
{
    public partial class Form1 : Form
    {
        private TextBox textBox1;
        private Button button1;

        public Form1()  //constructor
        {
            InitializeComponent();
        }

        [System.Runtime.InteropServices.DllImport("winmm.DLL", EntryPoint = "PlaySound", SetLastError = true,
CharSet = CharSet.Unicode, ThrowOnUnmappableChar = true)]
        private static extern bool PlaySound(string szSound, System.IntPtr hMod, PlaySoundFlags flags);

        [System.Flags]
        public enum PlaySoundFlags : int
        {
            SND_SYNC = 0x0000,
            SND_ASYNC = 0x0001,
            SND_NODEFAULT = 0x0002,
            SND_LOOP = 0x0008,
            SND_NOSTOP = 0x0010,
            SND_NOWAIT = 0x00002000,
            SND_FILENAME = 0x00020000,
            SND_RESOURCE = 0x00040004
        }

        private void button1_Click (object sender, System.EventArgs e)
        {
            OpenFileDialog dialog1 = new OpenFileDialog();

            dialog1.Title = "Browse to find sound file to play";
            dialog1.InitialDirectory = @"c:\";
            dialog1.Filter = "Wav Files (*.wav)|*.wav";
            dialog1.FilterIndex = 2;
            dialog1.RestoreDirectory = true;

            if(dialog1.ShowDialog() == DialogResult.OK)
            {
                textBox1.Text = dialog1.FileName;
                PlaySound (dialog1.FileName, new System.IntPtr(), PlaySoundFlags.SND_SYNC);
            }
        }
    }
}
```

Compiling the Code

To compile the code

1. Create a new C# Windows Application project in Visual Studio and name it **WinSound**.
2. Copy the code above, and paste it over the contents of the `Form1.cs` file.
3. Copy the following code, and paste it in the `Form1.Designer.cs` file, in the `InitializeComponent()` method, after any existing code.

```
this.button1 = new System.Windows.Forms.Button();
this.textBox1 = new System.Windows.Forms.TextBox();
this.SuspendLayout();
//
// button1
//
this.button1.Location = new System.Drawing.Point(192, 40);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(88, 24);
this.button1.TabIndex = 0;
this.button1.Text = "Browse";
this.button1.Click += new System.EventHandler(this.button1_Click);
//
// textBox1
//
this.textBox1.Location = new System.Drawing.Point(8, 40);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(168, 20);
this.textBox1.TabIndex = 1;
this.textBox1.Text = "FIle path";
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(5, 13);
this.ClientSize = new System.Drawing.Size(292, 266);
this.Controls.Add(this.textBox1);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "Platform Invoke WinSound C#";
this.ResumeLayout(false);
this.PerformLayout();
```

4. Compile and run the code.

.NET Framework Security

For more information, see Security in .NET.

See also

- C# Programming Guide
- Interoperability Overview
- A Closer Look at Platform Invoke
- Marshaling Data with Platform Invoke

Walkthrough: Office Programming (C# and Visual Basic)

2/13/2019 • 11 minutes to read • Edit Online

Visual Studio offers features in C# and Visual Basic that improve Microsoft Office programming. Helpful C# features include named and optional arguments and return values of type `dynamic`. In COM programming, you can omit the `ref` keyword and gain access to indexed properties. Features in Visual Basic include auto-implemented properties, statements in lambda expressions, and collection initializers.

Both languages enable embedding of type information, which allows deployment of assemblies that interact with COM components without deploying primary interop assemblies (PIAs) to the user's computer. For more information, see Walkthrough: Embedding Types from Managed Assemblies.

This walkthrough demonstrates these features in the context of Office programming, but many of these features are also useful in general programming. In the walkthrough, you use an Excel Add-in application to create an Excel workbook. Next, you create a Word document that contains a link to the workbook. Finally, you see how to enable and disable the PIA dependency.

Prerequisites

You must have Microsoft Office Excel and Microsoft Office Word installed on your computer to complete this walkthrough.

If you are using an operating system that is older than Windows Vista, make sure that .NET Framework 2.0 is installed.

> **NOTE**
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

To set up an Excel Add-in application

1. Start Visual Studio.
2. On the **File** menu, point to **New**, and then click **Project**.
3. In the **Installed Templates** pane, expand **Visual Basic** or **Visual C#**, expand **Office**, and then click the version year of the Office product.
4. In the **Templates** pane, click **Excel <version> Add-in**.
5. Look at the top of the **Templates** pane to make sure that **.NET Framework 4**, or a later version, appears in the **Target Framework** box.
6. Type a name for your project in the **Name** box, if you want to.
7. Click **OK**.
8. The new project appears in **Solution Explorer**.

To add references

1. In **Solution Explorer**, right-click your project's name and then click **Add Reference**. The **Add Reference** dialog box appears.

2. On the **Assemblies** tab, select **Microsoft.Office.Interop.Excel**, version `<version>.0.0.0` (for a key to the Office product version numbers, see Microsoft Versions), in the **Component Name** list, and then hold down the CTRL key and select **Microsoft.Office.Interop.Word**, `version <version>.0.0.0`. If you do not see the assemblies, you may need to ensure they are installed and displayed (see How to: Install Office Primary Interop Assemblies).

3. Click **OK**.

To add necessary Imports statements or using directives

1. In **Solution Explorer**, right-click the **ThisAddIn.vb** or **ThisAddIn.cs** file and then click **View Code**.

2. Add the following `Imports` statements (Visual Basic) or `using` directives (C#) to the top of the code file if they are not already present.

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

```
Imports Microsoft.Office.Interop
```

To create a list of bank accounts

1. In **Solution Explorer**, right-click your project's name, click **Add**, and then click **Class**. Name the class Account.vb if you are using Visual Basic or Account.cs if you are using C#. Click **Add**.

2. Replace the definition of the `Account` class with the following code. The class definitions use *auto-implemented properties*. For more information, see Auto-Implemented Properties.

```
class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

```
Public Class Account
    Property ID As Integer = -1
    Property Balance As Double
End Class
```

3. To create a `bankAccounts` list that contains two accounts, add the following code to the `ThisAddIn_Startup` method in *ThisAddIn.vb* or *ThisAddIn.cs*. The list declarations use *collection initializers*. For more information, see Collection Initializers.

```
var bankAccounts = new List<Account>
{
    new Account
    {
        ID = 345,
        Balance = 541.27
    },
    new Account
    {
        ID = 123,
        Balance = -127.44
    }
};
```

```
Dim bankAccounts As New List(Of Account) From {
    New Account With {
                          .ID = 345,
                          .Balance = 541.27
                      },
    New Account With {
                          .ID = 123,
                          .Balance = -127.44
                      }
    }
```

To export data to Excel

1. In the same file, add the following method to the `ThisAddIn` class. The method sets up an Excel workbook and exports data to it.

```
void DisplayInExcel(IEnumerable<Account> accounts,
           Action<Account, Excel.Range> DisplayFunc)
{
    var excelApp = this.Application;
    // Add a new Excel workbook.
    excelApp.Workbooks.Add();
    excelApp.Visible = true;
    excelApp.Range["A1"].Value = "ID";
    excelApp.Range["B1"].Value = "Balance";
    excelApp.Range["A2"].Select();

    foreach (var ac in accounts)
    {
        DisplayFunc(ac, excelApp.ActiveCell);
        excelApp.ActiveCell.Offset[1, 0].Select();
    }
    // Copy the results to the Clipboard.
    excelApp.Range["A1:B3"].Copy();
}
```

```
Sub DisplayInExcel(ByVal accounts As IEnumerable(Of Account),
                   ByVal DisplayAction As Action(Of Account, Excel.Range))

    With Me.Application
        ' Add a new Excel workbook.
        .Workbooks.Add()
        .Visible = True
        .Range("A1").Value = "ID"
        .Range("B1").Value = "Balance"
        .Range("A2").Select()

        For Each ac In accounts
            DisplayAction(ac, .ActiveCell)
            .ActiveCell.Offset(1, 0).Select()
        Next

        ' Copy the results to the Clipboard.
        .Range("A1:B3").Copy()
    End With
End Sub
```

Two new C# features are used in this method. Both of these features already exist in Visual Basic.

- Method Add has an *optional parameter* for specifying a particular template. Optional parameters, new in C# 4, enable you to omit the argument for that parameter if you want to use the parameter's default value. Because no argument is sent in the previous example, `Add` uses the default template and creates a new workbook. The equivalent statement in earlier versions of C# requires a placeholder argument: `excelApp.Workbooks.Add(Type.Missing)`.

  For more information, see Named and Optional Arguments.

- The `Range` and `Offset` properties of the Range object use the *indexed properties* feature. This feature enables you to consume these properties from COM types by using the following typical C# syntax. Indexed properties also enable you to use the `Value` property of the `Range` object, eliminating the need to use the `Value2` property. The `Value` property is indexed, but the index is optional. Optional arguments and indexed properties work together in the following example.

  ```
  // Visual C# 2010 provides indexed properties for COM programming.
  excelApp.Range["A1"].Value = "ID";
  excelApp.ActiveCell.Offset[1, 0].Select();
  ```

  In earlier versions of the language, the following special syntax is required.

  ```
  // In Visual C# 2008, you cannot access the Range, Offset, and Value
  // properties directly.
  excelApp.get_Range("A1").Value2 = "ID";
  excelApp.ActiveCell.get_Offset(1, 0).Select();
  ```

  You cannot create indexed properties of your own. The feature only supports consumption of existing indexed properties.

  For more information, see How to: Use Indexed Properties in COM Interop Programming.

2. Add the following code at the end of `DisplayInExcel` to adjust the column widths to fit the content.

```
excelApp.Columns[1].AutoFit();
excelApp.Columns[2].AutoFit();
```

```
' Add the following two lines at the end of the With statement.
.Columns(1).AutoFit()
.Columns(2).AutoFit()
```

These additions demonstrate another feature in C#: treating `Object` values returned from COM hosts such as Office as if they have type dynamic. This happens automatically when **Embed Interop Types** is set to its default value, `True`, or, equivalently, when the assembly is referenced by the /link compiler option. Type `dynamic` allows late binding, already available in Visual Basic, and avoids the explicit casting required in Visual C# 2008 and earlier versions of the language.

For example, `excelApp.Columns[1]` returns an `Object`, and `AutoFit` is an Excel Range method. Without `dynamic`, you must cast the object returned by `excelApp.Columns[1]` as an instance of `Range` before calling method `AutoFit`.

```
// Casting is required in Visual C# 2008.
((Excel.Range)excelApp.Columns[1]).AutoFit();

// Casting is not required in Visual C# 2010.
excelApp.Columns[1].AutoFit();
```

For more information about embedding interop types, see procedures "To find the PIA reference" and "To restore the PIA dependency" later in this topic. For more information about `dynamic`, see dynamic or Using Type dynamic.

To invoke DisplayInExcel

1. Add the following code at the end of the `ThisAddIn_StartUp` method. The call to `DisplayInExcel` contains two arguments. The first argument is the name of the list of accounts to be processed. The second argument is a multiline lambda expression that defines how the data is to be processed. The `ID` and `balance` values for each account are displayed in adjacent cells, and the row is displayed in red if the balance is less than zero. For more information, see Lambda Expressions.

```
DisplayInExcel(bankAccounts, (account, cell) =>
// This multiline lambda expression sets custom processing rules
// for the bankAccounts.
{
    cell.Value = account.ID;
    cell.Offset[0, 1].Value = account.Balance;
    if (account.Balance < 0)
    {
        cell.Interior.Color = 255;
        cell.Offset[0, 1].Interior.Color = 255;
    }
});
```

```
DisplayInExcel(bankAccounts,
       Sub(account, cell)
           ' This multiline lambda expression sets custom
           ' processing rules for the bankAccounts.
           cell.Value = account.ID
           cell.Offset(0, 1).Value = account.Balance

           If account.Balance < 0 Then
               cell.Interior.Color = RGB(255, 0, 0)
               cell.Offset(0, 1).Interior.Color = RGB(255, 0, 0)
           End If
       End Sub)
```

2. To run the program, press F5. An Excel worksheet appears that contains the data from the accounts.

To add a Word document

1. Add the following code at the end of the `ThisAddIn_StartUp` method to create a Word document that contains a link to the Excel workbook.

```
var wordApp = new Word.Application();
wordApp.Visible = true;
wordApp.Documents.Add();
wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
```

```
Dim wordApp As New Word.Application
wordApp.Visible = True
wordApp.Documents.Add()
wordApp.Selection.PasteSpecial(Link:=True, DisplayAsIcon:=True)
```

   This code demonstrates several of the new features in C#: the ability to omit the `ref` keyword in COM programming, named arguments, and optional arguments. These features already exist in Visual Basic. The PasteSpecial method has seven parameters, all of which are defined as optional reference parameters. Named and optional arguments enable you to designate the parameters you want to access by name and to send arguments to only those parameters. In this example, arguments are sent to indicate that a link to the workbook on the Clipboard should be created (parameter `Link`) and that the link is to be displayed in the Word document as an icon (parameter `DisplayAsIcon`). Visual C# also enables you to omit the `ref` keyword for these arguments.

To run the application

1. Press F5 to run the application. Excel starts and displays a table that contains the information from the two accounts in `bankAccounts`. Then a Word document appears that contains a link to the Excel table.

To clean up the completed project

1. In Visual Studio, click **Clean Solution** on the **Build** menu. Otherwise, the add-in will run every time that you open Excel on your computer.

To find the PIA reference

1. Run the application again, but do not click **Clean Solution**.

2. Select the **Start**. Locate **Microsoft Visual Studio <version>** and open a developer command prompt.

3. Type `ildasm` in the Developer Command Prompt for Visual Studio window, and then press ENTER. The IL DASM window appears.

4. On the **File** menu in the IL DASM window, select **File** > **Open**. Double-click **Visual Studio <version>**, and then double-click **Projects**. Open the folder for your project, and look in the bin/Debug folder for *your project name*.dll. Double-click *your project name*.dll. A new window displays your project's attributes, in addition to references to other modules and assemblies. Note that namespaces `Microsoft.Office.Interop.Excel` and `Microsoft.Office.Interop.Word` are included in the assembly. By default in Visual Studio, the compiler imports the types you need from a referenced PIA into your assembly.

   For more information, see How to: View Assembly Contents.

5. Double-click the **MANIFEST** icon. A window appears that contains a list of assemblies that contain items referenced by the project. `Microsoft.Office.Interop.Excel` and `Microsoft.Office.Interop.Word` are not included in the list. Because the types your project needs have been imported into your assembly, references to a PIA are not required. This makes deployment easier. The PIAs do not have to be present on the user's computer, and because an application does not require deployment of a specific version of a PIA,

applications can be designed to work with multiple versions of Office, provided that the necessary APIs exist in all versions.

Because deployment of PIAs is no longer necessary, you can create an application in advanced scenarios that works with multiple versions of Office, including earlier versions. However, this works only if your code does not use any APIs that are not available in the version of Office you are working with. It is not always clear whether a particular API was available in an earlier version, and for that reason working with earlier versions of Office is not recommended.

> **NOTE**
> Office did not publish PIAs before Office 2003. Therefore, the only way to generate an interop assembly for Office 2002 or earlier versions is by importing the COM reference.

6. Close the manifest window and the assembly window.

To restore the PIA dependency

1. In **Solution Explorer**, click the **Show All Files** button. Expand the **References** folder and select **Microsoft.Office.Interop.Excel**. Press F4 to display the **Properties** window.
2. In the **Properties** window, change the **Embed Interop Types** property from **True** to **False**.
3. Repeat steps 1 and 2 in this procedure for `Microsoft.Office.Interop.Word`.
4. In C#, comment out the two calls to `Autofit` at the end of the `DisplayInExcel` method.
5. Press F5 to verify that the project still runs correctly.
6. Repeat steps 1-3 from the previous procedure to open the assembly window. Notice that `Microsoft.Office.Interop.Word` and `Microsoft.Office.Interop.Excel` are no longer in the list of embedded assemblies.
7. Double-click the **MANIFEST** icon and scroll through the list of referenced assemblies. Both `Microsoft.Office.Interop.Word` and `Microsoft.Office.Interop.Excel` are in the list. Because the application references the Excel and Word PIAs, and the **Embed Interop Types** property is set to **False**, both assemblies must exist on the end user's computer.
8. In Visual Studio, click **Clean Solution** on the **Build** menu to clean up the completed project.

See also

- Auto-Implemented Properties (Visual Basic)
- Auto-Implemented Properties (C#)
- Collection Initializers
- Object and Collection Initializers
- Optional Parameters
- Passing Arguments by Position and by Name
- Named and Optional Arguments
- Early and Late Binding
- dynamic
- Using Type dynamic
- Lambda Expressions (Visual Basic)
- Lambda Expressions (C#)
- How to: Use Indexed Properties in COM Interop Programming

- Walkthrough: Embedding Type Information from Microsoft Office Assemblies in Visual Studio (C#)
- Walkthrough: Embedding Types from Managed Assemblies
- Walkthrough: Creating Your First VSTO Add-in for Excel
- COM Interop
- Interoperability

Example COM Class (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

The following is an example of a class that you would expose as a COM object. After this code has been placed in a .cs file and added to your project, set the **Register for COM Interop** property to **True**. For more information, see How to: Register a Component for COM Interop.

Exposing Visual C# objects to COM requires declaring a class interface, an events interface if it is required, and the class itself. Class members must follow these rules to be visible to COM:

- The class must be public.
- Properties, methods, and events must be public.
- Properties and methods must be declared on the class interface.
- Events must be declared in the event interface.

Other public members in the class that are not declared in these interfaces will not be visible to COM, but they will be visible to other .NET Framework objects.

To expose properties and methods to COM, you must declare them on the class interface and mark them with a `DispId` attribute, and implement them in the class. The order in which the members are declared in the interface is the order used for the COM vtable.

To expose events from your class, you must declare them on the events interface and mark them with a `DispId` attribute. The class should not implement this interface.

The class implements the class interface; it can implement more than one interface, but the first implementation will be the default class interface. Implement the methods and properties exposed to COM here. They must be marked public and must match the declarations in the class interface. Also, declare the events raised by the class here. They must be marked public and must match the declarations in the events interface.

Example

```
using System.Runtime.InteropServices;

namespace project_name
{
    [Guid("EAA4976A-45C3-4BC5-BC0B-E474F4C3C83F")]
    public interface ComClass1Interface
    {
    }

    [Guid("7BD20046-DF8C-44A6-8F6B-687FAA26FA71"),
        InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
    public interface ComClass1Events
    {
    }

    [Guid("0D53A3E8-E51A-49C7-944E-E72A2064F938"),
        ClassInterface(ClassInterfaceType.None),
        ComSourceInterfaces(typeof(ComClass1Events))]
    public class ComClass1 : ComClass1Interface
    {
    }
}
```

See also

- C# Programming Guide
- Interoperability
- Build Page, Project Designer (C#)