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

Visual Studio 2015

Visual C# 2010 introduces new 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 Visual Studio 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.

C#

```
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.

C#

```
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.

C#

```
// 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 Add has an optional parameter for specifying a particular template. Optional parameters, new in Visual C# 2010, 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)`.

   C#

   ```
   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.

   C#

   ```
   // 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.

   C#

   ```
   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.

   C#

   ```
   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.

   C#

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

   Visual C# 2010, 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`.

   C#

   ```
   // 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 Visual C# 2010, 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 Add and PasteSpecial. These calls incorporate two other new features introduced in Visual C# 2010 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 Visual C# 2010, 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 Visual C# 2010, 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.

C#

```
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# 2010 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.

C#

```
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`.

C#

```
// 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.

C#

```
// 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 (C# and Visual Basic).

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 (C# Programming Guide).

In Visual C# 2010, 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.

C#

```
((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.

C#

```
// 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.

C#

```
// 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.

C#

```
    // 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.xlRangeAutoFormatTa
        Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing,
        Type.Missing);
```

Example

The following code shows the complete example.

C#

```
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# 2010. 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# 2010 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 (C# Reference)
Using Type dynamic (C# Programming Guide)
Named and Optional Arguments (C# Programming Guide)
How to: Use Named and Optional Arguments in Office Programming (C# Programming Guide)