* 1. 

Hands-On Lab

Microsoft Office Programmability in C# and Visual Basic

* 1. Lab version: 1.0.0
  2. Last updated: 1/4/2011
  3. 

Contents

[Overview 3](#_Toc281930405)

[Exercise 1: Creating a Collection of Business Entities and Inserting their Values into an Excel Worksheet 5](#_Toc281930406)

[Task 1 – Creating a New Console Application and Reference the Microsoft Office Interop Assemblies 5](#_Toc281930407)

[Task 2 – Creating a Collection of Business Entities 6](#_Toc281930408)

[Task 3 – Creating an Excel Workbook and Populate it with Data from the Business Entities 10](#_Toc281930409)

[Exercise 1: Verification 13](#_Toc281930410)

[Exercise 2: Embedding an Excel Worksheet in Microsoft Word 14](#_Toc281930411)

[Task 1 – Embedding your Excel Worksheet into a New Microsoft Word Document 15](#_Toc281930412)

[Exercise 2: Verification 17](#_Toc281930413)

[Exercise 3: Removing the PIA Dependence from the Application 19](#_Toc281930414)

[Task 1 – Checking that PIA dependencies are removed by default 19](#_Toc281930415)

[Summary 20](#_Toc281930416)

Overview

* 1. Microsoft Office provides organizations an environment for rapidly creating business applications in an environment that is very familiar to end users. Building Office Business Applications present a large number of opportunities for organizations to take advantage of, including tying into Customer Relationship Management (CRM) systems, fronting data extracted from line of business applications, hosting business intelligence reports, and a wealth of other possibilities.
  2. While Office development provides a great number of value opportunities for organizations, previously it also presented a number of challenges for developing and deploying those applications. Earlier versions of managed languages also contributed to the friction involved in creating applications using Office.
  3. Calling numerous Office API methods was often painful due to the large number of parameters involved, many of which had no use. Visual Basic’s optional parameters feature made this easier, but developers in C# had to suffer through writing significant amounts of useless initializers or null statements to fill out required method signature parameters. If you were writing in Visual Basic, you had no access to an equivalent of C# 3.0’s lambda feature to supply inline methods when invoking delegates.
  4. In this lab, you will see how new features in Visual Studio 2010, C# 4.0, and Visual Basic 10 eliminate the issues in the previous paragraphs. Additionally, you will see a number of other powerful features, which speed other elements of Office development.

# Objectives

* 1. In this Hands-On Lab, you will learn how to:
  + Learn how new language features speed creation of business entities
  + Discover new language features that make it easier to have business entities interact with Excel
  + Utilize new language features which enable quick interaction between Office applications
  + See how the new build process allows easier deployment of Office applications to end users

# System Requirements

* 1. You must have the following items to complete this lab:
  + Microsoft Visual Studio 2010
  + .Net Framework 4
  + Microsoft Excel 2007
  + Microsoft Word 2007

# Setup

* 1. All the requisites for this lab are verified using the **Configuration Wizard**. To make sure that everything is correctly configured, follow these steps.
  2. Run the **Configuration Wizard** for the Training Kit if you have not done it previously. To do this, browse to **Source\Setup** folder of this lab, and double-click the **Dependencies.dep** file. Install any pre-requisites that are missing (rescanning if necessary) and complete the wizard.
     1. **Note:** The Configuration Wizard is used for checking dependencies and setting up the environment. If the Configuration Wizard is not installed on your machine, you must install it running the DependencyChecker.msi file located on the %VS2010TKInstallationFolder%\Assets\DependencyChecker folder (e.g. by default the Training Kit is installed under C:\VS2010TrainingKit).
     2. For convenience, much of the code you will be managing along this lab is available as Visual Studio code snippets. The **Dependencies.dep** file launches the Visual Studio installer file that installs the code snippets.

# Exercises

* 1. This Hands-On Lab is comprised of the following exercises:
  2. Creating a collection of account classes and display the account information in Microsoft Excel
  3. Embed the Microsoft Excel worksheet in a Microsoft Word document
  4. Alter the VS project files to enable the program to be deployed without PIA
  5. Estimated time to complete this lab: **60 minutes**.

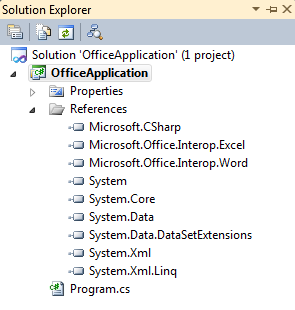
# Next Step

Exercise 1: Creating a Collection of Business Entities and Inserting their Values into an Excel Worksheet

Exercise 1: Creating a Collection of Business Entities and Inserting their Values into an Excel Worksheet

* 1. Business entities are a commonly used pattern to carry data through an application or system. Business entities might reflect a bank account, with a balance and account holder, or they might reflect a shopping cart with a number of items in it, each of those items being its own business entity.
  2. In this exercise, you will learn how to create a collection of business entities and write the values from those entities to an Excel worksheet. Along the way, you will be introduced to new language features in C# 4.0 and Visual Basic 10 that greatly speed the creation of business entities.
  3. **Note:** To verify that each step is correctly performed, it is recommended to build the solution at the end of each task.

Task 1 – Creating a New Console Application and Reference the Microsoft Office Interop Assemblies

* 1. In this task, you will create a new console application and add in the assemblies needed to begin Office development.
  2. Open Microsoft Visual Studio 2010 from **Start** | **All Programs** | **Microsoft Visual Studio 2010** | **Microsoft Visual Studio 2010**.
  3. On the Microsoft Visual Studio Start Page, click the **New Project** icon.
     1. **Note:** This lab will be presented in both C# and Visual Basic. Feel free to use which ever language you are more comfortable with. Along the way new features of each language and Visual Studo 2010 will be pointed out.
  4. In either the C# or Visual Basic projects Templates list, select **Windows | Console Application**.
  5. In the **Name** field, enter the name “OfficeApplication”.
  6. In the upper right hand corner, ensure that version 4.0 of the .NET Framework is selected.
  7. Accept the default location and Solution Name. Click the **OK** button. The solution will be created with a single console application project.
  8. Right-click on the **OfficeApplication** project node and select **Add Reference…**.
  9. Under the **.NET** tab in the **Add Reference** dialog box, select **Microsoft.Office.Interop.Excel**. Hold down the **CTRL** key and click **Microsoft.Office.Interop.Word.** Click the **OK** button.
     1. **Note:** Since the lab is targeted to Office 2007,make sure version 12.0.0.0 is selected when adding the references.
     2. When complete, the list of references for your console application should look like this:
     3. 
     4. Figure
     5. References for Office development

Task 2 – Creating a Collection of Business Entities

* 1. In this task, you will create a collection of business entities that will be pre-populated with data. Pre-populating simulates loading the entities with data from a separate source. We are shortcutting here to focus on things other than data access.
  2. You will start by creating an individual entity.
  3. In the Solution Explorer, right click on the node for the **OfficeApplication** project and select **Add…** | **Class** from the menu.
  4. The **Add New Item** dialog box appears with the Class template already selected. Change the name to *Account.cs* (for C#) *Account.vb* (for Visual Basic) and click the **Add** button.
     1. If you are doing the lab in C#, change the definition of the **Account** class to be **public**, as shown below:
        1. C#
        2. **public class Account**
        3. {
        4. }
  5. Add public properties for **ID** (**integer**), **Balance** (**double**) and **AccountHolder** (**string**) to the **Account** as shown:
     1. *(Code Snippet – Office Programmability Lab - Ex1 Account properties CSharp)*
     2. C#
     3. public class Account
     4. {
     5. **public int ID { get; set; }**
     6. **public double Balance { get; set; }**
     7. **public string AccountHolder { get; set; }**
     8. }
     9. *(Code Snippet – Office Programmability Lab - Ex1 Account properties VB)*
     10. Visual Basic
     11. Public Class Account
     12. **Property ID As Integer**
     13. **Property Balance As Double**
     14. **Property AccountHolder As String**
     15. End Class
     16. **Note:** A new feature in Visual Basic is the ability to declare a property without declaring a separate field to store the value in. These automatically-implemented properties work the same as if the developer had declared a private field to hold the data and implemented simple “getters” and “setters.” They are called the same way a traditional property is called. If a more robust logic is required for property access, traditional properties that store data in private fields can still be created. Comparable Visual Basic code from VS 2008 would look like the following example:
     17. Visual Basic
     18. ' This is the old style using backing variables instead of
     19. ' auto-implemented properties
     20. Public Class Account
     21. Private \_id As Integer
     22. Private \_balance As Double
     23. Private \_accountHolder As String
     24. Property ID() As Integer
     25. Get
     26. Return \_id
     27. End Get
     28. Set(ByVal value As Integer)
     29. \_id = value
     30. End Set
     31. End Property
     32. Property Balance() As Double
     33. Get
     34. Return \_balance
     35. End Get
     36. Set(ByVal value As Double)
     37. \_balance = value
     38. End Set
     39. End Property
     40. Property AccountHolder() As String
     41. Get
     42. Return \_accountHolder
     43. End Get
     44. Set(ByVal value As String)
     45. \_accountHolder = value
     46. End Set
     47. End Property
     48. End Class
  6. The **Account** class represents your individual entity type. Normally an application would create some sort of collection of these entities and populate them with data from a data store of some sort. To keep this lab concise and focused you will populate your list by pushing values into the individual objects at creation. In the *Module1.vb* (VB) or *Program.cs* (C#) file, create a new method called **CreateAccountList** as shown in the following code:
     1. *(Code Snippet – Office Programmability Lab - Ex1 CreateAccountList VB)*
     2. Visual Basic
     3. **Private Function CreateAccountList() As List(Of Account)**
     4. **Dim checkAccounts As New List(Of Account) From {**
     5. **New Account With {**
     6. **.ID = 1,**
     7. **.Balance = 285.93,**
     8. **.AccountHolder = "John Doe"**
     9. **},**
     10. **New Account With {**
     11. **.ID = 2,**
     12. **.Balance = 2349.23,**
     13. **.AccountHolder = "Richard Roe"**
     14. **},**
     15. **New Account With {**
     16. **.ID = 3,**
     17. **.Balance = -39.46,**
     18. **.AccountHolder = "I Dunoe"**
     19. **}**
     20. **}**
     21. **Return checkAccounts**
     22. **End Function**
     23. **Note:** The code above highlights another new feature in Visual Basic: collection initializers. This enables developers to save large amounts of time by declaring the list and initializing the list’s contents directly at that point. There is no need to instantiate the list and then individually “new” up items and add them to the list.
     24. *(Code Snippet – Office Programmability Lab - Ex1 CreateAccountList CSharp)*
     25. C#
     26. **private static List<Account> CreateAccountList()**
     27. **{**
     28. **var checkAccounts = new List<Account> {**
     29. **new Account{**
     30. **ID = 1,**
     31. **Balance = 285.93,**
     32. **AccountHolder = "John Doe"**
     33. **},**
     34. **new Account {**
     35. **ID = 2,**
     36. **Balance = 2349.23,**
     37. **AccountHolder = "Richard Roe"**
     38. **},**
     39. **new Account {**
     40. **ID = 3,**
     41. **Balance = -39.46,**
     42. **AccountHolder = "I Dunoe"**
     43. **}**
     44. **};**
     45. **return checkAccounts;**
     46. **}**
  7. Call the **CreateAccountList** method from the Main method in the *Module1.vb* (VB) or *Program.cs* (C#) file as show:
     1. Visual Basic
     2. Sub Main()
     3. **Dim checkAccounts = CreateAccountList()**
     4. End Sub
     5. C#
     6. static void Main(string[] args)
     7. {
     8. **var checkAccounts = CreateAccountList();**
     9. }

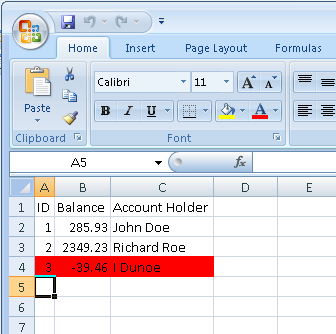
Task 3 – Creating an Excel Workbook and Populate it with Data from the Business Entities

* 1. In this task, you will create a new Excel workbook. You will then loop through the collection of business objects and insert the data into the Excel workbook. Finally, you will format the worksheet.
  2. The first step is to import the Office namespaces with **Imports** (VB) or **using** (C#) statements. Enter the following code at the top of the *Module1.vb* (VB) or *Program.cs* (C#) file:
     1. *(Code Snippet – Office Programmability Lab - Ex1 Namespaces VB)*
     2. Visual Basic
     3. **Imports Microsoft.Office.Interop**
     4. **Imports System.Runtime.CompilerServices**
     5. *(Code Snippet – Office Programmability Lab - Ex1 Namespaces CSharp)*
     6. C#
     7. **using Microsoft.Office.Interop;**
     8. **using Excel = Microsoft.Office.Interop.Excel;**
     9. **using Word = Microsoft.Office.Interop.Word;**
  3. The code to insert the data into Excel is implemented as an **Extension** method.
     1. The first parameter of this (and all) extension methods indicates the type that the method operates on (your list of accounts). The second argument, **DisplayFunc**, is a type of **delegate** called **Action**. **Action** delegates can take any number of arguments, but always return **void**.
     2. This block of code creates a new Excel application with a workbook, injects some header text into the workbook, calls a helper method to display the values of your **Account** objects, then sets each column to AutoFit mode to better display each column’s values.
     3. Enter the following code in *Module1.vb* (vb) or *Program.cs* (C#) after the **Main** method:
     4. *(Code Snippet – Office Programmability Lab - Ex1 DisplayInExcel VB)*
     5. Visual Basic
     6. **<Extension()>**
     7. **Sub DisplayInExcel(ByVal accounts As IEnumerable(Of Account), ByVal DisplayFunc As Action(Of Account, Excel.Range))**
     8. **With New Excel.Application**
     9. **.Workbooks.Add()**
     10. **.Visible = True**
     11. **.Range("A1").Value = "ID"**
     12. **.Range("B1").Value = "Balance"**
     13. **.Range("C1").Value = "Account Holder"**
     14. **.Range("A2").Select()**
     15. **For Each ac In accounts**
     16. **DisplayFunc(ac, .ActiveCell)**
     17. **.ActiveCell.Offset(1, 0).Select()**
     18. **Next**
     20. **.Columns(1).AutoFit()**
     21. **.Columns(2).AutoFit()**
     22. **.Columns(3).AutoFit()**
     23. **End With**
     24. **End Sub**
     25. **Note:** In C #, it is necessary to change the declaration of your **Program** class in *Program.cs* to be **static**.
     26. *(Code Snippet – Office Programmability Lab - Ex1 DisplayInExcel CSharp)*
     27. C#
     28. **static void DisplayInExcel(this IEnumerable<Account> accounts,**
     29. **Action<Account, Excel.Range> DisplayFunc)**
     30. **{**
     31. **var x1 = new Excel.Application();**
     32. //see the Note below
     33. **x1.Workbooks.Add();**
     34. **x1.Visible = true;**
     35. **x1.get\_Range("A1").Value2 = "ID";**
     36. **x1.get\_Range("B1").Value2 = "Balance";**
     37. **x1.get\_Range("C1").Value2 = "Account Holder";**
     38. **x1.get\_Range("A2").Select();**
     39. **foreach (var ac in accounts)**
     40. **{**
     41. **DisplayFunc(ac, x1.ActiveCell);**
     42. **x1.ActiveCell.get\_Offset(1, 0).Select();**
     43. **}**
     44. **((Excel.Range)x1.Columns[1]).AutoFit();**
     45. **((Excel.Range)x1.Columns[2]).AutoFit();**
     46. **((Excel.Range)x1.Columns[3]).AutoFit();**
     47. **}**
     48. **Note:** The statement **x1.Workbooks.Add();** reflects an important change in the .NET Framework 4.0 and C# 4.0: optional parameters. Previous versions of the Microsoft.Office.Interop.Excel.Workbook COM interop assembly required passing in a parameter. Besides, C#’s previous versions had no support for optional parameters, something Visual Basic developers have enjoyed for years. Earlier the statement would have appeared as **x1.Workbooks.Add(Missing.Value);** The new interop assemblies in the .NET Framework 4.0, coupled with C# 4.0’s new support for optional parameters lets you use a more concise, clearer syntax.
  4. The next step is to call your extension method from the **Main** method of your program. Add the following code to the bottom of the **Main** method:
     1. (Code Snippet – *Office Programmability Lab - Ex1 DisplayInExcel call VB*)
     2. Visual Basic
     3. **checkAccounts.DisplayInExcel(Sub(account, cell)**
     4. **cell.Value2 = account.ID**
     5. **cell.Offset(0, 1).Value2 = account.Balance**
     6. **cell.Offset(0, 2).Value2 = account.AccountHolder**
     7. **If account.Balance < 0 Then**
     8. **cell.Interior.Color = RGB(255, 0, 0)**
     9. **cell.Offset(0, 1).Interior.Color = RGB(255, 0, 0)**
     10. **cell.Offset(0, 2).Interior.Color = RGB(255, 0, 0)**
     11. **End If**
     12. **End Sub)**
     13. **Note:** Here you see another new feature in Visual Basic; statement lambdas. A statement lambda is an anonymous block of code that is treated like a variable in that it can be passed as an argument to a method, as you are doing in this example. The syntax for a lambda is to “declare” a series of arguments, in this case **account** and **cell**, and then to supply the logic for that lambda. When using statement lambdas **function** or **sub** names are not required and argument types are inferred.
     14. *(Code Snippet – Office Programmability Lab - Ex1 DisplayInExcel call CSharp)*
     15. C#
     16. **checkAccounts.DisplayInExcel((account, cell) =>**
     17. **{**
     18. **cell.Value2 = account.ID;**
     19. **cell.get\_Offset(0, 1).Value2 = account.Balance;**
     20. **cell.get\_Offset(0, 2).Value2 = account.AccountHolder;**
     21. **if (account.Balance < 0)**
     22. **{**
     23. **cell.Interior.Color = 255;**
     24. **cell.get\_Offset(0, 1).Interior.Color = 255;**
     25. **cell.get\_Offset(0, 2).Interior.Color = 255;**
     26. **}**
     27. **});**
     28. The **DisplayInExcel** extension method takes one argument, an **Action** **delegate** which in turn takes two arguments. The **Action** delegate’s arguments are an instance of your **Account** object and an instance of a cell in an Excel worksheet. This extension method is in reality a unit of functionality that will be called for each **Account** object in your list by the **for each** loop in the body of the **DisplayInExcel** function.
     29. The **DisplayInExcel** function contains the logic for creating and performing setup of the Excel workbook. The delegate you are providing contains the logic to take the individual values from the **Account** object and place them in the Excel worksheet.

# Next Step

* 1. Exercise 1: Verification

### Exercise 1: Verification

* 1. In this verification, you will run the console application and observe that an Excel worksheet is opened and your account data is populated in a worksheet.
  2. In Visual Studio, press the F5 key to run the application in debug mode.
  3. Observe that the application starts an instance of Microsoft Excel and inserts your data into the first worksheet:
     1. 
     2. Figure
     3. Account data in an Excel worksheet
     4. **Note:** Notice that the column widths have been adjusted to be sized based on the contents. Also notice per the logic in the **for each** loop of the **delegate** passed to the **DisplayInExcel** function, the account with a negative balance has a red background.
  4. Close Excel. Do not save any changes.

# Next Step

* 1. Exercise 2: Embedding an Excel Worksheet in Microsoft Word

Exercise 2: Embedding an Excel Worksheet in Microsoft Word

* 1. In this exercise, you will embed your Excel worksheet in a Microsoft Word document. This exercise will also introduce C# developers to optional parameters.

**Note:** To verify that each step is correctly performed, it is recommended to build the solution at the end of each task.



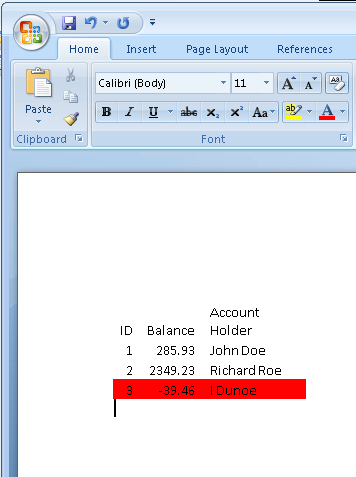
Task 1 – Embedding your Excel Worksheet into a New Microsoft Word Document

* 1. Open Microsoft Visual Studio 2010 from **Start** | **All Programs** | **Microsoft Visual Studio 2010** | **Microsoft Visual Studio 2010**.
  2. Open the **OfficeApplication.sln** solution file. By default, this file is located in the folder: **%TrainingKitInstallFolder%*\Labs\Dev10Office\Source\Ex02-EmbeddingWorksheet\Begin\*** and choose the language of your preference. Optionally, you can continue working the solution you created in the previous exercise.
  3. Copy your Excel worksheet so that it can be pasted into the Word document. The **Copy()** method accomplishes this for you. Add the following code to the end of the **DisplayInExcel** method (in the Visual Basic method, this command must be within the **With** block):
     1. Visual Basic
     2. .Range("A1:C4").Copy()
     3. C#
     4. x1.get\_Range("A1:C4").Copy();
  4. Now that your worksheet is copied to the clipboard, you need to create a Word document and past the worksheet into it. In the Module1.vb (VB) or Program.cs (C#) file, create a new method called EmbedInWordDocument as shown in the following code:
     1. *(Code Snippet – Office Programmability Lab - Ex2 EmbedInWordDocument VB)*
     2. Visual Basic
     3. **Private Sub EmbedInWordDocument()**
     4. **Dim word As New Word.Application**
     5. **word.Visible = True**
     6. **word.Documents.Add()**
     7. **word.Selection.PasteSpecial(Link:=True, DisplayAsIcon:=False)**
     8. **End Sub**
     9. *(Code Snippet – Office Programmability Lab - Ex2 EmbedInWordDocument CSharp)*
     10. C#
     11. **private static void EmbedInWordDocument()**
     12. **{**
     13. **var word = new Word.Application();**
     14. **word.Visible = true;**
     15. **word.Documents.Add();**
     16. **word.Selection.PasteSpecial(Link: true, DisplayAsIcon: false);**
     17. **}**
     18. **Note:** C# developers will recognize a new feature to C# 4.0 in the call to **PasteSpecial**: the use of optional parameters. **PasteSpecial** actually has seven parameters. In most cases, you do not need to supply values for these parameters, as the defaults are adequate for the application. In previous version of C# these values still had to be provided even if the provided value was of type **System.Reflection.Missing.Value**, which resulted in called to **PasteSpecial** that look like this:
     19. C#
     20. object iconIndex = System.Reflection.Missing.Value;
     21. object link = true;
     22. object placement = System.Reflection.Missing.Value;
     23. object displayAsIcon = false;
     24. object dataType = System.Reflection.Missing.Value;
     25. object iconFileName = System.Reflection.Missing.Value;
     26. object iconLabel = System.Reflection.Missing.Value;
     27. word.Selection.PasteSpecial(ref iconIndex,
     28. ref link,
     29. ref placement,
     30. ref displayAsIcon,
     31. ref dataType,
     32. ref iconFileName,
     33. ref iconLabel);
     34. **Note:** C# developers should also note that while parameters to **PasteSpecial** are still by reference, it is no longer necessary to specify this with the **ref** keyword.
  5. Call the **EmbedInWordDocument** method from the end of the Main method in the Module1.vb (VB) or Program.cs (C#) file as show:
     1. Visual Basic
     2. Sub Main()
     3. Dim checkAccounts = CreateAccountList()
     4. checkAccounts.DisplayInExcel(Sub(account, cell)
     5. cell.Value2 = account.ID
     6. cell.Offset(0, 1).Value2 = account.Balance
     7. cell.Offset(0, 2).Value2 = account.AccountHolder
     8. If account.Balance < 0 Then
     9. cell.Interior.Color = RGB(255, 0, 0)
     10. cell.Offset(0, 1).Interior.Color = RGB(255, 0, 0)
     11. cell.Offset(0, 2).Interior.Color = RGB(255, 0, 0)
     12. End If
     13. End Sub)
     14. **EmbedInWordDocument()**
     15. End Sub
     16. C#
     17. static void Main(string[] args)
     18. {
     19. var checkAccounts = CreateAccountList();
     20. checkAccounts.DisplayInExcel((account, cell) =>
     21. {
     22. cell.Value2 = account.ID;
     23. cell.get\_Offset(0, 1).Value2 = account.Balance;
     24. cell.get\_Offset(0, 2).Value2 = account.AccountHolder;
     25. if (account.Balance < 0)
     26. {
     27. cell.Interior.Color = 255;
     28. cell.get\_Offset(0, 1).Interior.Color = 255;
     29. cell.get\_Offset(0, 2).Interior.Color = 255;
     30. }
     31. });
     32. **EmbedInWordDocument();**
     33. }

# Next Step

* 1. Exercise 2: Verification

### Exercise 2: Verification

* 1. In this verification, you will run the console application and observe that in addition to the activities your application performed in the previous Verification, your Excel worksheet is also embedded in a Microsoft Word application.
  2. In Visual Studio, press the F5 key to run the application in debug mode.
  3. Observe that the application starts an instance of Microsoft Excel and inserts your data into the first worksheet as before.
  4. When finished creating the Excel worksheet, the application will start in instance of Microsoft Word and insert your data into a document:
     1. 
     2. Figure
     3. Account data in a Word document
  5. Close Excel and Word. Do not save any changes.

# Next Step

* 1. Exercise 3: Removing the PIA Dependence from the Application

Exercise 3: Removing the PIA Dependence from the Application

* 1. Office programmability relies on types from the Office Primary Interop Assembly (PIA) to be able to interface with the various applications in the office suite. In the past, this has dramatically impacted deployment of customized Office applications for a number of reasons. First, different versions of Office use different versions of the PIA, creating complexity for managing the assemblies. Secondly, IT personnel had to figure out deployment strategies to ensure required PIA resources were available across their support base. Additionally, deploying the PIA with an application resulted in a larger installation than necessary due to the extra, unused features.
  2. **Note:** Primary Interop Assemblies (PIAs) are a critical part of any .NET framework application needing to interface with COM components. A regular Interop Assembly is used for describing the COM types, allowing managed code to bind to the COM types at compile time. Interop Assemblies also help the CLR understand how those COM types should be marshaled at run time. The Primary Interop Assembly differs from regular Interop Assemblies in that it is the “official” holder of all type descriptions and is signed by the vendor to ensure its validity.
  3. Development in Visual Studio 2010 eases these burdens by removing this PIA dependence from applications. When the dependency is removed, the compiler will import whatever types are needed by your application from the PIA directly into your application; you simply deploy your application knowing that all the PIA classes you need are included with your application’s assemblies.

Task 1 – Checking that PIA dependencies are removed by default

* 1. In Visual Studio 2010 PIA dependencies are removed by default, to verify this:
  2. Open Microsoft Visual Studio 2010 from **Start** | **All Programs** | **Microsoft Visual Studio 2010** | **Microsoft Visual Studio 2010**.
  3. Open the **OfficeApplication.sln** solution file. By default, this file is located in the folder: **%TrainingKitInstallFolder%*\Labs\Dev10Office\Source\RemovingPIA\Begin\*** and choose the language of your preference. Optionally, you can continue working the solution you created in the previous exercise.
  4. Expand the project’s **References**.
     1. In Visual Basic, you should select the **OfficeApplication** project node and click the **Show All Files** button in the Solution Explorer in order to see the References node.
  5. Right-Click on **Microsoft.Office.Interop.Excel** in your project’s references, and choose **Properties**.
  6. Notice that the value of **Embed Interop Types** is set to **True**.
  7. Right-click on **Microsoft.Office.Interop.Word** in your project’s references, and choose **Properties**.
  8. Notice that the value of **Embed Interop Types** is set to **True**.

# Next Step

* 1. Summary

Summary

* 1. In this lab, you have seen the ease of working with Office to create custom applications. You have worked with new language features like Visual Basic’s new auto-properties and collection initializers which enable you to more quickly build up your classes and methods.
  2. Among other language features, you have seen C# 4.0’s new use of optional parameters, and seen how it makes working with the APIs of Office’s Primary Interop Assemblies much more concise and less error prone. Furthermore, you have seen how Visual Basic’s addition of multi-line lambdas and statement lambdas enables you to succinctly deal with handing delegates a suitable statement.
  3. Finally, you have seen how Interop types can be embedded in an Office Business Application to drastically ease the burden of deploying your Office application. This is a vast improvement over previous Office development environments which required you to deal with the hardship of deconflicting multiple version dependencies.
  4. All these improvements in the tools, platform, and languages in the latest release of Visual Studio 2010 should boost your ability to create powerful, flexible Office Business Applications.