C# 4.0

Estimated time for completion: 45 minutes

Overview:

In this lab you will investigate how optional and default parameters combined with dynamic typing make COM interop easier and how dynamic typing can simplify parsing hierarchical data.

Goals:

- Learn how optional parameters and dynamic typing aid COM interop
- Build a dynamic type for parsing XML

Lab Notes:

N/A

Part 1: COM Interop

In the first part of the lab you will drive Excel to render a supplied .csv file as a graph

1. In Visual Studio 2010, open the starter solution **before/csharp4.sln.** You will notice that it contains two projects. The first part of the lab will use the Interop project
2. The project contains a .csv file called rates.csv. Open the file. This file contains interest rates showing a financial yield curve generated by linear interpolation. Your task will be to use Excel to render this into a graph
3. Add a reference to the Excel type library from the COM tab in the Add Reference dialog. If you are using Office 2007 this will be the **Microsoft Excel 12.0 Object Library** and for Office 2003 it will be the **Microsoft Excel 11.0 Object Library**
4. In `Main` create a new instance of the Excel `Application` class. Make sure you add a using statement for the Excel one rather than the VBE one
5. Because you are performing COM interop you will need to make sure you get hold of every intermediate object in the COM hierarchy to allow you to release these objects efficiently so create a local variable of type `Workbooks` and assign the application `Workbooks` collection to it
6. Get the full file path to the rates file by calling the supplied `GetRatesFilePath` helper method
7. Call `Open` on the `Workbooks` collection object passing the rates file path. Notice that intellisense will show you that the `Open` method can actually take 15 parameters. Before C# 4.0 you would have to have passed 14 instances of `Type.Missing` to this method but now, due to the new support for optional parameters, you only need pass the one

mandatory file name. The `Open` method returns a `Workbook` reference so store this in a local variable

```
Application xl = new Application();
Workbooks books = xl.Workbooks;

string ratesFile = GetRatesFilePath();

Workbook book = books.Open(ratesFile);
```

8. To work with the loaded file we need a reference to its `Worksheet` object. You can get this by accessing the first element of the workbook's `Worksheets` collection. Remember that Excel collections starts at 1 rather than 0. This collection actually returns an `object` which prior to C# 4.0 we would have to have cast to a `Worksheet` object. However, with dynamic support in C# we can now store the reference in a `dynamic` variable

```
dynamic sheet = book.Worksheets[1];
```

9. Finally we need to set the format of the first cell in the sheet to a date. You do this by getting hold of a `Range` object that represents a group of cells and then setting its `NumberFormat`.
    a. Call the `Range` method of the worksheet passing `"A:A"` as the value to get the first column and store the result in a variable of type `Range`
    b. Set the `NumberFormat` property of the range to `"yyyy-mm-dd"`

```
Range range = sheet.Range("A:A");
range.NumberFormat = "yyyy-mm-dd";
```

10. The data is now ready to be rendered into a graph. To do this you will need to get the `ChartObjects` collection from the worksheet. Remember to keep this in a local variable
11. Now add a new chart into the collection by calling `Add` on the `ChartObjects` collection object. You will need to pass in position and dimensions for the chart `(100, 80, 700, 400)` should work for most screen resolutions. The `Add` method returns a `ChartObject` which you should store in a local variable
12. Now cache the actual `Chart` from the chart object in a local variable

```
ChartObjects xlCharts = sheet.ChartObjects;
ChartObject myChart = xlCharts.Add(100, 80, 700, 400);
Chart chartPage = myChart.Chart;
```

13. Finally, for building the chart you need to specify the data for it to render and what type of chart it is. Start by retrieving a `Range` object from the sheet for the first two columns – the syntax for the first two columns is: `"A:A,B:B"`
14. Now, using the `Chart`, set its data source to the range you just retrieved
15. To complete the chart definition set its `ChartType` to `xlChartType.xlArea`
16. Make the spreadsheet visible by setting the `Visible` property on the application object to `true`
17. Add a Console.WriteLine / Console.ReadLine pair to stop the process exiting. After these lines of code you will shortly put the code to release of the Excel objects

```
Range chartRange = sheet.Range("A:A,B:B");
chartPage.SetSourceData(chartRange);
chartPage.ChartType = XlChartType.xlArea;

xl.Visible = true;

Console.WriteLine("Press enter to quit Excel ...");
Console.ReadLine();
```

18. You should be able to compile and test your code and see the graph being rendered although you will have to close Excel manually
19. To close Excel firstly you need to hide it and then tell Excel that visually you are finished with it.
    a. Set the application `Visible` property to `false`
    b. Call the `Quit` method on the application
20. Now you need to release all of the COM objects that you used when constructing the graph using `Marshal.ReleaseComObject`

```
xl.Visible = false;
xl.Quit();

Marshal.ReleaseComObject(chartRange);
Marshal.ReleaseComObject(chartPage);
Marshal.ReleaseComObject(myChart);
Marshal.ReleaseComObject(xlCharts);
Marshal.ReleaseComObject(range);
Marshal.ReleaseComObject(sheet);
Marshal.ReleaseComObject(book);
Marshal.ReleaseComObject(books);
Marshal.ReleaseComObject(xl);
```

21. Running the code should now tear down Excel after you press enter at the command line prompt

Part 2: Creating a Dynamic XML Parser

One of the powerful features of dynamic languages is the ability to wire up parsing code at runtime for data being processed. The dynamic support in C# 4.0 brings this power to C# where we can simulate methods dynamically. In this part of the lab you will build an XML parser that supports dynamic parsing of an XML file

1. For this part of the lab you will be using the `DynamicParsing` project in the `csharp4` solution. It contains the data to be parsed in the form of an **order.xml** file. Take a look at the file and familiarize yourself with its contents
2. Add a new class to the project calling it `DynamicXmlParser`
3. This class needs to support dynamic behavior so derive it from `DynamicObject`
4. Add a member variable to the class of type `XElement` called `element`
5. Add a `public` constructor to the class that takes a file name as a `string`. Inside the constructor call `XElement.Load` on the file to initialize the `element` member variable
6. Add a `private` constructor that takes an `XElement` as a parameter and use this to initialize the `element` member variable. We will use this version internally during parsing

```
class DynamicXmlParser : DynamicObject
{
  XElement element;
  public DynamicXmlParser(string file)
  {
    element = XElement.Load(file);
  }

  private DynamicXmlParser(XElement el)
  {
    element = el;
  }
}
```

7. The idea of dynamic parsing is we want to allow the consumer to walk through the XML data using properties. We will dynamically create those properties as requested. In fact all we really have to do it to return the appropriate object as if that property had actually existed. To perform this dynamic property wire up we need to `override` the `TryGetMember` `virtual` method of `DynamicObject`
8. `TryGetMember` returns a `bool`. We need to return `true` if we support the property and `false` if we do not. In effect this means that if the sub-element they have asks for exists we return `true`, else we return `false`. So the first job it to try to get the requested sub-element. The name of the requested element is in the `binder` parameter's `Name` property. Use the `Element` method of the `element` member variable to attempt to get the sub-element
9. If the sub-element does not exist then the result will be `null`. In this case we need to initialize the `result` to `null` (it must be initialized as it is an out param) and return `false`

10. If the element does exist we need to create a new instance of the `DynamicXmlParser` wrapped around this sub-element and set the `result` to it. This allows the consumer to recursively use this propery syntax as they walk through the document. Don't forget to return `true` as we do support the property

```
public override bool TryGetMember(GetMemberBinder binder, out object result)
{
  XElement sub = element.Element(binder.Name);
  if (sub == null)
  {
    result = null;
    return false;
  }
  else
  {
    result = new DynamicXmlParser(sub);
    return true;
  }
}
```

11. Finally we need to provide a way for the consumer to actually get at the data in the document – the data being the text nodes and the attributes. We will override `ToString` for the text node and use an indexer for the attributes
    a. Override `ToString` and return the `Value` property of the `element` member variable
    b. Add an indexer that returns a `string` and takes a `string` index. Pass the index to the `Attribute` method on the `element` member variable and return its `Value` property

```
public override string ToString()
{
  return element.Value;
}

public string this[string attr]
{
  get { return element.Attribute(attr).Value; }
}
```

12. We have now finished the parser so we can use it in `Main`. Create an instance of the `DynamicXmlParser` class passing the file path to the **order.xml** file. Assign this object to a `dynamic` local variable. This will enable the dynamic behavior as we use the variable
13. Use the parser to get the ordered attribute and some of the text nodes from the document

```
dynamic parser = new DynamicXmlParser(@"..\..\order.xml");
Console.WriteLine(parser["orderId"]);
Console.WriteLine(parser.customer.name);
Console.WriteLine(parser.orderItem.product);
Console.WriteLine(parser.orderItem.supplier);
```

14. Compile and test your code

Solutions

after\csharp4.sln