Hands-On Lab

Introduction To Windows Workflow Foundation in .NET 4

Lab version: 1.0.0

Last updated: 12/10/2010
Task 1 – Writing the Test to Verify that the Workflow Thread ID is Returned as an Out Argument . 31
Task 2 – Returning the WorkflowThread as an Argument ................................................................. 33
Task 3 – Modifying the Test to use WorkflowApplication .............................................................. 37
Exercise 6: Verification 40

EXERCISE 7: IF/ELSE LOGIC ............................................................................................................. 41
Task 0 – Opening the Solution ........................................................................................................... 41
Task 1 – Writing the Test for a New Requirement ............................................................................. 42
Task 2 – Implementing the New Requirement in the Workflow ....................................................... 43
Exercise 7: Verification 47

EXERCISE 8: ERROR HANDLING ................................................................................................... 48
Task 0 – Opening the Solution ........................................................................................................... 48
Task 1 – Writing a Test to Observe Error Behavior .......................................................................... 48
Task 2 – Adding the Try/Catch Activity to the Workflow ................................................................. 50
Exercise 8: Verification 54

EXERCISE 9: CUSTOM ACTIVITIES AND DESIGNERS .................................................................. 54
Task 0 – Opening the Solution ........................................................................................................... 55
Task 1 – Add Activity Designer Library ............................................................................................ 55
Task 2 – Create Composite Activity in XAML .................................................................................. 56
Task 3 – Add Existing Activities ...................................................................................................... 58
Task 4 – Observe Custom Activities without Designers ................................................................. 59
Task 6 – Explore Activity Designers ................................................................................................ 60
Task 7 – Linking the Activity Designers to the Activity ................................................................. 64
Exercise 9: Verification 65
Task 1 – Prepare the test projects ..................................................................................................... 65
Task 2 – Create DiagnosticTrace Test ............................................................................................ 65
Task 3 – Add a DiagnosticTrace activity to the SayHello Workflow ............................................. 68
Task 4 – Add the PrePostSequence Test ......................................................................................... 68

EXERCISE 10: HOSTED DESIGNER ............................................................................................... 72
Task 0 – Opening the Solution ........................................................................................................... 73
Task 1 – Adding new WPF Application ............................................................................................. 73
Exercise 10: Verification 80
Overview

Welcome to Windows Workflow Foundation 4 (WF) Microsoft’s programming model, engine and tools for quickly building workflow-enabled applications. This release of WF in .NET Framework 4 changes several development paradigms from previous versions. Workflows are now easier to create, execute, and maintain.

In this lab, you will get to know the basics of creating, hosting and running a workflow. This lab is also an introduction to the new workflow authoring constructs in the .NET Framework 4 and Visual Studio 2010, including the new Workflow Designer, expressions, variables and arguments. Additionally, you will explore the use of some basic built-in activities, create custom activities, activity designers and even re-host the workflow designer in a WPF application.

Do I have to do all the exercises?

This lab is long. To do all of it will probably take close to 2 hours. This lab has been designed so that each exercise is independent of the others. Just open the solution found under the Begin folder for the exercise. For example, if you wanted to learn how to create a custom activity designer you would start with Exercise 9 and open the solution from the Source\Ex9-ActivityDesigner\Begin folder.

Objectives

In this Hands-On Lab, you will learn:

- How to create a sequential workflow using the designer and XAML or, alternatively, using plain C# or Visual Basic code
- How to run and test the sequential workflow using the WorkflowApplication and WorkflowInvoker classes
- How to pass InArguments and receive OutArguments from a workflow
- How Expressions and Variables are used
- How to use the WriteLine, If, TryCatch, Catch<T> and Throw activities
- How to load and run an activity from a xaml file
- How to create an activity designer
- How to host the WorkflowDesigner in your own application
**System Requirements**

You must have the following items to complete this lab:

- Microsoft Visual Studio 2010
- Microsoft .NET Framework 4

**Setup**

All the requisites for this lab are verified using the Dependency Checker. To make sure that everything is correctly configured, follow these steps:

**Note:** To perform the setup steps you need to run the scripts in a command window with administrator privileges.

1. Run the Dependency checker for the Training Kit if you haven't done it previously. To do this, run the `CheckDependencies.cmd` script located under the `\Source\Setup` folder. Install any pre-requisites that are missing (rescanning if necessary) and complete the wizard.

**Note:** For convenience, much of the code you will be managing along this hands-on lab is available as Visual Studio code snippets.

To use code snippets with XAML, right click on the xaml file, select Open With and choose XML Editor. Then you can insert XML snippets into the file.

**Exercises**

This Hands-On Lab comprises the following exercises:

**Note:** each exercise builds on the one before it but you can start at any point in the lab by opening the solution in the Begin folder for the exercise.

1. Hello Workflow
2. Refactoring Workflow
3. The CodeActivity
4. Dynamic Workflows with XAML
5. Testing Workflows
Estimated time to complete this lab: **120 minutes.**

### Starting Materials

- **Visual Studio solutions.** Depending on the exercise you will find Visual Studio solutions for C# and Visual Basic that you can use as starting point for the exercises.

### What if I get stuck?

The source code that accompanies this hands-on lab includes an end folder where you can find a Visual Studio solution with the code that you would obtain if you complete the steps in each exercise. You can use this solution as a guide if you need additional help working through the exercises.

Remember to build the solutions before opening any file using the workflow designer.

### Next Step

**Exercise 1: Hello Workflow**

Workflows execute a business process. Each step in that process is implemented by an **Activity.**

In this exercise, you will create and test a simple “Hello World” process using Windows Workflow Foundation 4.

**Task 1 – Creating a Simple Hello Workflow Application**

In this task you will create a very simple workflow that will be the equivalent to this code:

```csharp
private static void SayHello()
{
    Console.WriteLine("Hello Workflow 4");
}
```
Visual Basic

Private Shared Sub SayHello()
    Console.WriteLine("Hello Workflow 4")
End Sub

1. Start **Microsoft Visual Studio 2010** from **Start | All Programs | Microsoft Visual Studio 2010**.
2. Select **File / New Project** Ctrl+Shift+N and set the project options
   a. Installed Templates: select either **Visual C#** or **Visual Basic** then select **Workflow** under the language
   b. Template: **Workflow Console Application**
   c. Name: **HelloWorkflow**.
   d. Location: `%TrainingKitInstallFolder%\Labs\IntroToWF\Ex1-HelloWorkflow\Begin`
   e. Solution: **HelloWorkflow**

![Figure 1](image)

*Creating a new Workflow Console Application (C#)*
3. Since your business process is a 1-step process, you can simply add a **WriteLine** activity to implement the process. From the **Toolbox**, drag a **WriteLine** activity and drop it on the design surface.

![Diagram of the Toolbox with a WriteLine activity highlighted]

**Figure 3**  
*Adding a WriteLine activity*

---

**Note:** If the Toolbox window is not visible, select **Toolbox** from the **View** menu.
4. Set the **WriteLine** **Text** property to "Hello Workflow 4".

![WriteLine Activity]

*Figure 4*

*Setting the Text property*

**WriteLine Activity**

The **WriteLine** activity is a simple activity that will display a message on the console. The text property is an **Expression** which could be the result of calling a function or evaluating the property of an object. In this case, the expression is a literal string so you must quote the string.

---

**Next Step**

**Exercise 1 Verification**

In order to verify that you have correctly performed all steps of exercise 1, proceed as follows:

1. Press **CTRL+F5** to build and run the workflow without debugging. The application should run in a console window and print the message “Hello Workflow 4”.

![Hello Workflow 4]

*Figure 5*

*The completed HelloWorkflow application*
Next Step

Exercise 2: Refactoring Workflow

Exercise 2: Refactoring Workflow

Even though you have a working application, you might want to make a few improvements. In this exercise, as Workflow1 is not a very descriptive name, you will change it to SayHello and see how the changes to the workflow name affect the rest of the program.

Task 0 – Opening the Solution

To begin this exercise you can use the solution you finished from Exercise 1. Alternatively, you can follow the following steps to begin with Exercise 2.

2. Open the starting solution for Exercise 2 located under the Source\Ex2-Refactoring\Begin folder and use it as the starting point for this exercise.
3. Press CTRL+SHIFT+B to build the solution.

Task 1 – Renaming Workflow1 to SayHello

1. To change the name of the workflow defined in Workflow1.xaml, Open it with the workflow designer and click on an empty area the designer surface. This will allow you to change the properties of the workflow.
Click on an empty area of the designer surface so you can set the name of the workflow.

2. In the Properties window, set the Name property of the Workflow to **HelloWorkflow.SayHello** (C#) or **SayHello** (VB)

---

**Figure 6**
*Click on an empty area of the designer surface so you can set the name of the workflow*

**Figure 7**
*Set the Name property of the workflow (C#)*
3. While it is not required, it is a good idea for the name of the .xaml file to match the name of the activity contained in it. In Solution Explorer, right-click Workflow1.xaml, select Rename and type SayHello.xaml.

¡Watch Out

When you rename C# or Visual Basic files, Visual Studio asks you if you want refactoring to change the name of the class also. It does not do this when renaming XAML files, so if you want to change the name of the workflow also you need to do this manually.

4. Press CTRL+SHIFT+B to build the solution. It will fail to compile because you changed the name of the workflow.

¿Why does the application fail to compile when I change the workflow name?\n
Your workflow is actually a class declared in XAML that inherits from System.Activities.Activity. The name of the workflow is the name of the class. When you set the Name property, you changed the name of the class.
Task 2 – Updating the Main Method to Use the New Name

WF4 runs a workflow with the *Workflow Runtime*. The easiest way to invoke the runtime is to use the *WorkflowInvoker* class which is what the Workflow Console Application template uses.

1. To fix the build break, open *Program.cs* (C#) or *Module1.vb* (Visual Basic) and modify it to use the new *SayHello* class name. To do this, locate the *WorkflowInvoker.Invoke* method call, and modify it to use *SayHello* instead of *Workflow1* as shown in the following code (shown in **bold**).

```csharp
static void Main(string[] args)
{
    WorkflowInvoker.Invoke(new SayHello());
}
```

```vbnet
Shared Sub Main()
    WorkflowInvoker.Invoke(New SayHello())
End Sub
```

**Why does Visual Studio warn me that the type SayHello is not defined?**

If you change Main() before compiling the workflow with the new SayHello name, you may notice a warning about the type SayHello not being defined. The reason for this is that Visual Studio is not aware of types declared in XAML until you build the solution.

Next Step

Exercise 2: Verification

In order to verify that you have correctly performed all steps of exercise 2, proceed as follows:

1. Press **CTRL+SHIFT+B** to build the solution. It should now build without errors.
2. Press **CTRL+F5** to run it. You should see the same message as before “Hello Workflow 4”
Exercise 3: The CodeActivity

As you have seen so far, WF4 consists of a designer that edits .xaml files and a runtime that invokes activities. When you author a workflow, you are creating a new kind of activity and because activities are just classes that inherit from System.Activities.Activity or one of its subclasses, you can declare workflows using C#, VB or XAML. In this exercise, you will implement your “greeting” business process by creating an activity in C# or VB.

As previously said, activities implement a business process. Some activities will implement the process by invoking other activities. For example, your SayHello activity did not actually write text to the console but instead used the WriteLine activity to do the work. could have implemented the same SayHello activity in C# or VB by inheriting from System.Activities.Activity and creating an Implementation as you see in the following examples

```csharp
public sealed class SayHelloActivity : Activity
{
    WriteLine writeLine = new WriteLine() { Text = "Hello Workflow 4" };

    public SayHelloActivity()
    {
        Implementation = () => { return writeLine; };
    }
}
```
This approach is useful if you are creating a workflow out of other activities that do the work as did in exercise 1 with the `WriteLine` activity. However, there may be times when you want to create an activity that implements the business logic in itself or by calling another class that is not an activity to do the work. To do this, you will inherit from a different base class `System.Activities.CodeActivity` and override the `Execute` method.

**Task 0 – Opening the Solution**

To begin this exercise you can use the solution you finished from Exercise 2. Alternatively, you can follow the following steps to begin with Exercise 3.

1. Start **Microsoft Visual Studio 2010** from **Start | All Programs | Microsoft Visual Studio 2010**.

2. Open the starting solution for Exercise 3 located under the **Source\Ex3-CodeActivity\Begin** folder and use it as the starting point for this exercise.

3. Press **CTRL+SHIFT+B** to build the solution.

**Task 1 – Creating the SayHelloInCode Activity**

In this task, you will create an activity in code and write the text to the console using `Console.WriteLine`

1. Right-click the **HelloWorkflow** project, select **Add / New Item**. From the **Workflow** templates select **Code Activity** and name the file **SayHelloInCode**.

2. Delete the **Text** property from the template – you will not need it.

3. **CodeActivity** is an **abstract** class. That means that when you inherit from it, you must override the **Execute** method. This is the place where you will do the work of your activity. Replace the default implementation of the class with the following code.

   *(Code Snippet - Introduction to WF4 Lab - SayHelloInCode Class CSharp)*

   ```c#
   public sealed class SayHelloInCode : CodeActivity
   ```
```csharp
{ 
    protected override void Execute(CodeActivityContext context) 
    { 
        Console.WriteLine("Hello Workflow 4 in code"); 
    }
}
```

(Code Snippet - Introduction to WF4 Lab - SayHelloInCode Class VB)

### Visual Basic

Public NotInheritable Class SayHelloInCode
    Inherits CodeActivity

    Protected Overrides Sub Execute(ByVal context As CodeActivityContext)
        Console.WriteLine("Hello Workflow 4 in code")
    End Sub
End Class

---

**Task 2 – Updating Main to Invoke SayHelloInCode**

1. Change Program.cs (C#) or Module1.vb (Visual Basic) to use the new SayHelloInCode class. To do this, locate the WorkflowInvoker.Invoke method code, and replace it with the following code:

   (Code Snippet - Introduction to WF4 Lab - InvokeSayHelloInCode Class CSharp)

   ```csharp
   static void Main(string[] args) 
   { 
       WorkflowInvoker.Invoke(new SayHelloInCode()); 
   }
   ```

   (Code Snippet - Introduction to WF4 Lab - InvokeSayHelloInCode Class VB)

   ```vb
   Shared Sub Main(ByVal args() As String) 
       WorkflowInvoker.Invoke(New SayHelloInCode())
   End Sub
   ```

---

**Next Step**

Exercise 3: Verification
Exercise 3: Verification

1. Press **CTRL+F5** to run the workflow without debugging. The application should run in a console window and print the message “Hello Workflow 4 in code”.

![Image of command prompt]

**Figure 12**
The completed HelloWorkflow application

---

Why would I create a code activity?

Writing business logic in code is nothing new. Why would you go to the effort to write it in a special class that inherits from CodeActivity? The reason is that by doing so now your business logic can be composed into other larger business processes that use the workflow runtime. As you will see later in the lab it will also benefit from a model for threading and data management that provides for highly scalable and long running business applications.

---

Next Step

Exercise 4: Dynamic Workflows with XAML

Exercise 4: Dynamic Workflows with XAML

Up to this point, you have been authoring workflows in .xaml, .cs or .vb files. These files are compiled into types that are included in the assembly of the project and run by the workflow runtime.

While it might look like the format of the source files does not matter, .xaml files offer some distinct advantages over authoring workflows in C# or VB.
The Workflow Designer works with .xaml files only, workflows authored in C# or VB have no designer support.

- XAML can be loaded and run dynamically without compiling it into an assembly.

Dynamic workflows provide some interesting possibilities for programs that want to generate business logic or make a runtime decision on which business logic to load and run.

**Task 0 – Opening the Solution**

To begin this exercise you can use the solution you finished from Exercise 3. Alternatively, you can follow the following steps to begin with Exercise 4.

1. Start **Microsoft Visual Studio 2010** from **Start | All Programs | Microsoft Visual Studio 2010**.
2. Open the starting solution for Exercise 4 located under the **Source\Ex4-XAML\Begin** folder and use it as the starting point for this exercise.
3. Press **CTRL+SHIFT+B** to build the solution.

**Task 1 – Modifying SayHello.xaml Files Properties**

In this task you will modify the HellowWorkflow program to load and run the SayHello.xaml file. Then you will modify the text of SayHello.xaml and observe the change in the message the next time you run the application.

1. You need to tell Visual Studio to treat **SayHello.xaml** as deployed content rather than code. To do this
   - In Solution Explorer select **SayHello.xaml**
   - Build Action: **Content**
   - Copy To Output Directory: **Copy Always**
   - Set the Custom Tool to blank

   ![Properties](image.png)

   **Figure 13**

   *Change the properties of SayHello.xaml to treat it as content*
Task 2 – Modifying Main() to Load the SayHello.xaml File

Previously your workflow was compiled into a type. To invoke a workflow from a .xaml file you will need to use `ActivityXamlServices` to load the .xaml file into memory and create an instance of an activity that `WorkflowInvoker` can invoke. Keep in mind that any assemblies that your .xaml file references must be available when the workflow is invoked.

1. Add the following namespace directives to the program.cs or Module1.vb file.

```csharp
using System.Activities.XamlIntegration;
```

```vbnet
Imports System.Activities.XamlIntegration
```

2. Modify program.cs or Module1.vb to use `ActivityXamlServices` and also add a call to `Console.ReadKey` this will make it easier to see what is happening with your app when running it from Windows Explorer.

(Code Snippet - Introduction to WF4 Lab - `ActivityXamlServices CSharp`

```csharp
static void Main(string[] args)
{
    WorkflowInvoker.Invoke(ActivityXamlServices.Load("SayHello.xaml"));
    Console.ReadKey(false);
}
```

(Code Snippet - Introduction to WF4 Lab - `ActivityXamlServices VB`

```vbnet
Sub Main()
    WorkflowInvoker.Invoke(ActivityXamlServices.Load("SayHello.xaml"))
    Console.ReadKey(False)
End Sub
```

Next Step

Exercise 4: Verification
Exercise 4: Verification

In this verification, you will run the HelloWorld application, then modify the deployed SayHello.xaml file from the bin\Debug folder and see if the new text appears in the console window.

1. Press **CTRL+F5** to run the workflow without debugging. The application should run in a console window and print the message “Hello Workflow 4”.

2. Navigate to the **Bin\Debug** directory under your project folder and locate **SayHello.xaml**.

![Figure 14](image)

*Locate the SayHello.xaml file in the project’s Bin\Debug directory*

3. Right click on **SayHello.xaml** and select **Edit** to open it in Notepad

![Figure 15](image)

*Right click and select Edit*

4. In Notepad change the Text property of the WriteLine activity to “Hello Workflow 4 XAML” then save and close it.

![Figure 16](image)

*Change the Text property in Notepad*
5. In Windows Explorer, run HelloWorkflow.exe and it should now say “Hello Workflow 4 from XAML”. Press any key to exit.

![Figure 17](image)

*HelloWorkflow.exe showing a new message from the .xaml file*

6. Go back to Visual Studio and restore **SayHello.xaml** to the default settings. You will need this for the rest of the labs.
   a. Build Action: XamlAppDef
   b. Copy to Output Directory: Do not copy
   c. Custom Tool: MSBuild:Compile

![Figure 18](image)

*Restore SayHello.xaml to the default properties for .xaml files*

7. In Solution Explorer, select SayHello.xaml and set the build action to XamlAppDef. Once this is done you can delete Activity1.xaml from the project.

**Next Step**

*Exercise 5: Testing Workflows*
Exercise 5: Testing Workflows

Up to this point, the application is not very interesting. It is only working in a console and does not receive any input arguments. Most applications that do meaningful work will have to deal with input and output arguments. Additionally, the application is not easily tested in its current form.

In this task, you will modify the SayHello activity to use arguments and prepare it to be useful in other non-console application environments by returning a greeting rather than writing directly to the Console with the WriteLine activity. You will be doing this using the "write the test first" approach. First, you will create a test that fails and then add the necessary code to make it pass.

The resulting application will be the equivalent of the following code.

```csharp
private static string SayHello(string name)
{
    return "Hello " + name + " from Workflow 4";
}
```

```visualbasic
Private Shared Function SayHello(ByVal name As String) As String
    Return "Hello " & name & " from Workflow 4"
End Function
```

Task 0 – Opening the Solution

To begin this exercise you can use the solution you finished from Exercise 4. Alternatively, you can follow the following steps to begin with Exercise 5.

2. Open the starting solution for Exercise 5 located under the Source\Ex5-Testing\Begin folder and use it as the starting point for this exercise.
3. Press CTRL+SHIFT+B to build the solution.

Task 1 – Creating the Unit Test Project

1. To begin with, create a unit test for the workflow to verify the behavior. In Solution Explorer, right-click the HelloWorkflow solution and select Add / New Project and set the project options.
   a. Installed Templates: select either Visual C# or Visual Basic then select Test under the language
b. Template: **Test Project**

c. Name: **HelloWorkflow.Tests**

![Adding a new Test Project to the solution (C#)](Image19)

**Figure 19**
*Adding a new Test Project to the solution (C#)*

![Adding a new Test Project to the solution (Visual Basic)](Image20)

**Figure 20**
*Adding a new Test Project to the solution (Visual Basic)*

2. Right-click the **HelloWorkflow.Tests** project and click **Add Reference**. Using the **Projects** tab, add a project reference to the **HelloWorkflow** project. Repeat these steps, using the **.NET** tab instead to add a reference to the **System.Activities** library.

3. Right-click **UnitTest1.cs** (C#) or **UnitTest1.vb** (Visual Basic), click **Rename** and change its name to **SayHelloFixture.cs** (C#) or **SayHelloFixture.vb** (Visual Basic). When prompted to rename the **UnitTest1** class select **Yes**.
Task 2 – Creating the Test

In this task you will create the test before you actually implement the behavior in the activity.

1. Add the following namespace directive to the SayHelloFixture.cs (C#) file or SayHelloFixture.vb (Visual Basic):

   C#
   ```
   using System.Activities;
   using HelloWorkflow;
   ```
   
   Visual Basic
   ```
   Imports System.Activities
   Imports HelloWorkflow
   ```

2. Create a test to make sure that the workflow properly greets you. To do this, open SayHelloFixture.cs (C#) or SayHelloFixture.vb (Visual Basic), locate the TestMethod1 method and rename it to ShouldReturnGreetingWithName.

3. The SayHello activity does not accept any arguments yet, but you will write the code to invoke it as though it does. This will allow you to think about what the interface to your activity feels like when consuming it. Do not worry if Visual Studio warns you about the UserName property not being defined yet. Replace the implementation as shown

   (Code Snippet - Introduction to WF4 Lab - ShouldReturnGreetingImpl CSharp)

   C#
   ```
   [TestMethod]
   public void ShouldReturnGreetingWithName()
   {
       IDictionary<string, object> output;

       output = WorkflowInvoker.Invoke(
               new SayHello()
               {
                   UserName = "Test"
               });
       Assert.AreEqual("Hello Test from Workflow 4", output["Greeting"]);
   }
   ```

   (Code Snippet - Introduction to WF4 Lab - ShouldReturnGreetingImpl VB)

   Visual Basic
   ```
   <TestMethod()> Public Sub ShouldReturnGreetingWithName()
       Dim output = WorkflowInvoker.Invoke(
               New SayHello() With {.UserName = "Test"})
   ```
How do I pass arguments to an activity?

You can create the activity and initialize the arguments (which are public properties) using object initialization or you can pass a Dictionary<string, object> (C#) or Dictionary(Of String, Object) (Visual Basic) of input parameters that map to the names of the input arguments of the activity.

How do I get data from output?

The output variable is an IDictionary<string, object> (C#) or IDictionary(Of String, Object) (Visual Basic) that contains a map of output variables using the name of the variable as the key.

Task 3 – Getting the Application to Compile

The interface to your Activity (in terms of input and output arguments) looks good but your application doesn’t compile. Your first goal is to get the app into a state where it will compile.

1. Press CTRL+SHIFT+B to build the application it should fail with a compile error

![Figure 21](image1.png)

*UserName has not been defined yet (C#)*

![Figure 22](image2.png)

*UserName has not been defined yet (VB)*
2. Open `SayHello.xaml` in the designer

3. Open the arguments pane by clicking on **Arguments**

![Figure 23](image)

*Click on Arguments to open the arguments pane*

4. Add the **UserName** and **Greeting** arguments as shown

![Figure 24](image)

*Declare the arguments to the activity*

5. Press **CTRL+SHIFT+B** to build the solution, it should now build without errors.

**Task 4 – Seeing the Test Fail**
It is important to see your test fail because it is possible that you have a bug in your test that would cause it to always succeed. In this task, you will insure that your test does indeed fail.

1. Press **CTRL+R,T** to run the unit tests in the current context. The test will run but it will fail, because the activity is not returning anything in the Greeting out argument.

![Figure 25](image)

*ShouldReturnGreetingWithName test*

**Task 5 – Making the Test Pass**

Now that you know your test is working, you need to do the simplest possible thing you can to make the test pass.

1. Writing text to the console with WriteLine isn’t going to make the test pass so delete the **WriteLine** activity by right-clicking on it and choosing **Delete** (❌).
2. You need to set the value of the **Greeting** out argument. To do this, drag an **Assign** activity from the **Primitives** group in the Toolbox and drop it onto the designer surface.

![Figure 26](image)

*Drag an Assign activity to the design surface*

3. Set the **To** property to **Greeting**

![Figure 27](image)

*Type Greeting in the left box*
4. You could type the greeting expression into the design surface but since the expression is longer, use the properties window. Click the button to the right of the Value property to open the expression editor.

Figure 28
Click the button to the right of the Value property

5. In the expression editor, you can enter longer expressions, including expressions that use the VB line continuation character “_”. Set the expression to "Hello " & UserName _ & " from Workflow 4"

Figure 29
The Assign activity that sets the Greeting argument

If you are a C# developer you may not be familiar with the line oriented nature of Visual Basic syntax. If you want an expression to break across lines in Visual Basic you must use an underscore to continue the expression on the next line.

6. Press CTRL+SHIFT+B to build the solution, it should compile without errors.
Next Step

Exercise 5: Verification

Now you are ready to run the test and see if your activity passes the test

1. Open the Test View window. To do this, on the Test menu point to Windows and click Test View ( ).

2. In the Test View, select the ShouldReturnGreetingWithName test and click the Run Selection button ( ).

![Figure 30](image)

Selecting and running the ShouldReturnGreetingWithName test

3. Verify that the test run passes.

![Figure 31](image)

The test passed

Next Step

Exercise 6: WorkflowApplication
Exercise 6: WorkflowApplication

To this point in the labs you have focused on creating an activity and invoking it in the simplest way possible with the WorkflowInvoker class. The WorkflowInvoker.Invoke method is simple because it is synchronous and invokes the workflow on the same thread as the caller.

Another way to invoke a workflow is with the WorkflowApplication class. This class allows you to run a workflow on a separate thread and to supply delegates to be invoked when the workflow completes, goes idle, terminates or has an unhandled exception. This allows you to create multi-threaded server or client programs more easily than you can without using WF4.

In this exercise, you will modify the host application to run your SayHello activity with WorkflowApplication and observe the threading behavior. You now have two requirements for your workflow.

1. Return a personalized greeting
2. Return a non-zero Int32 value containing the managed thread ID that the workflow was invoked on

Using the “write the test first” approach, you will begin by writing a test that verifies your new requirement for the workflow thread ID.

Task 0 – Opening the Solution

To begin this exercise you can use the solution you finished from Exercise 5. Alternatively, you can follow the following steps to begin with Exercise 6.

2. Open the starting solution for Exercise 6 located under the Source\Ex6-WorkflowApplication\Begin folder and use it as the starting point for this exercise.
3. Press CTRL+SHIFT+B to build the solution.

Task 1 – Writing the Test to Verify that the Workflow Thread ID is Returned as an Out Argument

In this task, you will create a test that verifies that the workflow returns a non-zero integer in the output arguments named WorkflowThread.

1. Open SayHelloFixture.cs and add the following namespace directives

```csharp
using System.Threading;
using System.Diagnostics;
```
2. Add the **ShouldReturnWorkflowThread** test as shown

(Code Snippet - Introduction to WF4 Lab - ShouldReturnWorkflowThread Test CSharp)

```csharp
/// <summary>
/// Verifies that the workflow returns an Out Argument
/// Name: WorkflowThread
/// Type: Int32
/// Value: Non-Zero
/// </summary>
[TestMethod]
public void ShouldReturnWorkflowThread()
{
    var output = WorkflowInvoker.Invoke(new SayHello()
    {
        UserName = "Test"
    });

    Assert.IsTrue(output.ContainsKey("WorkflowThread"),
    "SayHello must contain an OutArgument named WorkflowThread");

    // Don't know for sure what it is yet
    var outarg = output["WorkflowThread"];

    Assert.IsTrue(outarg, typeof(Int32),
    "WorkflowThread must be of type Int32");

    Assert.AreEqual(0, outarg,
    "WorkflowThread must not be zero");

    Debug.WriteLine("Test thread is "+
    Thread.CurrentThread.ManagedThreadId);
    Debug.WriteLine("Workflow thread is "+ outarg.ToString());
}
```

(Code Snippet - Introduction to WF4 Lab - ShouldReturnWorkflowThread Test VB)

```visualbasic
''' <summary>
''' Verifies that the SayHello workflow contains an Out Argument
''' Name: WorkflowThread
```
''' Type: Int32
''' Value: Non-Zero
''' </summary>
''' <remarks></remarks>
<TestMethod()>
Public Sub ShouldReturnWorkflowThread()

    Dim output = WorkflowInvoker.Invoke(_
        New SayHello() With {.UserName = "Test"})

    Assert.IsTrue(output.ContainsKey("WorkflowThread"),
        "SayHello must contain an OutArgument named WorkflowThread")

    ' Don't know for sure what it is yet
    Dim outarg = output("WorkflowThread")

    Assert.IsInstanceOfType(outarg, GetType(Int32),
        "WorkflowThread must be of type Int32")

    Assert.AreNotEqual(0, output("WorkflowThread"),
        "WorkflowThread must not be zero")

    Debug.WriteLine("Test thread is " & _
        Thread.CurrentThread.ManagedThreadId)

    Debug.WriteLine("Workflow thread is " & outarg.ToString())

End Sub

3. Press CTRL+R, A to run all tests. You should see the new test fail because you have not added
the WorkflowThread out argument yet.

Figure 32
One test passes, one test fails because you have no WorkflowThread argument

Task 2 – Returning the WorkflowThread as an Argument

Now that you have a test to verify the behavior, you need modify your workflow to make the test pass.

1. Open SayHello.xaml and add an Out argument named WorkflowThread of type Int32
Until now, your workflow has been just one activity. Now you need two activities, one to assign the greeting and another to assign the workflow thread. You need to modify the workflow to use an activity that will contain the two assign activities and there are several activities you could choose from to do this, but let’s begin with the simplest one, the Sequence activity.

2. You cannot drop a sequence on the designer until you remove the Assign activity that is already there. You will use this assign activity inside the sequence so you need to cut it, drop the sequence and then and paste it back. Right click on the Assign activity and select Cut.

3. Drag a Sequence and drop it on the design surface
4. Right click inside the sequence and select **Paste** to put the Assign activity inside the sequence.

---

**Figure 35**
*Drop a Sequence on the designer*

**Figure 36**
*Paste the assign activity inside the sequence*
5. Import the `System.Threading` namespace into your workflow. Click on **Imports** and add `System.Threading`. To do this type the first few letters of the namespace and when the correct namespace is highlighted press **Enter**

![Imports](image)

**Figure 37**
*Click on Imports and add System.Threading*

**Do I have to add System.Threading to Imports?**

No, this is optional just as it would be in any C# or VB project. If you don’t import the namespace you will have to fully qualify classes from the name space as in `System.Threading.Thread`

6. Now you need to assign the current managed thread ID to the **WorkflowThread** out argument. Drop an **Assign** activity on the Sequence below the first Assign activity and set the properties as shown

![Assign](image)

**Figure 38**
*Set the properties of the second Assign activity as shown*

7. Press **CTRL+SHIFT+B** to build the solution
8. Press **CTRL+R,A** to run all tests. The tests will now pass. If you want to see the thread returned by the workflow, double click on the **ShouldReturnWorkflowThread** test. The Debug output will appear in the test results. The actual thread ID will vary on your computer but you will notice that the thread ID for the test is the same as the thread ID for the Workflow because WorkflowInvoker invokes the workflow synchronously on the calling thread.

![Debug Trace](image)

**Figure 39**
*Debug trace output appears in test results*

**Task 3 – Modifying the Test to use WorkflowApplication**

Your test is good but it has one weakness. It verifies that the WorkflowThread returned is non-zero but it does not verify that it returns the actual managed thread ID that the workflow ran on. For example, your test would pass if the workflow always returned 1 in the WorkflowThread out argument.

If you want to verify the actual thread ID you will need to use **WorkflowApplication** to run the workflow. In this task you will modify the test to capture the thread of the workflow by obtaining it during a call to the **WorkflowApplication.Completed** action and then comparing that value to the value returned from the workflow.

1. Open **SayHelloFixture.cs** (C#) or **SayHelloFixture.vb** (VB) and locate the **ShouldReturnWorkflowThread** test. Modify it as shown to use the **WorkflowApplication** class to run the workflow. This code will use the **WorkflowApplication.Completed** action to capture the output arguments and thread ID.

(Code Snippet - Introduction to WF4 Lab - ShouldReturnWorkflowThread WFApp Test CSharp)

**C#**

```csharp
/// <summary>
/// Verifies that the workflow returns an Out Argument
/// Name: WorkflowThread
/// Type: Int32
```
/// Value: Non-Zero, matches thread used for Completed action
/// </summary>
[TestMethod]
public void ShouldReturnWorkflowThread()
{
    AutoResetEvent sync = new AutoResetEvent(false);
    Int32 actionThreadID = 0;
    IDictionary<string, object> output = null;

    WorkflowApplication workflowApp =
        new WorkflowApplication(
            new SayHello()
            {
                UserName = "Test"
            });

    // Create an Action<T> using a lambda expression
    // To be invoked when the workflow completes
    workflowApp.Completed = (e) =>
    {
        output = e.Outputs;
        actionThreadID = Thread.CurrentThread.ManagedThreadId;

        // Signal the test thread the workflow is done
        sync.Set();
    };

    workflowApp.Run();

    // Wait for the sync event for 1 second
    sync.WaitOne(TimeSpan.FromSeconds(1));

    Assert.IsNotNull(output,
        "output not set, workflow may have timed out");

    Assert.IsTrue(output.ContainsKey("WorkflowThread"),
        "SayHello must contain an OutArgument named WorkflowThread");

    // Don't know for sure what it is yet
    var outarg = output["WorkflowThread"];

    Assert.IsTrue(outarg is Int32,
        "WorkflowThread must be of type Int32");

    Assert.AreEqual(actionThreadID, (int)outarg,
        "WorkflowThread should equal actionThreadID");

    Debug.WriteLine("Test thread is " +
            Thread.CurrentThread.ManagedThreadId);
Debug.WriteLine("Workflow thread is " + outarg.ToString());
}

(Code Snippet - Introduction to WF4 Lab – ShouldReturnWorkflowThread WFApp Test VB)

Visual Basic

''' <summary>
''' Verifies that the SayHello workflow contains an Out Argument
''' Name: WorkflowThread
''' Type: Int32
''' Value: Non-Zero, matches thread used for Completed action
''' </summary>
''' <remarks></remarks>
<TestMethod()>
Public Sub ShouldReturnWorkflowThread()

Dim sync As New AutoResetEvent(False)
Dim actionThreadID As Int32 = 0
Dim output As IDictionary(Of String, Object) = Nothing
Dim workflowApp As New WorkflowApplication( _
    New SayHello() With {.UserName = "Test"})

workflowApp.Completed = _
    Function(e)
        output = e.Outputs
        actionThreadID = Thread.CurrentThread.ManagedThreadId

        ' Signal the test thread the workflow is done
        sync.Set()

        ' VB requires a lambda expression to return a value
        ' It is not used with Action(Of T)
        Return Nothing
    End Function

workflowApp.Run()

' Wait for the sync event for 1 second
sync.WaitOne(TimeSpan.FromSeconds(1))

Assert.IsNotNull(output, "output not set, workflow may have timed out")

Assert.IsTrue(output.ContainsKey("WorkflowThread"),
    "SayHello must contain an OutArgument named WorkflowThread")

' Don't know for sure what it is yet
Dim outarg = output("WorkflowThread")

Assert.IsInstanceOfType(outarg, GetType(Int32),
    "WorkflowThread must be of type Int32")
Assert.AreEqual(actionThreadID, DirectCast(output("WorkflowThread"), Integer), "WorkflowThread should equal actionThreadID")

Debug.WriteLine("Test thread is " & _ Thread.CurrentThread.ManagedThreadId)
Debug.WriteLine("Workflow thread is " & outarg.ToString())
End Sub

---

**Delegates, not events**

`WorkflowApplication.Completed` and other properties of `WorkflowApplication` are not events but delegates. Handling them requires you to provide a method, anonymous method or lambda expression.

---

**Next Step**

**Exercise 6: Verification**

Now you are ready to run the test and see if your activity passes the test with the new requirements:

1. Press **CTRL+SHIFT+B** to build the solution
2. Press **CTRL+R, A** to run all tests
3. Verify that all tests pass

![Test Results](image)

**Figure 40**

*All tests pass*
4. Double click on the ShouldReturnWorkflowThread test result to view the debug messages about the threads

![Debug Trace](image)

Figure 41
*The test results show the workflow ran on a different thread*

---

**Next Step**

Exercise 7: If/Else Logic

---

**Exercise 7: If/Else Logic**

In the previous exercise, you created an enhanced Hello Workflow application with a custom hello message. In this exercise, you will add If/Else logic to the workflow to display a different Hello message depending on a custom condition.

This exercise will use the "write the test first" approach, that is, first writing a test for the new requirement and then implementing the necessary code to make it pass.

**Task 0 – Opening the Solution**

To begin this exercise you can use the solution you finished from Exercise 6. Alternatively, you can follow the following steps to begin with Exercise 7.


2. Open the starting solution for Exercise 7 located under the \Source\Ex7-IfElse\Begin folder and use it as the starting point for this exercise.
3. Press **CTRL+SHIFT+B** to build the solution.

**Task 1 – Writing the Test for a New Requirement**

You have a new requirement for your application. If the name has an odd number of letters then you want the first word of the greeting to be "Greetings", otherwise the first word should be "Hello". As an example, now your workflow will be the equivalent of this code:

**C#**

```csharp
private static string SayHello(string userName)
{
    string FirstWord = null;
    if (userName.Length % 2 == 0)
        FirstWord = "Hello";
    else
        FirstWord = "Greetings";
    return FirstWord + ", " + userName + " from Workflow 4";
}
```

**Visual Basic**

```vbnet
Private Shared Function SayHello(ByVal userName As String) As String
    Dim FirstWord As String = Nothing

    If userName.Length Mod 2 = 0 Then
        FirstWord = "Hello"
    Else
        FirstWord = "Greetings"
    End If

    Return FirstWord & ", " & userName & " from Workflow 4"
End Function
```

1. Create a test to verify the new requirement. To do this, open the **SayHelloFixture.cs** file (**C#**) or **SayHelloFixture.vb** (**Visual Basic**), located under the HelloWorkflow.Test project, and add the following test.

(Code Snippet - Introduction to WF4 Lab - ShouldReturnGreetingWithOddLengthName Test CSharp)

**C#**

```csharp
[TestMethod]
public void ShouldReturnGreetingWithOddLengthName()
{
    var output = WorkflowInvoker.Invoke(
```
```csharp
    new SayHello() { UserName = "Odd" });

    string greeting = output["Greeting"].ToString();
    Assert.AreEqual("Greetings Odd from Workflow 4", greeting);
}
```

(Code Snippet - *Introduction to WF4 Lab - ShouldReturnGreetingWithOddLengthName Test VB*)

**Visual Basic**

```csharp
<TestMethod()>
Public Sub ShouldReturnGreetingWithOddLengthName()
    Dim output = WorkflowInvoker.Invoke(
        New SayHello() With {.UserName = "Odd"})
    Assert.AreEqual("Greetings Odd from Workflow 4", output("Greeting").ToString())
End Sub
```

2. Right-click over test method and select **Run Tests**. The test will fail because you have not modified the workflow yet to return a different hello message in each case.

![Test run failed](image)

**Figure 42**

*ShouldReturnGreetingWithOddLengthName test failing*

**Task 2 – Implementing the New Requirement in the Workflow**

1. Open *SayHello.xaml* in the workflow designer. To do this, double-click the file in Solution Explorer.

2. Add a **FirstWord** variable to store the first word of the hello message, whether it is "Hello" or "Greetings". To do this, follow these steps:
   a. Select the **Sequence** by clicking over the shape surface.
   b. Click the **Variables** button. A panel that displays the available variables for the **Sequence** activity appears.
   c. Click **Create Variable**.
   d. Type **FirstWord** in the Name box.
Adding the FirstWord variable of type String to Sequence

Variables

In Windows Workflow Foundation (WF), variables represent the storage of data. Arguments, on the other side, represent the flow of data into and out of an activity. Variables have a scope just as they do in C# or Visual Basic. If you open the variables pane without selecting an activity you will not be able to add a variable. The activity selected in the designer will provide the scope for the variable. In this case, FirstWord belongs to the Sequence scope.

3. Now you need to test the UserName argument to see if it has an even or odd number of characters. To do this, drag an If activity from the Control Flow group in the toolbox and drop it into the Sequence activity above the Assign activity.

4. Set the If activity DisplayName to If UserName is Even by selecting the text on the shape and editing it inline or by setting it in the property grid (select the activity and press F4 to display it).

Note: The Workflow designer allows you to give the shape a more readable name by setting its DisplayName property.
5. Supply an expression for the **If** condition. To do this, double-click the **If** activity to open it and type the following expression in the **Condition** box. This will check if the name length is odd or even.

**Visual Basic**

```vbnet
UserName.Length Mod 2 = 0
```

6. Update the **FirstWord** variable value for even length names, which should be greeted with a "Hello" message. Drag an **Assign** activity from **Primitives** group in the toolbox and drop it into the **Then** area. After that, type **FirstWord** in the **To** box (left side) and "Hello " (including the space at the end) in the **Value** box (right side).

7. Now update the **FirstWord** variable value for odd length names, which should be greeted with a "Greetings" message. Drag and **Assign** activity from **Primitives** group in the toolbox and drop it
into the Else area. Then type **FirstWord** in the **To** box (left side) and "**Greetings** " (including the space at the end) in the **Value** box (right side).

![If activity](image)

**Figure 45**
*The completed If activity*

8. Modify the final **Assign** activity to customize the hello message displayed based on the **FirstWord** value. To do this, in the **Assign** activity located below the **If** activity replace the content of the **Value** box (right side) with the following expression:

<table>
<thead>
<tr>
<th>Visual Basic</th>
</tr>
</thead>
</table>
| FirstWord & UserName _
& " from Workflow 4" |
9. Press **CTRL+SHIFT+B** to build the solution.

**Next Step**

Exercise 7: Verification

In order to verify that you have correctly performed all the steps of the exercise you will run the all the tests of the solution and ensure they pass.
1. From the menu select Test / Run / All Tests In Solution or press CTRL+R,A
2. Verify that all the tests pass.

![Figure 47](image-url)  
Tests pass

Next Step

**Exercise 8: Error Handling**

You may have noticed a potential bug in your simple application. What happens if you do not pass a UserName to the workflow? In this exercise, you will add some error handling capabilities to your workflow by using the Try/Catch, Catch<T> and Throw built-in activities.

**Task 0 – Opening the Solution**

To begin this exercise you can use the solution you finished from Exercise 7. Alternatively, you can follow the following steps to begin with Exercise 8.

2. Open the starting solution for Exercise 8 located under the Source\Ex8-ErrorHandling\Begin folder and use it as the starting point for this exercise.
3. Press CTRL+SHIFT+B to build the solution.

**Task 1 – Writing a Test to Observe Error Behavior**

Notice that if you pass an empty string (""") the application should function correctly. The only way to get a null name passed to the workflow is to create it without supplying the in argument for UserName.
1. Create a test to observe what happens when you do not pass the name argument to the workflow. To do this, using Solution Explorer, open `SayHelloFixture.cs (C#)` or `SayHelloFixture.vb (Visual Basic)` located under the `HelloWorkflow.Tests` project and add the following test.

(Code Snippet - Introduction to WF 4 Lab - ShouldHandleNullUserName Test CSharp)

```csharp
[TestMethod]
public void ShouldHandleNullUserName()
{
    // Invoking with no arguments
    WorkflowInvoker.Invoke(new SayHello());
}
```

(Code Snippet - Introduction to WF 4 Lab - ShouldHandleNullUserName Test VB)

```visualbasic
<TestMethod()>
Public Sub ShouldHandleNullUserName()

    ' Invoking with no arguments
    WorkflowInvoker.Invoke(New SayHello())

End Sub
```

2. Now run the test. Press Ctrl+R, T or Right-click over test method name and select Run Tests.

(Figure 48

Run the test

3. The result is that the test fails with a NullReferenceException because you have an expression that uses UserName.Length in the If activity and UserName was null.
Task 2 – Adding the Try/Catch Activity to the Workflow

To handle the error, you could validate the name argument prior to using it or you could simply catch the exception and deal with it. In this task, you will be catching the exception.

1. Open **SayHello.xaml** in the designer and drag a **TryCatch** activity from Toolbox and drop it at the top of the sequence.
Try/Catch Activity

Workflows can use the **TryCatch** activity to handle exceptions that are raised during the execution of a workflow. These exceptions can be handled or they can be re-thrown using the **Throw** activity. Activities in the **Finally** section are executed when either the **Try** section or the **Catches** section completes.

2. Now you need to move the **If** activity inside the **Try** block.
   a. Collapse the **If** activity (↑) to make the diagram smaller
   b. Drag-and-drop the **If** activity into the **Try** block.
3. Now you need to catch a `NullReferenceException`. To do this, follow these steps:
   a. Click on Add new catch.
   b. Select `System.NullReferenceException` from the combo box and press Enter

4. In the Catch section of the activity you need to decide how to handle the error. In this case, you are going to replace the exception with an `ArgumentNullException` and throw it. This lets the
caller know that the exception was their fault for not supplying the **UserName** argument. Drag a **Throw** activity from the Toolbox and drop it into the **Catches** area.

5. Set the **Throw** activity expression. To do this, select the **Throw** activity and in the Properties windows type the following expression in the **Exception** box.

**Visual Basic**

New ArgumentNullException("UserName")

![Figure 54](image)

*Adding a Throw activity*

6. Now you need to fix the **ShouldHandleNullUserName** test so that it expects this exception. To do this, open **SayHelloFixture.cs** and add an **ExpectedException** annotation to the test method as shown in the following code.

**C#**

```csharp
[TestMethod]
[ExpectedException(typeof(ArgumentNullException))]
public void ShouldHandleNullUserName()
{
    // Invoking with no arguments
    WorkflowInvoker.Invoke(new SayHello());
}
```

**Visual Basic**

```vbnet
<TestMethod(), ExpectedException(GetType(ArgumentNullException))>
Public Sub ShouldHandleNullUserName()
    ' Invoking with no arguments
    WorkflowInvoker.Invoke(New SayHello())
End Sub
```
Next Step

Exercise 8: Verification

1. Press **CTRL+SHIFT+B** to build the solution.
2. From the menu select **Test / Run / All Tests In Solution** or press **CTRL+R,A**
3. Verify that all the tests pass.

![Figure 55](image)

*Tests pass*

Next Step

Exercise 9: Activity Designers

Exercise 9: Custom Activities and Designers

As you have already seen, Windows Workflow allows you to build custom activities in code. Depending on your need, there are several base classes you can use.

<table>
<thead>
<tr>
<th>Base Class</th>
<th>Used For</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity</td>
<td>Activities that are composed of other activities</td>
</tr>
<tr>
<td>CodeActivity</td>
<td>Activities that want to control execution</td>
</tr>
<tr>
<td>AsyncCodeActivity</td>
<td>Activities that want to do async work during execution</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>NativeActivity</td>
<td>Activities that contain other activities or need advanced services from the workflow runtime</td>
</tr>
</tbody>
</table>

**AsyncCodeActivity**

Activities that want to do async work during execution

| NativeActivity | Activities that contain other activities or need advanced services from the workflow runtime |

**Task 0 – Opening the Solution**

To begin this exercise you can use the solution you finished from Exercise 8. Alternatively, you can follow the following steps to begin with Exercise 9.

1. Start **Microsoft Visual Studio 2010** from **Start | All Programs | Microsoft Visual Studio 2010**.
2. Open the starting solution for Exercise 9 located under **\Source\Ex9-ActivityDesigner\Begin** folder and use it as the starting point for this exercise.
3. Press **CTRL+SHIFT+B** to build the solution.

**Task 1 – Add Activity Designer Library**

1. Right click on the **HelloWorkflow** solution file and select **Add / New Project...**
2. Select the **Workflow** templates and choose **Activity Designer Library**. Name the project **HelloWorkflow.Activities**.

![Add New Project](image)

*Figure 56*

Add a new Activity Designer Library project named HelloWorkflow.Activities (C#)
Add a new Activity Designer Library project named HelloWorkflow.Activities (VB)

Why add an Activity Designer Library instead of an Activity Library?
If you intend to create designers using the Activity Designer Library project template is a better choice because it includes the necessary references to the WPF libraries in addition to the System.Activities and related libraries.

3. Delete ActivityDesigner1.xaml, you will not need it for this lab.

Task 2 – Create Composite Activity in XAML
Activities can be created by composing other activities. In this task, you will create a XAML activity that consists of other activities.

1. Right click on the HelloWorkflow.Activities project and select Add/New Item... and add a new activity using the following values

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installed Templates</td>
<td>Workflow</td>
</tr>
<tr>
<td>Template</td>
<td>Activity</td>
</tr>
<tr>
<td>Name</td>
<td>WriteTwoLines.xaml</td>
</tr>
</tbody>
</table>
2. **WriteTwoLines.xaml** will be opened in the designer. To complete it do the following
   a. From the Control Flow toolbox category drop a **Sequence** activity
   b. Add two In arguments of type string named **Line1** and **Line2**
   c. From the primitives group, drop two **WriteLine** activities
d. Set the first activity Text to **Line1** and the second to **Line2**

![Figure 60](image)

*The completed WriteTwoLines activity*

**Task 3 – Add Existing Activities**

In this task you will add some pre-built code activities

1. Add the existing **PrePostSequence** and **DiagnosticTrace** files. To do this, in Solution Explorer right-click the **HelloWorkflow.Activities** project, and select **Add / Existing Item**... browse to `Source\Assets` folder of this lab (choosing the folder that matches the language of your preference) and select both the **PrePostSequence** and **DiagnosticTrace** files.

   **What do these activities do?**

   The **PrePostSequence** activity is like a Sequence activity but also includes a Pre activity that will execute before the body and a Post activity that will execute after the body.

   The **DiagnosticTrace** activity outputs a message to **System.Diagnostics.Trace** as well as a **CustomTracking** record

2. Press **CTRL+SHIFT+B** to build the solution

3. The **HelloWorkflow** project will need a reference to the **HelloWorkflow.Activities** project. To add the reference, right click on the **HelloWorkflow** project and select Add Reference... in the projects tab select **HelloWorkflow.Activities** and click **OK** to add the reference. Then repeat and add another reference to the WPF library **PresentationFramework**
4. Open HelloWorkflow / SayHello.xaml in the designer and notice that the toolbox now contains the new activities.

Task 4 – Observe Custom Activities without Designers

In this task you will see how custom activities behave without a designer. Some can be used without a designer while others may benefit from one or may require one for intended functionality.

**Note:** You won’t be saving SayHello.xaml in this task – we will close it without saving at the end. The point is just to observe the behavior.

1. Drop the WriteTwoLines activity on the design surface after the last activity. Note that you can initialize the arguments for Line1 and Line2 using the property grid.

   ![Figure 61](image)

   *Initializing the arguments for WriteTwoLines using the property grid.*

   **Do I have to create an activity designer?**

   No, you don’t. For many activities, the property grid will be sufficient allowing developers to initialize arguments to your activity. However, adding a designer can improve the experience of using your custom activity making it easier to understand and use.

2. Drop the DiagnosticTrace activity on to the design surface after the last Assign activity. This activity is a little more complex with an enumerated value but you can still set the properties using the property grid. You will notice that this activity also provides some validation of the required argument "Text". This type of activity could benefit from a designer.
3. Drop the PrePostSequence activity on to the surface after the last Assign activity. The activity will work but there is no way to set the body Activities it will execute. You need to create a custom designer to make this activity fully functional.

4. Close SayHello.xaml without saving it.

Task 5 – Explore Activity Designers

In this task, you will add existing custom activity designers for the DiagnosticTrace and PrePostSequence activity to your project and explore how they work.

1. Add the existing DiagnosticTraceDesigner.xaml and PrePostSequenceDesigner.xaml files. To do this,
   a. Right click on the HelloWorkflow.Activities project and select Add/New Folder
   b. Name the folder Designers
c. Right-click the Designers folder, and select Add / Existing Item... browse to Source\Assets folder of this lab (choosing the folder that matches the language of your preference) and select both XAML files. Be sure to select the file type XAML Files in the dialog as shown.

![Add Existing Item - HelloWorkflow.Activities](image)

**Figure 64**
Add the existing XAML Designer files

2. The HelloWorld.Activities project Designers folder should have the two designers added.

![HelloWorkflow.Activities](image)

**Figure 65**
The HelloWorkflow.Activities folder with two activity designers

3. Next let’s take a tour of these designers and learn about how they work. Open DiagnosticTraceDesigner.xaml

4. Notice the DataTemplates – there is one for an Expanded view and one for a Collapsed view. The templates contain the body of your controls.
5. There is a lot to see in this designer but let’s focus on two things. The first is the **ExpressionTextBox**. This control allows the user to edit an `InArgument<T>` value with design time support for Intellisense (when used within Visual Studio) and compile checking of the expression. The DiagnosticTrace activity uses an `InArgument<string>` type for the Text property. Below you can see the Collapsed data template using the ExpressionTextBox.

**XAML**

```xml
<DataTemplate x:Key="Collapsed">
  <StackPanel Orientation="Horizontal">
    <!-- Show only the text when collapsed -->
    <TextBlock VerticalAlignment="Center" Margin="5">
      Text
    </TextBlock>
    <Sapv:ExpressionTextBox HintText="Text"
      Expression="{Binding Path=ModelItem.Text, Mode=TwoWay,
      Converter={StaticResource ArgumentToExpressionConverter},
      ConverterParameter=In }
      ExpressionType="s:String"
      OwnerActivity="{Binding Path=ModelItem}"
      Width="300"
      Margin="0,5"
      MaxLines="1"
    />
  </StackPanel>
</DataTemplate>
```

6. The next thing to note is found in the **Expanded** data template. It contains a ComboBox that will display the list of trace values. The DiagnosticTrace activity uses a CLR property named “Level” for this trace value. To bind to a CLR property you need to use `ModelItem.(property name)` and the `ModelToObjectValueConverter` as you see below.

**XAML**

```xml
<Style x:Key="ExpandOrCollapsedStyle" TargetType="{x:Type ContentPresenter}">
  <Setter Property="ContentTemplate" Value="%DynamicResource Collapsed"/>
  <Style.Triggers/>
</Style>
```

**Figure 66**

*The DataTemplates contain the body of the controls for the designer*
7. The PrePostSequence designer is more complex. Because it is a composite activity, it supports dropping other activities on the design surface. Open PrePostDesigner.XAML

8. This activity designer uses data binding to bind to the properties of the PrePostSequence class. The Pre and Post properties are a single activity so the designer uses WorkflowItemPresenter to enable a design surface for them which allows the user to drop a single activity.

XAML

```xml
<sap:WorkflowItemPresenter
    Margin="3"
    Items="{Binding Path=ModelItem.Pre, Mode=TwoWay}"
    HintText="Drop activity here"/>
```

9. The Activities collection uses the WorkflowItemsPresenter to create a design surface that can hold a collection of activities. In the XAML below you see how you can create a design surface that looks similar to the Sequence.

XAML

```xml
<sap:WorkflowItemsPresenter
    Margin="3"
    Items="{Binding Path=ModelItem.Activities}"
    HintText="Drop activities here">
    <sap:WorkflowItemsPresenter.SpacerTemplate>
        <DataTemplate>
            <Grid>
                <Path Fill="Red"
                      Width="20"
                      Height="30"
                      HorizontalAlignment="Center"
                      Data="M 0,10 l 20,0 l -10,10 Z"/>
            </Grid>
        </DataTemplate>
    </sap:WorkflowItemsPresenter.SpacerTemplate>
</sap:WorkflowItemsPresenter.ItemsPanel>
```
10. Press **CTRL+SHIFT+B** to build the solution

**Task 6 – Linking the Activity Designers to the Activity**

Now you have designers for your activities but they will not be used until you link the designers to your custom activities. There are two ways to do this, a declarative way and an imperative way. We will use the declarative approach.

1. Open **DiagnosticTrace.cs (C#)** and add the following namespace directives

   ```csharp
   using HelloWorkflow.Activities.Designers;
   ```

2. Add the **Designer** attribute to the class as shown

   ```csharp
   [Designer(typeof(DiagnosticTraceDesigner))]
   public sealed class DiagnosticTrace : CodeActivity
   {
   }
   ```

   **Visual Basic**

   ```vb
   <Designer(GetType(DiagnosticTraceDesigner))>
   Public NotInheritable Class DiagnosticTrace
   ```

3. Open **PrePostSequence.cs (C#)** and add the following namespace directives

   ```csharp
   using HelloWorkflow.Activities.Designers;
   ```

4. Add the **Designer** attribute to the **PrePostSequence** class.

   ```csharp
   [ContentProperty("Activities")]
   [Designer(typeof(PrePostSequenceDesigner))]
   public sealed class PrePostSequence : NativeActivity
   ```
Next Step

Exercise 9: Verification

Exercise 9: Verification

Task 1 – Prepare the test projects

1. Press **CTRL+SHIFT+B** to build the solution.
2. Add the existing **HelloWorkflow.Activities.Tests** project to your solution. To do this, right click on the Solution and select **Add/Existing Project...** browse to **Source\Assets** folder of this lab (choosing the folder that matches the language of your preference) and select the **HelloWorkflow.Activities.Tests** project
3. Right click on the **HelloWorkflow.Activities.Tests** project and select **Add Reference...** and add a project reference to **HelloWorkflow.Activities**
4. The **HelloWorkflow.Tests** project will also need a reference to the custom activity project. Right click on **HelloWorkflow.Tests** and select **Add Reference**. From the project tab select **HelloWorkflow.Activities**.
5. Press **Ctrl+R, A** to run all tests. All the tests should pass.

---

How do these Activity tests work?

Take a moment to explore the unit tests from **HelloWorkflow.Activities.Tests**. These tests create workflows using code instead of XAML. This is another option for testing workflows if you prefer to use it.

---

Task 2 – Create DiagnosticTrace Test

You need to create a test that verifies the runtime behavior of the **DiagnosticTrace** activity. There are two elements of the behavior you want to test for. The activity should emit a trace via **System.Diagnostics.Trace** and also a tracking record via the workflow tracking. You will use the classes provided in **HelloWorkflow.Activities.Tests** to provide a means for testing this behavior.

1. Right click on **HelloWorkflow.Tests** and select Add Reference... and add a project reference to **HelloWorkflow.Activities.Tests**
2. We will first write the test and then update the workflow. Add the following namespace directives to `SayHelloFixture.cs` (C#) or `SayHelloFixture.vb` (VB) file as shown.

**C#**

```csharp
using System.Activities.Tracking;
using HelloWorkflow.Activities.Tests;
```

**Visual Basic**

```vbnet
Imports System.Activities.Tracking
Imports HelloWorkflow.Activities.Tests
```

3. Add a new test to the `SayHelloFixture.cs` (C#) or `SayHelloFixture.vb` (VB) file as shown.

*(Code Snippet - Introduction to WF4 Lab – ShouldOutputVerboseTrace Test CSharp)*

**C#**

```csharp
[TestMethod()]
public void ShouldOutputVerboseTrace()
{
    // Arrange
    string testCategory = "Test";
    string expectedData = "From SayHello";
    string expectedTrace = string.Format("{0}: {1}", testCategory, expectedData);
    var listener = new TestTraceListener();
    var tracker = new TestTrackingParticipant();
    var profile = new TrackingProfile()
    {
        Name = "TestTrackingProfile",
        Queries =
        {
            new CustomTrackingQuery()
            {
                Name="Test",
                ActivityName = "*
            }
        }
    };

    tracker.TrackingProfile = profile;

    Trace.Listeners.Add(listener);

    var target = new SayHello()
    {
        UserName = "Test"
    };
```
```csharp
var workflow = new WorkflowInvoker(target);
workflow.Extensions.Add(tracker);

// Act
workflow.Invoke();

// Assert System.Diagnostics.Trace
Assert.AreEqual(1, listener.Traces.Count);
Assert.AreEqual(expectedTrace, listener.Traces[0]);

// Assert Tracking Records
Assert.AreEqual(1, tracker.Records.Count);
var customRecord = tracker.Records[0] as CustomTrackingRecord;
Assert.AreEqual(expectedData, customRecord.Data["Text"]);
Assert.AreEqual(testCategory, customRecord.Data["Category"]);
}
```

**Visual Basic**

```vbnet
<TestMethod()> Public Sub ShouldOutputVerboseTrace()
    ' Arrange
    Dim testCategory As String = "Test"
    Dim expectedData As String = "From SayHello"
    Dim expectedTrace As String = String.Format("{0}: {1}", testCategory, expectedData)
    Dim listener = New TestTraceListener()
    Dim tracker = New TestTrackingParticipant()
    Dim profile = New TrackingProfile() With {.Name = "TestTrackingProfile"
        profile.Queries.Add(New CustomTrackingQuery() With {.Name = testCategory, .ActivityName = "*"})
    tracker.TrackingProfile = profile
    Trace.Listeners.Add(listener)
    Dim target = New SayHello() With {.UserName = "Test"}
    Dim workflow = New WorkflowInvoker(target)
    workflow.Extensions.Add(tracker)

    ' Act
    workflow.Invoke()

    ' Assert System.Diagnostics.Trace
    Assert.AreEqual(1, listener.Traces.Count)
    Assert.AreEqual(expectedTrace, listener.Traces(0))

    ' Assert Tracking Records
```
4. Run the new test by right clicking on the test name `ShouldOutputVerboseTrace` and clicking on Run Tests. The test will fail because you have not added the DiagnosticTrace activity to the SayHello workflow yet.

**Task 3 – Add a DiagnosticTrace activity to the SayHello Workflow**

1. Open `HelloWorkflow / SayHello.xaml` in the workflow designer.
2. Drop a DiagnosticTrace activity at the bottom of the sequence and set the properties:
   a. Text: "From SayHello"
   b. Category: Test
   c. Trace Level:Verbose

   ![Figure 67](image)
   *Add the DiagnosticTrace activity*

3. Press Ctrl+R, A to run all tests. All the tests should now pass. If the debugger breaks on exceptions during the test this is expected – press F5 to continue.

**Task 4 – Add the PrePostSequence Test**
1. Next you will add a test to verify your PrePostSequence behavior. Once again you will write the test first and then update the workflow. Add a new test to the SayHelloFixture.cs (C#) or SayHelloFixture.vb (VB) file as shown.

(Code Snippet - Introduction to WF4 Lab – ShouldreturnPrePostMessages Test CSharp)

C#

[TestMethod]
public void ShouldReturnPrePostMessages()
{
    IDictionary<string, object> output;

    output = WorkflowInvoker.Invoke(new SayHello()
    {
        UserName = "Test"
    });

    Assert.AreEqual("This is Pre-Sequence Ordinal:1", output["PreMessage"]);
    Assert.AreEqual("PrePost is Cool Ordinal:2", output["PrePostBody"]);
    Assert.AreEqual("This is Post-Sequence Ordinal:3", output["PostMessage"]);
}

(Code Snippet - Introduction to WF4 Lab – ShouldreturnPrePostMessages Test VB)

Visual Basic

<TestMethod()>
Public Sub ShouldReturnPrePostMessages()

    Dim output As IDictionary(Of String, Object)

    output = WorkflowInvoker.Invoke(New SayHello() With {.UserName = "Test"})

    Assert.AreEqual("This is Pre-Sequence Ordinal:1", output("PreMessage"))
    Assert.AreEqual("PrePost is Cool Ordinal:2", output("PrePostBody"))
    Assert.AreEqual("This is Post-Sequence Ordinal:3", output("PostMessage"))

End Sub

2. Run the new test by right clicking on the test name ShouldReturnPrePostMessages and clicking on Run Tests. The test will fail because you have not added the PrePostSequence to SayHello yet.

3. Open HelloWorkflow / SayHello.xaml

4. Drop a PrePostSequence after the last activity, you should see the new PrePostSequence Activity Designer
5. To test that your activity is working you will add 3 new out arguments to hold a message from the Pre and Post areas of the **PrePostSequence**. Add **PreMessage**, **PrePostBody** and **PostMessage** as shown.

<table>
<thead>
<tr>
<th>Name</th>
<th>Direction</th>
<th>Argument type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserName</td>
<td>In</td>
<td>String</td>
<td>Enter a VB expression</td>
</tr>
<tr>
<td>Greeting</td>
<td>Out</td>
<td>String</td>
<td>Default value not supported</td>
</tr>
<tr>
<td>WorkflowThread</td>
<td>Out</td>
<td>Int32</td>
<td>Default value not supported</td>
</tr>
<tr>
<td>PreMessage</td>
<td>Out</td>
<td>String</td>
<td>Default value not supported</td>
</tr>
<tr>
<td>PostMessage</td>
<td>Out</td>
<td>String</td>
<td>Default value not supported</td>
</tr>
<tr>
<td>PrePostBody</td>
<td>Out</td>
<td>String</td>
<td>Default value not supported</td>
</tr>
</tbody>
</table>

Create Argument

---

**Figure 68**
Add the PreMessage, PrePostBody and PostMessage out arguments

6. To test that the activities are invoked in the correct order you will need a variable to store the ordinal. Add the **Ordinal** variable with a **default value of 1** to the **PrePostSequence** scope as shown.

<table>
<thead>
<tr>
<th>Name</th>
<th>Variable type</th>
<th>Scope</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ordinal</td>
<td>Int32</td>
<td>PrePostSequence</td>
<td>1</td>
</tr>
<tr>
<td>FirstWord</td>
<td>String</td>
<td>Sequence</td>
<td>Enter a VB expression</td>
</tr>
</tbody>
</table>

Create Variable

---

**Figure 69**
Add the Ordinal variable

7. Now you will store some values in the arguments you created that can be evaluated by the unit tests. Drop an **Assign** activity from the **Primitives** group on the **Pre-Activities** area and set the properties

   a. **DisplayName**: **Set the PreMessage**
   b. **To**: **PreMessage**
   c. **Value**: "This is Pre-Sequence Ordinal:" & Ordinal

8. Increment the **Ordinal** variable by dropping an **Assign** activity in the **Activities** area of the **PrePostSequence**.
a. DisplayName: **Increment the Ordinal**

b. To: **Ordinal**

c. Value: **Ordinal + 1**

9. Drop another **Assign** activity on the **Activities** area and set the properties

   a. DisplayName: **Set the PrePostBody**
   
   b. To: **PrePostBody**
   
   c. Value: "**PrePost is Cool Ordinal:** & Ordinal"

10. Copy the **Increment the Ordinal** activity by right clicking on it and select copy

11. Paste the activity below the **Set the PrePostBody** activity. You must right click on the Red arrow to paste the activity.

   ![Figure 70](image.jpg)

   **Copy then Paste the Increment Ordinal activity**

12. Drop another **Assign** activity on the Post-Activities area and set the properties

   a. DisplayName: **Set the PostMessage**
   
   b. To: **PostMessage**
   
   c. Value: "**This is Post-Sequence Ordinal:** & Ordinal"
Figure 71

The Completed PrePostSequence.

13. Press CTRL+SHIFT+B to build the solution

14. Press Ctrl+R, A to run all tests. All the tests should now pass.

Next Step

Exercise 10: Hosted Designer

Exercise 10: Hosted Designer

What if you wanted to create a custom activity like the PrePostSequence and have someone else who does not have Visual Studio use it? Many products allow end users to customize workflows. Windows
Workflow Foundation 4 allows you to host the designer in an application quite easily. In this exercise, you will host the designer and use your custom activities.

**Task 0 – Opening the Solution**

To begin this exercise you can use the solution you finished from Exercise 9. Alternatively, you can follow the following steps to begin with Exercise 10.

1. Start **Microsoft Visual Studio 2010** from **Start | All Programs | Microsoft Visual Studio 2010**.
2. Open the starting solution for Exercise 10 located under the **Source\Ex10-HostedDesigner\Begin** folder and use it as the starting point for this exercise.
3. Press **CTRL+SHIFT+B** to build the solution.

**Task 1 – Adding new WPF Application**

1. Right click on the **HelloWorkflow** solution and select **Add / New Project...**
2. From the **Windows** templates Add new WPF Application named **HelloDesigner**
3. Set the **HelloDesigner** project as the Startup project
4. Add references to the following assemblies
   1. **System.Activities.Presentation**
   2. **System.Activities.Core.Presentation**
   3. **System.Activities**
5. From the Projects tab, add the following assemblies:
   1. **HelloWorkflow.Activities**
   2. **HelloWorkflow.Activities.Designers**
6. Open **MainWindow.xaml** and modify it as shown

(Code Snippet - Introduction to WF4 Lab – MainWindow XAML CSharp)

**XAML (C#)**

```xml
<Window x:Class="HelloDesigner.MainWindow"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       Title="MainWindow" Height="600" Width="1000">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="4*" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Grid Name="grid1">
      <Grid.ColumnDefinitions>
        <Grid.ColumnDefinitions/>
    </Grid>
</Window>
```
7. Open MainWindow.xaml.cs (C#) or MainWindow.xaml.vb

8. Add the following namespace directives

(Code Snippet - Introduction to WF4 Lab – MainWindow Namespaces CSharp)

```csharp
using System.Activities.Presentation;
using System.Activities.Statements;
using System.Activities.Presentation.Toolbox;
using System.Activities.Core.Presentation;
using System.Activities.Presentation.Metadata;
using System.ComponentModel;
```
using HelloWorkflow.Activities;
using HelloWorkflow.Activities.Designers;

(Code Snippet - Introduction to WF4 Lab – MainWindow Namespaces VB)

Visual Basic

Imports System.Activities.Presentation
Imports System.Activities.Statements
Imports System.Activities.Presentation.Toolbox
Imports System.Activities.Core.Presentation
Imports System.Activities.Presentation.Metadata
Imports System.ComponentModel
Imports HelloWorkflow.Activities
Imports HelloWorkflow.Activities.Designers

9. Add a field of type WorkflowDesigner

C#

public partial class MainWindow : Window
{
    WorkflowDesigner workflowDesigner = new WorkflowDesigner();
}

Visual Basic

Class MainWindow
    Private workflowDesigner As WorkflowDesigner = New WorkflowDesigner()

10. Create a new function RegisterMetadata as shown. This function enables the designer metadata store.

(Code Snippet - Introduction to WF4 Lab – RegisterMetadata method CSharp)

C#

private void RegisterMetadata()
{
    DesignerMetadata metaData = new DesignerMetadata();
    metaData.Register();
    AttributeTableBuilder builder = new AttributeTableBuilder();
    MetadataStore.AddAttributeTable(builder.CreateTable());
}

(Code Snippet - Introduction to WF4 Lab – RegisterMetadata method VB)

Visual Basic

Class MainWindow
    Private workflowDesigner As WorkflowDesigner
    Private designerMetadata As New DesignerMetadata()
    Private attributeTableBuilder As New AttributeTableBuilder()
    Private MetadataStore As New MetadataStore()

    Public Sub RegisterMetadata()
        designerMetadata.Register()
        MetadataStore.AddAttributeTable(attributeTableBuilder.CreateTable())
    End Sub
Public Sub RegisterMetadata()
    Dim metaData As DesignerMetadata = New DesignerMetadata()
    metaData.Register()
    Dim builder As AttributeTableBuilder = New AttributeTableBuilder()
    MetadataStore.AddAttributeTable(builder.CreateTable())
End Sub

Note: When you host the designer you can control the toolbox. You choose what controls will appear, the categories they appear in and even the names of the controls. Add the CreateToolboxControl function as shown.

(Code Snippet - Introduction to WF4 Lab – CreateToolboxControl method CSharp)

C#

private ToolboxControl CreateToolboxControl()
{
    return new ToolboxControl()
    {
        Categories =
        {
            new ToolboxCategory("Hello Workflow")
            {
                Tools =
                {
                    new ToolboxItemWrapper(
                        "System.Activities, Version=4.0.0.0, Culture=neutral,
                        PublicKeyToken=31bf3856ad364e35",
                        null,
                        "Assign"),
                    new ToolboxItemWrapper(
                        "System.Activities, Version=4.0.0.0, Culture=neutral,
                        PublicKeyToken=31bf3856ad364e35",
                        null,
                        "Sequence"),
                    new ToolboxItemWrapper(
                        "System.Activities, Version=4.0.0.0, Culture=neutral,
                        PublicKeyToken=31bf3856ad364e35",
                        null,
                        "Try It"), // Can use a different name
                    new ToolboxItemWrapper(
                        "System.Activities.Statements.TryFinally",
                        "System.Activities, Version=4.0.0.0, Culture=neutral,
                        PublicKeyToken=31bf3856ad364e35",
                        null,
                        "Try Finally"),
                    new ToolboxItemWrapper(
                        "System.Activities, Version=4.0.0.0, Culture=neutral,
                        PublicKeyToken=31bf3856ad364e35",
                        null,
                        "Try Else"),
                }
            }
        }
    }
"HelloWorkflow.Activities.PrePostSequence",
"HelloWorkflow.Activities",
null,
"PrePostSequence"),

    new ToolboxItemWrapper(
"HelloWorkflow.Activities.DiagnosticTrace",
"HelloWorkflow.Activities",
null,
"Diagnostic Trace")
    }
}]
]
]

(Code Snippet - Introduction to WF4 Lab – CreateToolboxControl method VB)

**Visual Basic**

**Private Function** CreateToolboxControl() As ToolboxControl

'Create the ToolboxControl
    Dim ctrl As ToolboxControl = New ToolboxControl()

'Create a collection of category items
    Dim category As ToolboxCategory = New ToolboxCategory("Hello Workflow")

'Creating toolboxItems
    category.Add(New ToolboxItemWrapper(
        "System.Activities, Version=4.0.0.0, Culture=neutral,
        PublicKeyToken=31bf3856ad364e35",
        Nothing,
        "Assign")

    category.Add(New ToolboxItemWrapper(
        "System.Activities, Version=4.0.0.0, Culture=neutral,
        PublicKeyToken=31bf3856ad364e35",
        Nothing,
        "Sequence")

    category.Add(New ToolboxItemWrapper(
        "System.Activities, Version=4.0.0.0, Culture=neutral,
        PublicKeyToken=31bf3856ad364e35",
        Nothing,
        "Try It")') Can use a different name


'Adding the category to the ToolBox control.
ctrl.Categories.Add(category)
Return ctrl
End Function

11. Your Designer window will display the XAML of the workflow as it built. To do this you will need a function to handle the ModelChanged event. Add the ShowWorkflowXAML function as shown.

(Code Snippet - Introduction to WF4 Lab – ShowWorkflowXAML method CSharp)

C#

private void ShowWorkflowXAML(object sender, EventArgs e)
{
    workflowDesigner.Flush();
    textXAML.Text = workflowDesigner.Text;
}

Visual Basic

Private Sub ShowWorkflowXAML(ByVal sender As Object, ByVal e As EventArgs)
    workflowDesigner.Flush()
    textXAML.Text = workflowDesigner.Text
End Sub

12. Create the AddDesigner function as shown. This function will add the designer to your window

(Code Snippet - Introduction to WF4 Lab – AddDesigner method CSharp)

C#

private void AddDesigner()
{
    //Create an instance of WorkflowDesigner class
    this.workflowDesigner = new WorkflowDesigner();
//Place the WorkflowDesigner in the middle column of the grid
Grid.SetColumn(this.workflowDesigner.View, 1);

// Show the XAML when the model changes
workflowDesigner.ModelChanged += ShowWorkflowXAML;

//Load a new Sequence as default.
this.workflowDesigner.Load(new Sequence());

//Add the WorkflowDesigner to the grid
grid1.Children.Add(this.workflowDesigner.View);

// Add the Property Inspector
Grid.SetColumn(workflowDesigner.PropertyInspectorView, 2);
grid1.Children.Add(workflowDesigner.PropertyInspectorView);

// Add the toolbox
ToolboxControl tc = CreateToolboxControl();
Grid.SetColumn(tc, 0);
grid1.Children.Add(tc);

// Show the initial XAML
ShowWorkflowXAML(null, null);
}

(Code Snippet - Introduction to WF4 Lab – AddDesigner method VB)

Visual Basic

Private Sub AddDesigner()

'Create an instance of WorkflowDesigner class
workflowDesigner = New WorkflowDesigner()

'Place the WorkflowDesigner in the middle column of the grid
Grid.SetColumn(workflowDesigner.View, 1)

' Setup the Model Changed event handler
AddHandler workflowDesigner.ModelChanged, AddressOf ShowWorkflowXAML

'Load a new Sequence as default.
workflowDesigner.Load(New Sequence())

'Add the WorkflowDesigner to the grid

' Add the Property Inspector
13. Modify the constructor to call the functions you added

(Code Snippet - Introduction to WF Lab – MainWindow method CSharp)

C#

```csharp
public MainWindow()
{
    InitializeComponent();
    RegisterMetadata();
    AddDesigner();
}
```

(Code Snippet - Introduction to WF Lab – MainWindow method VB)

Visual Basic

```vbnet
Public Sub New()
    InitializeComponent()
    RegisterMetadata()
    AddDesigner()
End Sub
```

Next Step

Exercise 10: Verification

In this verification you will run the HelloDesigner Application and modify workflow

1. Right click on the HelloDesigner project and select Set as StartUp Project

2. Press F5 to launch the application in the debugger
3. When the design window appears drop a **DiagnosticTrace** and **PrePostSequence** on the designer surface.

4. You should see the following:

![Image of HelloDesigner application hosting the WorkflowDesigner](image)

**Figure 72**

*The HelloDesigner application hosting the WorkflowDesigner*

---

**Next Step**

**Summary**

---

**Summary**
In this Hands-On Lab you have learned the basics of workflow authoring in Windows Workflow Foundation 4 by creating a simple Windows Workflow application. During the lab you learned how to create a workflow using the new Visual Studio 2010 workflow designer, and as an alternative, you also learned how to create it using C# or VB code. You have explored the use of input and output arguments, seen how to add if/else logic to your workflow with the If activity and, finally, you have learned how to handle error conditions in your workflows by using exception-handling specific activities like the Try/Catch, Catch<T> and Throw activities. Of course, there is much more to Windows Workflow 4. You recommend you look at the Introduction to Workflow Services lab for more information about building Workflow enabled applications.

Feedback

What do you think of this lab? What do you think of the new Windows Workflow 4? Your feedback is important; it will help us build the best product for you. Please take a moment to provide it. Send your comments to wfwcfhol@microsoft.com