

Windows Workflow Foundation

Hands-On Lab

Lab Manual

Lab 05 – Communicating with the Workflow Host Application in C#

Information in this document, including URL and other Internet Web site references, is subject to change without notice. This document supports a preliminary release of software that may be changed substantially prior to final commercial release, and is the proprietary information of Microsoft Corporation.

This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, AS TO THE INFORMATION IN THIS DOCUMENT.

The entire risk of the use or the results from the use of this document remains with the user. Complying with all applicable copyright laws is the responsibility of the user.  Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document.  Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

© 2005 Microsoft Corporation.  All rights reserved.

Microsoft, MS-DOS, Windows, Windows NT, Windows Server, Visual C# and Visual Studioare either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

**Contents**

[Overview 2](#_Toc218633652)

[Objectives 2](#_Toc218633653)

[System Requirements 3](#_Toc218633654)

[Setup 3](#_Toc218633655)

[Physical Folder Structure 3](#_Toc218633656)

[Code Snippets 3](#_Toc218633657)

[Starting Material 3](#_Toc218633658)

[Acronyms Used in this Lab 3](#_Toc218633659)

[Scenario 3](#_Toc218633660)

[Exercise 1 – Communicating with a Local Service 4](#_Toc218633661)

[Task 1 – Create the Host Application 4](#_Toc218633662)

[Task 2 – Define the Local Service 5](#_Toc218633663)

[Task 3 – Define the Workflow 8](#_Toc218633664)

[Exercise 2 – Correlating Communication 18](#_Toc218633665)

[Task 1 – Enabling the Interface for Correlation 18](#_Toc218633666)

[Task 2 – Modifying the Workflow with an Additional Voter 21](#_Toc218633667)

[Exercise 3 – Roles and Security in Workflow 27](#_Toc218633668)

[Task 1 – Create a new Workflow Project 27](#_Toc218633669)

[Task 2 – Build the workflow 27](#_Toc218633670)

[Task 3 – Add an Exception Handler to workflow 29](#_Toc218633671)

[Task 4 – Building, Running and Validating the Roles Workflow 31](#_Toc218633672)

[Exercise 4 – Deferring a Long Running Operation to the Host 32](#_Toc218633673)

[Task 1 – Create a new Workflow Project 32](#_Toc218633674)

[Task 2 – Define our FactoringService 32](#_Toc218633675)

[Task 3 – Create a Workflow 35](#_Toc218633676)

[Lab Summary 42](#_Toc218633677)

# Overview

Estimated time to complete this lab: **80 minutes**

The project files for this lab are in the *C:\WF\WF 3.5 Labs\Lab05* folder.

For this lab we are using the **SQLProvider** for role management, before running the lab you must ensure that you have a configured database installed, we will be using the default database name ‘*aspnetdb*’.

To automatically create and configure the database run the **aspnet\_regsql.exe** tool found at:   
  
*\WINDOWS\Microsoft.NET\Framework\v2.0.50727\aspnet\_regsql.exe*   
  
Install the configured database in *localhost\SQLExpress* and select the default settings.

## Objectives

The objective of this lab is to introduce Windows Workflow Foundation messaging and local services. Messaging involves an event system implemented by Windows Workflow Foundation to be used by workflows and the host of the runtime. There are two main scenarios for how events are used.  First, there is the DoSomething method / SomethingDone event pattern where the first is a method invoked on the local service and the latter is an event raised back to state that the request has gone through and completed.  Typically, in this scenario, the DoSomething method which runs in the local service will start up a worker thread which then performs some work. This worker thread in turn will raise the event to indicate that the work has been completed.

Second, there is a more generic inbound event scenario.  Consider that DoSomething method takes an extended or unknown period to do the work, e.g. a document is assigned to someone to review.  In this scenario the host itself is a generic raiser of events.  The person assigned to review the document gets an e-mail with a document to review. The user reviews the document and clicks some button to indicate approval or rejection.  The host gets information from the button click that is associated with the document review, e.g. workflow instance ID, and uses this to do a reflection based method call that raises an event.

After completing this lab, you will be able to:

* Create a simple local service that exposes a method in which it does some work and also publishes an event which it fires when its work is complete.

Created a more complex workflow that uses correlated messaging to route message to the appropriate activity within a workflow.

Added role based authorization and security to a Workflow scenario

Created a simple Workflow that actions a long running process

More information about Windows Workflow Foundation can be found at <http://msdn.microsoft.com/workflow>

## System Requirements

* Microsoft Visual Studio 2008

## Setup

Unzip the lab to your local hard drive. Everything needed to complete the lab is in the zip file.

### Physical Folder Structure

File paths referenced in this lab assume the lab is installed in the following folder:

*C:\WF\WF 3.5 Labs\Lab[Number]*

Within the ***Lab[Number]*** folder, several child folders are available:

* ***CSharp*** – The lab written for C#
* ***VB*** – The lab written for VB
* ***resources*** –Any files referenced in the lab can be found in the Resources subdirectory, including source code for custom assemblies referenced in the exercises.

Within each *[Language]* folder, several child folders are available:

* ***before*** – The work area for completing the HOL
* ***after*** – The fully completed HOL

### Code Snippets

All code required for this lab consisting of more than 2 lines is available as code snippets. To learn more about code snippets including how to install them and how to use them, see the snippet guide document for the language of your choice in the folder:

*C:\WF\WF 3.5 Labs\Snippets*

## Starting Material

### Acronyms Used in this Lab

* WF – Windows Workflow Foundation

### Scenario

# Exercise 1 – Communicating with a Local Service

## Task 1 – Create the Host Application

1. Open Visual Studio 2005 by going to the **Start Menu | Programs | Microsoft Visual Studio 2005 | Microsoft Visual Studio 2005**
2. In Visual Studio 2005, select the **File | New | Project** menu command.
3. Select **Visual C# | Workflow |** **Sequential Workflow Console Application.** Change the Name to *CommunicationsWorkflow*.
4. Set the location to *C:\WF\WF 3.5 Labs\Lab05\* and click **OK** to create the solution.
5. Add **System.Windows.Forms** as a reference.
6. In the *Program.cs* file, set the using statements to only the following: (*NOTE: Make sure you don’t have duplicate ‘using’ statements after inserting the snippet.*)

(Snippet: “WFLab05\_Exercise01\_Task01\_UsingStatements”)

using System;

using System.Collections.Generic;

using System.Threading;

using System.Workflow.Runtime;

using System.Workflow.Activities;

1. In the *Program.cs* file, delete the code inside the *Program* class, and replace it with the following code. It is fundamentally the same as the generated template code but has additional functionality for displaying the creation and completion of a workflow instance.

(Snippet: “WFLab05\_Exercise01\_Task01\_ProgramClass”)

static AutoResetEvent waitHandle = new AutoResetEvent(false);

static VotingService votingService;

static void Main(string[] args)

{

WorkflowRuntime workflowRuntime = new WorkflowRuntime();

ExternalDataExchangeService dataService = new ExternalDataExchangeService();

workflowRuntime.AddService(dataService);

votingService = new VotingService();

dataService.AddService(votingService);

workflowRuntime.WorkflowCreated += new EventHandler<WorkflowEventArgs>(workflowRuntime\_WorkflowCreated);

workflowRuntime.WorkflowCompleted += new EventHandler<WorkflowCompletedEventArgs>(workflowRuntime\_WorkflowCompleted);

Type type = typeof(VotingWorkflow);

workflowRuntime.CreateWorkflow(type).Start();

waitHandle.WaitOne();

Console.WriteLine("Press any key to exit...");

Console.Read();

}

static void workflowRuntime\_WorkflowCompleted(object sender, WorkflowCompletedEventArgs e)

{

Console.WriteLine("Workflow " + e.WorkflowInstance.InstanceId.ToString() + " completed.");

waitHandle.Set();

}

static void workflowRuntime\_WorkflowCreated(object sender, WorkflowEventArgs e)

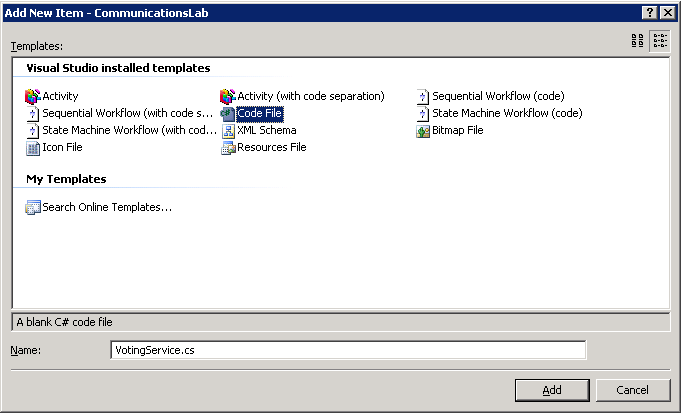
{

Console.WriteLine("Workflow " + e.WorkflowInstance.InstanceId.ToString() + " created.");

}

## Task 2 – Define the Local Service

1. Add a code file to the project by right clicking the project and choosing **Add | New Item**
2. Choose a **Code File** and name the file *VotingService.cs*



1. *VotingService.cs* will be a blank file. Paste the following code or use the snippet to define the message, interface, and local service used by the workflow. The *VotingEventArgs* class defines the message passed between the service and the workflow and stores the instance id of the workflow as well as the alias of the person that is voting on the proposal. The *IVotingService* is an interface that defines how the service and workflow communicate. The workflow calls the service to create a new ballot using the *CreateBallot* method; the service raises the events into the workflow to indicate whether the person approved or rejected the proposal.

(Snippet: “WFLab05\_Task02\_Exercise01\_VotingService”)

using System;

using System.Threading;

using System.Workflow.ComponentModel;

using System.Workflow.Runtime;

using System.Workflow.Activities;

using System.Windows.Forms;

namespace CommunicationsWorkflow

{

[Serializable]

public class VotingEventArgs : ExternalDataEventArgs

{

private string alias;

public VotingEventArgs(Guid InstanceID, string alias) : base(InstanceID)

{

this.alias = alias;

}

public string Alias

{

get { return this.alias; }

set { this.alias = value; }

}

}

[ExternalDataExchange]

internal interface IVotingService

{

event EventHandler<VotingEventArgs> ApproveProposal;

event EventHandler<VotingEventArgs> RejectProposal;

void CreateBallot(string alias);

}

internal class VotingService : IVotingService

{

public event EventHandler<VotingEventArgs> ApproveProposal;

public event EventHandler<VotingEventArgs> RejectProposal;

public void CreateBallot(string alias)

{

Console.WriteLine("Ballot created for " + alias + ".");

ThreadPool.QueueUserWorkItem(ShowVotingDialog, new VotingEventArgs(WorkflowEnvironment.WorkflowInstanceId, alias));

}

public void ShowVotingDialog(object o)

{

DialogResult result;

VotingEventArgs votingEventArgs = o as VotingEventArgs;

Guid instanceId = votingEventArgs.InstanceId;

string alias = votingEventArgs.Alias;

result = MessageBox.Show("Approve Proposal, " + alias + "?", alias + " Ballot", MessageBoxButtons.YesNo);

if (result == DialogResult.Yes)

{

if (ApproveProposal != null)

ApproveProposal(null, votingEventArgs);

}

else

{

if (RejectProposal != null)

RejectProposal(null, votingEventArgs);

}

}

}

}

## Task 3 – Define the Workflow

The workflow defines the voting logic for a proposal. For this workflow all proposals must be approved by Jim and the workflow manages when a ballot is created and completed. It creates a ballot for Jim and then waits for either an approval or rejection of the proposal.

1. **Rename** the *Workflow1.cs* file to *VotingWorkflow.cs* by right clicking on the filename and choosing **Rename**.
2. This will also rename the class within the file and you will see a dialog confirming this.

**Note:** Since we have not built the solution yet you *may* get a dialog stating the solution does not build in its current state, press Cancel to continue.

1. Right click the *VotingWorkflow.cs* file and choose **View Code**. Copy and paste the highlighted code below into the *VotingWorkflow* class.

(Snippet: “WFLab05\_Exercise01\_Task03\_OnRejected”)

public sealed partial class VotingWorkflow: SequentialWorkflowActivity

{

public VotingWorkflow()

{

InitializeComponent();

}

private void OnRejected(object sender, ExternalDataEventArgs e)

{

Console.WriteLine("Proposal Rejected");

}

private void OnApproved(object sender, ExternalDataEventArgs e)

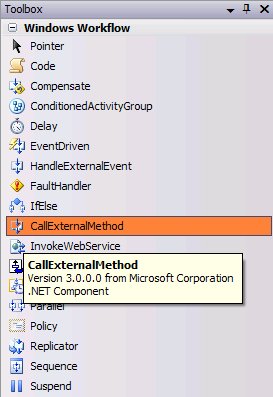
{

Console.WriteLine("Proposal Approved");

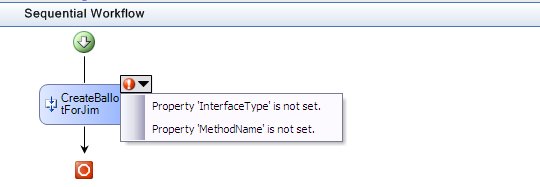
}

}

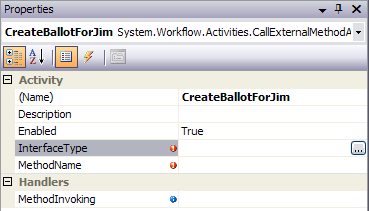
1. Double click the *VotingWorkflow.cs* file to bring up the workflow designer.
2. The ballot is created by calling the *VotingService.CreateBallot* method. To call a method on a local service from a workflow choose the **CallExternalMethodActivity** activity from the **Toolbox** and drag it on to the workflow.



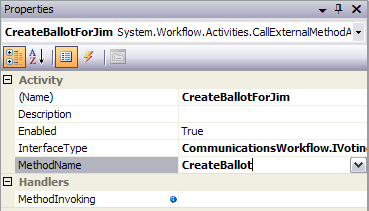
1. Change the Name of the **CallExternalMethodActivity** activity from *callExternalMethodActivity1* to *CreateBallotForJim*



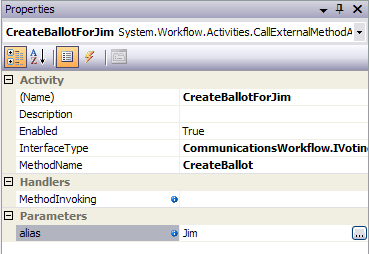
1. Note, the activity’s validation logic is indicating we need to set several of the activity properties. To call a method on a local service we first need to set the interface property. Choose the ellipsis (…) next to **InterfaceType** and choose the ***IVotingService*** interface in the project.



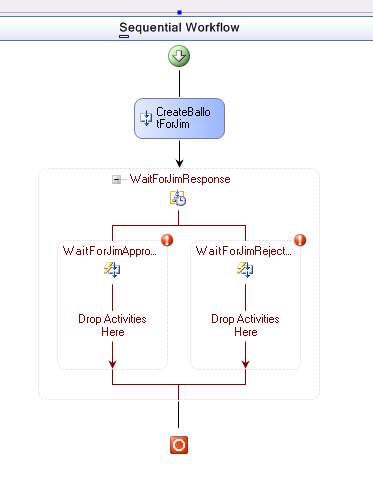
1. Once an interface is selected, the method to invoke can be set. To do this, choose the *CreateBallot* method name for the **MethodName** property.



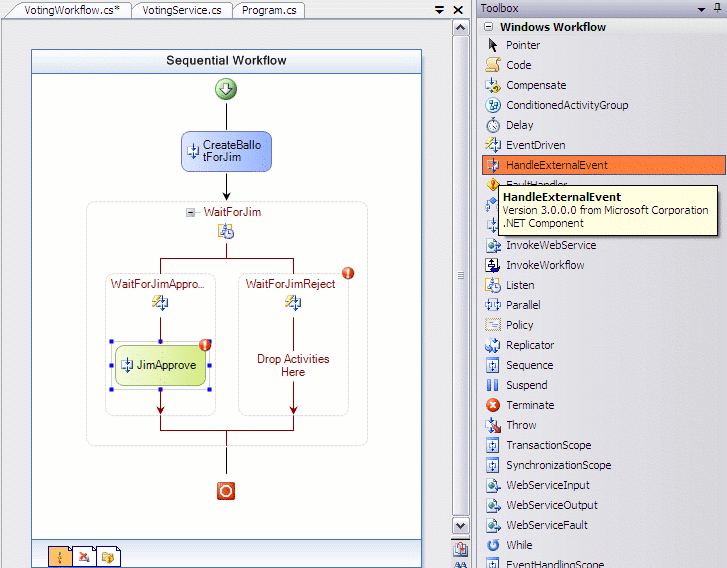
The *CreateBallot* method takes a single string parameter called **alias**. Type in *Jim* for the value of this property.



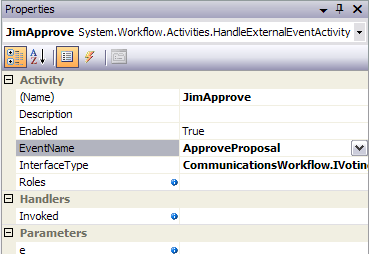
1. The workflow now will create a ballot for Jim but it needs to get the response back. To do this, add a **Listen** activity after the **CallExternalMethodActivity** activity, name it *WaitForJimResponse*. Rename the left branch of the **Listen** activity to *WaitForJimApproval* and the right branch to *WaitForJimRejection*. The **Listen** activity will wait for a response from one inbound event before proceeding. In this workflow it will wait for the ballot response from Jim – i.e. to approve or reject the proposal.



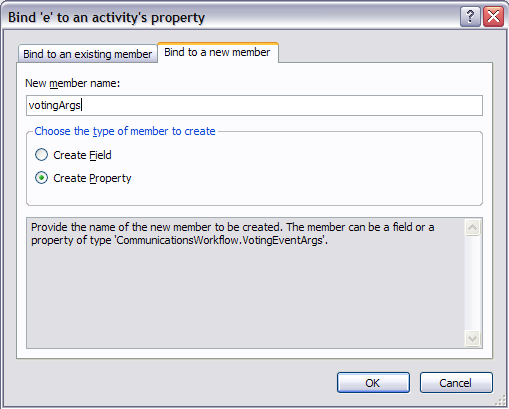
1. The **Listen** activity needs to be configured with **HandleExternalEvent** activities. Add a **HandleExternalEvent** activity to the left branch by dragging and dropping the **HandleExternalEvent** activity to the left branch. Rename this **HandleExternalEvent** activity to *JimApprove*.



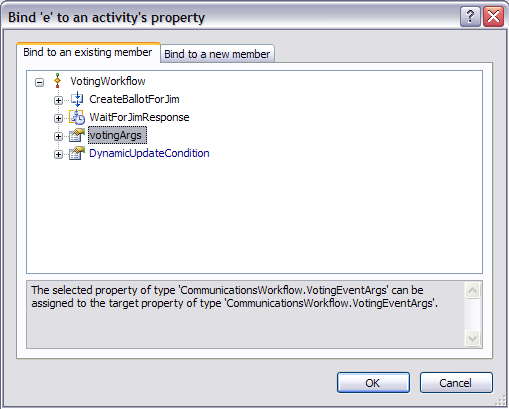
1. In order for the workflow to get a message from a local service it must be bound to an event defined by an interface. This is done in a similar fashion as the **CallExternalMethodActivity** activity. Set the **InterfaceType** property to *IVotingService* and set the **EventName** to *ApproveProposal* by choosing *ApproveProposal* from the dropdown menu.



1. For the **e** property select the property and click the ellipsis button next to it to bring up the activity binding interface. Select the **Bind to a new member** tab as shown below:



1. Enter *votingArgs* in the **New member name** box and click **OK.**
2. To view progress as the workflow executes we will have the *OnApproved* method called after an approval event is raised. To do this, select the **Invoked** parameter and type in *OnApproved*.
3. Add a **HandleExternalEvent** activity to the right branch calling it *JimReject*. Set its **InterfaceType** to *IVotingService* and **EventName** to *RejectProposal*.
4. Set the **Invoked** parameter to *OnRejected*, and set the **e** value to *votingArgs* by clicking on the ellipsis and using the activity binding interface to select *VotingWorkflow.votingArgs* as shown below:



1. Modify the *OnApproved* and *OnRejected* methods to include the name alias value.

public sealed partial class VotingWorkflow: SequentialWorkflowActivity

{

public VotingWorkflow()

{

InitializeComponent();

}

private void OnRejected(object sender, ExternalDataEventArgs e)

{

Console.WriteLine(string.Format("Proposal Rejected by {0}",

votingArgs.Alias));

}

private void OnApproved(object sender, ExternalDataEventArgs e)

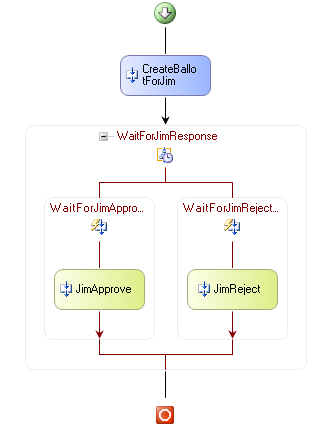
{

Console.WriteLine(string.Format("Proposal Approved by {0}",

votingArgs.Alias));

}

1. The completed Workflow should look like the following:



1. Compile the solution and run.
2. Approve the proposal when the **MessageBox** pops up.
3. The output should look something like the following:

**Workflow 272443dd-40fe-4806-aea1-aaa65c87641c created.**

**Ballot created for Jim.**

**Proposal Approved by Jim**

**Workflow 272443dd-40fe-4806-aea1-aaa65c87641c completed.**

**Press any key to exit...**

## 

# Exercise 2 – Correlating Communication

In this exercise we will add an additional voter to the process. Both voters will get their ballots in parallel and both must vote on the proposal. The ballot creation and voting process is exactly the same as above and uses the same local service, interface, methods, and events thus it is necessary to correlate the responses between the local service and the workflow. For this workflow this means that the Jim responses from the local service go to Jim’s **HandleExternalEvent** activities and Alice’s response to Alice’s. This is done by decorating the interface with the correlation key information and adding a correlation reference object for each voter. This object is referenced by the corresponding **CallExternalEvent** and **HandleExternalEvent** activities.

## Task 1 – Enabling the Interface for Correlation

1. Add a new interface called *IVotingServiceCorrelated* in the *VotingService.cs* file just at the top of the namespace:

(Snippet: “WFLab05\_Exercise02\_Task01\_IVotingServiceCorrelated”)

[ExternalDataExchange]

[CorrelationParameter("alias")]

internal interface IVotingServiceCorrelated

{

[CorrelationAlias("alias", "e.Alias")]

event EventHandler<VotingEventArgs> ApproveProposal;

[CorrelationAlias("alias", "e.Alias")]

event EventHandler<VotingEventArgs> RejectProposal;

[CorrelationInitializer]

void CreateBallot(string alias);

}

1. Change the inheritance of the *VotingService* so that it inherits from *IVotingServiceCorrrelated*.

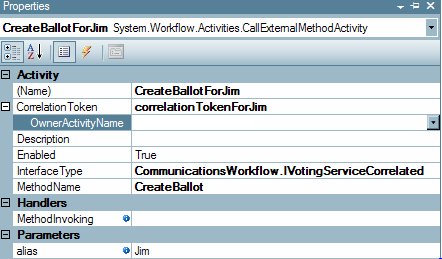
internal class VotingService : IVotingServiceCorrelated

{

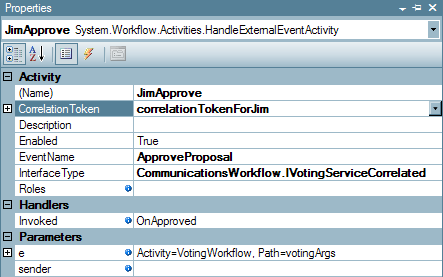
…

}

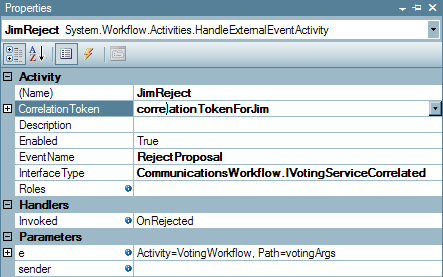
1. We will now have to update our workflow to reflect the change from using the *IVotingService* interface to the *IVotingServiceCorrellated* interface.
2. In the workflow designer, select the ***CreateBallotForJim*** activity.
   1. Change the **InterfaceType** to *IVotingServiceCorrelated*
   2. Set the **MethodName** to *CreateBallot*
   3. Set the **alias** to *Jim*
   4. Set the **CorrelationToken** to *correlationTokenForJim*.



1. Select the **JimApprove** activity
   1. Change **InterfaceType** to *IVotingServiceCorrelated.*
   2. Set **EventName** to *ApproveProposal.*
   3. Set **e** to *votingArgs* by clicking the ellipsis and using the activity binding interface.
   4. Set **CorrelationToken** to *correlationTokenForJim* using the drop down menu.

**

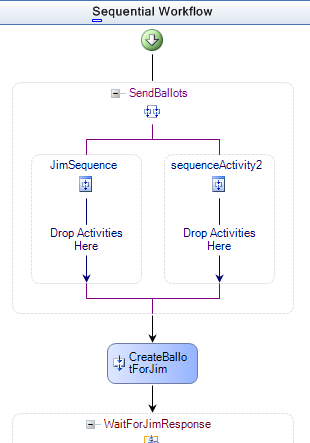
1. Select the **JimReject** activity
   1. Change the **InterfaceType** to *IVotingServiceCorrelated*
   2. Set the **EventName** to *RejectProposal*
   3. Set **e** to *votingArgs* by clicking the ellipsis and using the activity binding interface.
   4. Set the **CorrelationToken** to *correlationTokenForJim*

**

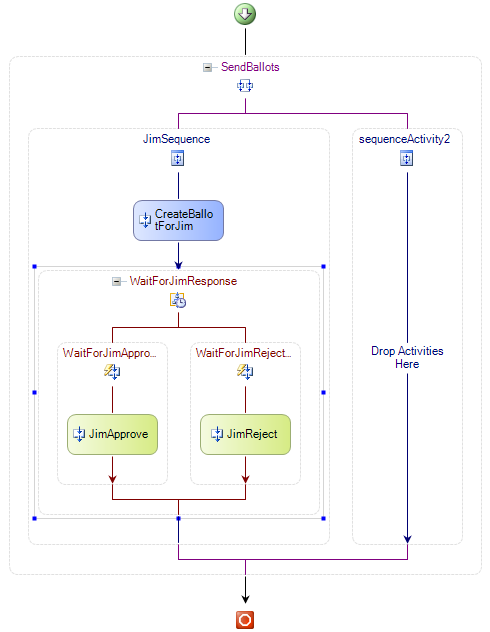
1. You will notice the workflow designer has errors on each of the activities, ignore these for now as we will correct them shortly.

## Task 2 – Modifying the Workflow with an Additional Voter

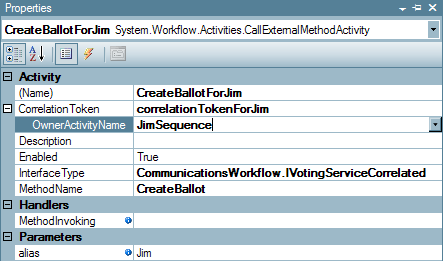
1. In order to get both ballots for the proposal to execute in parallel, one for Jim and a second for Alice, we need to create parallel execution tracks for creating the ballot and listening for approval/rejection events. To do this first start by creating a Parallel activity at the root of the workflow. **Rename** the *parallelActivity1* activity to *SendBallots* and the *sequenceActivity1* activity in the parallel activity to *JimSequence*.



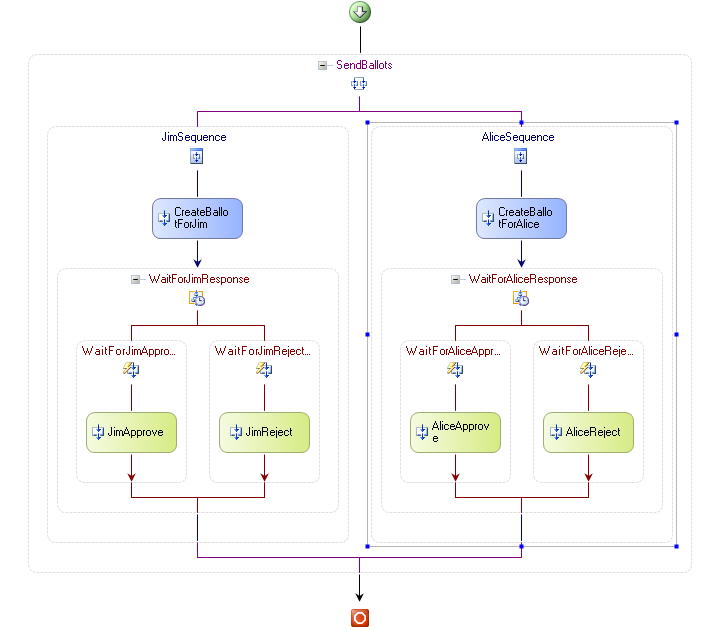
1. Next, move the activities associated with Jim’s ballot into the left branch of the parallel activity by dragging and dropping the **CallExternalMethod** and **Listen** activities into the left branch.



1. For all three activities for Jim (*CreateBallotForJim*, *JimApprove* and *JimReject*) set the **OwnerActivityName** property (under **CorrelationToken**) to *JimSequence*.



1. Rename *sequenceActivity2* to *AliceSequence***,** who is the second voter.
2. Drag on a **CallExternalMethod** and **Listen** activity from the **Toolbox** for Alice.
3. Name the CallExternalMethod activity *CreateBallotForAlice* and the listen activity *WaitForAliceResponse*. Name the left sequence activity *WaitForAliceApproval* and the right sequence activity *WaitForAliceRejection*.
4. For the CreateBallotForAlice activity
   1. Set the **InterfaceType** to the *IVotingServiceCorrelated* interface
   2. Set the **MethodName** to *CreateBallot*.
   3. Set the **alias** parameter to *Alice*
   4. Set the **CorrelationToken** to *correlationTokenForAlice*
   5. Set the **OwnerActivityName** to *AliceSequence* from the drop-down list.
5. Add two **HandleExternalEvent** activities for Alice in the **Listen** activity and name them *AliceApprove* and *AliceReject*.
6. Also for each **HandleExternalEvent** activity
   1. Set the **InterfaceType** to the *IVotingServiceCorrelated* interface
   2. Set the **EventName** for each to *ApproveProposal* and *RejectProposal*
   3. Set each **e** to *votingArgs* using the activity binding interface.
   4. Set each **CorrelationToken** to *correlationTokenForAlice*.
   5. Set the **Invoked** property for each to *OnApproved* and *OnRejected*.
7. The completed workflow is below.



1. Compile the solution and run.
2. Use the **MessageBoxes** that pop up to accept or reject each proposal. You can test that the service is correctly separating the correlation tokens by approving one and rejecting the other.
3. The output should look something like the following:

**Workflow e9001953-0b08-4005-8105-47cd0b15a003 created.**

**Ballot created for Jim.**

**Ballot created for Alice.**

**Proposal Approved by Alice**

**Proposal Approved by Jim**

**Workflow e9001953-0b08-4005-8105-47cd0b15a003 completed.**

**Press any key to exit...**

## 

# Exercise 3 – Roles and Security in Workflow

Windows Workflow Foundation supports the execution of activities based on the membership of a user in a given role. In this exercise, you will add role based authorization to a Workflow scenario. The Workflow scenario that is illustrated is the initiation of a purchase order. You will create a Workflow which requests a purchase order. The initiator will only be able to raise a purchase order if they are part of the Role that is configured for the **HandleExternalEvent** activity.

You will create a Workflow which has two activities. The first is a code activity which creates a Role of valid PO Initiators. The second activity is a **HandleExternalEvent** activity which is provided from the resources folder. The **HandleExternalEvent** activity is created with one interface to receive an event raised by the main program. The main program raises this event to request the initiation of a Purchase Order. You will have to ensure that the Role of valid PO Initiators includes the user name you specify.

If the user is part of the role then the Workflow completes successfully. If the user is not part of role then an exception is thrown by the **HandleExternalEvent** activity that is caught by the Workflow exception handler.

## Task 1 – Create a new Workflow Project

1. Copy the *Exercise3* folder from the resources directory.

**From source directory**: *C:\WF\WF 3.5 Labs\Lab05\Resources\Exercise3*

**To destination directory:** *C:\WF\WF 3.5 Labs\Lab05\Exercise3*

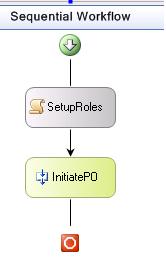
1. Open the solution file **PurchaseOrderWorkflow.sln** under *C:\WF\WF 3.5 Labs\Lab05\Exercise3*
2. The *POInterface* project defines the interface used for communicating information from the host. The **HandleExternalEvent** activity that is in the workflow uses the *POInterface* defined in this project

## Task 2 – Build the workflow

1. Build the solution so the *POInterface* and the *PurchaseOrderWorkflow* projects get built.
2. Select the designer view by double-clicking *PurchaseOrderWorkflow.cs* in the **Solution Explorer**
3. Add a **Code** activity to the workflow by dragging and dropping one from the **Toolbox**.
4. **Rename** the *codeActivity1* activity to *SetupRoles* in the **Properties** windows for this activity.
5. Create an **ExecuteCode** handler called *OnSetupRoles* for the *SetupRoles* **Code** activity by typing *OnSetupRoles* for **ExecuteCode** property
6. Select the designer view by double-clicking *PurchaseOrderWorkflow.cs* in the **Solution Explorer**.
7. Add a **HandleExternalEvent** Activity to the workflow and drag it under the *SetupRoles* Activity of the workflow
8. Select the properties windows of *handleExternalEventActivity1*activity
9. Set the properties as follows:
   1. Set the **Name** property to *InitiatePO*
   2. Set the **InterfaceType** property to *IStartPurchaseOrder*under the referenced POInterface assembly
   3. Set the **EventName** property to *InitiatePurchaseOrder*by selecting from the drop-down list
   4. Set the **Roles** property to *poInitiators*by clicking the ellipsis and using the activity binding interface to bind to *poInitiators*.
   5. Set the **Invoked** Handler to *OnInitiatePO* by typing the value in. The handler *OnInitiatePO* will be created in *PurchaseOrderWorkflow.cs* file. Add the following line to this method:

Console.WriteLine("Purchase Order initiated successfully");

1. Now your workflow should look like the following:



1. Add the following code in the *OnSetupRoles* method (or insert using the code snippet).

(Snippet: “WFLab05\_Exercise03\_Task02\_POInitiators”)

WebWorkflowRole poInitiatorsRole = new WebWorkflowRole("Clerk");

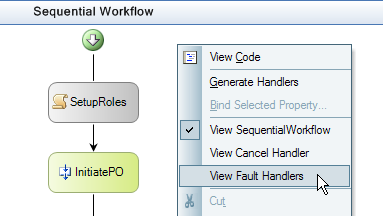
// Add the role to the RoleCollection representing the POInitiators

poInitiators.Add(poInitiatorsRole);

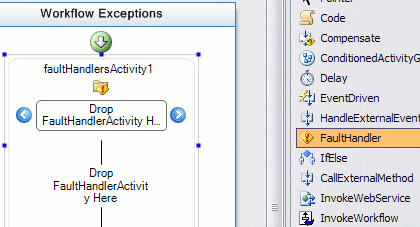
### 

## Task 3 – Add an Exception Handler to workflow

1. Select the workflow and right click **View Fault Handlers** to view the Fault Handlers.



1. Select and drag a **Fault Handler** into the workflow. It will be named as *faultHandlerActivity1*.



1. Select the properties windows for the *faultHandlerActivity1*and for the **FaultType** property for *faultHandlerActivity1*, specify *System.Workflow.Activities.WorkflowAuthorizationException*
2. Select and drag a **Code** Activity into the *faultHandlerActivity1*. **Rename** the *codeActivity1* activity that appears in **Workflow Exceptions** view to *ExceptionHandlerCode*
3. Type in *AuthExceptionHandler* for the **ExecuteCode** property. This will create the code handler for this activity named as *AuthExceptionHandler*
4. Add the following line to the method *authExceptionHandler*

Console.WriteLine("Exception message: {0}", faultHandlerActivity1.Fault.Message.ToString());

### 

## Task 4 – Building, Running and Validating the Roles Workflow

1. Save the solution and build it.
2. For this sample we are using the **SQLProvider** for role management, before running the example you must ensure that you have a configured database installed, we will be using the default database name ‘*aspnetdb*’. To automatically create and configure the database run the aspnet\_regsql.exe tool found at:   
     
   *\WINDOWS\Microsoft.NET\Framework\v2.0.50727\aspnet\_regsql.exe*   
     
   Install the configured database in localhost\SQLExpress and select the default settings, If you change the name of the database you will have to update the connection string found the app.config.
3. Run the workflow by pressing **F5.**
4. You should see the following:

**Purchase Order initiated by: Amanda**

**Purchase Order initiated successfully**

1. Press **Enter** to exit the program.
2. Modify the name *“Amanda”* in the *SendPORequestMessage()* method in *Program.cs* to *“William”*
3. Build and run the workflow by pressing **F5.**

You should see the following:

**Purchase Order initiated by: William**

**Exception message: Authorization failed for message received for event activity InitiatePO, from principal William**

1. Press **Enter** to exit the program.

The request failed for the user William because he was not a member of the authorized role.

# Exercise 4 – Deferring a Long Running Operation to the Host

In this exercise we will use the workflow to raise an event to the host application to act as a long running process. The workflow will run parallel activities, one executing a short 5 second delay and the other executing a long running service, factoring the prime numbers found in 100000000. Once both parallel activities have completed the workflow will finish.

## Task 1 – Create a new Workflow Project

1. Open Visual Studio 2005 by going to the **Start Menu | Programs | Microsoft Visual Studio 2005 | Microsoft Visual Studio 2005**
2. In Visual Studio 2005, select the **File | New | Project** menu command.
3. Select **Visual C# | Workflow |** **Sequential Workflow Console Application.** Change the Name to *LongRunningWorkflow*.
4. Set the location *C:\WF\WF 3.5 Labs\Lab05\.*

## Task 2 – Define our FactoringService

1. Next we will define our interface, local service and custom **ExternalDataEventArgs** classes. Add a **Code file** to the project by right clicking the project and choosing **Add | New Item**
2. Choose a **Code File** and name the file *FactoringService.cs*
3. *FactoringService.cs* will be a blank file. Copy and paste the following using statements and namespace declaration into the *FactoringService.cs* file.

(Snippet : “WFLab05\_Exercise04\_Task02\_UsingStatements”)

using System;

using System.Collections;

using System.Collections.Generic;

using System.Text;

using System.Threading;

using System.Workflow.Activities;

using System.Workflow.Runtime;

namespace LongRunningWorkflow

{

}

1. Next we will add our custom **ExternalDataEventArgs** class which we will call *PrimeFactoringEventArgs*.

Copy and paste or use the code snippet provided to add our *PrimeFactoringEventArgs* class within the namespace declaration.

(Snippet : “WFLab05\_Exercise04\_Task02\_EventArgs”)

[Serializable]

public class PrimeFactoringEventArgs : ExternalDataEventArgs

{

private int primeCount;

public int PrimeCount

{

get { return primeCount; }

}

public PrimeFactoringEventArgs(Guid instanceId, int primeCount) : base(instanceId)

{

this.primeCount = primeCount;

}

}

1. Now we will add our interface *IPrimeFactoring*, cut and paste the following code or use the code snippet provided to add *IPrimeFactoring* within the namespace declaration.

(Snippet: “WFLab05\_Exercise04\_Task02\_IPrimeFactoring”)

[ExternalDataExchange]

public interface IPrimeFactoring

{

void FactorPrimes();

event EventHandler<PrimeFactoringEventArgs> FactoredPrimes;

}

1. Finally we will add our implementation of the *IPrimeFactoring* interface. This service will be called from the workflow and will begin factor prime numbers; once the process is complete it will raise an event to inform the workflow that it has completed. Cut and paste or use the code snippet provided to add the *FactoringService* class to *FactoringService.cs*, remember to include the class within the namespace declaration.

(Snippet: “WFLab05\_Exercise04\_Task02\_FactoringService”)

[Serializable]

public class FactoringService : IPrimeFactoring

{

#region IPrimeFactoring Members

public void FactorPrimes()

{

ThreadPool.QueueUserWorkItem(FactorPrimes, WorkflowEnvironment.WorkflowInstanceId);

}

public event EventHandler<PrimeFactoringEventArgs> FactoredPrimes;

#endregion

private void FactorPrimes(object instanceId)

{

Console.WriteLine("Beginning Factoring Prime Numbers");

DateTime start = DateTime.Now;

int topNumber = 100000000;

BitArray numbers = new BitArray(topNumber, true);

for (int i = 2; i < topNumber; i++)

{

if (numbers[i])

{

for (int j = i \* 2; j < topNumber; j += i)

{

numbers[j] = false;

}

}

}

int primes = 0;

for (int i = 1; i < topNumber; i++)

{

if (numbers[i])

{

primes++;

}

}

Console.WriteLine("Finished Factoring Prime Numbers (" + Math.Round(DateTime.Now.Subtract(start).TotalSeconds, 0) + " seconds)");

if (FactoredPrimes != null)

{

this.FactoredPrimes(this, new PrimeFactoringEventArgs((Guid)instanceId, primes));

}

}

}

1. Now that we have created our *FactoringService*, save the project and build the new service.

## Task 3 – Create a Workflow

1. Open the **Solution Explorer** and double click the file *Program.cs***.**
2. Add a using statement to the top of the *Program.cs* file

using System.Workflow.Activities;

1. Add the highlighted code within the **Main** method. Cut and paste or use the code snippet provided for the first section and add the line at the end manually.

(Snippet: “WFLab05\_Exercise04\_Task03\_MainMethod”)

using(WorkflowRuntime workflowRuntime = new WorkflowRuntime())

{

ExternalDataExchangeService dataService = new ExternalDataExchangeService();

workflowRuntime.AddService(dataService);

dataService.AddService(new FactoringService());

AutoResetEvent waitHandle = new AutoResetEvent(false);

workflowRuntime.WorkflowCompleted += delegate(object sender, WorkflowCompletedEventArgs e) {waitHandle.Set();};

workflowRuntime.WorkflowTerminated += delegate(object sender, WorkflowTerminatedEventArgs e)

{

Console.WriteLine(e.Exception.Message);

waitHandle.Set();

};

WorkflowInstance instance = workflowRuntime.CreateWorkflow(typeof(LongRunningWorkflow.Workflow1));

instance.Start();

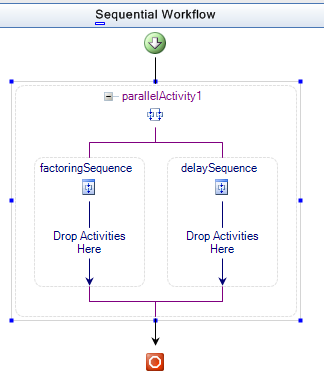
waitHandle.WaitOne();

Console.ReadLine();

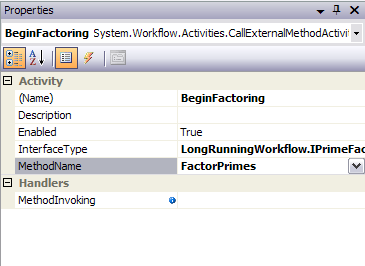
}

The code is essentially the same although if you notice the three lines of highlighted code above we added an **ExternalDataExchangeService** and our *FactoringService* into the worklow.

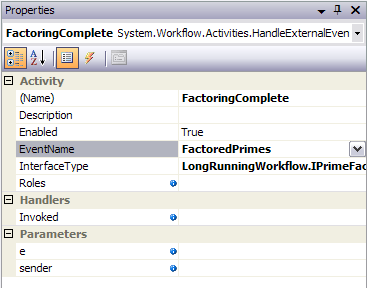
1. Open the **Solution Explorer** and double click on the file named *Workflow1.cs* to open the workflow in design mode.
2. Our workflow is going to branch into two paths, one path will invoke our service which we created in task1 and begin the time consuming task of factoring prime numbers, the second path will simply delay for 5 seconds. Once both paths have completed the workflow will be complete. As both paths will be running simultaneously we will use the **Parallel** activity, open the **Toolbox** and drag and drop a **Parallel** activity onto the workflow.
3. Click on the section named *sequenceActivity1* and, using the properties window, set the ***Name*** property to *factoringSequence*. Similarly **rename** *sequenceActivity2* to *delaySequence*.
4. Your Workflow should now look similar to this



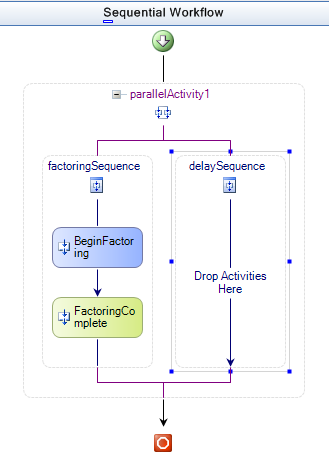
1. Now we will add some activities to utilize our factoring service. Drag and drop a **CallExternalMethod** activity into our *factoringSequence*.
2. Change the **Name** of the **CallExternalMethodActivity** activity from *callExternalMethodActivity1* to *BeginFactoring*.
3. Set the **InterfaceType** to *IPrimeFactoring* by clicking the ellipsis (…) next to **InterfaceType** and choose the *IPrimeFactoring* interface in the project.
4. Using the drop down menu, set the **MethodName** to *FactorPrimes*.



1. Now we need to handle the event raised by the *FactoringService* when it has completed Factoring the prime numbers. Drag and drop a **HandleExternalEvent** activity onto the workflow just below the *BeginFactoring* activity and still within the *factoringSequence*.
2. **Rename** *handleExternalEventActivity1* to *FactoringComplete* and set the **InterfaceType** to *IPrimeFactoring*.
3. Using the drop down menu, set the **EventName** to *FactoredPrimes*.



1. Your workflow should now look similar to this:



1. Next we will create the *delaySequence*, drag and drop a **Delay** activity from the **Toolbox** into the *delaySequence*.
2. Set the **TimeoutDuration** property of *delayActivity1* to *5* seconds.
3. Save the workflow and build the project.
4. For demonstration purposes we will add a number of Code activities to our workflow so we can watch the progress in the console.
5. Right click the file *Workflow1.cs* in the **Solution Explorer** and select **View Code**.
6. Cut and paste or use the code snippet provided to add the following 4 event handlers to the *Workflow1* class

(Snippet: “WFLab05\_Exercise04\_Task03\_EventHandlers”)

private void BeforeDelay\_Execute(object sender, EventArgs e)

{

Console.WriteLine("Beginning Delay");

}

private void AfterDelay\_Execute(object sender, EventArgs e)

{

Console.WriteLine(string.Format("Finished Delay ({0} seconds)", this.delayActivity1.TimeoutDuration.Seconds));

}

private void AfterBranch\_Execute(object sender, EventArgs e)

{

Console.WriteLine("Finished Branch");

}

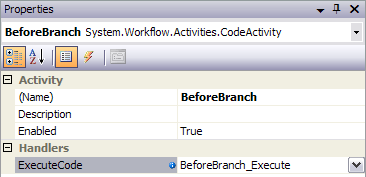
private void BeforeBranch\_Execute(object sender, EventArgs e)

{

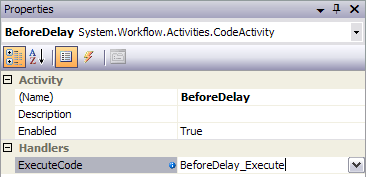
Console.WriteLine("Beginning Branch");

}

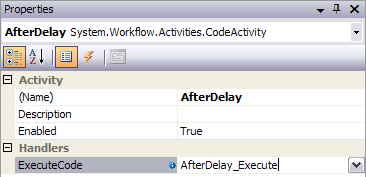
1. Press **SHIFT + F7** to switch to design mode and we will add a code activity for each of these.
2. Drag and drop a **Code** activity onto your workflow just before *parallelActivity1*, **rename** the activity from *codeActivity1* to *BeforeBranch.*
3. In the **ExecuteCode** property use the drop down menu and select *BeforeBranch\_Execute* event handler.



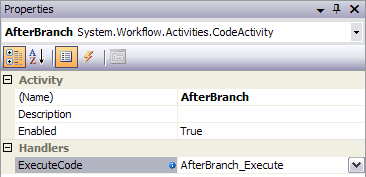
1. Drag and drop a second **Code** activity onto your workflow just before *delayActivity1* within the *delaySequence*, **rename** the activity from *codeActivity1* to *BeforeDelay.*
2. Set the **ExecuteCode** property to *BeforeDelay\_Execute*.



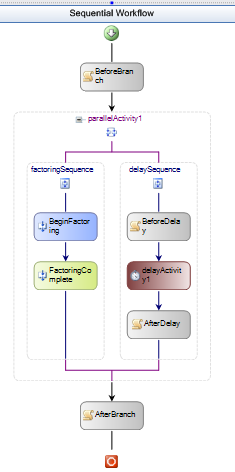
1. Drag and drop a third **Code** activity onto your workflow just after *delayActivity1* within the *delaySequence*, **rename** the activity from *codeActivity1* to *AfterDelay.*
2. Set the **ExecuteCode** property to *AfterDelay\_Execute*.



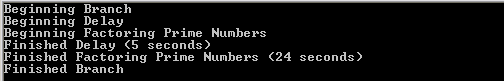
1. Finally, drag and drop a **Code** activity onto your workflow just after *parallelActivity1*, **rename** the activity from *codeActivity1* to *AfterBranch.*
2. Set the **ExecuteCode** property to *AfterBranch\_Execute*.



1. Your finished Workflow should look similar to this:



1. Save and run the Workflow by pressing **F5**, you should see the progress of your workflow in the console as it executes



1. Press **Enter** to exit the program.

As you can see, the long running process was carried out by the host application while the workflow was idled. We could, of course, add a PersistenceService to our Host and allow the runtime to persist out the workflow while the long running process was taking place. In this way we can potentially support processes that may span several hours, days or even weeks.

# Lab Summary

The objective of this lab was to introduce Windows Workflow Foundation communication with the host application using a local service. A local service is a way to provide host application functionality to a workflow. Communication to a workflow is accomplished using events raised by the local service; a local service is called by a workflow via methods.

During this lab, you should have:

Created a simple local service that exposes a method in which it does some work and also publishes an event which it raises when its work is complete.

Created a more complex workflow that uses correlated messaging to route message to the appropriate activity within a workflow.

Added role based authorization and security to a Workflow scenario

Created a simple Workflow that actions a long running process