

Windows Workflow Foundation

Hands-On Lab

Lab Manual

Lab 07 – Creating Rules Driven Workflows 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 Studio are 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 1](#_Toc218630400)

[Objectives 1](#_Toc218630401)

[System Requirements 1](#_Toc218630402)

[Setup 1](#_Toc218630403)

[Physical Folder Structure 1](#_Toc218630404)

[Code Snippets 2](#_Toc218630405)

[Starting Material 2](#_Toc218630406)

[Acronyms Used in this Lab 2](#_Toc218630407)

[Scenario 2](#_Toc218630408)

[Exercise 1 – Using the ConditionedActivityGroup Activity 3](#_Toc218630409)

[Task 1 – Create a new Sequential Workflow Console Application 3](#_Toc218630410)

[Task 2 – Add the ConditionedActivityGroup and Code activities 4](#_Toc218630411)

[Task 3 – Define the conditions 8](#_Toc218630412)

[Task 4 – Run the workflow 11](#_Toc218630413)

[Exercise 2 – Using the Replicator Activity 12](#_Toc218630414)

[Task 1 – Create a new Sequential Workflow Console Application 12](#_Toc218630415)

[Task 2 – Add the Replicator 13](#_Toc218630416)

[Task 3 – Run the workflow 17](#_Toc218630417)

[Exercise 3 – Using the Policy Activity 19](#_Toc218630418)

[Task 1 – Create a new Sequential Workflow Console Application 19](#_Toc218630419)

[Task 2 – Add and configure a PolicyActivity 20](#_Toc218630420)

[Task 3 – Run the workflow 22](#_Toc218630421)

[Lab Summary 24](#_Toc218630422)

# Overview

Estimated time to complete this lab: **45 minutes**

The project files for this lab are in the *C:\WF\WF 3.5 Labs\Lab07* folder.

## Objectives

The objective of this lab is to introduce several of the more advanced activities provided with Windows Workflow Foundation. After completing this lab, you will be able to:

* Use the **ConditionedActivityGroup** activity.

The **ConditionedActivityGroup** activity allows you to provide condition-driven execution behavior to a collection of activities. This behavior allows you to define the condition under which a given activity will execute, as opposed to defining a strict sequencing or flow of activity execution.

* Use the **Replicator** activity to launch multiple copies of other activities.

The **Replicator** activity allows you to dynamically generate multiple instances of an activity based on the data provided in a collection at runtime. The instances can be created in sequential or parallel fashion.

* Use a **Policy** based activity to implement application logic

The **Policy** activity encapsulates a forward chaining rules engine. Creating a custom activity that inherits from the **Policy** activity allows you to define sophisticated application logic that is then available for use in your workflows.

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

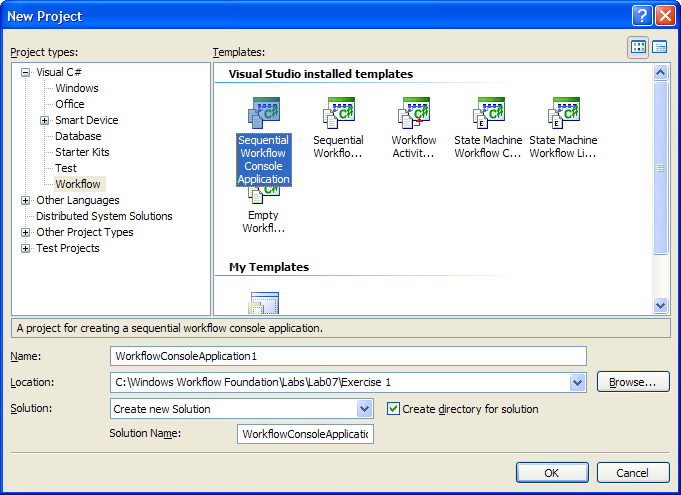
### Scenario

# Exercise 1 – Using the ConditionedActivityGroup Activity

In this exercise you will create a simple console application to instantiate the Windows Workflow Foundation runtime and start a **ConditionedActivityGroup** based workflow.

## Task 1 – Create a new Sequential Workflow Console Application

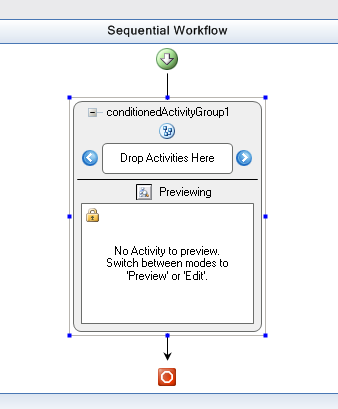
1. Open **Visual Studio 2005** by going to the **Start Menu | Programs | Microsoft Visual Studio 2005 | Microsoft Visual Studio 2005**
2. In **Visual Studio 2005**, select the **File** | **New** | **Project** menu command.
3. **Visual Studio** will display the **New Project** dialog window.
4. In the **New Project** dialog window, expand **Visual C# | Workflow** in the Project Types tree on the left side.
5. Select the template named **Sequential Workflow Console Application** and accept the default name for the project but change the location to *C:\WF\WF 3.5 Labs\Lab07\Exercise 1* and leave the **Create directory for solution** box checked. This will create a subdirectory to *C:\WF\WF 3.5 Labs\Lab07\Exercise 1* and place the project and solution files there. The directory will be named*WorkflowConsoleApplication1*.



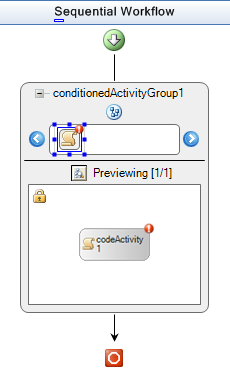
1. Click **OK** and the basic project will be generated.

## Task 2 – Add the ConditionedActivityGroup and Code activities

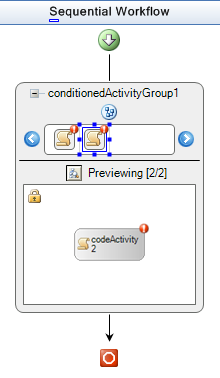
1. Let's start by adding some activities to the workflow. First drag and drop a **ConditionedActivityGroup** activity onto the workflow from the toolbox (select **View | Toolbox** from the menu if the toolbox is not shown). It should look like this:



1. Now we need to add two **Code** activities to the **ConditionedActivityGroup**. In the screen shot above you can see a section of the **ConditionedActivityGroup** activity that has the words **Drop Activities Here**. Drag a **Code** activity from the toolbox to that area and you should see this:



1. The little red circles with exclamation marks are an indication that we have not yet connected the **Code** activity for *codeActivity1* to an event handler. We will do that next, but first add another **Code** activity so that your workflow design looks like this:



1. Next we will add the code handlers for the two **Code** activities that were added. Place the code below within the *Workflow1* class in *Workflow1.cs*. You can go to *Workflow1.cs* by right clicking on the design surface and choose **View Code**.

(Code snippet: “WFLab07\_Ex01\_Task02\_CodeHandlers”)

public enum NumberIs { Odd = 0, Even = 1}

private NumberIs currentNumber = NumberIs.Even;

private bool userRequestsExit;

//number entered is odd

public void codeActivity1\_ExecuteCode(object sender, EventArgs e)

{

Console.WriteLine("the number is odd");

GetNextNumber();

}

//number entered is even

public void codeActivity2\_ExecuteCode(object sender, EventArgs e)

{

Console.WriteLine("the number is even");

GetNextNumber();

}

//evaluate number entered by user and set currentNumber variable

private void GetNextNumber()

{

int numIn;

Console.WriteLine("enter a number.");

string charIn = Console.ReadLine();

if (Int32.TryParse(charIn, out numIn))

{

if ((numIn % 2) == 0)

{

currentNumber = NumberIs.Even;

}

else

{

currentNumber = NumberIs.Odd;

}

}

else if (charIn.Contains("x"))

{

userRequestsExit = true;

}

else

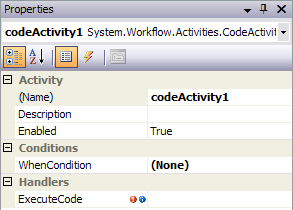
{

GetNextNumber();

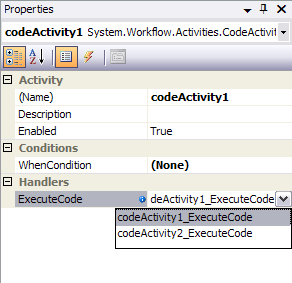
}

}

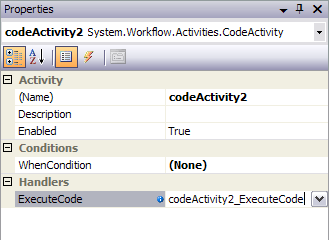
1. Change back to the designer window.
2. Right click on the **ConditionedActivityGroup** activity and choose **Edit Activity** from the context menu. You can now select each of the code activities within the **ConditionedActivityGroup** so that their properties can be set.
3. Go to the property window for *codeActivity1*. It should look like this;



1. Click on the **ExecuteCode** property in the **Handlers** section. The drop down list should contain the *codeActivity1\_ExecuteCode* and *codeActivity2\_ExecuteCode* methods that we just added.
2. Select *codeActivity1\_ExecuteCode*. It should look like this:



1. You can see that there is a red circle with an exclamation mark next to *codeActivity2*. Now you can click on the *codeActivity2* activity and connect its **ExecuteCode** handler to *codeActivity2\_ExecuteCode*. It should look like this when you are done:



## Task 3 – Define the conditions

The next step is to write the functions that will be called when *codeActivity1* and *codeActivity2*'s **WhenCondition** is evaluated. The activity will execute if the **WhenCondition** evaluates to true. There is also an **UntilCondition** for the **ConditionedActivityGroup** activity. If the **ConditionedActivityGroup UntilCondition** returns true then the **ConditionedActivityGroup** is finished and it exits.

This example has a very simple scenario. The user is asked to enter a number. That number is evaluated as to whether it is even or odd. If it is odd then *currentNumber* is set to *NumberIs.Odd*. If it is even then *currentNumber* is set to *NumberIs.Even*. The **UntilCondition** method bases its return value on the state of the *currentNumber* variable. If the user types an 'x' then *userRequestsExist* is set to true, in which case both the **UntilCondition** evaluates to true and the **ConditionedActivityGroup** exits.

1. Right click on the *Workflow1.cs* file and select **View Code**.
2. Place the following 3 functions within the *Workflow1* class: *oddCondition*, *evenCondition* and *timeToExit*

(snippet: “WFLab07\_Ex01\_Task03\_CodeConditions”).

//used to control execution of codeActivity1\_ExecuteCode1

public void oddCondition(object sender, ConditionalEventArgs args)

{

args.Result = ((currentNumber == NumberIs.Odd));

}

//used to control execution of codeActivity2\_ExecuteCode1

public void evenCondition(object sender, ConditionalEventArgs args)

{

args.Result = ((currentNumber == NumberIs.Even));

}

//used to control execution of the ConditionedActivityGroup

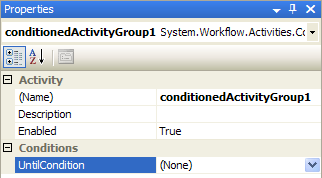
public void timeToExit(object sender, ConditionalEventArgs args)

{

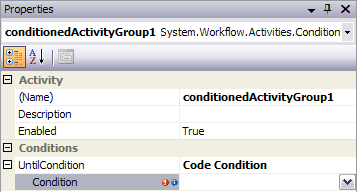
args.Result = userRequestsExit;

}

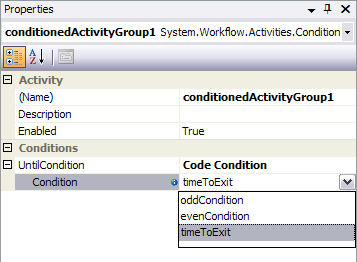
1. You should now connect these functions to the corresponding condition handlers so switch back to the designer.
2. Click on the *conditionedActivityGroup1* activity.
3. In the properties window, the **UntilCondition** for the *conditionedActivityGroup1* will read **(None)**.



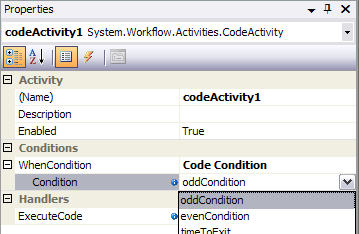
1. Click on the down arrow and select **System.Workflow.Activities.CodeCondition**. Now you can expand this property section by clicking on the plus sign to the left of **UntilCondition**. When it expands you will see the **Condition** property just below **UntilCondition**.



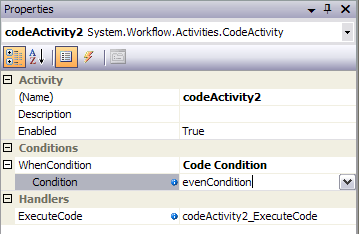
1. Click on the drop down and you should see a list of functions to choose from. Select *timeToExit*. You should see this when you click the drop down:



1. Now when this workflow runs the **UntilCondition** of the *conditionedActivityGroup1* is evaluated. The *conditionedActivityGroup1* will continue to execute as long as the **UntilCondition** evaluates to false.
2. The **WhenCondition** of *codeActivity1* will be set to the *oddCondition* handler and the **WhenCondition** of *codeActivity2* will be set to the *evenCondition* handler- we’ll do this in a similar fashion to the way in which we set the **CodeCondition** above. As the name suggests, *codeActivity1* will execute when an odd number is passed in, and *codeActivity2* will execute when an even number is passed in.
3. Connect the **WhenCondition** for *codeActivity1*. The properties window for *codeActivity1* will look like this:



1. Connect the **WhenCondition** for *codeActivity2*. It should look like this:



## Task 4 – Run the workflow

When you run the application, you can enter numbers and the **ConditionedActivityGroup** will run the correct code handler based on whether you entered an even or an odd number. When you want to exit type a lower case "x". When you type "x" the *timeToExit* function returns true and the workflow exits returning execution to the caller. In this case we return to **Main**, the **WorkflowCompleted** function is called, we signal our event and **Main** exits. For simplicity we set *currentNumber = NumberIs.Even* so the first time through the assumption is that the number is even and we will print out the number is even although you have not typed a number yet.

1. We haven't added any code to *Program.cs* yet but we need to add the following code highlighted in gray to the end of the **Main** method- just after the **WaitOne** call on the *waitHandle*.

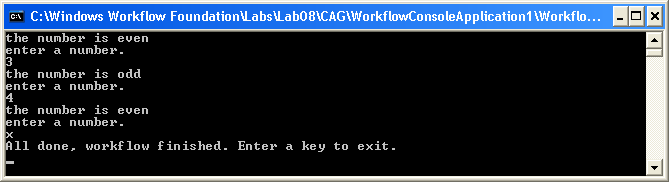
waitHandle.WaitOne();

**Console**.WriteLine("All done, workflow finished. Press Enter to exit.");

**Console**.Read();

}

1. Now you should be able to compile and run the application by pressing F5 or Ctrl-F5. Try entering some numbers and then type ‘x’ to finish the workflow. You should see something like this:

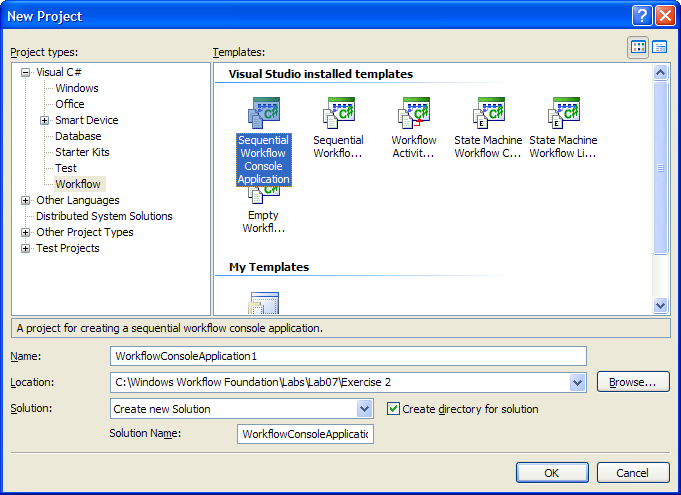


# Exercise 2 – Using the Replicator Activity

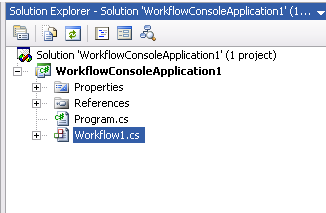
The **Replicator** activity can be used to launch multiple copies of other activities. In this example, for simplicity, we will launch copies of a **Code** activity.

## Task 1 – Create a new Sequential Workflow Console Application

1. Open Visual Studio 2005 by going to the **Start Menu | Programs | Microsoft Visual Studio | Microsoft Visual Studio 2005**
2. In **Visual Studio 2005**, select the **File** | **New** | **Project** menu command.
3. **Visual Studio** will display the **New Project** dialog window.
4. In the **New Project** dialog window, expand **Visual C# | Workflow** in the Project Types tree on the left side.
5. Select the template named **Sequential Workflow Console Application** and accept the default name for the project but change the location to *C:\WF\WF 3.5 Labs\Lab07\Exercise 2* and leave the **Create directory for solution** box checked. This will create a subdirectory to *C:\WF\WF 3.5 Labs\Lab07\Exercise 2* and place the project and solution files there. The directory will be named *WorkflowConsoleApplication1*.

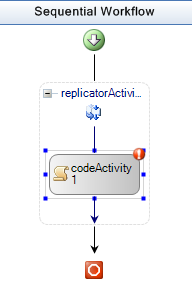


1. You should now have a new solution and workflow console application project. The project generated for you will contain 2 files of interest to us. *Program.cs* which contains the **Main** method and the workflow is contained in *Workflow1.cs*.



## Task 2 – Add the Replicator

1. Let's start with the workflow. Add a **Replicator** activity to your workflow and add a **Code** activity into the **Replicator** activity. Your drawing surface should look like the following picture. The little red circles with the exclamation marks indicate that we have not yet fully configured the activities.



1. To connect the handlers and get rid of the little red circles with exclamation marks let's start with the replicator's **Initialized** handler.
2. Switch to the code view of the workflow by right clicking on *Workflow1.cs* in the solution explorer and choosing **View Code**.
3. Add the following code to *Workflow1.cs* within the *workflow1* class declaration.

(snippet: “WFLab07\_Ex02\_Task02\_InitializedHandler”):

public void replicatorActivity1\_Initialized(object sender, **EventArgs** e)

{

// Populate the data used for each instance of the Replicator's

// child instance that are created

**ArrayList** children = new **ArrayList**();

children.Add("Child Instance 1");

children.Add("Child Instance 2");

children.Add("Child Instance 3");

children.Add("Child Instance 4");

children.Add("Child Instance 5");

children.Add("Child Instance 6");

replicatorActivity1.InitialChildData = children;

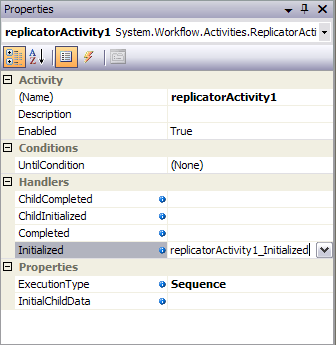
// Specify how the child instances should be created - in parallel

// or in a sequence

replicatorActivity1.ExecutionType = ExecutionType.Parallel;

}

1. This code adds 6 children to the **ArrayList** which we then assign to *replicatorActivity1*’s **InitialChildData** property. We also specify that the **ExecutionType** is **ExecutionType.Parallel**. We could have specified **ExecutionType.Sequence**. You could try both to see the difference- the main difference is that **Sequence** means that each child will finish executing before the next one starts. You can add more children to the list if you like. The number of **Code** activities launched is based on the number of children added to the **ArrayList** children.
2. Switch back to design view and connect the **Initialized** handler to the replicator in *replicatorActivity1*'s property window.
3. When you click on the drop down arrow you should see *replicatorActivity1\_Initialized*, click on it to set the **Initialized** handler.



1. Next we need to hook up the **Code** activity's code handler. Add the following code to *Workflow1.cs* within the *Workflow1* class.

(snippet: “WFLab07\_Ex02\_Task02\_ExecuteCode”):

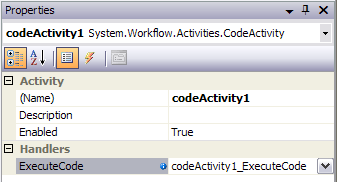
public void codeActivity1\_ExecuteCode(object sender, EventArgs e)

{

Console.WriteLine("In codeActivity1\_ExecuteCode");

}

1. Back in design view click on the **Code** activity and then click on its **ExecuteCode** handler. You should see *codeActivity1\_ExecuteCode* in the properties dropdown. Select that and the handler is connected. It should look like this when you are done.



1. The **Replicator** has 2 more code handlers that we want to hook up: **ChildCompleted** and **ChildInitialized**. Add the following to *Workflow1.cs* within the *Workflow1* class.

(snippet: “WFLab07\_Ex02\_Task02\_ChildEvents”)

private void childCompleted(object sender, ReplicatorChildEventArgs e)

{

Console.WriteLine("Completed {0}", e.InstanceData.ToString());

}

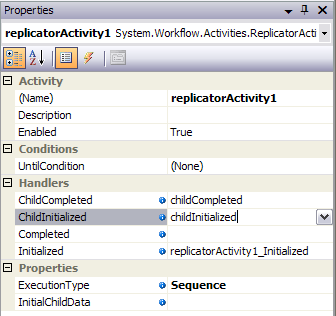
private void childInitialized(object sender, ReplicatorChildEventArgs e)

{

Console.WriteLine("Initialized {0}", e.InstanceData.ToString());

}

1. Connect **ChildInitialized** to *childInitialized* and **ChildCompleted** to *childCompleted* on the **Replicator**'s handler section. It should look like this:



1. The workflow itself has a handler we want to hook into. Add this function to the others within the *Workflow1* class in the *Workflow1.cs* file.

(snippet: “WFLab07\_Ex02\_Task02\_WorkflowCompleted”).

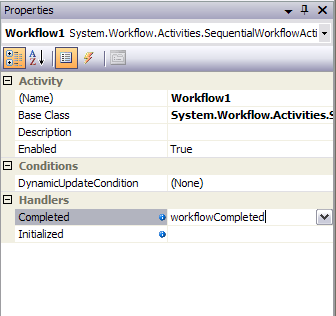
private void workflowCompleted(object sender, EventArgs e)

{

Console.WriteLine("Workflow finished");

}

1. Click anywhere outside of the replicator activity on the drawing surface, or select *Workflow1* from the combo box in the properties window. Click on the **Completed** handler for *Workflow1* in the properties window and select *workflowCompleted*.



## Task 3 – Run the workflow

1. Now we have just a minor change to make to **Main** in the *program.cs* file. Add the following code highlighted in gray to the end of the **Main** method- just after the **WaitOne** call.

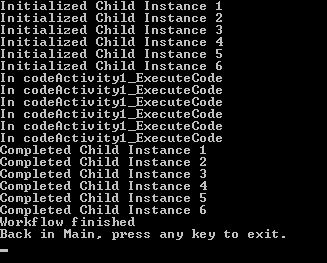
waitHandle.WaitOne();

**Console**.WriteLine("Back in Main, press any key to exit.");

**Console**.Read();

}

1. At this point you should be able to compile and run the project you should see the following:



1. Change the code in *Workflow1.cs* as follows:

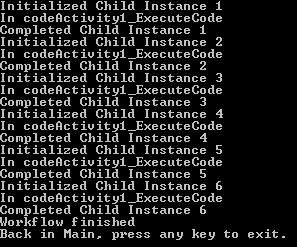
from

replicatorActivity1.ExecutionType = ExecutionType.Parallel;

to

replicatorActivity1.ExecutionType = ExecutionType.Sequence;

1. You will see something like the following:

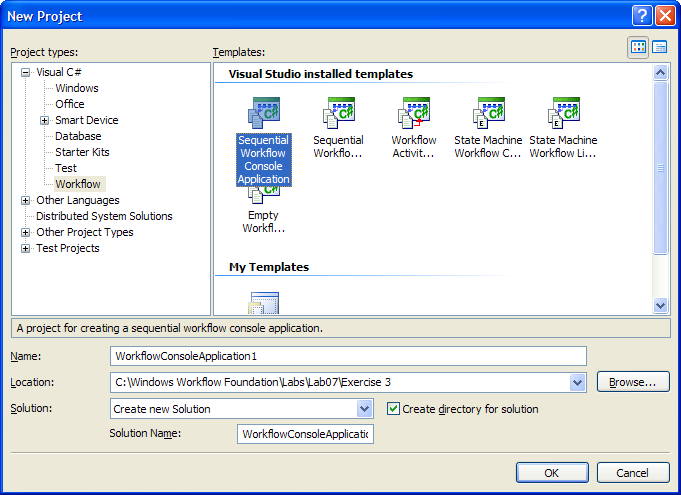


# Exercise 3 – Using the Policy Activity

In this exercise you will create a simple console application to instantiate the Windows Workflow Foundation runtime and start a workflow with a **Policy** based activity.

## Task 1 – Create a new Sequential Workflow Console 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. **Visual Studio** will display the **New Project** dialog window.
4. In the **New Project** dialog window, expand **Visual C# | Workflow** in the Project Types tree on the left side.
5. Select the template named **Sequential Workflow Console Application** and accept the default name for the project but change the location to *C:\WF\WF 3.5 Labs\Lab07\Exercise 3* and leave the **Create directory for solution** box checked. This will create a subdirectory to *C:\WF\WF 3.5 Labs\Lab07\Exercise 3* and place the project and solution files there. The directory will be named*WorkflowConsoleApplication1*.



1. Click **OK** and the basic project will be generated.

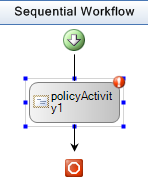
## Task 2 – Add and configure a PolicyActivity

1. Right-click on *Workflow1.cs* in the **Solution Explorer** and select **View Code**.
2. Add the following lines to the beginning of the *Workflow1* class declaration. These are the parameters that our **PolicyActivity** will use:

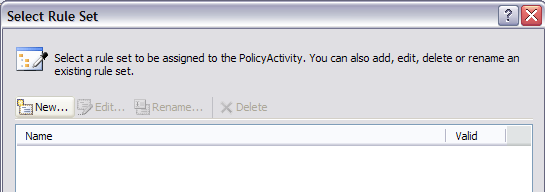
public int orderValue = 600;

public int discount = default(int);

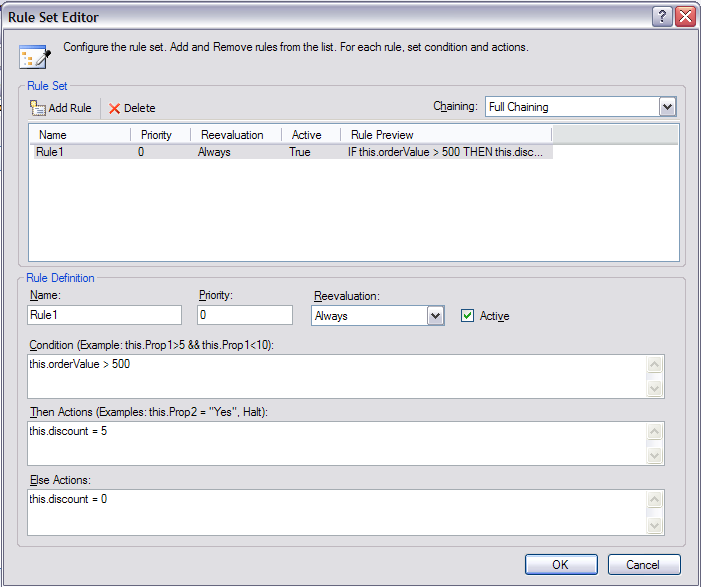
1. Go to the designer view for *Workflow1.cs*. You can do this by right-clicking on it in the **Solution Explorer** and selecting **View Designer**.
2. Add a **Policy** activity to the workflow. It should now look like this:



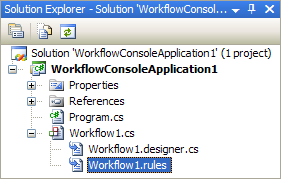
1. In the properties window click on the ellipses next to the **RuleSetReference** property of *policyActivity1*.
2. The **Select RuleSet** dialog will appear. Click the **New RuleSet…** button as shown below:



1. This will bring up a new dialog, **Rule Set Editor**. Click **Add Rule** and enter the values for the lower three editable regions shown in the image below:



1. Click **OK** then **OK** again to set the rules for the policy. Notice that a file called *Workflow1*.rules has been added to your solution (shown below). This file contains an XML definition of any rules and rule sets for *Workflow1*.



1. Select the outer **SequentialWorkflow** and in the **Properties** window set the **Completed** handler to **WorkflowCompleted** and hit **Enter** to go to the *Workflow1.cs* code (or right click on the workflow and select **View Code**). Add the following code to the handler:

private void WorkflowCompleted(object sender, EventArgs e)

{

Console.WriteLine("OrderValue = {0}", this.orderValue);

Console.WriteLine("Discount = {0}", this.discount);

}

## Task 3 – Run the workflow

1. Now we have just a minor change to make to **Main** in the *Program.cs* file. Add the two lines of code highlighted in gray at the end of the **Main** method, just after the call to **WaitOne**:

instance.Start();

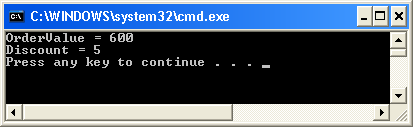
waitHandle.WaitOne();

**Console**.WriteLine("Back in Main, press Enter to exit.");

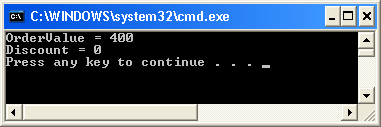
**Console**.Read();

}

1. You should now be able to compile and run the application. After pressing Ctrl-F5, you should see something like this:



1. If you change the *orderValue* (set in task 2, step 2) in *Workflow1.cs* to less than 500 and re-compile and run the application you will see something like:



# Lab Summary

The objective of this lab has been to introduce several of the advanced built in Activities provided with Windows Workflow Foundation. You should know how to use:

The **ConditionedActivityGroup** activity to provide condition-driven behavior to an activity set.

The **Replicator** activity to launch multiple copies of another activity.

A **Policy** based activity to implement application logic.