developmentor

Workflow 4.5

Estimated time for completion: 45 minutes

Overview:
In this lab you will be creating an application that uses a workflow to list and filter the processes that are running on the machine

Goals:
- Learn how to use the new workflow editor
- Understand how to create flowchart based workflows
- Write custom activities

Lab Notes:
This lab will be using Visual Studio 2010.

Part 1: Creating the Initial Project

In the first part of the lab you will create an initial project within which to do the rest of the work

1. Open Visual Studio
2. Create a New Project of type Workflow Console Application (this is in the workflow section of the new project dialog)
3. From the Flow Control section of the toolbox drag and drop a Sequence on to the design surface. This will act as the top level execution mode of your workflow.

Part 2: Creating the GetProcessesActivity

In this part of the lab you will create a custom activity that wraps the System.Diagnostics.Process class to get a list of the processes that are running on the machine

1. Add a new class to the project called `GetProcessesActivity`
2. Mark the class as `public` and derive it from `CodeActivity`
3. Add a new public property of type `OutArgument<List<Process>>` called `Processes` (you will need to add a using statement for System.Diagnostics for this to compile)
4. Implement the abstract `Execute` method from `CodeActivity`
5. In the `Execute` method call the static `GetProcesses` member of the `Process` class, convert the returned array into a `List` and assign it to the `Processes` property using its `Set` method

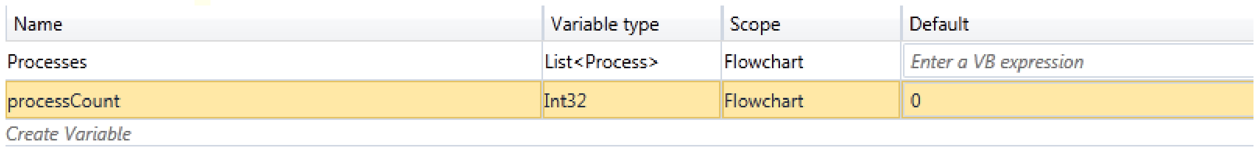
```
public class GetProcessesActivity : CodeActivity
{
    public OutArgument<List<Process>> Processes { get; set; }

    protected override void Execute(CodeActivityContext context)
    {
        Processes.Set(context, Process.GetProcesses().ToList());
    }
}
```

6. Compile the code to ensure there are no syntax errors

Part 3: Creating the Initial Workflow

In this section of the lab you will create the initial workflow to list and print the processes running on the machine. You will be using a FlowChart based workflow to do this.

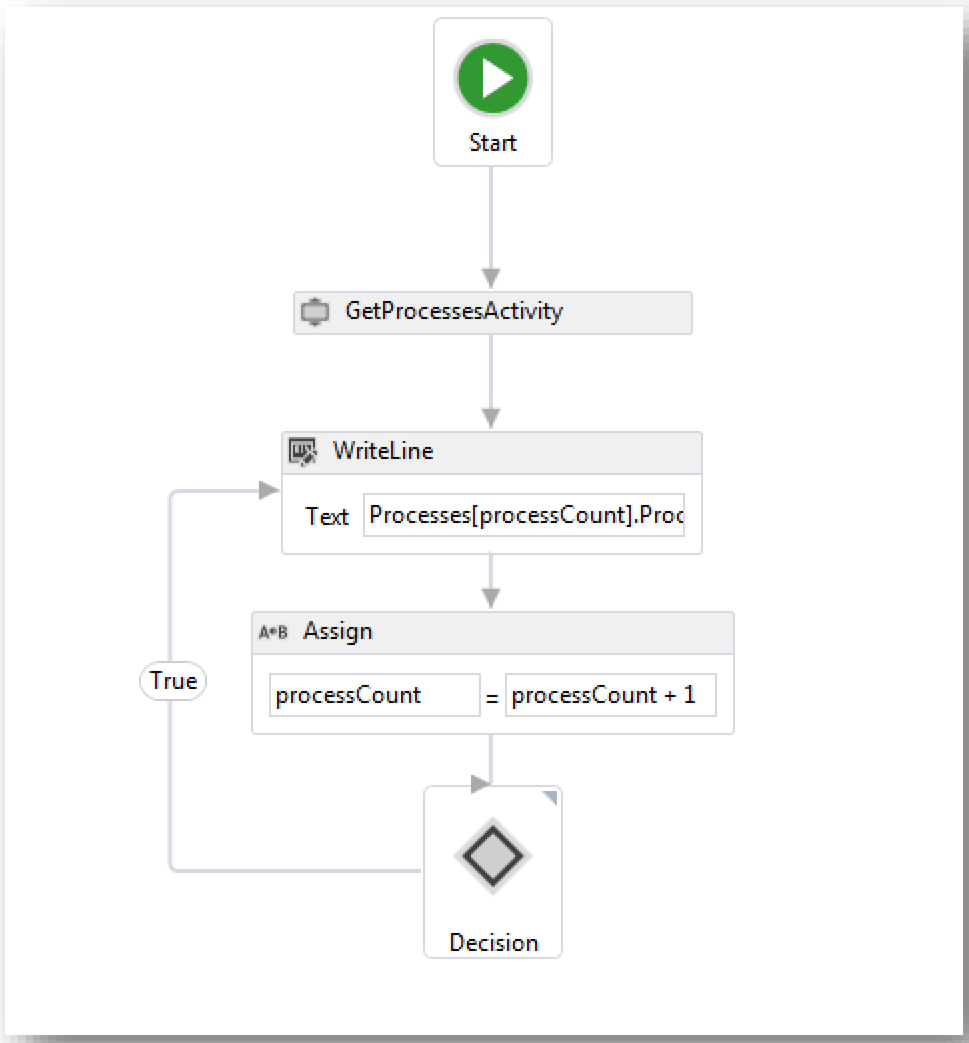
1. Open the **Workflow1.xaml** file. This will put you in the workflow designer
2. Drag a FlowChart on to the design surface and double click on it. This will use the new breadcrumb feature of the designer to focus on the design of the flowchart.
3. Select the flowchart (by clicking on its background) and then bring up the variables window (pictured). Add two variables: one called `Processes` of type `List<Process>` (you will need to use the type browser to find the List class in mscorlib and the Process class in System); the other called `processCount` of type `Int32`. For `processCount` set its initial value to `0`.



Name	Variable type	Scope	Default
Processes	List<Process>	Flowchart	Enter a VB expression
processCount	Int32	Flowchart	0
Create Variable			

4. Drag a `GetProcessesActivity` on to the flowchart from the toolbox. In the properties window set it's `Processes` output argument to the `Processes` variable you just created
5. Drag a `WriteLine` from the toolbox on to the flowchart below the `GetProcessesActivity`. Set its `Text` property to the expression `Processes[processCount].ProcessName` .
6. Drag an `Assign` on to the design surface under the `WriteLine` and set `processCount` equal to `processCount + 1`
7. Drag a `FlowDecision` on to the flowchart under the `Assign` and set its `Condition` property to the expression `processCount < Processes.Count`.
8. You now need to connect up the shapes on the flow chart. When you hover over a shape connectors appear on the sides of the shape. You click on the connector and drag it to its destination where target connectors will appear.
9. Connect the green start circle to the `GetProcessesActivity`
10. Connect the `GetProcessesActivity` to the `WriteLine`
11. Connect the `WriteLine` to the `Assign`
12. Connect the `Assign` to the `FlowDecision`

13. Connect the true branch of the `FlowDecision` to the `WriteLineActivity` to loop back through the processing



14. Compile and test your workflow. You should see the names of the processes running on the machine printed out

Part 4: Filtering the Process List with a Pluggable Filter

In the last part of the lab you will create another custom activity that checks a Process against a pluggable scheme to allow filtering. The filtering scheme is provided by an extension which can be associated with the workflow instance.

1. Add a new class to the project. Call it `ProcessFilterExtension`.

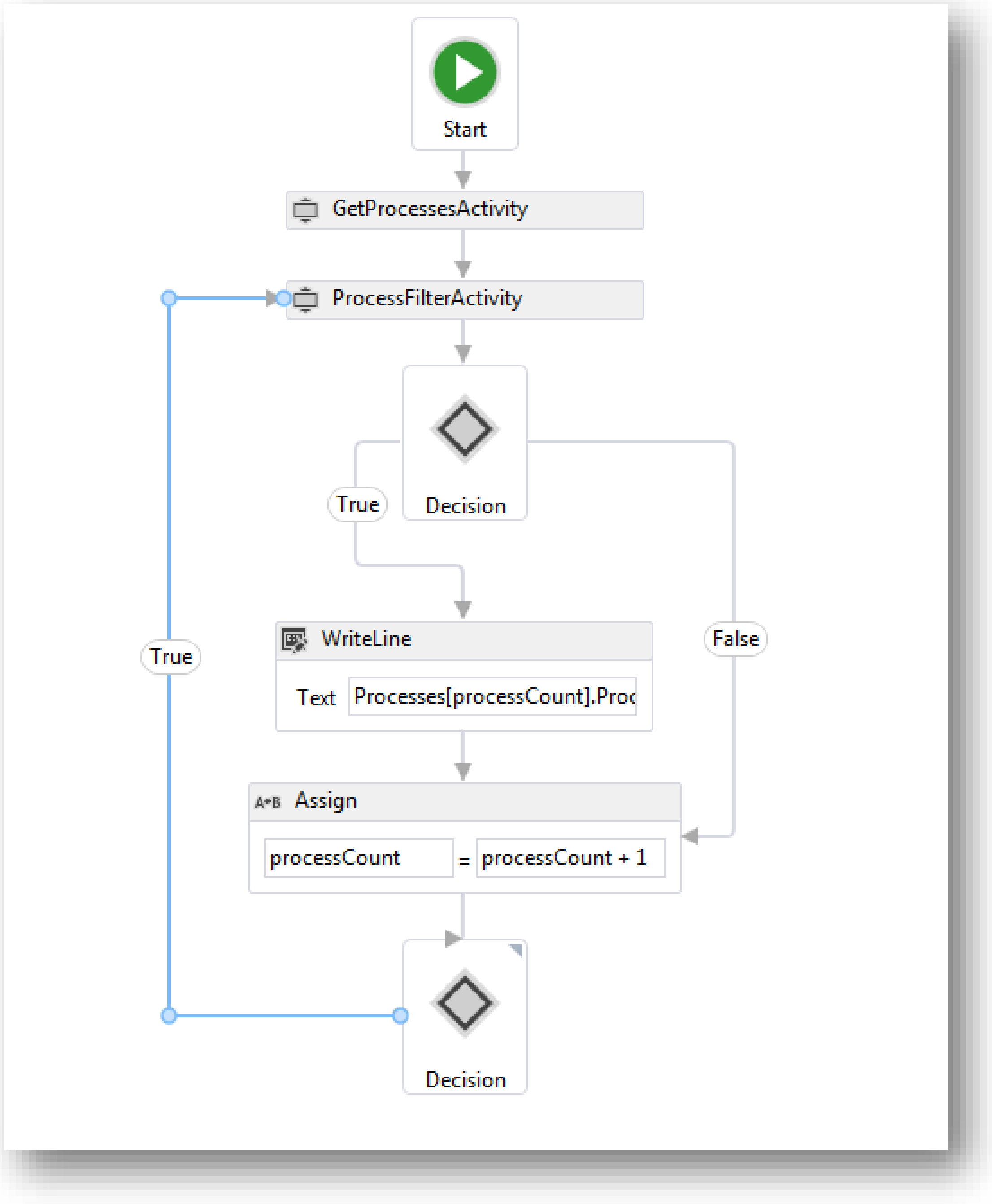
2. Make this class `abstract` and provide a single `abstract` method `IsMatchingProcess` that takes a `Process` and returns a Boolean
3. In the same file create another class that derives from the `ProcessFilterExtension` abstract class. Call this class `StartsWithFilter`.
4. Create a member variable in the `StartsWithFilter` class of type `string` called `startsWith`.
5. Add a constructor to the `StartsWithFilter` class that takes a `string` and initialize the `startsWith` member with this string
6. Override the abstract `IsMatchingProcess` method testing to see if the name of the process passed starts with the `startsWith` field value. Return `true` if it does, `false` if it does not.

```
abstract class ProcessFilterExtension
{
    public abstract bool IsMatchingProcess(Process process);
}

class StartsWithFilter : ProcessFilterExtension
{
    string startsWith;
    public StartsWithFilter(string startsWith)
    {
        this.startsWith  = startsWith;
    }

    public override bool IsMatchingProcess(Process process)
    {
        return process.ProcessName.StartsWith(startsWith);
    }
}
```

7. Next you will create the activity that utilizes this extension. Add a new class to the project called `ProcessFilterActivity`
8. Mark the class as `public` and derive from CodeActivity
9. This activity will have two argunents:
10. Create a public property of type `InArgument<Process>` called `Process`
11. Create a public property of type `OutArgument<bool>` called `IsMatch`
12. Override `Execute`. Attempt to retrieve the ProcessFilterExtension by calling `GetExtension<>()` on the `CodeActivityContext`
13. If the extension has not been added to the workflow instance (`GetExtension` returns `null`) then set the `IsMatch` argument value to `true` (using the `Set` method)
14. If the extension has been added, use it calling `IsMatchingProcess`, passing the `Process` argument value (using the `Get` method) and set the `IsMatch` argument value to the value returned from `IsMatchingProcess`.

```
public class ProcessFilterActivity : CodeActivity
{
    public InArgument<Process> Process { get; set; }
    public OutArgument<bool> IsMatch { get; set; }

    protected override void Execute(CodeActivityContext context)
    {
        ProcessFilterExtension pf =
                            context.GetExtension<ProcessFilterExtension>();
        if (pf == null)
        {
            Match.Set(context, true);
        }
        else
        {
            IsMatch.Set(context, pf.IsMatchingProcess(Process.Get(context)));
        }
    }
}
```

15. Now we need to amend the flowchart to use the filter. Open **Workflow1.xaml**, this will open the workflow designer. Double click on the flowchart to edit it as a breadcrumb.
16. Add a new variable to the flowchart (using the Variable window) of type `Boolean` called `processMatch`. We will use this to store the result of the filter
17. Drag a `ProcessFilterActivity` on to the flowchart between the `GetProcessesActivity` and the `WriteLineActivity` (you may want to move everything apart from the `GetProcessesActivity` down in the designer)
18. With the `ProcessFilterActivity` highlighted set its `Process` argument to the expression `Processes[processCount]` and its `IsMatch` argument to `processMatch`. This allows the filter to test the current process in the iteration
19. Drag a `FlowDecision` on to the flowchart below the `ProcessFilterActivity` and set its `Condition` property to `processMatch`
20. Delete the link between the `GetProcessesActivity` and the `WriteLineActivity` and the link from the first `FlowDecision` you added. You delete a link by selecting it and pressing the Delete key.
21. Create a link from the `GetProcessesActivity` to the `ProcessFilterActivity`
22. Create a link from the `ProcessFilterActivity` to the new `FlowDecision`
23. Create a link from the new `FlowDecision`'s `true` branch to the `WriteLineActivity`
24. Create a link from the new `FlowDecisions`'s `false` branch to the `Assign`
25. Create a link from the original `FlowDecision`'s `true` branch to the `ProcessFilterActivity`
26. You can move the activities and links around by selecting them and dragging to make the flowchart clearer



27. Finally we will add the `StartsWithFilter` to the workflow instance. Open the **Program.cs** file.
28. By default the lightweight execution model is used. To add custom extensions you need to create an instance of `WorkflowInvoker` passing the workflow to its constructor
29. Create an instance of the `StartsWithFilter` class and add it to the `Extensions` collection of the workflow invoker
30. Call `Invoke` on the workflow invoker object

```
static void Main(string[] args)
{
    Activity workflow1 = new Workflow1();
    var invoker = new WorkflowInvoker(workflow1);

    invoker.Extensions.Add(new StartsWithFilter("s"));
    invoker.Invoke();
}
```

31. Compile and run the application. You will see only those processes listed that start with the letter(s) you specified. If you have time create another filter (perhaps testing memory working set of the process and plug that one in instead of the `StartsWithFilter`)

Solutions

after\WF45.sln