Join the discussion @ p2p.wrox.com

Wrox Programmer to Programmer™

Real World .NET, C#, and Silverlight®

Indispensible Experiences from 15 MVPs

Bill Evjen, Dominick Baier, György Balássy, Gill Cleeren, David Giard, Alex Golesh, Kevin Grossnicklaus, Caleb Jenkins, Jeffrey Juday, Vishwas Lele, Jeremy Likness, Scott Millett, Christian Nagel, Christian Weyer, Daron Yöndem

12 The WF Programming Language

by Vishwas Lele

Windows Workflow Foundation 4.0 (WF 4) is a programming language based on higher-level abstractions suitable for implementing business processes. Even though the WF 4 programs are likely to be authored using graphical tools (such as the Workflow Designer), developers must understand the fundamentals of the Extensible Application Markup Language (XAML) based declarative syntax to make the most of the features offered by WF 4.

This chapter starts with the notion of `Activity` as a basic construct of a WF 4 program. In this chapter, you learn about language fundamentals, including arguments and variables, scoping rules, expression support, and flow control constructs. In addition, you learn ways to extend the WF 4 framework by building domain-specific custom activities. Finally, this chapter examines hosting WF 4 programs in Windows AppFabric.

This chapter is designed for .NET developers who want to incorporate WF 4 into their solutions to ease the authoring of business processes. This chapter assumes that you have a good understanding of C# and .NET.

You can download all the code snippets illustrated in this chapter as part of the download package for this book located on this book's companion website (`www.wrox.com`).

GETTING STARTED

The key objective of any line-of-business (LOB) application is to implement the underlying business processes. As you can imagine, business processes come in all sizes and shapes. Interestingly, though, they exhibit two common traits:

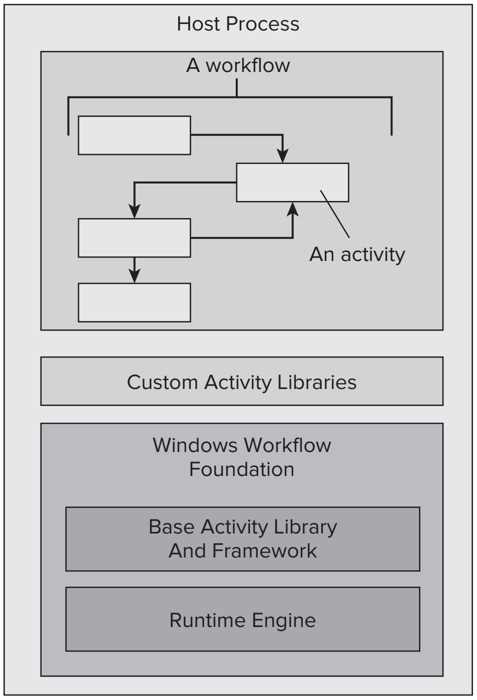
- Business processes are *interactive*.
- Business processes are *long running*.

Given this commonality, it makes sense to provide a consistent framework to implement business processes. This is where Windows WF 4 comes in.

WF 4 is a part of the .NET Framework 4.0 designed to ease the implementation of business processes. It may be helpful to think of WF 4 as a language that raises the abstraction level for implementing interactive, long-running applications. Similar to a traditional software program assembled using language statements, WF 4 can be assembled using activities (units of work).

Consider a traditional execution environment where a program is executed. Again, the WF 4 runtime performs a similar role. The WF 4 runtime executes the workflow activities. In addition, it also provides a set of services, such as automatically loading and unloading long-running programs, persisting the state, and flow control.

Figure 12-1 shows a conceptual model of WF 4. WF 4 programs can be hosted with an operating system process. This includes hosting within a custom application process (referred to as *self-hosting*), or within a system-provided host such as Windows Process Activation Service (WAS). The runtime engine is responsible for executing a workflow program. WF 4 comes with a set of activities that is part of the base activity library. The most fundamental extensibility point of WF 4 is the capability to build custom activities.


Host Process
A workflow
An activity
Custom Activity Libraries
Windows Workflow Foundation
Base Activity Library And Framework
Runtime Engine
FIGURE 12-1: WF 4 conceptual model

In this chapter, the terms *WF 4 program* and *workflow* are used interchangeably. Both refer to a program composed of a set of activities.

Now start by writing a WF 4 program that is the same for all languages: print the words `hello, world`.

As mentioned, an activity is a building block for a WF 4 program. The WF 4 program, whatever its size, is a collection of activities. WF 4 programs are typically specified using the declarative XAML. Interestingly, the root element is also an activity. In other words, the WF 4 program is itself an activity.

In Listing 12-1, the parent activity is made up on a `Sequence` activity. A sequence activity, as the name suggests, is made up of a sequence of activities sequentially executed. In this example, the sequence activity consists of a single `WriteLine` activity responsible for writing the text `"hello, world"`.

LISTING 12-1: HelloWorld in WF 4

```
<Activity x:Class="ProgrammingWF4.HelloWorld"
     xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Sequence >
    <WriteLine Text="hello, world" />
  </Sequence>
</Activity>
```

Now that you have this WF 4 program ready, it is time to run it. Listing 12-2 shows a console program that runs the workflow.

LISTING 12-2: Running HelloWorld

```
static void Main(string[] args)
{
wf = new HelloWorld();
AutoResetEvent syncEvent = new AutoResetEvent(false);

WorkflowApplication wfApp = new WorkflowApplication(wf);

// Handle the desired lifecycle events.
wfApp.Completed = delegate(WorkflowApplicationCompletedEventArgs e)
{
    syncEvent.Set();
};

// Start the workflow.
wfApp.Run();

// Wait for Completed to arrive
syncEvent.WaitOne();


}
```

First, you create an instance of the `WorkflowApplication` class. This class acts as the host for a single WF 4 program instance. The constructor of `WorkflowApplication` class takes an instance of an activity as a parameter. As mentioned, the WF 4 program is also an activity. This enables you to pass an instance of the `HelloWorld` WF 4 program. After the `WorkflowApplication` instance has been created, you can simply run `HelloWorld` by calling the `Run` method.

You must wait for the workflow instance to complete before exiting the program. This is because the workflow instance is being run on a thread other than the main thread. So, even though the execution on the main thread is complete, the thread running the workflow may still be active.

You achieve this by subscribing to the workflow completion event. Within the workflow completion event handler, you set the event to a `Signaled` state, thus allowing the function `Main` to complete.

Relying on a host class (such as the `WorkflowApplication`) to run a workflow and to subscribe for completion events are patterns that are common across WF 4 programming. Later in this chapter, you learn about more advanced versions of the host class (such as `WorkflowServiceHost`) that support hosting of multiple WF 4 program instances.

WF 4 also supports a lightweight way to invoke workflow programs using the `WorkflowInvoker` class, as shown here:

```
wf = HelloWorldInCode();
WorkflowInvoker wi = new WorkflowInvoker(wf);
wi.Invoke();
```

Declarative Workflow Syntax

WF 4 offers a number of ways to author workflows. Workflows can be developed imperatively in code using a managed language such as C#. Workflows can also be written using declarative XAML. The program to print `"hello, world"` can be represented in code as shown here:

```
Activity wf = new Sequence
             {
                 Activities =
                 {
                     new WriteLine
                     {
                         Text = "hello, world"
                     }
                 }
             };
```

However, the preferred authoring mode is the declarative mode. This is mainly because of the following:

- Declarative programs are easier to analyze and can be manually edited. Remember that the main reason for using WF 4 is to implement business processes. Authoring workflows in a declarative syntax makes it is easier for business analysts — the vast majority of them being nondevelopers — to follow the business logic.
- Declarative programs are easily transferable to different execution environments. For example, a XAML-based WF 4 program hosted within Windows AppFabric can be transferred to CRM 5.0 by a simple file copy operation.
- Because XAML is based on XML, you can represent the hierarchies in a human- and machine-readable way. For example, it is easy to represent a hierarchy of activities nested inside a parent activity.
- Declarative syntax makes it easier to author the workflow using a visual designer. Figure 12-2 shows the workflow designer view of the WF 4 program.

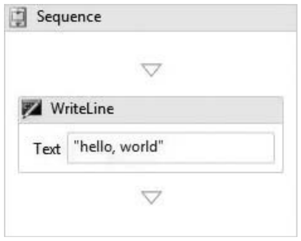


FIGURE 12-2: Workflow designer view of the example WF 4 program

Variables and Arguments

You can extend the example program to pass in parameters. For example, instead of hard-coding the text "hello, world" inside the WriteLine activity, you can pass a string as an argument. Listing 12-3 illustrates the use of parameters within a WF 4 program.

LISTING 12-3: Passing Arguments into HelloWorld

```
<Activity  x:Class="ProgrammingWF4.HelloWorld2"
           xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
           xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <x:Members>
    <x:Property Name="argument1" Type="InArgument(x:String)" />
  </x:Members>
  <Sequence >
    <Sequence.Variables>
      <Variable x:TypeArguments="x:String" Name="var1" />
    </Sequence.Variables>
    <Assign >
      <Assign.To>
        <OutArgument x:TypeArguments="x:String">[var1]</OutArgument>
      </Assign.To>
      <Assign.Value>
        <InArgument x:TypeArguments="x:String">[argument1]</InArgument>
      </Assign.Value>
    </Assign>
    <WriteLine Text="[var1]" />
  </Sequence>
</Activity>
```

You declare an incoming argument called argument1 of type string. Because this is an incoming argument, its value is set by the runtime before the activity is invoked. You have added a variable called var1 under the Sequence activity. The scope of this variable is limited to the lifetime of the Sequence activity. So, when the Sequence activity completes its execution, var1 cannot be accessed.

Next, you added an Assign activity that sets the value of var1 based on the incoming argument. Finally, you have modified the WriteLine activity to print var1, instead of the hard-coded text. Notice that var1 appears within square brackets. This represents an expression that must be evaluated. The result of the expression is stored in the Text property of the WriteLine activity.

Now that the WF 4 program includes the notion of an incoming argument, change the program that invokes it. Listing 12-4 shows how the parameters are passed in when a new instance of a WF 4 program is created.

LISTING 12-4: Passing Arguments into HelloWorld

```
Dictionary<string, object> arguments = new Dictionary<string, object>();
arguments.Add("argument1", "hello, world");
WorkflowApplication wfApp =
            new WorkflowApplication( new HelloWorld(),arguments);
```

Here you use an overloaded constructor of the `WorkflowApplication` that enables you to supply a WF 4 program instance and a dictionary of arguments.

Expressions

As mentioned, the square brackets represent an expression. By default, expressions can be literal values, or Visual Basic code that contains variables, arguments, constants, and so on, combined with operators to yield a value. Earlier, you saw an example of a literal value when you passed `"hello, world"` as the argument to the `WriteLine` activity.

The following code snippet shows two Visual Basic expressions. `[var1]` represents an expression in Visual Basic syntax that evaluates to the left side of the assignment expression. `[UCase(argument1)]` represents an expression in Visual Basic syntax that evaluates to the right side of the assignment expression.

```
<Assign >
      <Assign.To>
             <OutArgument x:TypeArguments="x:String">[var1]</OutArgument>
      </Assign.To>
      <Assign.Value>
             <InArgument
x:TypeArguments="x:String">[UCase(argument1)]</InArgument>
      </Assign.Value>
    </Assign>
```

Visual Basic expressions are compiled in-memory by the WF 4 runtime and included as part of the workflow logic. For the Visual Basic compiler to resolve the function `UCase`, you must import the appropriate assemblies, as shown here:

```
<Activity  x:Class="ProgrammingWF4.HelloWorld3"
     mva:VisualBasic.Settings="Assembly references
     and imported namespaces serialized as XML namespaces"
 xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 xmlns:mv="clr-namespace:Microsoft.VisualBasic;assembly=System"
 xmlns:mva="clr-amespace:Microsoft.VisualBasic.Activities;
     assembly=System.Activities">
```

Properties

You have now seen how to use parameters for passing values into an activity. Another way to achieve this is by setting any Common Language Runtime (CLR) properties exposed by an activity.

The limitation of this approach is that values passed into an activity are known at compile time. This also means that, for all instances of the WF 4 program, the property remains the same. This is different from the usage of arguments, where it is possible to pass a value at the time a WF 4 program is instantiated.

The following code snippet shows an example of a CLR property `MethodName` exposed by the `InvokeMethod` activity. The `MethodName` property is set to a method named `print` at compile time, and is the same for all instances of the WF 4 program.

```
<InvokeMethod DisplayName="Instance Method Call" MethodName="print">
 <InvokeMethod.TargetObject>
```

```
        <InArgument x:TypeArguments="msi:TestClass">[New
TestClass()]</InArgument>
      </InvokeMethod.TargetObject>
    </InvokeMethod>
```

This snippet also provides another example of a Visual Basic expression, as mentioned earlier. As the name suggests, the `InvokeMethod` activity can be used to call a public method of a specified object or type. In the previous example, Visual Basic expression `[New TestClass()]` instantiates the target object of type `TestClass`:

```
public class TestClass
 {
     public void Print ()
     {
         Console.WriteLine("hello, world");
     }
 }
```

The `InvokeMethod` activity then executes the method based on the value of the `MethodName` property.

"Dynamic" Properties

Properties and arguments enable data to be passed into a workflow. However, these values cannot be changed after the workflow execution starts.

To dynamically vary the passed-in data during the course of execution of the workflow, you can use the `ActivityFunc` activity. `ActivityFunc` is an activity that represents a callable method (delegate) that returns an argument. The delegate is defined by the workflow application, and passed into the workflow as a property. So, when the workflow references the property, the delegate gets invoked.

This pattern is illustrated in Listing 12-5.

LISTING 12-5: Dynamically Varying the Values Passed into a Workflow

```
<Activity>
<x:Members>
    <x:Property Name="Text" Type="ActivityFunc(x:String)" />
  </x:Members>
  <Sequence>
    <Sequence.Variables>
      <Variable x:TypeArguments="x:String" Name="PrintString" />
    </Sequence.Variables>
    <InvokeFunc x:TypeArguments="x:String" Result="[PrintString]">
      <PropertyReference x:TypeArguments="ActivityFunc(x:String)"
           PropertyName="Text" />
    </InvokeFunc>
    <WriteLine Text="[PrintString]" />
  </Sequence>
</Activity>
```

Here you declare a property called `Text` of an `ActionFunc` type. Notice the definition of the `Text` property. It is a type of a delegate that returns a string. Later, in the workflow definition, you invoke `ActionFunc` using the `InvokeFunc` activity. The `Result` is set to a workflow variable called `PrintString`. The last step of the workflow is to print the `PrintString` variable using the `WriteLine` activity.

Now that you have seen the workflow code, here is the code to invoke the workflow. You create the activity as part of the property initialization, and set the `Text` property to an instance of `ActivityFunc`.

```
Activity wf = new HelloWorld5
{
     Text = new ActivityFunc<string>
     {
        Handler = new TextGenerator
        {

        }
     }
 };

 WorkflowApplication wfApp = new WorkflowApplication(wf);
```

CONTROLLING THE FLOW OF EXECUTION

In the earlier sections, you saw how variables and properties can be defined within a WF 4 program. You also learned about the `Assign` activity that enabled you to define expressions, as well as set variables and properties.

Now look at activities that you can use to control the flow of execution of a WF 4 program. WF 4 supports two styles of flow control — *procedural styles* and *flowchart styles*. Now review each workflow style.

Procedural Style

The procedural style offers constructs similar to the ones offered by other procedural languages, such as the `while`, `if else`, and `switch`. Listing 12-6 shows an example of a procedural workflow.

LISTING 12-6: Procedural Style Constructs

```
<Activity>
  <x:Members>
    <x:Property Name="argument1" Type="InArgument(s:String[])" />
  </x:Members>
  <Sequence >
```

```
            <Sequence.Variables>
              <Variable x:TypeArguments="x:Int32" Default="[0]" Name="count" />
              <Variable x:TypeArguments="x:Boolean" Default="[false]" Name="finished" />
            </Sequence.Variables>
            <While Condition="[Not finished]">
              <Sequence >
                <If Condition="[count &lt; argument1.Length]" >
                  <If.Then>
                    <WriteLine  Text="[&quot;Hello &quot; &amp;
                         argument1(count).ToString()]" />
                  </If.Then>
                  <If.Else>
                    <Assign x:Key="0" >
                      <Assign.To>
                        <OutArgument x:TypeArguments="x:Boolean">[finished]</OutArgument>
                      </Assign.To>
                      <Assign.Value>
                        <InArgument x:TypeArguments="x:Boolean">[true]</InArgument>
                      </Assign.Value>
                    </Assign>
                  </If.Else>
                </If>
                <Assign >
                  <Assign.To>
                    <OutArgument x:TypeArguments="x:Int32">[count]</OutArgument>
                  </Assign.To>
                  <Assign.Value>
                    <InArgument x:TypeArguments="x:Int32">[count + 1]</InArgument>
                  </Assign.Value>
                </Assign>
              </Sequence>
            </While>
          </Sequence>
        </Activity>
```

An array of a string is passed in as a parameter. The workflow is composed of a single `Sequence`. Within the scope of the `Sequence` activity, you define two variables — `count` and `finished`. The procedural steps nested within the `Sequence` activity are self-explanatory.

A `While` activity is used to loop over until the expression [`Not Finished`] returns `false`. Within each iteration of the loop, an `If` activity is used to check if the `count` variable is less than the length of the array passed in as an argument. If this is indeed the case, the `WriteLine` activity is used to print the greeting. Alternatively, if the count becomes equal to the length of the array passed in, the `Assign` activity is used to set the finished variable to `true`. Each iteration of the loop ends by incrementing the count variable by `1` using the `Assign` activity.

Because the code in Listing 12-6 is rather long, reviewing the equivalent WF designer view shown in Figure 12-3 may be helpful.

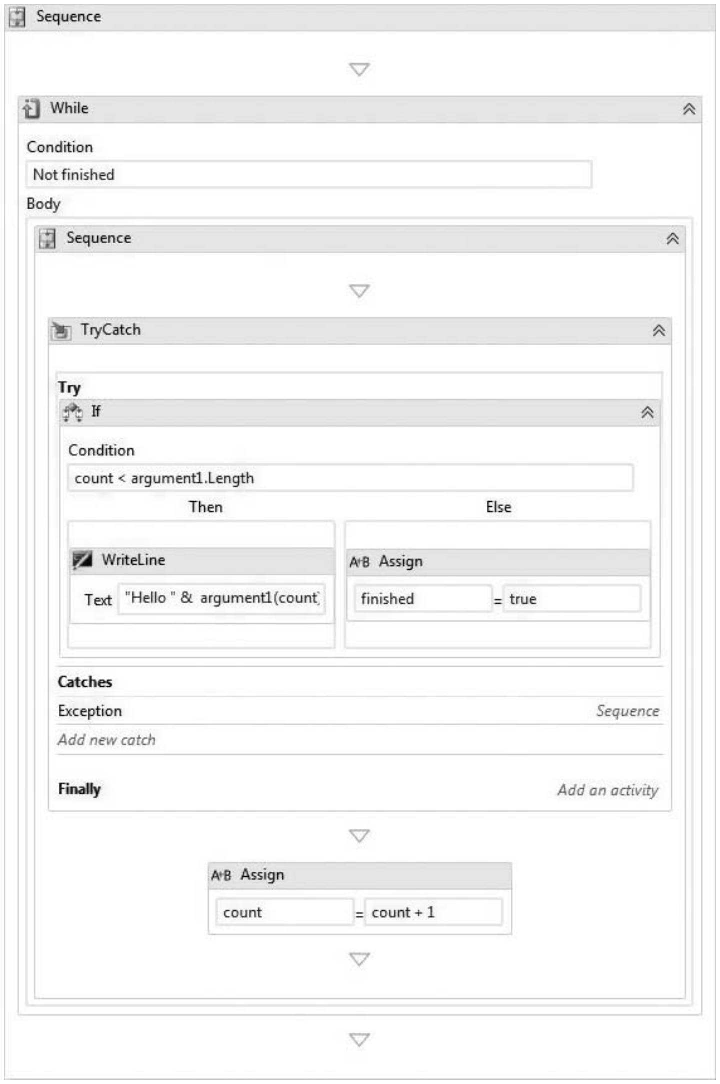


FIGURE 12-3: Procedural style in WF designer view

Exception Handling

Similar to other programming languages, WF 4 offers exception-handling constructs. As you learned earlier, most of the constructs in WF 4 are activities. Exception handling is no different. WF 4 provides the following activities for exception handling: `TryCatch`, `Throw`, and `Finally`.

You can easily extend the previous example to handle exceptions that may be thrown during the execution of the workflow steps. Listing 12-7 shows the use of the `TryCatch` activity to wrap the procedural steps discussed earlier.

LISTING 12-7: Exception Handling

```
        <TryCatch >
            <TryCatch.Try>
              <If Condition="[count &lt; argument1.Length]" >
                <If.Then>
                  <WriteLine  Text="[&quot;Hello &quot; &amp;
    argument1(count).ToString()]" />
                </If.Then>
                <If.Else>
                  <Assign >
                    <Assign.To>
                      <OutArgument x:TypeArguments=
                           "x:Boolean">[finished]</OutArgument>
                    </Assign.To>
                    <Assign.Value>
                      <InArgument x:TypeArguments="x:Boolean">[true]</InArgument>
                    </Assign.Value>
                  </Assign>
                </If.Else>
              </If>
            </TryCatch.Try>
            <TryCatch.Catches>
              <Catch x:TypeArguments="s:Exception" >
                <ActivityAction x:TypeArguments="s:Exception">
                  <Sequence >
                    <WriteLine  Text="[&quot;Exception thrown&quot;]" />
                    <TerminateWorkflow  Reason="exception thrown" />
                  </Sequence>
                </ActivityAction>
              </Catch>
            </TryCatch.Catches>
          </TryCatch>
```

In case an exception is thrown, the `Catch` activity that takes an argument of the `Exception` type is invoked. The `Catch` activity, in turn, contains steps that are executed when an exception is caught. In Listing 12-7, the `WriteLine` activity is used to write diagnostic text to `console`, followed by calling the `TerminateException` activity to terminate the workflow execution.

Transaction and Compensation

WF 4 provides constructs to ensure the integrity of the data touched by the workflow program. This includes support for transactions and compensation.

WF 4 provides support for transactions in the form of the `TransactionScope` activity. The idea behind this activity is not different from the notion of transactions that developers are already familiar with. A WF program can embed one or more activities within the `TransactionScope` activity to ensure that they all execute as part of a single transaction.

Listing 12-8 shows an example of `TransactionScope` in action.

LISTING 12-8: TransactionScope

```
<Activity>
  <Sequence>
    <WriteLine>["    Begin workflow"]</WriteLine>
    <TransactionScope >
      <TransactionScope.IsolationLevel>3</TransactionScope.IsolationLevel >
        <TransactionScope.AbortInstanceOnTransactionFailure >true
</TransactionScope.AbortInstanceOnTransactionFailure>
      <Sequence>
        <WriteLine>["    Begin TransactionScope"]</WriteLine>
        <ntsa:PrintTransactionId/>
        <WriteLine>["    End TransactionScope"]</WriteLine>
      </Sequence>

    </TransactionScope>
    <WriteLine>["    End workflow"]</WriteLine>
  </Sequence>
</Activity>



   public sealed class PrintTransactionId : NativeActivity
    {
        protected override void Execute(NativeActivityContext context)
        {
//Access to the current transaction in Workflow is through the
//GetCurrentTransaction method on a RuntimeTransactionHandle
              RuntimeTransactionHandle rth =
                   context.Properties.Find(typeof(RuntimeTransactionHandle)
                   .FullName) as RuntimeTransactionHandle;
             Console.WriteLine("    TransactionID: "
               +rth.GetCurrentTransaction(context).
TransactionInformation.LocalIdentifier.ToString());
        }
    }
```

In this example, the `Sequence` activity is placed within the `TransactionScope`. The `Sequence` activity, in turn, contains two `WriteLine` activities and a custom activity called `PrintTransactionId` that prints the transaction identifier. The code for `PrintTransactionId`, taken from the MSDN WF 4 sample `NestedTransactionScopeSample`, is provided in the listing. The `PrintTransactionID` activity inherits from the `NativeActivity` base class. You learn about the `NativeActivity` base class later in this chapter.

Listing 12-8 also shows how additional properties can be set on the `TransactionScope` activity. For example, the isolation level is set to `3` (`ReadUncommitted` within the workflow designer). Additionally, the code indicates to the WF runtime that the workflow instance must be aborted in case the transaction fails.

You can also nest `TransactionScope` activities. Nested `TransactionScope` activities reuse the ambient transaction (as opposed to spawning a new transaction). However, nested `TransactionScope` activities should not cause a conflict with the outer `TransactionScope` activity.

For example, the nested TransactionScope should have the same isolation level as the outside TransactionScope.

As powerful as the TransactionScope activity is, it is not suitable for all scenarios. This is especially true for scenarios in which the duration of the transaction is long, thereby making it expensive to hold on to the resources participating in the transaction. In such situations, compensation — explicitly reversing an action — is more suitable. WF 4 provides rich constructs to help workflow authors model compensation logic into their workflows. The key construct related to compensation is the CompensationActivity. Listing 12-9 shows its usage.

LISTING 12-9: CompensationActivity

```
<Activity>
    <Sequence>
    <Sequence.Variables>
      <Variable x:TypeArguments="CompensationToken" Name="token" />
    </Sequence.Variables>

    <WriteLine>["Start of workflow."]</WriteLine>

    <CompensableActivity Result="[token]">

      <WriteLine>["CompensableActivity: Do Work."]</WriteLine>

      <CompensableActivity.CompensationHandler>
        <WriteLine>["CompensableActivity: Undo Work."]</WriteLine>
      </CompensableActivity.CompensationHandler>

      <CompensableActivity.ConfirmationHandler>
        <WriteLine>["CompensableActivity: Do Work Confirmed."]</WriteLine>
      </CompensableActivity.ConfirmationHandler>

      <WriteLine>["CompensableActivity: Do Work."]</WriteLine>

    </CompensableActivity>

    <Compensate Target="[token]" />

    <WriteLine>["End of workflow."]</WriteLine>

  </Sequence>
</Activity>
```

In this example, the activity to be compensated is a WriteLine activity that prints the text "Do Work". Now you must specify a CompensationHandler property that represents the undo activity that will be invoked when compensation is to be performed. In this example, the compensation handler is also a WriteLine activity that simply prints the text "Undo Work".

In addition, you have another handler called the ConfirmationHandler that can be used to specify custom logic that will execute when a CompensationActivity is confirmed. The workflow has reached a point in which the compensation action associated with the CompensationActivity is no longer to be invoked. In this example, the `ConfirmationHandler` simply prints the text `"Do Work Confirmed"`.

In Listing 12-9, just below the `CompensationActivity`, is an activity called `Compensate` that is used to explicitly invoke the compensation handler. The `Compensate` activity takes as a parameter a token that points to the instance of the `CompensationActivity` whose action must be reversed. This token was populated by the outgoing parameter of the `CompensationActivity` called `Result`.

If you run this program, you can see the normal execution of the `CompensationActivity`, followed by the compensation action. If you had placed a `Confirm` activity in place of the `Compensate` activity, you would also see the normal execution of the `CompensationActivity`, followed by the confirmation action. As discussed, after the `ConfirmationHandler` has been invoked, calling `Compensate` would result in an error. This is to be expected because the invocation of the `Compensation` activity has already indicated to the WF 4 runtime that no further compensation is needed.

So far, you have been explicitly invoking compensation. This is not always needed because the WF 4 runtime can implicitly invoke the compensation behavior based on whether the workflow completed successfully. If there were an error in the execution of the workflow after the `CompletionActivity` successfully completed, WF 4 runtime invokes the compensation handler.

Parallel Execution

All the examples so far have been based on sequential execution. In other words, the WF 4 runtime executes the next workflow step only after the preceding step is complete. This begs the question about a scenario in which a parallel (or concurrent) execution would be needed.

As mentioned earlier, WF 4 is a language based on higher-level abstractions. This is why there are no built-in constructs to create and synchronize threads. In fact, the WF 4 runtime, by design, allows only a single workflow activity to execute at any given point in time. But that does not mean a WF 4 program cannot schedule more than one workflow activity for execution.

Listing 12-10 shows the use of the `Parallel` activity that makes it possible to schedule more than one workflow activity.

LISTING 12-10: Parallel Execution

```
<Activity>
  <Sequence >
    <Parallel >
      <Sequence >
        <WriteLine  Text="Parallel Path1 start" />
        <WriteLine  Text="Parallel Path1 end" />
      </Sequence>
      <Sequence >
        <WriteLine  Text="Parallel Path2 start" />
        <WriteLine  Text="Parallel Path2 end" />
      </Sequence>
    </Parallel>
  </Sequence>
</Activity>
```

As shown in the example, the `Parallel` activity is a collection of `Sequence` activities. The `Parallel` activity, acting as the parent activity, iterates over the collection of `Sequence` activities and schedules each of them for execution. As mentioned, only one of them gets to execute at any time. So, scheduling multiple activities is not actually doing any good. In effect, this program is going to execute sequentially.

To take advantage of the parallel execution, you need one of the executing activities to yield control back to the runtime. This typically happens when the executing activity is waiting for an event to complete.

For example, an activity that invokes a web service asynchronously can yield control back to the workflow runtime while it waits for its completion. When that happens, the workflow runtime can execute another activity that has been scheduled for execution.

Because this discussion has not addressed invoking web services up until now, try to simulate the "yield" behavior by invoking a `Delay` activity.

```
<WriteLine  Text="Parallel Path1 start" />
        <Delay Duration="00:00:01"  />
        <WriteLine  Text="Parallel Path1 end" />
```

As the name suggests, you can use the `Delay` activity to pause the execution of a workflow for a specified period of time. The paused activity, in turn, prompts the workflow runtime to execute another activity that has been scheduled for execution.

The order in which scheduled activities are executed is indeterminate.

Flowchart Style

As discussed, the procedural style mimics the common programming language constructs, such as conditionals, looping, and exception handling. This makes it easy to author well-structured business processes.

The concept of a well-structured process is familiar to developers — for example, they have been taught to use a looping construct when there is a need to return the flow of execution to an earlier location in the workflow. However, nondevelopers may find it difficult to accomplish this because it involves adding a looping construct, setting up a looping condition, and so on. They need an easier approach to alter the flow of execution.

This is where the flowchart style workflows come in. As the name suggests, this style mimics the well-known paradigm for designing programs. In a nutshell, a flowchart consists of nodes and arcs. Nodes represent the steps in a flowchart, whereas the arcs represent the potential path of execution through the nodes. In WF 4, the node maps to a `FlowStep` activity, and the arc maps to a `FlowDecision` or `FlowSwitch` activities.

Now take a look at an example that highlights the flexibility offered by the flowchart style. Implement a WF 4 program that mimics the business process shown in Figure 12-4.

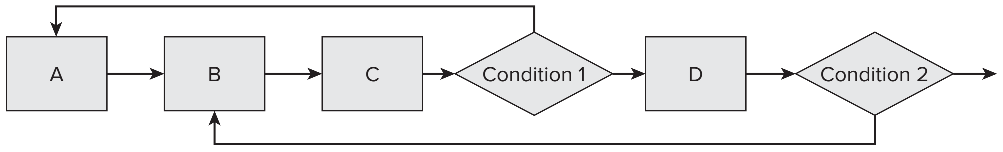


FIGURE 12-4: Business process suitable for flowchart style

Listing 12-11 shows the WF 4 program that implements the business process shown in Figure 12-4.

LISTING 12-11: Flowchart Style

```
  <Activity  ">
   <Flowchart >
     <Flowchart.StartNode>
       <FlowStep x:Name="__ReferenceID0">
         <Sequence >
           <WriteLine  Text="Step A" />
           <WriteLine  Text="Step B" />
           <WriteLine  Text="Step C" />
         </Sequence>
         <FlowStep.Next>
           <FlowDecision x:Name="__ReferenceID1" Condition="[True = True]" >
             <FlowDecision.True>
               <FlowStep x:Name="__ReferenceID2">
                 <Sequence >
                   <WriteLine  Text="Step D" />
                 </Sequence>
                 <FlowStep.Next>
                   <FlowDecision x:Name="__ReferenceID3" Condition="[True = True]" >
                     <FlowDecision.False>
                       <x:Reference>__ReferenceID0</x:Reference>
                     </FlowDecision.False>
                   </FlowDecision>
                 </FlowStep.Next>
               </FlowStep>
             </FlowDecision.True>
             <FlowDecision.False>
               <x:Reference>__ReferenceID0</x:Reference>
             </FlowDecision.False>
           </FlowDecision>
         </FlowStep.Next>
       </FlowStep>
     </Flowchart.StartNode>
   </Flowchart>
 </Activity>
```

The `Flowchart` activity is like a container that can hold any number of `Flowstep` activities. Each `FlowStep` activity models a step within the workflow. Notice the first `FlowStep` activity named `__ReferenceID0`. It executes a sequence activity that contains steps A, B, and C from Figure 12-4. There are two things to note about the `Sequence` activity.

First, you can mix procedural and flowchart styles. In this instance, the `FlowStep` activity is responsible for executing the `Sequence` activity. It is also possible to place a `Flowchart` within a `Sequence` activity.

Second, steps A, B, and C are placed within a `Sequence` activity. Although you could model each step as a distinct `FlowStep` activity, placing them inside a `Sequence` activity allows you to potentially narrow the scope. You can define variables that are visible only within the `Sequence` activity.

The `FlowStep.Next` property points to the next step in the flow chart. In this example, the `FlowDecision` evaluates an expression `True = True` (admittedly contrite to keep the example simple). If the expression evaluates to `true`, you execute another `Sequence` activity that, in turn, executes step D. Alternatively, if the expression evaluates to `false` (not possible in this example), it would cause the flowchart execution to return to the first `FlowStep` activity named `__ReferenceID0`.

The last step in the workflow is to add a `FlowDecision` activity after step D. Again, to keep things simple, you use a hard-code `true = true` expression. If the expression associated with this `Flow.Decision` evaluates to `false`, you would return to the `FlowStep` named `__ReferenceID0`.

Herein lies the most flexible aspect of the flowchart style — the capability to execute a workflow step by name, as shown in the following code snippet. This construct is what enables you to easily implement the business process shown in Figure 12-4.

```
<FlowDecision.False>
  <x:Reference>__ReferenceID0</x:Reference>
</FlowDecision.False>
```

You could have implemented the aforementioned business process by relying only on procedural constructs. Figure 12-5 shows an implementation based on procedural constructs. The complexity is evident from the block diagram — a XAML implementation would be more complicated.

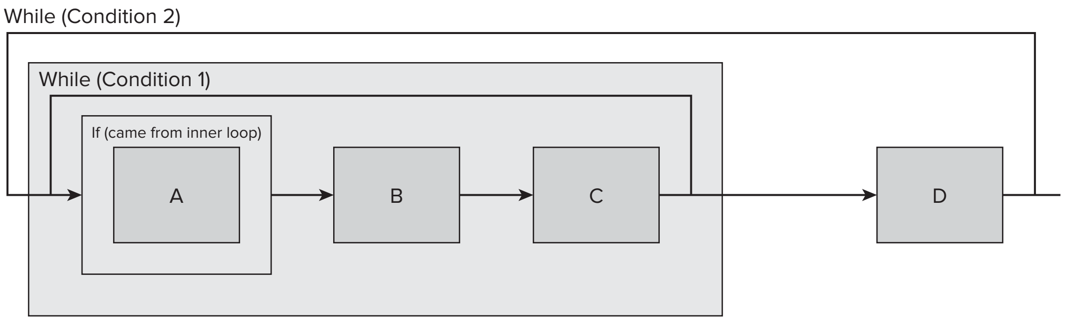


FIGURE 12-5: Implementation based on procedural style

Messaging

As you would expect, WF 4 provides messaging constructs designed to enable workflow programs to communicate with external entities. Consistent with the rest of the WF 4 theme, messaging constructs are also activities. These activities rely on Windows Communication Foundation (WCF) for providing the communication stack. Fortunately for the workflow authors, though, the messaging activities abstract many of the WCF details. For example, a WCF contract can be automatically inferred based on the set of messaging activities that make a workflow.

There are two core messaging activities — `Send` (used to send messages) and `Receive` (used to receive messages). In addition to the core activities, there are two additional activities, `SendReply` and `ReceiveReply`, that enable WF 4 programs to send and receive a response for a preceding invocation of `Receive` and `Send` activities, respectively. Together, these messaging activities enable workflow authors to model a variety of message-exchange patterns, including request-response, bidirectional, and one-way patterns.

Listing 12-12 shows an example of using `Receive` and `SendReply` activities. In addition, this example also illustrates the concept of *correlation* — a mechanism that associates activities based on a shared context.

LISTING 12-12: Receive and SendReply

```
        <PickBranch Trigger="{x:Reference __ReferenceID0}">

            <p:SendReply >

              <p:SendReply.Request>

                <p:Receive x:Name="__ReferenceID0"  CanCreateInstance="True"
                CorrelatesWith="[TwitterIdHandle]"  OperationName="Hello"
               ServiceContractName="ISendGreeting">

                  <p:Receive.CorrelatesOn>

                    <p:MessageQuerySet>

                      <p:XPathMessageQuery x:Key="TwitterId">

                        <p:XPathMessageQuery.Namespaces>

                          <ssx:XPathMessageContextMarkup>

                            <x:String x:Key="xgSc">http://tempuri.org/</x:String>

                          </ssx:XPathMessageContextMarkup>

                        </p:XPathMessageQuery.Namespaces>

                        sm:body()/xgSc:Hello/xgSc:TwitterHandle

                      </p:XPathMessageQuery>

                    </p:MessageQuerySet>

                  </p:Receive.CorrelatesOn>

                  <p:ReceiveParametersContent>

                    <OutArgument x:TypeArguments=
                                    "x:String" x:Key="TwitterHandle" />

                              </p:ReceiveParametersContent>

                            </p:Receive>

                          </p:SendReply.Request>

                          <p:SendParametersContent>

                            <InArgument x:TypeArguments="x:String"
                                x:Key="Value">[NumOfRequests.ToString()] </InArgument>

                          </p:SendParametersContent>

                        </p:SendReply>

                      </PickBranch >
```

The functionality implemented in this example is simple. It can receive a message, sent to it by an external entity. In response, it provides a count of requests received within a given context. In other words, it returns the total number of correlated incoming requests. Before reviewing the messaging code, however, let's consider one other concept used in this example.

This example provides an opportunity to look at another important control-flow construct called the `Pick` activity. The `Pick` activity is similar to the `Parallel` activity in that it enables creation of multiple parallel branches for workflow execution. However, it is specifically designed to wait concurrently for multiple event triggers. Each parallel branch represents an instance of `PickBranch` activity.

For example, consider the `Pick` activity that is modeled — one `PickBranch` waits for a delay trigger to arrive, while another `PickBranch` waits for the `Receive` activity to be invoked. Of course, given that workflow runtime ensures causality, only one `PickBranch` can execute at any given time. The `Pick` activity is useful when implementing a state *machine style workflow* — a workflow style in which the flow is based on a discrete set of states.

For simplicity, Listing 12-13 shows an elided view of the code that includes the code nested within the `PickBranch` activity.

LISTING 12-13: Messaging

```
<Activity>

    <Sequence >

        <Sequence.Variables>

           <Variable x:TypeArguments="x:Boolean" Default="[True]"
                Name="Running" />
           <Variable x:TypeArguments="x:Int32"  Name="NumOfRequests" />

        </Sequence.Variables>
```

```
                <While >

                    <While.Variables>

                        <Variable x:TypeArguments="p:CorrelationHandle"
                                                              Name="TwitterIdHandle" />

                    </While.Variables>

                   <While.Condition>[Running]</While.Condition>

                   <Sequence>
                       <Assign >
                           <Assign.To>
                               <OutArgument
                                    x:TypeArguments=
                                         "x:Int32">[NumOfRequests]</OutArgument>
                           </Assign.To>
                           <Assign.Value>
                               <InArgument x:TypeArguments="x:Int32">
                                         [NumOfRequests + 1]</InArgument>
                           </Assign.Value>
                      </Assign>

                       <Pick >
                            Refer to Listing 12-12
                       </Pick >
                   </Sequence>
               </While >
         </Sequence>
     </Activity>
```

The `PickBranch` activity consists of two parts: a *trigger* and the *action*. In this example, the trigger that you are waiting on is a `Receive` activity with a name `__ReferenceID0`. This activity is responsible for receiving an incoming message. As stated earlier, based on the `Receive` activity attributes such as the `OperationName` and `ServiceContractName`, a WCF contract is inferred automatically.

The inferred WCF contract, combined with the WCF 4.0 feature to apply a default binding, means that there is no need for any WCF configuration (in code or in the configuration file) for this `Receive` activity. The `CanCreateInstance` attribute, when set to `true`, means that a new instance of the workflow can be created (if needed) to process the message received by the `Receive` activity.

The other two correlation-based attributes, `CorrelatesWith` and `CorrelatesOn`, need some explanation. Earlier you learned that correlation is about associating a set of activities. To understand why this is important, consider a scenario in which multiple external entities try to concurrently call the workflow program. As a result, multiple instances of the workflow program are created.

Now, it is up to the WF 4 runtime to deliver the incoming messages to appropriate instances. It does so by relating incoming requests that have a shared token (also referred to as the *correlation token*). Correlation, of course, is not limited to incoming messages; it could be about relating a request with a response. Correlation can be of two types:

- *Protocol-based correlation* relates messages based on data provided by the underlying infrastructure (for example, a token provided by the transport protocol).

- *Content-based correlation* relates messages based on the content of the message. (Listing 12-13 shows the use of content-based correlation.)

Now that you understand the notion behind correlation, continue the review of the remaining two correlation attributes.

Attribute `CorrelatesOn` specifies the part of the message content that will be used to relate the messages. In this instance, you rely on an XPath expression to specify the correlation token. The XPath expression points to the element `TwitterHandle` in the incoming SOAP request. So, if the SOAP request looked like the following example, the XPath expression would evaluate to `john`. Following is the value you correlate on.

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
     <s:Body>
         <Hello xmlns="http://tempuri.org/">
             <TwitterHandle>john</TwitterHandle>
         </Hello>
     </s:Body>
</s:Envelope>
```

The other attribute, `CorrelatesWith`, points to a workflow variable that stores the correlation token. The WF 4 runtime compares the result of the XPath expression to the variable pointed to by the `CorrelatesWith` attribute to determine where the message needs to be delivered.

Now that you understand the `Receive` activity, take a look at what's next in the workflow. The `SendReply` activity is responsible for sending a response to the caller. The most interesting part is the payload of the reply. You maintain a workflow variable `NumOfRequests` that is incremented each time you receive an incoming message. Listing 12-13 represents the overall flow of the workflow. As shown in Listing 12-12, the `SendReply` activity simply returns the current value of this workflow variable.

The last issue associated with messaging to examine is the client program used to call the workflow. Listing 12-14 shows the client code.

LISTING 12-14: Messaging Client

```
             BasicHttpBinding binding = new BasicHttpBinding();

            EndpointAddress address = new
EndpointAddress("http://localhost:8080/HelloWorldService");

            ChannelFactory<ISendGreeting> factory = new
                              ChannelFactory<ISendGreeting>();

            ISendGreeting proxy =
                  ChannelFactory<ISendGreeting>.CreateChannel(binding, address);

            using (proxy as IDisposable)
            {
                string res = proxy.Hello("john");
                res = proxy.Hello("john");
                 res = proxy.Hello("mary");
                 res = proxy.Hello("john");
            }
```

You use `ChannelFactory` to create a channel based on the endpoint information. Using the channel, you invoke the `Hello` method four times.

As discussed, the content-based routing scheme you have in place is based on a parameter being passed with the `Hello` method. This is why the first, second, and fourth calls (with parameter set to `"john"`) will all be associated to the same instance of the workflow. Consequently, the value returned upon the completion of the fourth call is `3`. The second call (with parameter set to `"mary"`) causes a separate workflow instance to be spawned. As a result, the value returned upon the completion of the second call is `1`.

BUILDING CUSTOM ACTIVITIES

So far in this chapter, you have learned about the use of built-in activities to author WF 4 programs. In addition to a rich set of built-in activities, WF 4 has also been designed to make it easy to develop and use custom activities.

As discussed, activities enable large workflows to be broken up into small steps that can be reused, thereby promoting reuse. Well-designed activities hide details of an operation from the workflow author. For example, a custom `SendMail` activity hides the implementation details associated with sending an e-mail. Workflow authors can simply add a `SendEmail` activity as a step within their workflow as needed.

Before looking at some code to create a custom activity, you need to recall that a workflow is also an activity. So, the examples of workflows you have seen so far can themselves be reused as activities by workflow authors (with adequate factoring of incoming and outgoing arguments). In other words, what you have seen so far are examples of custom activities declaratively assembled.

Building custom activities in this manner is a valid option. However, instances exist in which expressiveness of code is needed to building custom activities. The rest of this section focuses on building code-based custom activities.

WF 4 provides a set of base classes designed to accelerate the development of custom activities. Developers can select from this set, based on the type of activity being developed. Figure 12-6 shows the set of activity base classes provided by WF 4.

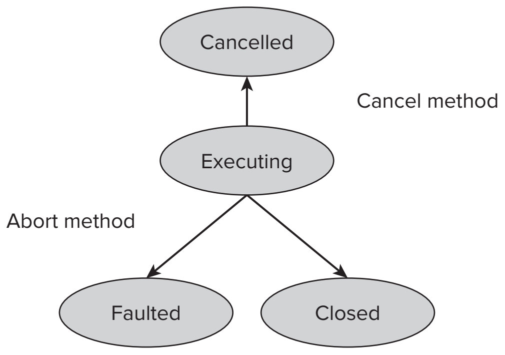


FIGURE 12-6: Activity base classes

Now look at each of these classes (`Activity`, `CodeActivity`, `AsyncCodeActivity`, and `NativeActivity`) in more detail.

Activity

`Activity` is an abstract base class that can be used to create activities using existing activities. You have already seen examples that use the `Activity` base class. The declarative workflows presented earlier had a root element of the `Activity` type. In essence, you simply added prebuilt activities (such as `While`, `Assign`, and `Sequence`) as needed to the `Activity` class. The `Activity` base class served as the composite or container class.

Now re-create the example from Listing 12-3 in which you declaratively authored a workflow. Use the `Activity` base class as a container, and rely on code-based constructs to re-create the workflow. Listing 12-15 shows the relevant code.

LISTING 12-15: Activity Base Class

```
     public sealed class HelloWorld : Activity
{

    public InArgument<string> argument1 { get; set; }public HelloWorld()
    {
    }
    protected override Func<Activity> Implementation
    {
        get
        {
            return () =>
           {Variable<String> var1 =
                   new Variable<String> { Name = "var1" };
            return new Sequence
               {

                   Variables =
                    {
                        var1
                    },

                   Activities =
                    {
                        new Assign<String>
                        {
                            To = new OutArgument<string> (ac =>
                                                            var1.Get(ac)),
                                                      Value = new
                                                           InArgument<string>
                                                           (ac=> argument1
                                                           .Get(ac))
                        },
                        new WriteLine
                        {
                            Text = new InArgument<string> (ac =>
                                                                  var1.Get(ac))
                        }

                    }

               };
           };
        }
        set
```

continues

LISTING 12-15 *(continued)*

```
                {
                    base.Implementation = value;
                }

            }
        }
```

The `Activity` base class has a property called `Implementation` that serves as the container for activities in the workflow. The `Get` accessor of the `Implementation` property defines the collection of activities that make up the workflow. Even though the initialization syntax is used to assemble the activities, the workflow logic is exactly the same.

You have a `Sequence` activity, which, in turn, contains an `Assign` activity that stores the incoming argument into a workflow `var1` variable. This is followed by a `WriteLine` activity that writes `var1` to `console`.

The only other noteworthy aspect of this listing is the pattern used to access the workflow variable and parameters. Rather than directly accessing the program properties (such as `argument1`), you must access them via the execution environment under which the activity is executing. Remember that the program is setting up only the workflow. The execution environment is accessed using the context (known as `ActvityContext`) provided by the `Activity` base class.

Now review an example of this pattern. To access the argument called `argument1` within the workflow, you must create an instance of `InArgument <string>` by passing in a lambda expression that takes the `ActivityContext` as a parameter and returns a string.

```
Value = new InArgument<string>(ac=> argument1.Get(ac))
```

In summary, this code-based example is similar to assembling a workflow declaratively — both rely on deriving from the base `Activity` class and setting the `Implementation` property. Another aspect is common to both; neither approach is about implementing code that executes as part of the custom activity. Rather, the custom activity logic is assembled using pre-existing activities.

Although the code-based approach based on the `Activity` class is more concise, it does come at a cost: additional complexity. Specifically, the manner in which activities must be stacked and then assigned to the `Implementation` property of the `Activity` class is not intuitive. Fortunately, WF 4 provides additional base classes that can simplify the development of custom activities.

CodeActivity

The `CodeActivity` base class is perhaps the simplest one to derive from. Custom activity classes that derive from `CodeActivity` are responsible for overriding the `Execute` method. In addition, they are responsible for implementing any arguments and variables as needed. Listing 12-16 shows an example of a custom activity that derives from the `CodeActivity` base class.

LISTING 12-16: Deriving from CodeActivity Class

```
public sealed class HelloWorld_CodeActivity : CodeActivity
{
    public InArgument<String> argument1 { get; set; }
    public OutArgument<String> result { get; set; }



    protected override void Execute(CodeActivityContext context)
    {
        private String _var1 = argument1.Get(context);
        Console.WriteLine(_var1);
        result.Set(context, _var1);
    }
}
```

The `Execute` method is overridden to supply an implementation for the custom activity. In this example, you store the incoming argument into a local variable called `_var1`. Notice that the incoming argument is of the `InArgument<String>` type. To access the value of the incoming parameter, you must invoke the `Get` method of the `InArgument`, and pass in the activity context. Setting a return value is similar, except you invoke the `Set` method, passing in the activity context and the value to be set.

You can now use the custom activity in the declarative workflow, as shown in Listing 12-17. The only difference from previous listings is that a namespace prefix called `"custom"` is added to include the namespace in which the custom activity is defined. This enables you to add the `HelloWorld_CodeActivity` to the workflow. The rest of the code is the same. The workflow has one incoming argument of type `string`, which, in turn, is passed to the `HelloWorld_CodeActivity`.

LISTING 12-17: Using Custom Activity within a Declaritive Worflow

```
<Activity  ..
           xmlns:custom="clr-namespace:ProgrammingWF4"
  <x:Members>
    <x:Property Name="argument1" Type="InArgument(x:String)" />
  </x:Members>
  <custom:HelloWorld_CodeActivity argument1="[argument1]" />
</Activity>
```

In summary, deriving from `CodeActivity` is straightforward. It requires providing custom execution logic via the `Execute` method of the base class. This approach is suitable for custom activities designed to perform short-lived, atomic operations.

AsyncCodeActivity

If a custom activity is designed for a long-running operation, it is advisable that it performs the bulk of its work on a separate thread. This ensures that workflow is not blocked while the long-running operation is completed. Recall the discussion of parallel activities earlier in this chapter. By moving the work to another thread, a custom activity yields control back to the WF 4 runtime. Based on this, the WF 4 runtime is now free to execute another activity that may have been scheduled to run in parallel.

To build long-running custom activities that perform asynchronously, WF 4 provides the `AsyncCodeActivity` base class. Now look at an example.

This time around, you do things a bit differently. In the previous section, the custom activity inherited from a `CodeActivity` class that was not generic. WF 4 provides generic versions of the activity base classes that are designed to further simplify the development of custom activities.

For example, by using the generic version, return types are predefined, based on a convention. For this example, build a custom activity that derives from the generic `AsyncCodeActivity<T>` base class. The code is shown in Listing 12-18.

LISTING 12-18: Deriving from AsyncCodeActivity Class

```
public sealed class HelloWorld_AsyncCodeActivity : AsyncCodeActivity<int>
{
        public InArgument<String> argument1 { get; set; }


        protected override IAsyncResult BeginExecute(
AsyncCodeActivityContext context,
AsyncCallback callback,
object state)
        {
            private String _var1 = argument1.Get(context);

            Func<string, int> LongRunningFunc = new Func<string, int>(DoWork);
            context.UserState = LongRunningFunc;
            return LongRunningFunc.BeginInvoke(_var1, callback, state);
        }


        protected override int EndExecute(
AsyncCodeActivityContext context,
IAsyncResult result)
        {
            Func<string, int> LongRunningFunc = (Func<string,
                 int>)context.UserState;
            return (int)LongRunningFunc.EndInvoke(result);
        }

        int DoWork(string argument)
        {
            int WorkDuration = new Random().Next(10) * 1000;
            Thread.Sleep(WorkDuration);
            Console.WriteLine(argument);
```

```
                return WorkDuration;
            }

        }
```

In an attempt to keep things simple, Thread.Sleep is combined with a Random class to simulate a long-running behavior. As you would guess, this is the portion of the code that will be executed on another thread. The custom activity, HelloWorld_AsyncCodeActivity, derives from AsyncCodeActivity<int>.

Defining arguments and variables is something you have seen already. In this example, you define an incoming argument called argument1 and a local variable called _var1 that is visible only inside the scope of the custom activity. Note, however, that you have not defined any return parameters. As mentioned, the benefit to using a generic base class is that the return value of type int activity is already defined for you. Furthermore, it is always consistently named Result.

Deriving from the AsyncCodeActivity requires you to override two methods from the base class: BeginExecute and EndExecute. Within the BeginExecute method, you create a delegate that points to the method that will be executed on another thread. You assign the delegate to the UserState property of the AsyncActivityContext. Storing the delegate reference within the activity context is what provides the continuity because the context is available throughout the execution of the custom activity.

The last step is to invoke the delegate and return control to the WF 4 runtime. At this point, DoWork is invoked on another thread. The interesting part about DoWork is that it does not have access to AsyncActivityContext because it is being executed on another thread. This is why you must pass the variable _var1 as part of the DoWork invocation.

Also noteworthy is the direct access to the argument passed into the DoWork method. You treated it as CLR type. Unlike the incoming argument, argument1, you did not need to access the argument via the context.

The rest of the code within DoWork is quite mundane — sleep for a period of time based on a random number, print the passed-in argument to console, and return the sleep interval back to the caller.

When the DoWork method is complete, the WF 4 runtime invokes the callback function, defined as part of the BeginInvoke call. This callback location was passed into the BeginExecute method by the WF 4 runtime. This chain of events prompts the WF runtime to invoke the EndExecute method.

This is an opportunity for the custom activity to retrieve the results of the DoWork method. You achieve this by retrieving a reference to the delegate you stored within the AsyncActivityContext and calling the EndInvoke method on it.

You have just completed the asynchronous custom activity. What is interesting, however, is that the declarative workflow you saw in Listing 12-11 can be used to invoke this activity. As you would imagine, to the consumer of the activity it makes no difference whether the work inside the activity is done in a synchronous or asynchronous manner. However, as stated earlier, the consuming workflow would benefit from the asynchronous behavior only if the workflow scheduled more than one activity for execution — as discussed in the earlier section, "Parallel Execution."

NativeActivity

As you have seen in the previous sections, deriving from the `CodeActivity` base class is the easiest way to build a custom activity. However, the ease of development also means that the access to some of the more advanced WF 4 runtime services is not available. This is to be expected; hiding complexity is job one for a good abstraction.

So, when would you need to access some of the more advanced features of WF 4 runtime? One example would be an activity that is waiting for an external event, such as a task completion or a return receipt notification. The operative word is "external" — there are events arriving from outside the execution context of the workflow.

Observant readers are probably wondering why this requirement cannot be met by deriving from the `AsyncCodeActivity`. After all, the asynchronous method `DoWork` in Listing 12-12 can be modified to wait for an external event. As a result, the activity will yield control back to the runtime, thus facilitating the execution of another scheduled activity.

Although this line of thought is indeed correct, this approach has limitations. Remember that, by yielding control, you can make it possible for another activity within the workflow to schedule. For example, it will be possible for the WF 4 runtime to execute an already scheduled activity nested within another `Parallel` activity branch. However, as far the WF 4 runtime is concerned, the workflow instance is still active.

The reason this is important is because each workflow instance consumes WF 4 runtime resources. To scale, the WF 4 runtime should only expend resources on active workflows. As you saw a moment ago, even though the custom activity yielded control, overall, the workflow is still active — the `DoWork` method is actively waiting for an external event.

It would be better if the logic to wait for an external event were moved out of the custom activity and into the WF 4 runtime. This way, the entire workflow would yield control back to the runtime.

This would be an improvement over the behavior examined earlier in the context of an asynchronous custom activity, wherein an activity yielded control to another activity within the workflow. But because the asynchronous thread was busy working on the long-running task, the workflow as a whole could not yield control back to the WF 4 runtime.

WF 4 runtime provides a mechanism that enables activities to completely offload the task of waiting for an event. This way, the workflow can be completely passive during the wait. It is the runtime that is responsible for listening to any incoming events. As the events arrive, the runtime activates the appropriate instance of the workflow, passing in the payload associated with the event.

This mechanism is called a `Bookmark` in WF 4. To take advantage of this mechanism, custom activities must derive from the base class `NativeActivity`. As the name suggests, this base class enables custom activity developers to get closer to the native execution environment of the WF 4 runtime.

Listing 12-19 shows an example of a custom class that derives from `NativeActivity`.

LISTING 12-19: Deriving from NativeCodeActivity Class

```
    class HelloWorld_NativeCodeActivity<TResult> : NativeActivity<TResult>
    {
         public HelloWorld_NativeCodeActivity()
             : base()
         {
         }

         protected override void Execute(NativeActivityContext context)
         {
             context.CreateBookmark(Guid.NewGuid().ToString(),
 new BookmarkCallback(this.Continue));
         }

         void Continue(NativeActivityContext context, Bookmark bookmark, object obj)
         {
             Console.WriteLine((string)obj);
             this.Result.Set(context, (TResult)obj);
         }

         // Must return true for a NativeActivity that creates a bookmark
         protected override bool CanInduceIdle
         {
             get { return true; }
         }

     }
```

In previous examples, you passed in a parameter to the workflow at the time of invocation. In this listing, you use the `Bookmark` mechanism to pass data into a running instance of the workflow. You override the `Execute` method to create a `Bookmark`. The name of the `Bookmark` is set to a new instance of `Guid` class. You also pass in the callback delegate `Continue` that is invoked when the `Bookmark` is resumed.

At this point, the `Execute` activity blocks for the `Bookmark` to be resumed. By default, the activity blocks after creating a `Bookmark`. However, you can specify an alternative behavior.

For example, a custom activity may explicitly request a nonblocking behavior if it intends to create more than one `Bookmark` that is active concurrently. It should be noted that the `CreateBookMark` method is available to the custom activity because it derives from the `NativeActivity` base class. The `Bookmark` function is not available to the custom activities that derive from base classes you saw in previous sections, including `CodeActivity` and `AsyncCodeActivity`.

As discussed, it is useful to think of the custom activity's current blocking state as the one in which it is waiting for an external trigger. This is also the state in which the WF 4 runtime can idle the entire workflow instance. Notice that you have overridden the `CanInduceIdle` method to return `true`. When the external trigger does indeed arrive, the `Continue` method will be invoked.

Listing 12-20 shows the code that resumes the `Bookmark`. This code assumes that the external trigger comes in the form of some text entered by the user. In addition to the text, the user also needs the name of the `Bookmark` to be resumed. This is needed because, as discussed, there can be more than one active `Bookmark` created by a workflow.

LISTING 12-20: Resuming Bookmark

```
                Console.WriteLine("Enter the name of the bookmark to resume");
                string bookmarkName = Console.ReadLine();

                if ((!String.IsNullOrEmpty(bookmarkName)) )
                {
                    Console.WriteLine("Enter the greeting '{0}'", bookmarkName);
                    string bookmarkPayload = Console.ReadLine();

                    BookmarkResumptionResult result !=
application.ResumeBookmark(bookmarkName, bookmarkPayload);
                    if (result != BookmarkResumptionResult.Success)
                    {
                        Console.WriteLine("BookmarkResumptionResult: " + result);
                    }
                }
```

Understanding When to Use Custom Activities

In summary, the `Activity` base class makes it is possible to develop custom activities declaratively using existing activities. This approach enhances reuse and, therefore, should be the preferred option. However, there are instances in which the activity must perform specialized work that requires the development of a code-based activity.

Based on the style of work, custom activities can derive from one of the three base classes:

- For work that is short-lived and can be completed without the need to yield control back to the runtime, deriving from the `CodeActivity` is recommended.
- For work that can benefit from being done on a separate thread asynchronously (such as invoking a web service), deriving from `AsyncCodeActivity` is recommended.
- For long-running work (such as waiting for an external event that can potentially take a significantly long time to complete), deriving from `NativeActivty` is recommended. In addition, deriving from `NativeActivity` is required for an advanced style of work, including creating nested activities, canceling or aborting child activity execution, and accessing custom tracking features.

Composite Activity

Although you have seen examples of prebuilt composite activities, (such as the `Sequence` activity that can contain other activities), you have not built a custom one yet. So, this section focuses on building custom composite activities.

Composite activities represent a parent-child relationship wherein the composite activities act as parents that can, in turn, invoke one or more child activities. The manner in which the child activities are invoked (serially or in parallel) and their order of execution (prioritized or by the order in which the child activities are stacked) is completely up to the author of the composite activity.

It is also common for a composite activity to serve as a control activity. Control activities can help control the order in which the activities are executed. Examples of control activities include `If`, `While`, and `ForEach`. Each of these activities, based on a condition, determine which child activities will be executed.

Because composite and control activities are closely related, now look at an example that combines the two. Build a composite activity called `HelloWorld_CompositeActivity`. This activity can contain zero or more instances of child activities. Additionally, to add a control flow aspect to the composite activity, support both sequential and parallel execution of child activities. Consumers of the `HelloWorld_CompositeActivity` can specify the execution mode by setting the property `IsExecutionSequential`.

The key aspect to develop a composite activity revolves around managing the life cycle of child activities. For all custom activities you have developed so far, you did not concern yourself with the life cycle of the custom activity — that is, the sequence in which the activity is created, scheduled for execution, and ultimately terminated. For the most part, you overrode the `Execute` method and relied on the fact that the WF 4 runtime would invoke it appropriately.

However, when building a composite (parent) activity, the activity author is responsible for managing the life cycle of the child activities. This includes scheduling the execution of child activities, reacting to their completion, and forwarding any cancellation or abort requests it receives on the child activities.

Life Cycle of an Activity

Figure 12-7 shows the life cycle of a WF 4 activity. An activity instance starts out in the `Executing` state. It remains in this state until all its pending work is complete including when it is persisted or unloaded. Upon successful completion, the activity instance transitions to a `Closed` state. If an exception is encountered during the execution, the runtime transitions the activity instance into a `Faulted` state. Likewise, if a cancellation is requested by the host, the runtime transitions the activity to the `Cancelled` state. All three states (`Closed`, `Cancelled`, and `Faulted`) are completion states. In other words, when an activity reaches a completion state, it cannot transition out of it.

Although all custom activities follow the aforementioned activity life cycle, there are differences in the level of control they can exert, based on the base activity class they derive from. For example, custom activities that derive from the `CodeActivity` base class cannot specify a cancellation or an abort handler — these handlers provide an opportunity for the custom activity to clean up before the activity is transitioned into the `Cancelled` or `Faulted` state.

Deriving from `NativeActivity` offers the most control by allowing the derived classes to supply a cancellation and abort handler. In addition, `NativeActivityContext` (an activity context available to activities that derive from `NativeActivity`) enables an activity to detect if a cancellation has been requested. Learning that a cancellation has been requested allows an activity in the `Executing` state to start a graceful shutdown of its ongoing work.

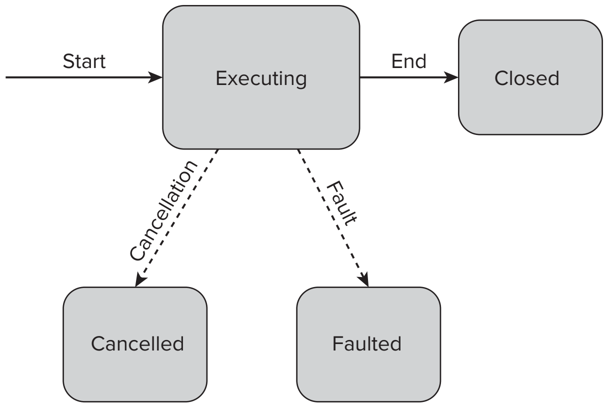


FIGURE 12-7: Life cycle of an activity

Access to the `NativeActivityContext` class also provides the capability to control the life cycle of a child activity. This includes a method for scheduling a child, canceling, and aborting a child activity.

Listing 12-21 shows the key aspects of a composite activity.

LISTING 12-21: Composite Activity

```
protected override void Execute(NativeActivityContext context)
{

            if (IsExecutionSequential)
            {
                // Schedule the first activity.
                if (this.Activities.Count > 0)
                    context.ScheduleActivity(this.Activities[0],
                                        this.OnChildCompleted);}
            else
            {
                foreach (Activity child in this.Activities)
                {
                    context.ScheduleActivity(child);}
            }
        }

      void OnChildCompleted(NativeActivityContext context,
           ActivityInstance completed)
        {// Calculate the index of the next activity to scheduled.
            int currentExecutingActivity = this.current.Get(context);
            int next = currentExecutingActivity + 1;
            // If index within boundaries....
            if (next < this.Activities.Count)
            {
                // Schedule the next activity.
                context.ScheduleActivity(this.Activities[next],
                                             this.OnChildCompleted);
                // Store the index in the collection of the activity executing.
                this.current.Set(context, next);
            }
        }
```

The `Execute` method is overridden as in the earlier custom activity examples. Using the local property `Activities`, you access the collection of child activities. Using the value of the `IsExecutionSequential` property, you know whether the consumer wants the child activities to be executed in sequence, or in parallel.

If a sequential execution mode is requested, `ScheduleActivity` is used to schedule the first child. As you would imagine, the parent waits for the currently executing child to complete before scheduling the next child.

The code to achieve this behavior is shown in the callback handler called `OnChildCompleted`. As the name suggests, this callback handler is invoked when the child activity completes. Inside the handler, you schedule the next child activity as needed. A noteworthy aspect of the `OnChildCompleted` method is the workflow variable called `current` that is defined inside the `HelloWorld_CompositeActivity`.

```
Variable<int> current = new Variable<int>() { Default = 0 };
```

By virtue that `NativeActivityContext` is available inside the callback handler, you can get and set the value of this variable, like so:

```
this.current.Set(context, next);
```

Alternatively, if a parallel execution mode is requested, all the child activities in the collection are scheduled immediately.

Listing 12-22 shows the code that creates an instance of a `HelloWorld_CompositeActivity` activity that contains two nested instances of `HelloWorld_NativeActivity` activities.

LISTING 12-22: Testing CompositeActivity

```
            return new HelloWorld_CompositeActivity
            {
                IsExecutionSequential = true,
                Activities =
                {
                    new HelloWorld_NativeActivity<string>() { },
                    new HelloWorld_NativeActivity<string>() { },

                }


            };
```

As you have seen, `HelloWorld_NativeActivity` uses a `Bookmark`-based approach to solicit the input from the `console`. In this example, you have set the `IsExecutionSequential` to `true` so that the child activities will be executed in sequence. Consequently, even if the resumption of `Bookmark` associated with the first child took a long time, the second child cannot be scheduled. Alternatively, if you set `IsExecutionSequential` to `false`, it would be possible to schedule the second child without waiting for the first child to complete.

USING PERSISTENCE

The previous section emphasized the benefit of placing a workflow in an idle state when it is waiting for an external trigger. But you have yet to see how the resources associated with an idle workflow can be reclaimed. This is where the persistence capability of WF 4 runtime comes in.

In a nutshell, persistence is about capturing the state of a workflow's instance and saving it in a durable store so that it can be re-created at a later time. After the workflow is persisted, it is possible to completely remove it from the host process, thus freeing up any associated resources.

Persistence has other benefits, too. For example, a persisted workflow can be re-created on a machine different from the one on which it was created. This capability derives from the fact that the workflow instance's captured state is bereft of any machine- or process-specific details. As you can imagine, the capability to re-create an instance on another machine can help with recovery in the event of hardware failure.

Although persistence is something that is, more often than not, initiated by the WF 4 runtime imperatively (when a workflow is idled, or when it completes the execution of a `TransactionScope` activity), it is also possible for a workflow to explicitly invoke persistence. This is achieved by invoking the `Persist` activity.

On the flip side, it is also possible for a workflow program to prevent persistence by using the *no-persist block*. (A no-persist block is actually a region of WF 4 code that sits between calls to `Enter` and `Exit` methods of the `NoPersistHandle` class.)

As you would guess, the WF 4 runtime needs a durable store to capture the workflow instance. The WF 4 runtime provides the `SqlWorkflowInstanceStore` class that is responsible for storing the workflow instance data to SQL Server. `SqlWorkflowInstanceStore` is an implementation of the abstract `InstanceStore` base class that represents an instance store. Here is the code to set up an instance store:

```
SqlWorkflowInstanceStore instanceStore =
new SqlWorkflowInstanceStore(
@"Data Source=.\SQLEXPRESS;Initial   Catalog=WFInstanceStore;Integrated
     Security=True;Asynchronous Processing=True");

            application.InstanceStore = instanceStore;
```

Based on where the workflow is hosted and the type of store, you can choose from a number of implementations of the `InstanceStore` class. Let's now take a look at the instance store provided as part of Windows AppFabric.

HOSTING WORKFLOWS INSIDE WINDOWS APPFABRIC

Windows AppFabric provides services for hosting and monitoring workflow programs. These services are built as extensions to the core Windows Server capabilities provided by Internet Information Server (IIS) and WAS. This section briefly describes how WF 4 developers can use these services.

All throughout this chapter, you have seen examples of WF 4 programs being hosted within a console application (also known as *self-hosting*). However, in a production setting, hosting the WF 4 programs within Windows AppFabric is generally preferable because of the benefits it offers, including auto-start, reliability, monitoring, and scalability. However, be aware that there are scenarios in which hosting within AppFabric may not be possible. For example, a workflow that relies on `WS-Discovery` cannot be hosted within the Windows AppFabric.

Earlier in the chapter, you learned how workflow instances that are idle can be persisted to a durable store. The earlier section, "Using Persistence," discussed the steps needed to enable persistence. Although the steps involved may not seem onerous at first, they can add up to be a significant overhead for the system administrator as the number of workflows grows.

Because a persistence store is part of Windows AppFabric installation by default, setting up persistence is easier. Furthermore, AppFabric provides a management UI to administer the persistence store. For example, there may be a need to configure multiple persistence database instances to avoid bottlenecks in the larger environments.

There is another aspect related to persistence that is worth mentioning here. Generally speaking, two types of events can cause a persisted workflow instance to be reloaded: the arrival of an external trigger (such as an incoming message) or the expiration of the elapsed time of a `Delay` activity. A workflow host must be active to process these events.

Fortunately, AppFabric can help with this as well. Workflow instances hosted within the AppFabric can automatically be started when a message arrives. This is a capability provided by WAS/IIS and is well known to developers. The handling of the expired `Delay` activities requires some additional explanation.

AppFabric installs a Windows service called the Workflow Management Service (WMS) that is responsible for monitoring the persistence store. It gets notified when a workflow is ready to be reloaded. Upon receiving a notification, it calls another AppFabric-provided WCF service called Service Management Service that is then responsible for reloading the workflow instance into memory.

Another store is associated with an AppFabric installation by default. This is the monitoring store. It is used for storing monitoring events emitted by the WF 4 runtime. Similar to the persistence store, the monitoring store can consist of multiple database instances. AppFabric provides tooling to aggregate the collected data that is helpful for performance monitoring and troubleshooting the workflow programs.

Another reason to host workflows within AppFabric relates to command queuing. The command queuing feature provides the capability to queue commands such as a cancellation to a running instance of a workflow program. In the self-hosted scenario, a cancellation request can be made by invoking the `Cancel` method on the class `WorkflowApplication`. As you would imagine, this approach requires some custom plumbing to be built for exposing the command queuing function to the system administrators. AppFabric reduces this burden by providing tooling to queue commands.

Finally, AppFabric offers the capability to set up a farm that includes multiple AppFabric servers. This enables handling of increased loads because the resources from multiple servers can be pooled together. One of the benefits of workflow persistence is that it enables for capturing the state of a workflow in a location-independent manner. (No information about the server executing the workflow is persisted.) This enables another node in the farm to resume the workflow at a later time. This means that a web farm can scale linearly by adding AppFabric nodes. Figure 12-8 shows the web farm made up to AppFabric nodes.

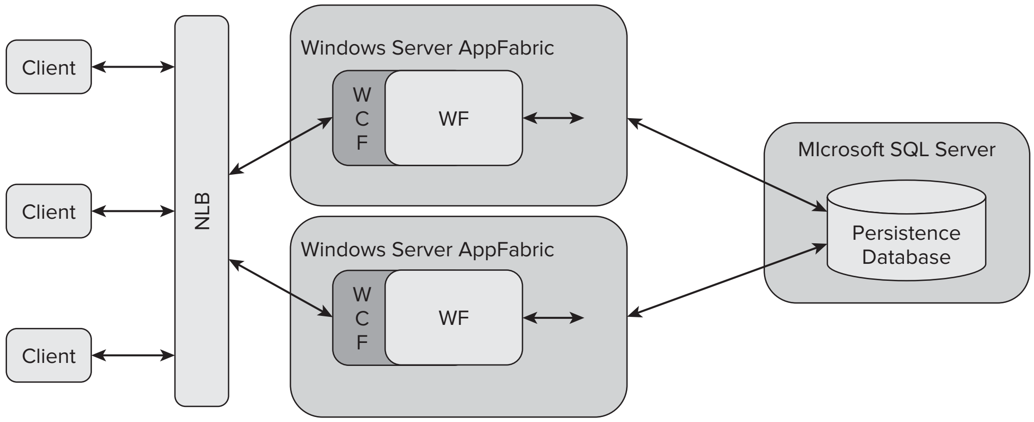


FIGURE 12-8: Scaling out AppFabric-hosted WF 4 programs

FURTHER READING

The primary target audience for this chapter is developers responsible for enabling the authoring of WF 4 programs. Typically, graphical design tools are used to author WF 4 programs. To assist developers in that goal, this chapter focused on the fundamentals of the WF 4 programming language — a behind-the-scene look at the code generated by the graphical design tools. Because of the focus on the language fundamentals, the aspects of the design experience were not explored.

For example, activity authors can build custom designers to make the authoring experience more productive. Another example of customizing the design experience is the capability to rehost the Workflow Designer tool inside a custom application.

As a next step, the following resources are recommended for developers looking to enhance the workflow design experience:

- *Custom activity designers* — You can find a collection of samples that use custom designers at `http://msdn.microsoft.com/en-us/library/dd759030.aspx`.
- *Designer rehosting* — A sample at `http://msdn.microsoft.com/en-us/library/dd699776.aspx` shows how to create the WPF layout to rehost the designer.

- "*Visual Design of Workflows with WCF and WF 4*" — In this article at `http://msdn.microsoft.com/magazine/ff646977`, Leon Welicki offers tips for authoring workflows within the Workflow Designer.

SUMMARY

In this chapter, you learned about the features of WF 4.0 from a perspective of a higher-level programming language. You got a "behind the Workflow Designer" look at how WF 4.0 program is structured and its core constructs, including the key `Activity` class. You learned about different styles of WF 4.0 programs, including flowchart and procedural styles.

For WF 4.0 programs that need alternate flows of execution, you learned about sequential and parallel flows of execution. You also built custom activities by inheriting from framework-provided base classes, including `CodeActivity`, `NativeActivity`, and `AsyncCodeActivity`. And, finally, you learned about hosting WF 4.0 programs in AppFabric.

ABOUT THE AUTHOR

Vishwas Lele is an AIS (`www.appliedis.com`) Chief Technology Officer, and is responsible for the company vision and execution of creating business solutions using .NET technologies. Lele has more than 20 years of experience and is responsible for providing thought leadership in his position. He has been at AIS for 17 years. A noted industry speaker and author, Lele is the Microsoft Regional Director for the Washington, D.C., area.

REAL WORLD .NET 4, C#, AND SILVERLIGHT®

Real World .NET 4, C#, and Silverlight® : Indispensible Experiences from 15 MVPs

Published by
John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2012 by John Wiley & Sons, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-1-118-02196-5
ISBN: 978-1-118-22189-1 (ebk)
ISBN: 978-1-118-23619-2 (ebk)
ISBN: 978-1-118-25264-2 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Control Number: 2011939645