



Your Short Cut to Knowledge

The following is an excerpt from a Short Cut published by one of the Pearson Education imprints.

Short Cuts are short, concise, PDF documents designed specifically for busy technical professionals like you.

We've provided this excerpt to help you review the product before you purchase. Please note, the hyperlinks contained within this excerpt have been deactivated.

Tap into learning—NOW!

Visit www.informit.com/shortcuts for a complete list of Short Cuts.



SAMS

Cisco Press

**IBM
Press™**

que®

Workflow Scheduling

Principles of Workflow Scheduling

At this point you should have a good idea of how to build simple workflows. We still have not yet discussed features such as error handling or compensation which are critical to building any real workflow. In order to do so, however, we need to explain how the workflow runtime schedules activities. Otherwise, you will have a mistaken notion of how the more complicated scenarios operate.

¹ Because of passivation, however, you should not assume that all activities execute on the same thread. For example, do not store anything in the current thread context.

² For those of you old enough to remember, this is how scheduling worked in early versions of Windows.

Each workflow instance runs on one thread.¹ The workflow runtime maintains a FIFO queue of activities that are ready to execute in a given workflow instance. In addition, activity scheduling is non-preemptive. Until an executing activity either yields because it is waiting for a notification of some event, or enters the closed state, no other activity in the scheduler's queue can run. For example, a `CodeActivity` that enters an infinite loop will cause the entire workflow to hang.²

If an activity yields because it is waiting for some external event to occur, the next activity in the scheduler's queue can execute. When the event that activity is waiting for arrives, the waiting activity is scheduled to run by placing it behind whatever activities are already in the scheduled to run queue. It has no priority over what is already in the scheduling queue.

Understanding this is important so that you understand how your workflow will operate. It is critical for understanding how fault handling, compensation, and other such activities are scheduled when we discuss them.

As noted earlier, the workflow itself is an activity. The first thing that the workflow scheduler does when a workflow instance is started (with the *Start* method of the **WorkflowInstance** class) is to place the parent activity of the workflow (the one passed to the *CreateWorkflow* method) on the scheduler queue to be executed.

SequenceActivity Scheduling

As you have seen in previous examples, a **SequenceActivity** contains one or more child activities that are executed in order. When the **SequenceActivity** executes, it sets up a notification request for the close event of its first child, and then schedules the first child activity. It then yields. When the first child closes, the **SequenceActivity** receives a notification. At that point, it sets up a notification request for the close of the second child, and then schedules the second child activity. It then yields. It continues with this pattern for all its children. At the close of the last child, the **SequenceActivity** itself goes into the Closed state.

ParallelActivity Example

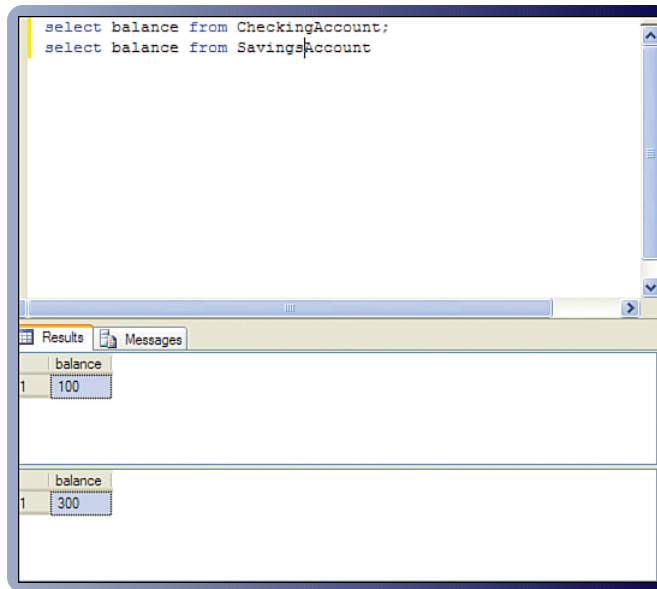
The **ParallelActivity** is more complicated. The **ParallelActivity** can have several branches, all of which must complete before the next activity is executed. This activity is illustrated with the **ParallelActivity** example (see Figure 27).

The **Start** code activity writes a message to the console when the workflow begins. The **Finish** code activity writes a message to the console at the end of the workflow. The **ParallelActivity** can have several branches, all of which must complete before the next activity is executed. Hence, the **Finish** activity will not write its message out until all branches of the **Parallel** activity have

SECTION #4

Activities That Define Workflow Scope

FIGURE 53
TransactionScope
Activity rolled back
the initial
withdrawal



Compensation

Long running workflows (as well as other service oriented systems) need transaction semantics under circumstances other than the ones databases can provide. Long running workflows often take a long time to execute because they may be waiting for a human interaction before they can commit. They would lock records for durations that would impede scalability. In addition, they often use multiple databases, and classic ACID transactions would time out under these circumstances. Workflows (and other service oriented systems) often use databases from other companies. Since this access often crosses a trust boundary, a company is usually reluctant to let another company lock records in its database even if the circumstances for classic ACID transactions would apply.

SECTION #4

Activities That Define Workflow Scope

¹⁰ Compensating transactions also are used with ACID transactions when one has to reverse a previously committed transaction because of an error in business logic. Suppose you decided that the withdrawal was a mistake. You deposit the cash back in the account. This type of compensation is analogous to the compensation of long running workflows. Of course compensating transactions are not always possible. If the action was to launch a cruise missile there may or may not be an abort sequence.

¹¹ The **CompensatableSequenceActivity** implements the *ICompensatableActivity* interface. Any activity that implements this interface can have a compensation handler.

Under these conditions, one uses another transaction to compensate, or reverse the effects of an already committed transaction. This is called a compensating transaction. Often ACID transactions are referred to just as transactions, and compensating transactions are called just compensation.¹⁰

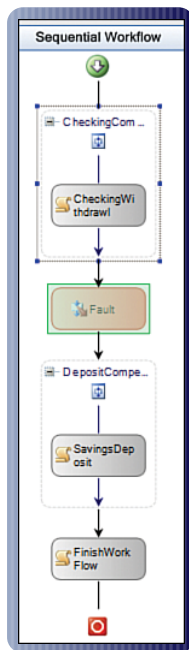


FIGURE 54
Compensation
Example
Workflow

Since this scenario is the more common one with long running workflows, WF comes with activities that support compensation.

The **CompensatableSequenceActivity** allows one to associate a compensation handler with a sequence of activities. This compensation handler can contain activities that you need to invoke in order to reverse the action taken in the **CompensatableSequenceActivity**.¹¹

The Compensation example has a simple illustration of how to use this activity. It uses the exact same example of a transfer from checking account to a savings account that was used by the Transaction example. Here the withdrawal and the deposit are in two separate **CompensatableSequenceActivities** to illustrate how this would work if these actions used two separate databases.

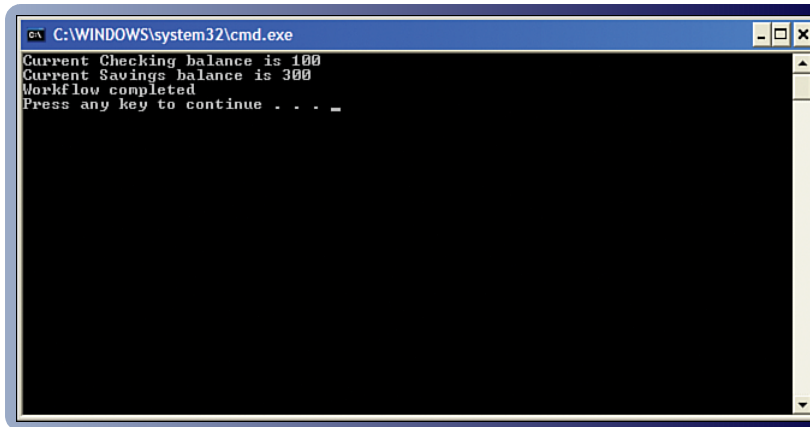
The workflow example is shown in Figure 54.

The code within the activities is identical to the code in the previous Transaction example except it is split into two compensatable sequences. As before there is a disabled **ThrowActivity** between them. Running the workflow yields the following result in Figure 55.

SECTION #4

Activities That Define Workflow Scope

FIGURE 55
Compensation
Example Output



```
C:\WINDOWS\system32\cmd.exe
Current Checking balance is 100
Current Savings balance is 300
Workflow completed
Press any key to continue . . . _
```

Since this code uses the same database tables as the previous example, the exact numbers you see will depend on how many times that the previous example is executed before this one. The output in Figure 55 has the original values in the accounts as \$200 each.

To view the compensation handler, select “View Compensation Handler” from each sequence’s context menu as shown in Figure 56.

SECTION #4

Activities That Define Workflow Scope

You can then drag and drop whatever activities you need into the compensation handler (see Figure 57). In this case we just have code activities that reverse the withdrawal and deposit transactions. Each **CodeActivity** also writes out a diagnostic message to indicate that it was invoked and writes out the restored balance.

FIGURE 56
Viewing the
Compensation
Handler

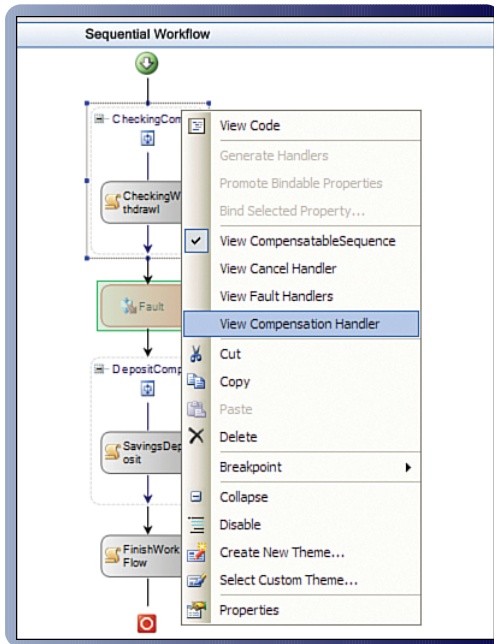
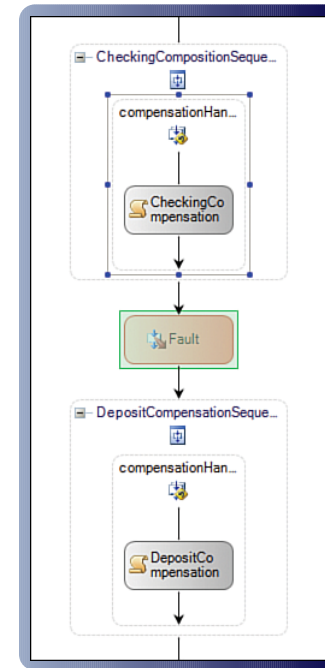


FIGURE 57
Compensation
Handlers

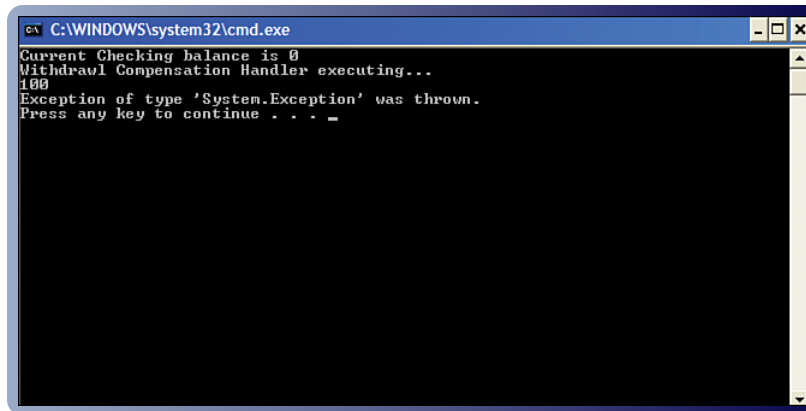


SECTION #4

Activities That Define Workflow Scope

Enable the **ThrowActivity** and then run the application:

FIGURE 58
Compensation
Example with Fault
After Checking
Withdrawal



```
C:\WINDOWS\system32\cmd.exe
Current Checking balance is 0
Withdrawal Compensation Handler executing...
100
Exception of type 'System.Exception' was thrown.
Press any key to continue . . . _
```

The withdrawal occurs and the balance is reduced from \$100 to 0. After the exception is thrown, the compensation handler is invoked and the balance is restored to \$100. Note that the **WorkflowTerminated** notification occurs because the exception is not handled by the workflow.

This compensation is handled entirely through the workflow mechanisms and unlike the **TransactionScopeActivity** the compensation handlers do not use any classes from the **System.Transaction** namespace. In addition, you do not have to have to use a persistence service because it is the responsibility of the compensation handler to restore the workflow to the appropriate state.

If you need both ACID transactions and compensation you can use the **CompensatableTransactionScopeActivity**. If you need to explicitly invoke the compensation handler of an already completed compensatable activity, you can use the **CompensateActivity**.