Chapter 4. Advanced Activity Execution

In the previous chapter, we learned about the basic lifecycle of an activity, which involves the Initialized, Executing, and Closed states of the activity automaton. We also explored the ramifications of the fact that the logical lifetime of activities differs from the lifetimes of CLR objects that transiently represent activities during those times that a WF program instance is not passivated. We developed several useful activities, including a few composite activities, and illustrated the central role of the `ActivityExecutionContext` as a facilitator of activity execution.

In this chapter, we will give more scrutiny to `ActivityExecutionContext` and introduce its vital role as a state boundary in WF programs. This will allow us to develop various iterative composite activities and other interesting control flow constructs. We will also learn about the other states of the activity automaton, and discuss activity cancellation, fault handling, and compensation.

Activity Execution Context

Until this point, we have straightforwardly used methods defined on `ActivityExecutionContext` (AEC), such as `GetService` and `ExecuteActivity`, in our activity examples. However, AEC has a larger role in the WF programming model that goes well beyond these methods. Arguably, AEC is overloaded with meanings and capabilities that could usefully be disentangled. Be that as it may, here we will catalog these meanings and capabilities, and provide a conceptual model for understanding and utilizing this key abstraction.

Let's begin with a brief summary of the functions of `ActivityExecutionContext` that we have already seen exercised.

The first role of AEC is as a container of services that is available to activities during their execution. This set of services is the same for all activities in all WF program instances (for a given application that hosts the WF runtime). Some services are provided by the WF runtime and are always obtainable from AEC; one such service we used in the `ReadLine` activity is the `WorkflowQueuingService`. Custom services can be offered by the application that hosts the WF runtime; such services are made available to activities, via AEC, by using the `AddService` method of `WorkflowRuntime`.

The fact that `ActivityExecutionContext` is a provider of services is made explicit by its implementation of the `System.IServiceProvider` interface, which defines the `GetService` method. `IServiceProvider` is a standard way to promote loose coupling between activities and the host application's implementations of services on which those activities depend.

The second role of `ActivityExecutionContext` is as an API surface through which activities can interact with the (internal) scheduler component of the WF runtime. For example, the `ExecuteActivity` method requests that a work item (corresponding to the invocation of the `Execute` method of a child activity) be added to the WF runtime's scheduler work queue. The `CloseActivity` method requests that the WF runtime finalize the current activity's transition to the Closed state, and resume the internal bookmark that notifies the parent composite activity of the activity's completion. AEC therefore abstracts the internal machinery of the WF runtime; even though we have explained the execution model of the WF runtime in terms of a scheduler and a work queue, these entities are not represented directly in the public API of the WF programming model.

To introduce the third, and subtlest, aspect of AEC, we need to return to the idea of WF program instances as continuations. The execution of a WF program instance is episodic, and at the end of each episode when the WF
program instance becomes idle, the instance can be persisted in durable storage as a continuation. This continuation, because it represents the entirety of the program instance's state that is necessary for resuming its execution, holds the relevant (internal) WF runtime execution state plus user-defined state, sometimes called the application state. The application state is nothing but the WF program instance's tree of activities (the actual CLR objects), which are usually stateful entities. The runtime state includes the state of the scheduler work queue, WF program queues, and bookkeeping information about internally managed bookmarks (such as subscriptions to the Activity.Closed event).

The resumption point of a bookmark is called an execution handler, so we can refer to the (heap-allocated) execution state required by an execution handler as its execution context. Because an execution handler is typically a method on an activity, we will often refer to this execution context as activity execution context.

ActivityExecutionContext is a programmatic abstraction for precisely this execution context. ActivityExecutionContext is passed to every execution handler either as an explicit argument (as for Activity.Execute) or as the sender parameter in the case of execution handlers that conform to a standard .NET Framework event handler delegate type.

In the examples we've developed thus far in the previous chapters, this "execution context" aspect of AEC has not been apparent. When the host application calls the WorkflowRuntime.CreateWorkflow method, the WF runtime creates a new activity execution context that represents the execution context for the newly created WF program instance. This execution context is managed by the WF runtime because its lifecycle corresponds precisely to the lifecycle of the WF program instance. We call this execution context the default execution context.

The relationship between a WF program instance and its execution context is depicted in Figure 4.1.

![Figure 4.1. WF program instance and the default execution context](View full size image)

The WF programming model might have stopped there, with a one-to-one mapping between every WF program instance and a corresponding execution context that is created and managed by the WF runtime. However, this
leaves a gap in the execution model for activities.

Instead, the WF programming model allows a composite activity to explicitly create subordinate execution contexts during the execution of a WF program instance. A subordinate execution context, like the default execution context, consists of application state for a set of activities, along with the necessary runtime state. The application state of a subordinate execution context consists of (a copy of) a subtree of activities a program fragment the root of which is a child activity of the composite activity that created the subordinate AEC.

The API for creating and managing execution contexts is found in the Activity-ExecutionContextManager type, which we will refer to as the execution context manager, or AECManager. The capabilities of AECManager are the basis for a wide range of activity composition patterns, ranging from familiar iterative control flow to more exotic patterns including coroutine style-interleaved iterations, graphs of activities, fork and join patterns, state machines, and activity compensation.

Activity Execution Context Manager

The default execution context for a WF program instance is created and managed internally by the WF runtime. AECManager, which is shown in Listing 4.1, can be utilized by composite activities to explicitly create and manage subordinate execution contexts.

Listing 4.1. ActivityExecutionContextManager

```csharp
namespace System.Workflow.ComponentModel
{
    public sealed class ActivityExecutionContextManager
    {
        public ActivityExecutionContext CreateExecutionContext(Activity activity);
        public ReadOnlyCollection<ActivityExecutionContext> ExecutionContexts { get; }
        public ActivityExecutionContext GetExecutionContext(Activity activity);
        public void CompleteExecutionContext(ActivityExecutionContext childContext);
        public void CompleteExecutionContext(ActivityExecutionContext childContext, bool forcePersist);
        public IEnumerable<Guid> PersistedExecutionContexts { get; }
        public ActivityExecutionContext GetPersistedExecutionContext(Guid contextGuid);
    }
}
```

AECManager is available to activity execution logic via the Execution-ContextManager property of ActivityExecutionContext, as shown in the following code fragment:

```csharp
protected override ActivityExecutionStatus Execute(ActivityExecutionContext context)
{
    ActivityExecutionContextManager manager = context.ExecutionContextManager;
    ...
```
The role of AECManager is to allow a composite activity to create and manage subordinate execution contexts, which act as boundaries for the execution of subtrees of activities within a WF program. Activity subtrees are essentially program fragments whose roots are child activities of a composite activity within a WF program instance.

A new execution context is created by calling the CreateExecutionContext method of AECManager, passing as a parameter the child activity that is to be the root of the activity subtree that will be represented by the new AEC. This activity is called the template activity of the create operation. The state of the new AEC is essentially a deep copy of the application state of the activity subtree, of which the template activity is the root.

When a new execution context is created, it becomes a member of a collection of active execution contexts that are managed by the current composite activity. This collection is represented by the ExecutionContexts property of AECManager.

The execution state of the activity subtree within the new execution context (which is represented most visibly by the ExecutionStatus and ExecutionResult properties of all the activity objects, but also includes data managed internally by the WF runtime) is guaranteed to be pristine when the CreateExecutionContext method returns. To be specific, the activity objects that form the new instance of the subtree (within the new execution context) each have their Initialize method called so that they properly enter the activity automaton. This is exactly like the initialization of the WF program instance (the WF program's activity tree) that occurs within the default execution context when WorkflowRuntime.CreateWorkflow is called.

In this way, a running WF program instance actually can be viewed, from one vantage point, as a hierarchy (tree) of execution contexts, a simple example of which is shown in Figure 4.2.

Exhibit 4.2. Hierarchy of execution contexts (the default AEC has an id of 1)
in the execution context tree occurs precisely where a composite activity uses the AECManager to create a new execution context.

Rather than drilling deeper into the mechanics of execution context management, we will use some examples of composite activities that leverage AECManager to help bring this aspect of the WF programming model to life.

**Iterative Control Flow**

None of the composite activities we have encountered so far has had to exhibit iterative behavior. For example, Sequence and Interleave both execute each of their child activities once. However, many control flow constructs such as the familiar while and foreach statements in C# require repeated execution of a nested (child) statement.

The activity automaton appears to, and in fact does, preclude such repetition because there are no loops present in the state transition diagram. In particular, there is no path that takes an activity from the Executing state to another state and then back again to the Executing state. There are good reasons why the activity automaton has this characteristic, and they should become apparent in the examples we develop.

AECManager provides a composite activity with a way of repeating the execution of a child activity (actually, the subtree of activities rooted at a child activity, which is exactly what is required for iteration). We will use the mechanism to implement control flow constructs such as while.

Before going further, let's clarify terminology. By the term activity iteration (or, simply iteration in the context of WF), we are referring to any circumstance in which an activity within a WF program instance needs to execute more than one time. As a simple example, we need to be able to represent in a WF program, and then of course execute, iterative logic such as that in the following C# code snippet:

```csharp
while (expression)
{
    string s = Console.ReadLine();
    Console.WriteLine(s);
}
```

Provided that we implement a While composite activity with the proper semantics, the representation of this Echo program in XAML is straightforward, as shown in **Listing 4.2**.

**Listing 4.2. WF Program that Uses the While Activity**

```xml
<While x:Name="while1" Condition="..." xmlns="http://EssentialWF/Activities"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:wf="http://schemas.microsoft.com/winfx/2006/xaml/workflow">
    <Sequence x:Name="s1">
        <ReadLine x:Name="r1" />
        <WriteLine x:Name="w1" Text="{wf:ActivityBind r1,Path=Text}" />
    </Sequence>
</While>
```

In **Chapter 7, "Advanced Authoring,"** we will show how the While activity can be associated with validation logic that limits its number of child activities to one. Here we will discuss how the While activity manages the repeated execution of its child activity.

The usual semantics of iteration requires a scoping boundary for data (which pulls along other requirements, such as variable binding rules, which we are ignoring here). This simply means that, within the scope of the
construction governing the iteration, a given iteration does not see residual effects of the execution of the prior iterations. Iterations can certainly influence one another via shared data, but that data must be defined outside the scope of the iterative construction. Consider the following C# code:

```csharp
while (expression)
{
    string s = "";
    Console.WriteLine(s);
    s = Console.ReadLine();
}
```

In order to compile this program with the C# compiler, the variable `s` must be assigned to prior to the `WriteLine` method call. More to the point, the value assigned to `s` by the `ReadLine` method call is lost when the first iteration finishes. The `WriteLine` will forever write the empty string, no matter what is entered at the console and read by the `ReadLine`.

The situation changes if the declaration of `s` occurs outside the `while`:

```csharp
string s = "";
while (expression)
{
    Console.WriteLine(s);
    s = Console.ReadLine();
}
```

When `s` becomes shared data, any iterations of the `while` loop beyond the first one see the value that is assigned to `s` during the previous iteration.

In the case of a WF program, the activities (or, rather, the fields of activity objects) are the program's data. So, the WF program of Listing 4.2 must exhibit behavior similar to the C# program in which `s` is local to the scope within the `while`. The `WriteLine` activity's `Text` property takes its value from the `Value` property of the `ReadLine` activity in the current iteration.

If we rewrite the WF program by swapping the order of the `WriteLine` and `ReadLine` activities, the `WriteLine` will forever print the empty string. If this were not the case, there would be unexpected scoping of data and as a result the local maneuverings of one iteration of the `While` activity would influence (interfere with) the execution of subsequent iterations.

Thus, in order for iterative composite activities to work properly, there must be a way to provide a clean slate of local application data (and execution state) to each iteration. Practically speaking, for a WF program this conceptually amounts to either using (a) a freshly initialized activity subtree for each iteration, or (b) facilitating a reset of the activity subtree used in one iteration for use in the next iteration.

The "reset" approach is problematic. For one thing, if the WF scheduler were used to achieve the reset behavior, the resetting of an activity subtree would unfold asynchronously and necessitate a new "Resetting" state that doesn't correspond to any aspect of a real-world process. For another reason, it would complicate activity development, as activity writers would need to implement potentially difficult reset logic. Properties of an activity can be bound to fields and properties of other activities, so that an activity's reset logic might affect another executing activity adversely. Finally, the reset strategy would preclude (or at least make much more convoluted) certain use cases, including but not limited to interleaved execution of multiple instances of the same child activity, and activity compensation, both of which we will discuss later in this chapter.

The WF programming model therefore equates the normal semantics of a looping construct like `while` as one use case for subordinate execution contexts. Each iteration of an activity like `While` is housed within a distinct execution context that holds the state defined by the activity subtree over which the iteration is occurring. This
scoping of state ensures the locality of the data, determines its lifetime, and establishes a framework for binding to data in other scopes. We will explore these aspects more fully in the examples to come.

It's time to write the While activity. We will assume that While has an associated validator component that assures only a single child activity is present within the While (this will be developed in Chapter 7).

The While activity is shown in Listing 4.3.

Listing 4.3. While Activity

```csharp
using System;
using System.Workflow.ComponentModel;

namespace EssentialWF.Activities
{
    public class While : CompositeActivity
    {
        public static readonly DependencyProperty ConditionProperty = DependencyProperty.Register(
            "Condition",
            typeof(ActivityCondition),
            typeof(While),
            new PropertyMetadata(DependencyPropertyOptions.Metadata)
        );

        public ActivityCondition Condition
        {
            get
            {
                return GetValue(ConditionProperty) as ActivityCondition;
            }
            set
            {
                SetValue(ConditionProperty, value);
            }
        }

        protected override ActivityExecutionStatus Execute(ActivityExecutionContext context)
        {
            if (Condition != null && Condition.Evaluate(this, context))
            {
                ExecuteBody(context);
                return ActivityExecutionStatus.Executing;
            }

            return ActivityExecutionStatus.Closed;
        }

        void ExecuteBody(ActivityExecutionContext context)
        {
            ActivityExecutionContextManager manager = context.ExecutionContextManager;

            ActivityExecutionContext newContext =
                manager.CreateExecutionContext(EnabledActivities[0]);

            Activity newActivity = newContext.Activity;

            newActivity.Closed += this.ContinueAt;
            newContext.ExecuteActivity(newActivity);
        }
    }
}
```
The While activity has a Condition property of type ActivityCondition, and continues to execute iterations of its child activity until the condition evaluates to false. Evaluation of the condition occurs when the While begins its execution and, subsequently, upon the completion of each iteration.

The ActivityCondition type, which is in the System.Workflow.ComponentModel namespace and is shown in Listing 4.4, represents a Boolean expression that can be evaluated by invoking ActivityCondition.Evaluate.

Listing 4.4. ActivityCondition

```csharp
namespace System.Workflow.ComponentModel
{
    public abstract class ActivityCondition : DependencyObject
    {
        protected ActivityCondition();

        public abstract bool Evaluate(Activity activity, IServiceProvider provider);
    }
}
```

ActivityCondition is useful in other ways during the development of composite activities. For example, branching logic of the if or switch variety can be easily implemented with a composite activity that uses an attached property of type ActivityCondition on its ordered list of child activities. ActivityCondition is an
abstract class. In Chapter 8, "Miscellanea," we will discuss the two generic derivatives of ActivityCondition provided by WF. You are also free to write your own custom derivative. If your activities requiring customizable conditional logic reference only the ActivityCondition base class, they are shielded from details of the various condition classes. This lets the user of the activity (the developer of the WF program) decide what is most appropriate for his solution. We can write a simplistic constant condition type that we can use throughout this chapter. This is shown in Listing 4.5.

Listing 4.5. ConstantLoopCondition

```csharp
using System;
using System.Workflow.ComponentModel;

namespace EssentialWF.Activities
{
    public class ConstantLoopCondition : ActivityCondition
    {
        int counter = 0;

        public static readonly DependencyProperty
            MaxCountProperty = DependencyProperty.Register(
                "MaxCount",
                typeof(int),
                typeof(ConstantLoopCondition),
                new PropertyMetadata(DependencyPropertyOptions.Metadata)
            );

        public int MaxCount
        {
            get { return (int) GetValue(MaxCountProperty); } 
            set { SetValue(MaxCountProperty, value); } 
        }

        public override bool Evaluate(Activity activity,
            IServiceProvider provider)
        {
            return (counter++ < this.MaxCount);
        }
    }
}
```

Now we can rewrite Listing 4.2 by using ConstantLoopCondition. Listing 4.6 is an executable WF program (unlike Listing 4.2) and will loop through the Sequence three times.

Listing 4.6. An Executable Version of Listing 4.2

```xml
<While x:Name="while1" xmlns="http://EssentialWF/Activities"
    xmlns:wf="http://schemas.microsoft.com/winfx/2006/xaml/workflow"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <While.Condition>
        <ConstantLoopCondition MaxCount="3"/>
    </While.Condition>
    <Sequence x:Name="s1">
        <ReadLine x:Name="r1" />
        <WriteLine x:Name="w1" Text="{wf:ActivityBind r1,Path=Text}" />
    </Sequence>
</While>
```
Figure 4.3 shows the three execution contexts that are created, one for each of the iterations of the While activity.

Returning to Listing 4.3, we see that in order to provide local state for each iteration, the While activity must use the AECManager to create a separate execution context for each iteration. The lone child activity of While is the template activity used in the creation of a new execution context. The newly created (subordinate) execution context contains a new instance of the template activity's subtree, and it is the root activity of this fresh subtree that is scheduled for execution. You can see that exactly the same pattern of subscribing to the Closed event of the scheduled activity that we used in Sequence and Interleave also is used here.

The subordinate AEC has an Activity property, which holds the new instance of the template activity. This new instance can be scheduled for execution, using the familiar ExecuteActivity method of AEC, so long as the AEC used to make the request is the execution context that corresponds to the newly created instance of the activity. The basic pattern is illustrated here:

```csharp
ActivityExecutionContextManager manager = context ExecutionContextManager;
Activity template = ...
ActivityExecutionContext newContext = manager.CreateExecutionContext(template);
Activity newInstance = newContext.Activity;
newInstance.Closed += this.ContinueAt;
newContext.ExecuteActivity(newInstance);
```

The important line of code is the last one; the newly created activity instance is scheduled for execution within the subordinate AEC.

The IsDynamicActivity property of Activity can be used to determine if a specific activity object is part of the default AEC, or instead has been manufactured as part of the creation of a dynamic execution context. In the
preceding code, the newInstance object is a reference to an activity within a subordinate AEC. We can verify this fact by looking at the value of the IsDynamicActivity property:

```csharp
System.Diagnostics.Debug.Assert(newActivity.IsDynamicActivity);
```

Because the While activity executes a new instance of the template activity for each iteration, the template activity itself (which is part of the default AEC) will always remain in the Initialized state. Only the instances of the template activity, which execute within subordinate execution contexts, move from the Initialized state to the Executing state and then to the Closed state.

This bookkeeping is not terribly difficult for the While activity to manage, but it does play a bit of havoc with the rest of the WF program's view of the While. There is no generalized way for code outside of While to locate the subordinate execution contexts that are dynamically created by While (or the activity objects within these execution contexts).

For this reason, it is a recommended practice for composite activities that dynamically create execution contexts to also provide helper properties or methods (depending upon what makes the most sense for a given composite activity) that assist external code in accessing appropriate iterations of a child activity. In the case of While, there is always at most one active iteration, so we can provide a property (as shown here) that returns the currently executing instance of the template activity:

```csharp
public Activity CurrentIteration
{
    get
    {
        Activity template = EnabledActivities[0];
        Activity[] list = GetDynamicActivities(template);

        if (list.Length == 0) return null;
        return list[0];
    }
}
```

The GetDynamicActivities method is a protected method defined on the CompositeActivity class. It can be used to obtain the set of active instances of a child activity for which execution contexts have been dynamically created. In the case of While, there will always be at most one, but as we will see in the next section, this might not always be the case. GetDynamicActivities also by its nature implies that separate execution contexts could be created for different child activities (for those composite activities that allow multiple child activities).

Returning again to Listing 4.3, in its ExecuteBody method, the While activity subscribes for the Closed event of the newly created activity instance and schedules this activity for execution in its newly created AEC. As we have seen, the template activity never gets scheduled for execution and always remains in the Initialized state; it acts as a true declaration from which instances are manufactured at runtime. When a dynamic activity instance transitions to the Closed state, the ContinueAt method (of While) will be scheduled. ContinueAt retrieves the subordinate AEC using the GetExecutionContext method of AECManager, and then removes it from the list of active execution contexts by calling CompleteExecutionContext. In other words, when the execution of the activity subtree within a subordinate AEC is completed, the iteration is complete and that AEC can be discarded.

One additional aspect of subordinate execution contexts is related to activity databinding (the mechanics of which are discussed in Chapter 7). ActivityBind objects, though they reference a target activity by name, always implicitly reference an activity within a visible AEC. In other words, even though the instances of activities within the subtrees of different iterations carry the same values for their Name property, there is never any ambiguity about which activity instance an ActivityBind object will use. This is depicted in Figure 4.4.

![Figure 4.4. Activity execution context and ActivityBind](file://C:\Users\gibba\AppData\Local\Temp\~hhE3BC.htm)
The current AEC and parent execution contexts (in the sense of the AEC hierarchy described earlier) are visible (as shown in Figure 4.5), but peer and subordinate AECs are not visible because this would introduce naming ambiguity. This is demonstrated in our example WF program (refer to Listing 4.6) that uses While, in which a property of the WriteLine activity is bound to a property of a peer ReadLine activity. In that example, resolution of the activity databinding expression always uses the ReadLine instance that is present within the same subordinate AEC as the WriteLine.

Figure 4.5. ActivityBind across an activity execution context hierarchy
Interleaved Iteration

The `While` activity demonstrates how the WF programming model enables scoped (iterative) execution of an activity subtree. Every iteration of `While` creates a new AEC, in which a new instance of the (template) child activity is executed.

There is nothing stopping us, though, from writing a composite activity that schedules more than one iteration of a child activity for execution at the same time. This does not make sense for `While`, but it is a quite natural and useful pattern when applied to the C# `foreach` construction. Instead of processing the items in a collection one after the other (for which use case we could write an ordinary `ForEach` activity that does what the C# `foreach` does), our `InterleavedForEach` activity will schedule the processing of all items in the collection simultaneously. Item processing is performed by a set of separately executed instances of the `InterleavedForEach` activity's single (template) child activity, with each instance comfortably ensconced within its own AEC.

Listing 4.7 shows an implementation of the `InterleavedForEach` activity.

Listing 4.7. `InterleavedForEach` Activity

```csharp
using System;
using System.Collections;
using System.Workflow.ComponentModel;

namespace EssentialWF.Activities
{
    public class InterleavedForEach : CompositeActivity
    {
        int totalCount = 0;
        int closedCount = 0;

        public static readonly DependencyProperty IterationVariableProperty =
            DependencyProperty.RegisterAttached(
                "IterationVariable",
                typeof(object),
                typeof(InterleavedForEach)
            );

        public static object GetIterationVariable(object dependencyObject)
        {
            return ((DependencyObject)dependencyObject).GetValue(IterationVariableProperty);
        }

        public static void SetIterationVariable(object dependencyObject, object value)
        {
            ((DependencyObject)dependencyObject).SetValue(IterationVariableProperty, value);
        }

        public static readonly DependencyProperty CollectionProperty =
            DependencyProperty.Register(
                "Collection",
                typeof(IEnumerable),
                typeof(InterleavedForEach)
            );

        public IEnumerable Collection
        {
            get { return (IEnumerable)GetValue(CollectionProperty); }
            set { SetValue(CollectionProperty, value); }
        }
    }
}```
protected override ActivityExecutionStatus Execute(
    ActivityExecutionContext context)
{
    if (Collection == null)
        return ActivityExecutionStatus.Closed;

    Activity template = this.EnabledActivities[0];
    ActivityExecutionContextManager manager =
        context.ExecutionContextManager;

    foreach (object item in Collection)
    {
        totalCount++;
        ActivityExecutionContext newContext =
            manager.CreateExecutionContext(template);
        Activity newActivity = newContext.Activity;

        InterleavedForEach.SetIterationVariable(newActivity, item);

        newActivity.Closed += this.ContinueAt;
        newContext.ExecuteActivity(newActivity);
    }

    if (totalCount == 0)
        return ActivityExecutionStatus.Closed;

    return ActivityExecutionStatus.Executing;
}

void ContinueAt(object sender,
    ActivityExecutionStatusChangedEventArgs e)
{
    e.Activity.Closed -= this.ContinueAt;

    ActivityExecutionContext context =
        sender as ActivityExecutionContext;
    ActivityExecutionContextManager manager =
        context.ExecutionContextManager;
    ActivityExecutionContext innerContext =
        manager.GetExecutionContext(e.Activity);
    manager.CompleteExecutionContext(innerContext);

    if (totalCount == ++closedCount)
        context.CloseActivity();
}

public Activity[] ActiveIterations
{
    get
    {
        if (this.EnabledActivities.Count > 0)
        {
            Activity template = EnabledActivities[0];
            return GetDynamicActivities(template);
        }
        return null;
    }
}
The code for InterleavedForEach is not unlike that of our Interleave activity. These two activities really are doing quite similar things, except that for InterleavedForEach, the number of iterations (number of instances of a single child activity) is determined dynamically (at runtime) by the size of the associated collection. In contrast, the list of child activities of an Interleave is described statically.

Just as with While, we should also implement a helper method on InterleavedForEach that provides external code with access to the currently running iterations. This is shown in Listing 4.8, which shows the implementation of an ActiveIterations property. In the case of the InterleavedForEach, because each iteration is effectively keyed by a data item (that was taken from the Collection property), we could also provide a useful lookup method that accepts this data item (or some representation of it) as a parameter and returns the activity of the appropriate iteration, if it is still executing.

After an iteration completes, the InterleavedForEach activity closes the corresponding subordinate execution context. That iteration’s activity object is then no longer an element of the ActiveIterations array.

Listing 4.8. InterleavedForEach.ActiveIterations

```csharp
public Activity[] ActiveIterations
{
    get
    {
        Activity template = EnabledActivities[0];
        return GetDynamicActivities(template);
    }
}
```

One important detail in the implementation of InterleavedForEach has to do with the handoff of data from the collection to an iteration of the template activity. A number of other viable strategies exist, but in this example we are taking a simple approach. The ith item in the collection is given to the ith iteration of the child activity in the form of an attached property, called IterationVariable. The item can then be easily retrieved programmatically, using the standard pattern for attached properties, by activity code within the corresponding AEC:

```csharp
Activity a = ...
object o = InterleavedForEach.GetIterationVariable(a);
```

The InterleavedForEach activity is a useful construction for modeling real-world control flow. For example, the participants in a document review might be indicated as a list of names. For each participant, a document review task should be assigned and managed until it is completed. In many such situations, the assignees are allowed to complete their tasks simultaneously, making the management of a single task a natural fit for the template activity within an InterleavedForEach. The complexity of managing multiple subordinate execution contexts is entirely hidden from the WF program developer.

One requirement that InterleavedForEach places upon the underlying WF execution model (which we discussed earlier in this chapter) is that peer execution contexts are isolated and cannot interfere with each other. Just as we described earlier for While, each iteration must operate on local data. Stated differently, each iteration executes a local instance of the activity subtree.

The WF runtime enforces that all subordinate execution contexts must be completed (the ExecutionContexts collection is empty) in order for a composite activity to move to the Closed state. This is just an extension of the statement that all child activities of a composite activity must be in the Initialized state or the Closed state in order for the composite activity to move to the Closed state.

**Completed Activity Execution Contexts**
Our examples so far assume that after a subordinate AEC is completed, it can be discarded. This simple AEC lifecycle is sufficient for the implementation of many composite activities. But it is also possible to serialize a completed AEC for retrieval at a later time. A completed AEC will be serialized (as part of the state of the WF program instance) instead of discarded if you call the overload of `CompleteExecutionContext` that accepts two parameters, passing `true` as the value of the second parameter. The name of this parameter is `forcePersist` and, as its name indicates, it determines whether the state of the completed AEC should be saved or not. The overload of `CompleteExecutionContext` that takes one parameter simply calls the second overload passing a value of `false` for `forcePersist`.

A completed AEC will be saved automatically (regardless of the value of `forcePersist` parameter) if there are successfully executed compensatable activities within the completed activity subtree of that AEC. In order for activity compensation to work, it must be possible to resurrect a previously completed AEC and continue its execution by scheduling its compensation logic. Thus, the potential for compensation overrides a `false` value for the `forcePersist` parameter. Compensation will be discussed fully later in this chapter; for now, it is enough to appreciate that compensation logic requires the restoration of a previously completed AEC.

Additional use cases for this advanced feature will be discussed in Appendix B, "Control Flow Patterns."

**AEC and WF Program Passivation**

Because multiple instances of an activity can execute (each in a different AEC), there is a need to manage the lifecycle of those activity objects in an AEC-specific manner.

The WF runtime will invoke `Activity.OnActivityExecutionContextLoad` on the activities of a newly created AEC as part of a composite activity's invocation of the `AECManager.CreateExecutionContext` method. This allows the dynamically created activities within the new AEC a chance to allocate resources specific to the lifetime of that AEC. Similarly, the WF runtime will invoke `Activity.OnActivityExecutionContextUnload` on the activities in an AEC as part of the `AECManager.CompleteExecutionContext` method.

**Listing 4.9** shows the definition of these methods on the `Activity` class.

### Listing 4.9. AEC Loading and Unloading

```csharp
namespace System.Workflow.ComponentModel
{
    public class Activity
    {
        protected virtual void OnActivityExecutionContextLoad(IServiceProvider provider);
        protected virtual void OnActivityExecutionContextUnload(IServiceProvider provider);
        /* *** other members *** */
    }
}
```

The execution lifetime of an AEC (including the default AEC) does not necessarily match the in-memory lifetime of the activity objects that are executed within the AEC. Specifically, the lifetime of a subordinate AEC (bracketed by invocation of `CreateExecutionContext` and `CompleteExecutionContext`) may span multiple passivation cycles for the WF program instance. Consider again the program of **Listing 4.6**. Each iteration of the `While` executes the `Sequence` activity (and its nested `ReadLine` and `WriteLine` activities) in a separate subordinate AEC. Each iteration blocks until the hosting application enqueues an item into the WF program queue that is created by the active instance of the `ReadLine` activity. Because the scheduler's work queue might become empty at this
point, the WF program instance can passivate. The program will be reactivated and resume the execution of the
ReadStream activity after the external stimulus arrives.

Upon WF program passivation, the default AEC and all active subordinate execution context (which in the case of
While will be at most one) will be saved. As a part of disposing the program instance, the WF runtime will invoke
OnActivity-ExecutionContextUnload on the activities in each of the execution contexts, allowing all activity
instances to clean up resources allocated for the in-memory lifecycle of the AEC. Correspondingly, the WF
runtime will call OnActivityExecutionContextLoad on the activities of all active execution contexts in the
program instance each time the program is brought back into memory.

A sequence diagram that shows a single iteration of While is depicted in Figure 4.6.

Exhibit 4.6. AEC lifetime, for a single iteration of the while activity in Listing 4.2

Cancellation

In real-world processes, it is common to begin two or more activities simultaneously, but then demand that only a
subset of the activities need to complete in order to reach the desired goal. For example, when you are trying to sell
your car, you might simultaneously buy a classified advertisement in the local newspaper, place a listing at the top
Internet used car marketplaces, and put a sign in the window of your car. If a neighbor sees the sign and offers to
buy the car, the other avenues used to locate buyers are no longer needed and should be abandoned.

In C# programs, it is certainly possible to create multiple threads in order to perform work concurrently. The basic
capabilities of threads, though, do not directly provide patterns for orderly cancellation of work; custom solutions
must be devised. After all, we don't want to pay for newspaper and Internet advertisements after the car is sold.
Furthermore, if the actual work to be done is performed outside of the C# program perhaps by remote web services
or by an application that waits for input from people (a car can take weeks to sell) then the use of threads to manage
this work is not workable anyway (as we discussed in Chapter 1, "Deconstructing WF").

We will now see how easy it is to write a WF composite activity that implements exactly the semantic required in the car sale scenario. Once you are on the road to developing these kinds of composite activities, you will be able to more precisely translate the requirements of real-world processes to the control flow of the WF programs you write.

The AnyOneWillDo composite activity used in the program of Listing 4.10 is very similar to Interleave because AnyOneWillDo will schedule all of its child activities for execution in a single burst during the AnyOneWillDo.Execute method. AnyOneWillDo differs from Interleave in that after a single child activity reports its completion (moves to the Closed state from the Executing state), the AnyOneWillDo activity is logically ready to report its own completion.

Listing 4.10. XAML that Uses an AnyOneWillDo Activity

```xml
<AnyOneWillDo...>
  <SellCarWithNewspaperAd/>
  <SellCarOnline/>
  <SellCarWithSignInWindow/>
</AnyOneWillDo>
```

There is a catch, though. The WF runtime will not allow any composite activity to move to the Closed state while child activities within that composite activity are still in the Executing state. A composite activity cannot logically be done with its work if its child activities are still executing (and are therefore not finished with their work).

A composite activity can only transition to the Closed state if every one of its child activities is either in the Initialized state (which means it was never asked to do its work) or in the Closed state. Because this enforcement applies recursively to those child activities that themselves are composite activities, it is assured that the entire subtree of a composite activity is quiet (that is, Initialized or Closed) before the composite activity itself moves to the Closed state.

To get around this, AnyOneWillDo could wait for the completion of all child activities, but that would defeat the purpose and take us back to exactly what Interleave does. Instead, AnyOneWillDo can request cancellation of the executing child activities after one of the child activities completes.

The Canceling State

The cancellation of a child activity is requested via the ActivityExecution-Context in exactly the same way as the request to schedule the execution of a child activity. Cancellation is a scheduled dispatch to the Cancel method of the child activity.

Listing 4.11 shows the CancelActivity method of AEC, along with ExecuteActivity, which we have used already.

Listing 4.11. ActivityExecutionContext.CancelActivity

```csharp
namespace System.Workflow.ComponentModel
{
    public sealed class ActivityExecutionContext
    {
        public void CancelActivity(Activity activity);
        public void ExecuteActivity(Activity activity);
    }
}
```
We will return in a moment to what the child activities might implement in their `Cancel` method; first let's look at the code in `AnyOneWillDo` that achieves the cancellation.

Listing 4.12 shows a modified version of the `ContinueAt` method that we first developed for `Interleave`. When `AnyOneWillDo` receives notification that a child activity has moved to the Closed state, it calls the `CompletionThresholdReached` method.

CompletionThresholdReached is assumed to be a private helper method that iterates through the child activities, checking their `ExecutionStatus` and `ExecutionResult` properties to see if the necessary condition for completion has been met. In the case of `AnyOneWillDo`, if one child activity has successfully completed its execution, then all other child activities that are not yet in the Closed state (the logic of the `Execute` method, which is the same as that of `Interleave`, ensures that no child activity is still in the Initialized state) are scheduled for cancellation.

Listing 4.12. `AnyOneWillDo.ContinueAt`

```csharp
public class AnyOneWillDo : CompositeActivity
{
    ...

    void ContinueAt(object sender, ActivityExecutionStatusChangedEventArgs e)
    {
        e.Activity.Closed -= this.ContinueAt;

        ActivityExecutionContext context = sender as ActivityExecutionContext;

        if (CompletionThresholdReached())
        {
            bool okToClose = true;
            foreach (Activity child in this.EnabledActivities)
            {
                ActivityExecutionStatus status = child.ExecutionStatus;
                if (status == ActivityExecutionStatusExecuting)
                {
                    okToClose = false;
                    context.CancelActivity(child);
                }
                else if (status != ActivityExecutionStatus.Closed)
                {
                    okToClose = false;
                }
            }

            if (okToClose)
                context.CloseActivity();
        }
    }
}
```
The call to `CancelActivity` immediately moves the child activity into the Canceling state from the Executing state. Just as with normal execution, it will be up to the child activity to decide when to transition to Closed from Canceling. When that transition does happen, the `AnyOneWillDo` activity will be notified of this change (it has already subscribed for the `Closed` event of the child activity, when it was scheduled for execution). Thus, only when all child activities are Closed, either by virtue of having completed their execution or by having been canceled, will the `AnyOneWillDo` activity report its own completion and move to the Closed state. At this point, only one of the child activities will have completed. The car has been sold, and the other work items that had started simultaneously have been canceled.

The `AnyOneWillDo` example illustrates how easily cancellation becomes a normal part of WF program execution. Cancellation is a natural and necessary characteristic of many real-world processes. In general, activities that begin their execution cannot assume that they will be allowed to complete their work (unless the work can be finished entirely within the `Execute` method).

In other words, real-world control flow has a natural notion of early completion. This stands in stark contrast to C# control flow. In composite activities that embody elements of real-world processes, the execution of some child activities must be canceled when the overall goal of the set of child activities has been met.

Let's now consider what happens to an activity that has been scheduled for cancellation. The relevant method of `Activity` is the `Cancel` method, which has the same signature as `Execute`:

```csharp
namespace System.Workflow.ComponentModel
{
    public class Activity : DependencyObject
    {
        protected internal virtual ActivityExecutionStatus Cancel(ActivityExecutionContext context);
        /* *** other members *** */
    }
}
```

As we learned in the previous section, the `Cancel` method is the execution handler that is scheduled when a composite activity calls the `CancelActivity` method of AEC, passing the child activity to be canceled as the parameter.

The default implementation of the `Cancel` method, defined by the `Activity` class, is to immediately return a value of `ActivityExecutionStatus.Closed`, indicating that cancellation is complete. Activities needing to perform custom cancellation logic can override this method and add whatever cancellation logic is required.

Figure 4.7 depicts the Canceling state within the activity automaton.

![Figure 4.7. The Canceling state in the activity automaton](image)

Just as with execution, activity cancellation logic might require interaction with external entities; an activity may therefore remain in the Canceling state indefinitely before reporting the completion of its cancellation logic and moving to the Closed state. This is exactly the same pattern as normal activity execution.

The `Wait` activity that we developed in Chapter 3, "Activity Execution," is a simple example of an activity that
should perform custom cancellation work. If a \texttt{Wait} activity is canceled, it needs to cancel its timer because the timer is no longer needed. The \texttt{Wait} activity can also delete the WF program queue that is used to receive notification from the timer service. The cancellation logic of the \texttt{Wait} activity is shown in \textbf{Listing 4.13}.

\textbf{Listing 4.13. \texttt{Wait} Activity's Cancellation Logic}

```csharp
using System;
using System.Workflow.ComponentModel;
using System.Workflow.Runtime;
	namespace EssentialWF.Activities
{
	public class Wait : Activity
{
	private Guid timerId;

	// other members per \textbf{Listing 3.11}
...

	protected override ActivityExecutionStatus Cancel(
		ActivityExecutionContext context)
	{
		WorkflowQueuingService qService =
			context.GetService<WorkflowQueuingService>();

		if (qService.Exists(timerId))
		{
			TimerService timerService =
				context.GetService<TimerService>();

				timerService.CancelTimer(timerId);
				qService.DeleteWorkflowQueue(timerId);
		}

		return ActivityExecutionStatus.Closed;
	}
}
}
```

The \texttt{Cancel} method of the \texttt{Wait} activity does two things: It cancels the timer that was previously established using the \texttt{TimerService}, and it deletes the WF program queue that served as the location for the \texttt{TimerService} to deliver a notification when the timer elapsed.

When a canceled activity's transition to Closed occurs, its \texttt{ExecutionResult} property will from that point forward have a value of Canceled.

There is one subtle aspect to this cancellation logic, which is represented by the fact that the aforementioned cleanup is wrapped in a conditional check to see whether the WF program queue actually exists. The WF program queue will only exist if the \texttt{Execute} method of the \texttt{Wait} activity was executed. It would seem that this is an assertion we can make. But in fact it is not so. It is possible, though certainly not typical, for a \texttt{Wait} activity to move to the Executing state (when its parent schedules it for execution), and then to the Canceling state (when its parent schedules it for cancellation) without its \texttt{Execute} method ever being dispatched.

If the scheduler dequeues from its work queue an item representing the invocation of an activity's \texttt{Execute} method, but that activity has already moved to the Canceling state, then the work item corresponding to the \texttt{Execute} method is discarded. Invocation of the \texttt{Execute} execution handler would violate the activity automaton (because the activity is now in the Canceling state), and the reasonable expectation of the parent composite activity
that had scheduled the cancellation of the activity. In short, the WF runtime enforces the activity automaton when scheduler work items are enqueued (for example, an activity cannot be canceled if it is already in the Closed state), and also when they are dequeued for dispatch.

In the case of the Wait activity, if the WF program queue does not exist, the Execute method was never actually invoked. If the Execute method was not invoked, no timer was established and hence there is no need to call the CancelTimer method of the TimerService.

We can illustrate the subtlety just described with a somewhat pathological composite activity, which schedules a child activity for execution and then requests its cancellation in consecutive lines of code. This activity’s Execute method is shown in Listing 4.14.


```csharp
using System;
using System.Workflow.ComponentModel;

class ChangedMyMind : CompositeActivity
{
    protected override ActivityExecutionStatus Execute(ActivityExecutionContext context)
    {
        Activity child = this.EnabledActivities[0];
        child.Closed += this.ContinueAt;
        PrintStatus(child);
        context.ExecuteActivity(child);
        PrintStatus(child);
        context.CancelActivity(child);
        PrintStatus(child);
        return ActivityExecutionStatus.Executing;
    }

    void ContinueAt(object sender, ActivityExecutionStatusChangedEventArgs e)
    {
        PrintStatus(e.Activity);
    }

    void PrintStatus(Activity a)
    {
        Console.WriteLine(a.Name + " is " + a.ExecutionStatus + " : " + a.ExecutionResult);
    }
}
```

Though an extreme case, the ChangedMyMind activity suffices to illustrate the situation we have described. A request to schedule invocation of an activity’s Cancel method is always made after a request to schedule invocation of that activity’s Execute method; however, the dispatcher is not able to invoke Execute (dispatch the first work item) before the cancellation request comes in. Hence, when the Cancel work item lands in the scheduler work queue, the Execute work item will still also be in the work queue. The Execute work item has been logically overtaken because the enqueue of the Cancel work item moves the activity to the Canceling state. When the Execute work item is dequeued by the scheduler, it is ignored (and is not dispatched) because the target activity is already in the Canceling state.

Now let’s consider the following WF program:
Let's assume that the `Trace` activity prints a message when its `Execute` method and `Cancel` method is actually called:

```csharp
using System;
using System.Workflow.ComponentModel;

public class Trace : Activity
{
    protected override ActivityExecutionStatus Execute(ActivityExecutionContext context)
    {
        Console.WriteLine("Trace.Execute");
        return ActivityExecutionStatus.Closed;
    }

    protected override ActivityExecutionStatus Cancel(ActivityExecutionContext context)
    {
        Console.WriteLine("Trace.Cancel");
        return ActivityExecutionStatus.Closed;
    }
}
```

When the WF program executes, we will see the following output:

e1 is Initialized : None
e1 is Executing : None
e1 is Canceling : None
Trace.Cancel
e1 is Closed : Uninitialized

The `Trace` activity never has the work item for invocation of its `Execute` method dispatched. Because it moves to the `Canceling` state before its `Execute` execution handler is dispatched, only the `Cancel` method is invoked.

When you write activities that have custom cancellation logic, it is probably not a bad idea to use a pathological composite activity (like the one shown previously) as a test case to help ensure the correctness of your cancellation logic.

**Composite Activity Cancellation**

We will now return to the execution logic of `Interleave` to see how it responds to cancellation. **Listing 4.15** shows the implementation of its `Cancel` method. As you can see, the `Cancel` method of `Interleave`, unlike that of `Wait`, has no cleanup of private data to perform. It does, however, need to propagate the signal to cancel to any executing child activities because `Interleave` will not be able to report the completion of its cancellation logic if it has child activities that are still executing. The `Interleave` activity will therefore remain in the `Canceling` state until all previously executing child activities are canceled.

**Listing 4.15. Interleave Activity's Cancellation Logic**

```csharp
using System;
using System.Workflow.ComponentModel;
```
The cancellation logic shown in Listing 4.15 is correct for many composite activities, not just for Interleave. For example, it is a correct implementation for Sequence (and we will assume that the logic of our Sequence activity is updated to reflect this), though the cancellation logic for Sequence could be written slightly differently because at most one child activity of Sequence can be in the Executing state at a time. Any child activities in the Executing state are asked to cancel. When all child activities are in either the Initialized state or Closed state, then the composite activity may report its completion.

We've already alluded to the fact that just as for activity execution, activity cancellation is a scheduled request made via AEC. The Interleave activity (and the Sequence activity) must subscribe to the Closed event of any child activity for which it invokes the CancelActivity method of AEC. This in turn requires us to modify the logic of ContinueAt. We must selectively act upon the Closed event depending upon whether the composite activity is in the Executing state or the Canceling state. In other words, we are using the same bookmark resumption point to handle the Closed event that is raised by child activities that complete and also the Closed event that is raised by child activities that are canceled.

It is possible for a composite activity (call it "A") to detect that it is in the Canceling state before its Cancel method is invoked. This will happen if its parent activity schedules "A" for cancellation, but the notification for the Closed event of a child activity of "A" reaches "A" before the dispatch of its Cancel method.

The following code snippet shows a cancellation-aware implementation of Interleave.ContinueAt:

```csharp
namespace EssentialWF.Activities
{
    public class Interleave : CompositeActivity
    {
        ...
        protected override ActivityExecutionStatus Cancel(
            ActivityExecutionContext context)
        {
            bool okToClose = true;
            foreach (Activity child in EnabledActivities)
            {
                ActivityExecutionStatus status = child.ExecutionStatus;
                if (status == ActivityExecutionStatus.Executing)
                {
                    context.CancelActivity(child);
                    okToClose = false;
                }
                else if ((status != ActivityExecutionStatus.Closed) && (status != ActivityExecutionStatus.Initialized))
                {
                    okToClose = false;
                }
            }
            if (okToClose)
                return ActivityExecutionStatus.Closed;
            return ActivityExecutionStatus.Canceling;
        }
    }
}
```

The cancellation logic shown in Listing 4.15 is correct for many composite activities, not just for Interleave. For example, it is a correct implementation for Sequence (and we will assume that the logic of our Sequence activity is updated to reflect this), though the cancellation logic for Sequence could be written slightly differently because at most one child activity of Sequence can be in the Executing state at a time. Any child activities in the Executing state are asked to cancel. When all child activities are in either the Initialized state or Closed state, then the composite activity may report its completion.

We've already alluded to the fact that just as for activity execution, activity cancellation is a scheduled request made via AEC. The Interleave activity (and the Sequence activity) must subscribe to the Closed event of any child activity for which it invokes the CancelActivity method of AEC. This in turn requires us to modify the logic of ContinueAt. We must selectively act upon the Closed event depending upon whether the composite activity is in the Executing state or the Canceling state. In other words, we are using the same bookmark resumption point to handle the Closed event that is raised by child activities that complete and also the Closed event that is raised by child activities that are canceled.

It is possible for a composite activity (call it "A") to detect that it is in the Canceling state before its Cancel method is invoked. This will happen if its parent activity schedules "A" for cancellation, but the notification for the Closed event of a child activity of "A" reaches "A" before the dispatch of its Cancel method.

The following code snippet shows a cancellation-aware implementation of Interleave.ContinueAt:

```csharp
namespace EssentialWF.Activities
{
    public class Interleave : CompositeActivity
    {
        // other members same as earlier
        ...
    }
}
```
void ContinueAt(object sender,
ActivityExecutionStatusChangedEventArgs e)
{
    e.Activity.Closed -= ContinueAt;

    ActivityExecutionContext context =
        sender as ActivityExecutionContext;

    if (ExecutionStatus == ActivityExecutionStatus.Executing)
    {
        foreach (Activity child in EnabledActivities)
        {
            if (child.ExecutionStatus !=
                ActivityExecutionStatus.Initialized &&
                child.ExecutionStatus !=
                ActivityExecutionStatus.Closed)
                return;
        }
        context.CloseActivity();
    }
    else // canceling
    {
        bool okToClose = true;
        foreach (Activity child in EnabledActivities)
        {
            ActivityExecutionStatus status = child.ExecutionStatus;
            if (status == ActivityExecutionStatus.Executing)
            {
                // This happens if invocation of our Cancel method
                // has been scheduled but is still sitting in the
                // scheduler work queue
                okToClose = false;
                context.CancelActivity(child);
            }
            else if ((status != ActivityExecutionStatus.Closed) &&
                     (status != ActivityExecutionStatus.Initialized))
            {
                okToClose = false;
            }
        }
        if (okToClose)
            context.CloseActivity();
    }
}

Early Completion

The AnyOneWillDo activity that we developed earlier in this topic is so similar to Interleave we might consider just building the early completion capability directly into Interleave. While we are at it, we can also generalize how the completion condition is expressed. Imagine a variant of the Interleave activity that carries a property, PercentMustComplete, that indicates the percentage of child activities that must complete in order for the Interleave to be considered complete. If such an Interleave activity is given six child activities, and PercentMustComplete is set to 50%, only three child activities need to complete before the Interleave activity can report its completion:

<Interleave PercentMustComplete="50%" ...>
    <A />
    <B />
    <C />
The `Execute` method of `Interleave` remains as we wrote it in Chapter 3; all child activities are immediately scheduled for execution. As the child activities complete their execution, the `Interleave` is notified via the `ContinueAt` callback. In the `ContinueAt` method, the `Interleave` can now check to see whether the `PercentMustComplete` threshold has been met (or exceeded), and based upon this check, decide whether to report its own completion.

It is easy to write many variants of `Interleave` in which the variation is confined to the criterion used to determine when a sufficient number of child activities have completed their execution. One way to generalize this idea of completion condition is to make it a customizable feature of the composite activity. It is easy to write a variant of `Interleave` that carries a property whose type is `ActivityCondition` (just like the `While` activity we developed has a property of type `ActivityCondition`).

The `Interleave` activity can evaluate this condition in its `Execute` method (in case no child activities need be executed), and again whenever the `ContinueAt` method is invoked, in order to decide when to complete. You can also imagine a variant of `Sequence`, or in fact a variant of just about any composite activity, that provides an early completion capability. Composite activities that create subordinate execution contexts (such as `InterleavedForEach`) need to perform cancellation of the activity instances within subordinate execution contexts as part of their early completion.

The actual syntax of condition types (derivatives of `ActivityCondition`) supported natively by WF is discussed in Chapter 8. Here we will use a stylized form of condition writing to communicate our intent. A completion condition allows composite activities to capture real-world control flow in a very flexible way:

```xml
<Interleave CompletionCondition="A OR (B AND (C OR D))" >
  <A />
  <B />
  <C />
  <D />
</Interleave>
```

This gives us the ability to model real-world processes that are a bit more involved in their completion logic than the simple car sale scenario we started with.

**Cancellation Handlers**

The composite activities we have written are general purpose in nature. In other words, when a `Sequence` activity or `Interleave` activity is used in a WF program, the developer of the program decides which child activities should be added to that occurrence of the composite activity. Although the execution logic of `Sequence` itself is unchanging, there are an infinite number of ways to use it because it accepts any list of child activities you wish to give it.

In this way, it may be said that both the developer of `Sequence` and the WF program developer who uses a `Sequence` activity have a say in what any particular occurrence of `Sequence` actually accomplishes. The WF programming model extends this same idea to activity cancellation, with the idea of a cancellation handler.

The concept is quite simple. When a composite activity is canceled by its parent, its `Cancel` method is scheduled for execution. As we know, this immediately moves the activity into the Canceling state. The composite activity remains in this state until all its child activities are quiet, at which time the composite activity reports the completion of its cancellation logic and moves to the Closed state. This much we have already covered. The extra
step we are introducing here is that if the composite activity has an associated cancellation handler, that handler is
executed as a final step prior to the activity's transition to the Closed state.

Any composite activity (unless its validation logic is written to explicitly prevent this) is allowed to have one
special child activity of type CancellationHandlerActivity. This activity type is defined in the
System.Workflow.ComponentModel namespace and is shown in Listing 4.16. The purpose of a cancellation
handler is to allow the WF program developer to model what should happen in the case of an activity cancellation.

Listing 4.16. CancellationHandlerActivity

```csharp
namespace System.Workflow.ComponentModel
{
    public sealed class CancellationHandlerActivity : CompositeActivity
    {
        public CancellationHandlerActivity();
        public CancellationHandlerActivity(string name);
    }
}
```

Because CancellationHandlerActivity is a composite activity, you may add whatever activities are required to
represent the necessary cancellation logic. The child activities of a CancellationHandlerActivity will execute
in sequential order.

In order to help prevent a composite activity (say, Sequence) from executing its CancellationHandlerActivity
as part of normal execution logic, the EnabledActivities collection of CompositeActivity will never include
the composite activity's cancellation handler. Only the WF runtime will schedule the execution of a cancellation
handler.

Let's take a look at an example to see how CancellationHandlerActivity can be used. Listing 4.17 shows a
simple WF program that starts two timers simultaneously.

Listing 4.17. CancellationHandlerActivity

```xml
<Interleave x:Name="i1" CompletionCondition="seq1 OR seq2"
xmlns="http://EssentialWF/Activities"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:wf="http://schemas.microsoft.com/winfx/2006/xaml/workflow">
    <Sequence x:Name="seq1">
        <Wait Duration="00:01:00" x:Name="delay1" />
        <wf:CancellationHandlerActivity x:Name="ch1">
            <WriteLine x:Name="w1" Text="Cancelling seq1" />
        </wf:CancellationHandlerActivity>
    </Sequence>
    <Sequence x:Name="seq2">
        <Wait Duration="00:00:10" x:Name="delay2" />
        <wf:CancellationHandlerActivity x:Name="ch2">
            <WriteLine x:Name="w2" Text="Cancelling seq2" />
        </wf:CancellationHandlerActivity>
    </Sequence>
</Interleave>
```

The Interleave activity carries a completion condition (expressed in stylized form), which says that either child
activity must complete in order for the Interleave to complete. When the first timer fires, the corresponding Wait
activity ("delay2" based on the Duration values in the example WF program) is notified and then reports its
completion. Sequence "seq2" will be notified of the completion of "delay2", which will cause "seq2" to report its completion. When the Interleave activity is notified of the completion of "seq2", it will evaluate the completion condition and decide that it can report its own completion. Before doing this, though, it must cancel the execution of the other Sequence, "seq1". When "seq1" is canceled, it will propagate the cancellation to "delay1". After "delay1" cancels its timer and reports its cancellation, the cancellation handler associated with "seq1" will be scheduled for execution. Only when this cancellation handler moves to the Closed state will "seq1" finally move to the Closed state.

Figure 4.8 illustrates the same sequence of operations in diagrammatic form.

To summarize, we have seen that activity cancellation can be a commonindeed, essentialpart of normal WF program execution. When a set of activities is executed in an interleaved manner, it is often the case that only some subset of these activities need to complete in order for the goal of that set of activities to be realized. The other in-progress activities must be canceled. Activity developers can add cancellation logic to their activities, and WF program developers can also have a say in what happens during cancellation by using a cancellation handler.

Fault Handling

The execution handlers that are invoked by the scheduler are just methods on activities, so each of them has the potential to throw an exception. An exception can occur in Execute, Cancel, ContinueAt, or any activity method that is scheduled for execution. If an exception does occur, and is not handled internally by the activity's code (which of course is free to employ try-catch constructs), then it is the scheduler that catches the exception (because it is the scheduler that invoked the method). When this happens, it indicates to the WF runtime that the activity that threw the exception cannot successfully complete the work it was asked to perform.

In the WF programming model, the handling of such an exception is an aspect of the activity automaton. Thus,
exception handling in WF programs has an asynchronous flavor that exception handling in C# programs does not have. As we shall see, exceptions propagate asynchronously (via the scheduler work queue) and are therefore handled by activities asynchronously. It’s important to keep this high-level characteristic of WF exception handling in mind as you write and debug activities and WF programs.

The Faulting State

The throwing of an exception by an activity is at some level comparable to an activity calling the CloseActivity method of AEC; both are signals from an activity to the WF runtime that indicate the activity’s decision to transition from one state to another. In the case of an exception, the offending activity immediately moves to the Faulting state.

As shown in Figure 4.9, there are transitions to the Faulting state from both the Executing state and the Canceling state.

![Figure 4.9. The Faulting state of the activity automaton](image)

Activity initialization is a special case that we discussed in Chapter 3. The Initialize method of an activity is not a scheduled execution handler. Activity initialization occurs synchronously as part of the creation of a WF program instance.

If an exception is thrown by the Initialize method of an activity, the call to WorkflowRuntime.CreateWorkflow (within which activity initialization occurs) throws an exception indicating that the WF program instance failed to initialize properly.

A note about terminology: There is no notion of a fault in WF that is in any way different than a CLR exception—the terms are synonyms in WF. However, the term fault handling is favored in WF over exception handling because the mechanism by which faults (exceptions) are handled in WF does differ in important ways from the familiar exception handling constructs in CLR languages like C#. These differences are explored in the remainder of this section.

Figure 4.9 shows that after an activity enters the Faulting state, the only possible transition it can subsequently make is to the Closed state. When this transition occurs, the value of the activity’s ExecutionResult property is Faulted. Figure 4.9 also implies that, if the Faulting state is like the Executing and Canceling states, an activity may remain in the Faulting state for an indefinite amount of time. Indeed this is the case. Only an activity can decide when it is appropriate to transition to the Closed state; this is true no matter whether an activity makes the transition to the Closed state via normal execution, cancellation, or the occurrence of a fault.

When an activity enters the Faulting state, its HandleFault method is scheduled by the WF runtime. The HandleFault method, like Execute and Cancel, is defined on the Activity class:

```csharp
namespace System.Workflow.ComponentModel
{
```
When the `HandleFault` execution handler is dispatched, it is expected that the activity will perform any cleanup work that is required prior to its transition to the Closed state. Just as with the `Execute` and `Cancel` execution handlers, the activity may (if the cleanup work is short-lived) return `ActivityExecutionStatus.Closed` from this method. If the cleanup work is long-lived, the activity returns `ActivityExecutionStatus.Faulting` and waits for required callbacks before ultimately calling `ActivityExecutionContext.CloseActivity`.

To illustrate the mechanics of an activity's transitions from the Executing state to the Faulting state to the Closed state, consider the activity shown in Listing 4.18, which throws an exception in its `Execute` method, and then another exception in its `HandleFault` method.

**Listing 4.18. An Activity that Always Faults**

```csharp
using System;
using System.Workflow.ComponentModel;

public class NeverSucceeds : Activity
{
    protected override ActivityExecutionStatus Execute(
        ActivityExecutionContext context)
    {
        throw new InvalidOperationException("told you so");
    }

    protected override ActivityExecutionStatus HandleFault(
        ActivityExecutionContext context, Exception exception)
    {
        Console.WriteLine(exception.Message);
        if (beenHereBefore)
            return ActivityExecutionStatus.Closed;

        beenHereBefore = true;
        throw new InvalidOperationException("second time");
    }
}
```

If we run a WF program that consists of only a `NeverSucceeds` activity, we will see the following output at the console:

told you so
second time

When the program starts, the `Execute` method of the `NeverSucceeds` activity is scheduled, which moves it into the Executing state. When the `Execute` method is invoked, an exception is of course thrown and is caught by the
WF scheduler. This moves the activity into the Faulting state, and also enqueues a work item corresponding to the
activity’s HandleFault method. When HandleFault is dispatched, the Message property of the current exception
("told you so") is written to the console and, because the beenHereBefore variable value is false, a new
exception is thrown. This second exception is also caught by the WF scheduler, and again the HandleFault
method of the NeverSucceeds activity is scheduled for execution. The second time through, the beenHereBefore
variable value is true (having been set during the first invocation of HandleFault) and so the activity reports its
completion (after again printing the Message property of the current exception, which now is "second time"). This
moves the activity from the Faulting state to the Closed state.

This example does illustrate that care must be taken to avoid an infinite loop in which exceptions occurring within
HandleFault cause repeated scheduling of additional calls to HandleFault. The potential for infinite looping is a
byproduct of the fact that an activity must be allowed to remain in the Faulting state for an indefinite amount of
time while it performs any necessary cleanup work.

The HandleFault method allows an activity to perform cleanup work when an exception occurs. However, the
behavior of HandleFault is not the same as that of the catch block familiar to C# programmers. The WF runtime
will schedule the propagation of the exception (to the parent of the faulting activity) only when the faulting activity
transitions to the Closed state.

The faulting activity can choose to suppress the propagation of an exception by setting the
ActivityExecutionContext.CurrentExceptionProperty to null:

```csharp
protected override ActivityExecutionStatus HandleFault(
    ActivityExecutionContext context, Exception exception)
{
    this.SetValue(ActivityExecutionContext.CurrentExceptionProperty, null);

    return ActivityExecutionStatus.Closed;
}
```

The WF runtime automatically populates the AEC.CurrentExceptionProperty with the exception before calling
HandleFault. It will clear the property when the faulting activity transitions to the Closed state. If the exception is
not suppressed by the faulting activity, the exception will be made available to the parent activity.

The implementation of HandleFault that is provided by the Activity class simply returns
ActivityExecutionStatus.Closed. Only when you need special cleanup to occur should you override this
implementation. This is very similar to the situation for cancellation, which we previously discussed. In fact, in
many cases it is useful to factor the cleanup logic for an activity like Wait into a helper method that is then called
from both Cancel and HandleFault.

Composite Activity Fault Handling

As you might expect, things are a bit different for composite activities.

We saw previously that a composite activity will not be able to transition to the Closed state unless every child
activity of that composite activity is either in the Initialized state or the Closed state. Therefore, when the
HandleFault method of a composite activity is dispatched, the default implementation inherited from Activity
will not be correct.

The implementation of the HandleFault method found in CompositeActivity, shown here, will call the Cancel
method of the composite activity in order to ensure cancellation of all currently executing child activities:

```csharp
ActivityExecutionStatus s = this.Cancel(context);
if (s == ActivityExecutionStatus.Canceling)
{
```
return ActivityExecutionStatus.Faulting;
}

return s;

As we saw in the previous section, it is critical that composite activities implement appropriate cancellation logic. The simple implementation of the Cancel method defined by Activity immediately returns a status of ActivityExecutionStatus.Closed. This will not work for composite activities, and can lead to an infinite loop due to the fault that will be thrown by the WF runtime when a composite activity tries to report its completion if it has child activities still doing work.

Just as with cancellation, the composite activity must wait for all active child activities to move to the Closed state before it can report its completion. In the case of a composite activity that is handling a fault, the call to the CloseActivity method of AEC will move the composite activity from the Faulting state to the Closed state.

A composite activity will therefore, by default, see its Cancel method called (from HandleFault) when a fault needs to be handled. It is expected that for many composite activities, the logic of Cancel corresponds exactly to what must happen during the handling of a fault; specifically, all child activities in the Executing state are canceled, and the composite activity's call to AEC.CloseActivity occurs only when all child activities are in either the Initialized state or the Closed state. The implementations of HandleFault in Activity and CompositeActivity are just a convenience, however, and can be overridden in cases where they do not provide the appropriate behavior.

**Fault Propagation**

Thus far, we have only discussed the throwing and handling of a fault in the context of the activity whose execution handler produced an exception. For this activity, there is a transition to the Faulting state (from either the Executing state or the Canceling state) and a subsequent transition to the Closed state. But when this activity moves to the Closed state, the fault can hardly be considered to have been handled. After all, the default implementations of HandleFault provided by Activity and CompositeActivity essentially just enable an activity to move to the Closed state from the Faulting state as quickly as possible.

What actually happens when an activity transitions to the Closed state from the Faulting state is that (unless the fault is suppressed using the technique outlined earlier) the WF runtime automatically propagates the exception one level up the activity hierarchy. Specifically, the WF runtime schedules a work item for the HandleFault method of the parent of the activity that faulted; when this occurs, the parent activity moves to the Faulting state. Here we find the key point of difference between fault handling in WF and exception handling in the CLR. The WF fault does not propagate until the faulting activity moves to the Closed state; as we know this might take an arbitrary amount of time during which other scheduled execution handlers in the same WF program instance will be dispatched. The execution of other activities proceeds, unaffected by the fact that a fault has occurred.

When the exception propagates one level up, the parent composite activity eventually has a work item for its HandleFault method dispatched. As expected, the fault-handling logic of this composite activity will cause the cancellation of any executing child activities. Only when all child activities are quiet will the composite activity move to the Closed state; when this occurs, the exception will propagate yet one more level up the activity tree. In this way, the WF fault-handling model provides for the orderly cleanup of work as the exception propagates up the tree, one composite activity at a time.

WF's execution model is stackless and is driven by the activity automaton. Because activities are organized hierarchically in a WF program, the WF runtime must propagate exceptions according to the constraints of the activity automaton; this ensures downward propagation (and, indeed, completion) of cancellation (orderly cleanup of work) prior to the upward propagation of an exception. The WF program developer must be aware of the innately asynchronous nature of WF program execution. This is especially true when the fault-handling capabilities of activities are utilized.
Fault Handlers

We saw earlier in this chapter that the WF runtime schedules the execution of a cancellation handler, if one is present, when a composite activity transitions from the Canceling state to the Closed state. In a similar fashion, a FaultHandlers-Activity, if present, is scheduled for execution as part of a composite activity's transition from the Faulting state to the Closed state.

A FaultHandlersActivity (note the plural) is just an ordered list of child activities of type FaultHandlerActivity. A FaultHandlerActivity allows the WF program developer to model the handling of a fault of a specific type much like a catch handler in C#.

The FaultHandlerActivity type is shown in Listing 4.19.

Listing 4.19. FaultHandlerActivity

```csharp
namespace System.Workflow.ComponentModel
{
    public sealed class FaultHandlerActivity : CompositeActivity
    {
        public Exception Fault { get; }
        public Type FaultType { get; set; }

        /* *** other members *** */
    }
}
```

The execution logic of FaultHandlersActivity is responsible for finding the child FaultHandlerActivity that can handle the current fault. If one is found, that activity is scheduled for execution, and the composite activity (whose FaultHandlersActivity is being executed) remains in the Faulting state until the fault handler completes its execution. If no matching fault handler is found, the fault is propagated by the WF runtime to the next outer (parent) composite activity, and the composite activity handling the fault moves from the Faulting state to the Closed state.

When a FaultHandlerActivity executes, it sets the CurrentException-Property of the faulting composite activity to null in order to suppress the propagation of the (successfully handled) exception.

Unhandled Faults

If the root activity of a WF program moves from the Faulting state to the Closed state without having had a FaultHandlerActivity successfully handle the fault, the WF program terminates. The exception propagates to the application hosting the WF runtime in the form of an event.

The host application can subscribe to the WorkflowRuntime.WorkflowTerminated event to see the exception:

```csharp
using (WorkflowRuntime runtime = new WorkflowRuntime())
{
    runtime.WorkflowTerminated += delegate(object sender, WorkflowTerminatedEventArgs e)
    {
        Exception exception = e.Exception;
        ...
    };

    ...
}
```
Modeled Faults

It is an inescapable fact that exceptions will sometimes be thrown by opaque activity code. Sometimes, though, it can be desirable to represent, or model, the throwing of a fault as an explicit part of a WF program.

A side effect of layering the WF runtime on top of the CLR is that the WF runtime must be able to recognize and deal with CLR exceptions raised, at the lowest level, via the Microsoft Intermediate Language (MSIL) `throw` instruction. Activities, after all, are compiled to MSIL instructions. Therefore, although the propagation of a fault and its eventual handling is governed by the laws of the WF runtime (and not the CLR’s), the mechanism of throwing a fault is still the MSIL `throw` instruction.

In a more pristine architecture, wherein activities might be written with an exclusive reliance on WF APIs, WF could provide an API for activity writers to notify the WF runtime of the occurrence of a fault. This API would be used instead of the MSIL `throw` instruction. Rather than invent this duplicate API, though, WF pragmatically uses the `throw` statement of languages like C# as the mechanism by which activities signal a fault. Purists might object to the absence of a WF API for signaling a fault; this is an issue\[1\] that illustrates the design tradeoffs inherent in the building of a meta-runtime such as the WF runtime on top of the CLR.

\[1\] The practicalities of activity object constructors and the `Dispose` method is another that we encountered earlier.

That said, from the perspective of the WF program developer, it remains useful to hide this debate and provide an activity whose job is to raise a fault. An activity analog of the C# `throw` statement, a `ThrowFault` activity, is shown in **Listing 4.20**.

**Listing 4.20. ThrowFault Activity**

```csharp
using System;
using System.Workflow.ComponentModel;

namespace EssentialWF.Activities
{
    public class ThrowFault : Activity
    {
        public static readonly DependencyProperty FaultProperty = 
            DependencyProperty.Register("Fault",
                typeof(Exception),
                typeof(ThrowFault)
            );

        public Exception Fault
        {
            get { return GetValue(FaultProperty) as Exception; } 
            set { SetValue(FaultProperty, value); } 
        }

        protected override ActivityExecutionStatus Execute(
            ActivityExecutionContext context)
        {
            if (Fault == null)
                throw newInvalidOperationException("Null Fault");

            throw Fault;
        }
    }
}
```
Setting the Fault property prior to the execution of a THRowFault activity is analogous to placing a reference to an object of type System.Exception on the stack prior to execution of the MSIL throw. If the exception object is null, an InvalidOperationException occurs.

The presence of THRowFault in a WF program indicates precisely where the program is predictably going to fault. We don't even have to run the program in Listing 4.21. Just by looking at it, you can predict the result. The program will write "hello, world" but will then terminate without writing "unreachable" to the console.

Listing 4.21. WF Program That Throws an Exception

```xml
<Sequence x:Name="s1" xmlns="http://EssentialWF/Activities"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <WriteLine x:Name="w1" Text="hello, world"/>
  <ThrowFault x:Name="throw1"/>
  <WriteLine x:Name="w2" Text="unreachable"/>
</Sequence>
```

The ThrowFault activity is useful as written, but (outside of using a default System.InvalidOperationException if no other is provided) it takes no responsibility for creating the exception that it throws. Although the WF program developer could certainly resort to code to create a specialized exception, which ThrowFault can then throw, this more or less defeats the purpose of modeling the throwing of the exception. In other words, it is preferable to declaratively specify both the creation and the throwing of an exception.

One way to go about this would be to write a CreateFault activity that manufactures a System.Exception object based upon certain inputs (such as the type of the exception to be created, and any constructor parameter values that should be used). The Fault property of a THRowFault activity could then be bound to the property of CreateFault that holds the manufactured exception.

This is a perfectly fine approach, but it makes reasoning about the types of exceptions thrown in a WF program a bit trickier because one must parse databinding expressions in order to determine the type of an exception that will be thrown. An alternative approach is to collapse both steps into a single activity that manufactures and throws an exception. A simple version of such an activity is shown in Listing 4.22.

Listing 4.22. THRowTypedFault Activity

```csharp
using System;
using System.Workflow.ComponentModel;

namespace EssentialWF.Activities
{
    public class ThrowTypedFault : Activity
    {
        public static readonly DependencyProperty FaultTypeProperty =
            DependencyProperty.Register("FaultType",
                                typeof(System.Type),
                                typeof(ThrowTypedFault),
                                new PropertyMetadata(DependencyPropertyOptions.Metadata));

        public static readonly DependencyProperty FaultMessageProperty =
            DependencyProperty.Register("FaultMessage",
                                typeof(string),
                                typeof(ThrowTypedFault),
                                new PropertyMetadata(DependencyPropertyOptions.Metadata));
    }
}```
When it executes, the `ThrowTypedFault` activity creates a new exception object whose type is determined by the value of its `FaultType` metadata property. It is assumed that the exception type has a constructor that takes a single string parameter.

A `ThrowTypedFault` activity with a `FaultType` of `InvalidOperationException` and a `FaultMessage` of "nice try" is essentially equivalent to this C# statement:

```csharp
throw new InvalidOperationException("nice try");
```

It is easy to augment the implementation of `ThrowTypedFault` to make it more robust, and to make it meet the needs of your own WF programs. Appropriate validation logic (discussed in Chapter 7) can ensure that the `FaultType` property holds a type reference to a derivative of `System.Exception`. Additional properties (for example, an `InnerException` property that becomes yet another constructor parameter for the manufactured exception) might be added as well.

When the throwing and handling of exceptions are both modeled in a WF program, it becomes possible to reason quite usefully about the behavior of the program without looking at any activity code. The WF program in Listing 4.23 shows such a program.

**Listing 4.23. WF Program with Exceptions**

```xml
<Sequence x:Name="s1" xmlns="http://EssentialWF/Activities" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```
Listing 4.23 is functionally similar to the following C# code:

```csharp
try {
    ...
    throw new InvalidOperationException("oops");
    ...
} catch (InvalidOperationException e) {
    Console.WriteLine(e.Message);
} catch (Exception e) {
    throw e;
}
```

The sequence diagram in Figure 4.10 shows the sequence of execution for the WF program shown in Listing 4.23.

Figure 4.10. Sequence diagram for Listing 4.23

[View full size image]
Keep in mind that the exception propagation mechanism of the WF runtime ensures orderly cleanup of executing activities beginning at the origin of the exception. In WF, both the execution and exception models are fundamentally asynchronous. Only when the exception bubbles up to the composite activity with a fault handler that is able to handle the fault (has a matching fault type) will the fault be handled.

When an exception occurs in a C# program, the transfer of control to the appropriate exception handler (however many nested program statements need to be traversed) is effectively instantaneous. In a WF program, propagation of a fault to a fault handler will take an indeterminate amount of time, depending upon the time it takes to clean up (cancel) running activities within the activity subtrees of the composite activities that must move from the Faulting state to the Closed state.

**ThrowActivity**

The `System.Workflow.ComponentModel` namespace includes an activity named `ThrowActivity` (by convention, the names of activity types included with WF are suffixed with "Activity"). `ThrowActivity` combines the functionality of the two activities we've written in this chapter and thus is an easy way to model the throwing of exceptions in WF programs.

The following XAML snippets illustrate the usage of `ThrowActivity`:

```xml
<ThrowActivity FaultType="{x:Type System.InvalidOperationException}" />
<ThrowActivity Fault="{wf:ActivityBind SomeActivity, Path=SomeProp}" />
```

You can also specify both `FaultType` and `Fault` when using `ThrowActivity`, in which case a validation error occurs if the value of the `FaultType` property and the type of the object referenced by the `Fault` property do not match. Where `Throw-Activity` does not meet the needs of your WF programs, though, hopefully you have seen in this section how easy it is to craft variants that deliver the functionality you require.

**Compensation**

As we know, a typical WF program will perform work (execute activities) episodically. The effects of the execution of these activities are not necessarily, but often, visible outside of the boundaries of the WF program. To take a few concrete examples, one activity in a WF program might send an email, another activity might create and
write to a file on the local hard drive, and yet another activity might call a web service.

If a WF program instance successfully executes these activities, but then later (perhaps a second later, or perhaps six months later) has its execution go awry, it might be desirable to try to undo the effects of the successfully completed activities. This is no different conceptually than what you might do in a C# program when you catch an exception. The logic of a C# `catch` block can attempt to undo the effects of the partially completed `Try` block with which it is associated. For example, if the code in your `Try` block created a file but failed during the writing of data to that file (and consequently threw an exception), the `catch` block might delete the file. Complicating this problem is the fact that your cleanup logic probably needs to figure out just how far the work in the `Try` block proceeded before the exception occurred.

You are free to take this approach in WF programs. We have already discussed the fault-handling model of WF, and though you know now that the WF execution model differs from that of the CLR because of the asynchronous nature of the WF scheduler, it's nevertheless true that the purpose of a WF `FaultHandlerActivity` is essentially the same as that of a C# `catch` block. When a `FaultHandler-Activity` is executed, it means that something unexpected has happened in the execution of the WF program instance, and it also means that you have a chance to sort out the potentially messy state in which your program instance now finds itself.

Another strategy is to employ transactions so that work performed under the auspices of a transaction can be rolled back if the transaction cannot commit. Just as they do in CLR programs, transactions do play an important role in WF programs (and we will discuss transactions in Chapter 6, "Transactions"). But they are not a panacea. It is simply not practical to encompass long-running work (that spans episodes) in a single transaction; locks cannot be held for the required duration. Further complicating the situation is the fact that not all (or perhaps none) of the work represented by activities is performed locally; think of our activity that calls a web service; organizations will never give direct control of their transactional resources to external entities. Standards have emerged to address the transactional aspects of (short-lived) remote work, but such capabilities are not yet commonplace infrastructure. Use of transactions is therefore complementary to (and not a replacement for) other aspects of WF programming such as fault handling.

The Compensating State

The WF programming model permits the association of compensation logic with an activity. In a nutshell, the way compensation logic is expressed is very much the same as cancellation logic. An activity can implement a `Compensate` method if it wishes to express compensation logic in code.

Additionally, a composite activity can be given a `CompensationHandler-Activity`, so that the developer of a WF program can provide custom compensation logic for that composite activity.

The rules that govern the execution of compensation logic are a bit more complicated than those governing cancellation, and will be demonstrated in the examples of this section. At the level of the activity automaton, things are pretty simple. A compensatable activity that is in the Closed state with a result of Succeeded is allowed (but not required) to make a transition from the Closed state to the Compensating state. From the Compensating state, a transition can be made either to the Faulting state, or back to the Closed state.

The expected path is from the Compensating state back to the Closed state, which leaves the activity with an execution result of Compensated (thus precluding a second transition to the Compensating state). These transitions are shown in Figure 4.11.

Figure 4.11. The Compensating state of the activity automaton

Compensatable Activities
Not all activities are compensatable. In some cases, it might not be possible to describe reasonable compensation logic for an activity. Think of the `Wait` activity, which just waits for a timer to fire. After the timer fires, and the `Wait` activity completes, there is no way to undo the fact that the timer fired. In other cases it might not be desirable to utilize compensation (for example, if one takes the approach of doing best-effort cleanup within fault handlers).

In order to keep the WF programming model simple, compensation is an opt-in model. An activity must implement the `ICompensatableActivity` interface in order to be considered compensatable.

The `ICompensatableActivity` type is shown in **Listing 4.24**.

**Listing 4.24. ICompensatableActivity**

```csharp
using System.Workflow.ComponentModel;

public interface ICompensatableActivity
{
    ActivityExecutionStatus Compensate(
        ActivityExecutionContext context);
}
}
```

`ICompensatableActivity` defines one method, `Compensate`. Like `Activity.Execute` and `Activity.Cancel`, this method takes an `ActivityExecutionContext` as a parameter and returns `ActivityExecutionStatus`. Also like `Execute` and `Cancel`, `Compensate` is an execution handler whose dispatch is scheduled via the WF runtime.

Just as with cancellation and fault handling, an activity can complete its compensation logic within the `Compensate` method and return `ActivityExecutionStatus.Closed`. An activity may alternatively implement long-running compensation logic, in which case a value of `ActivityExecutionStatus.Compensating` needs to be returned from the `Compensate` method.

As a simple example, consider the `SendEmail` activity shown in **Listing 4.25**.

**Listing 4.25. SendEmail Activity with Compensation Logic**

```csharp
using System;
using System.Workflow.ComponentModel;

namespace EssentialWF.Activities
{
    public class SendMail : Activity, ICompensatableActivity
    {
        // To, From, Subject, and Body properties elided...
    }
}
```
The `SendEmail` activity's `Compensate` method simply sends a second email indicating that the original email should be ignored. A different approach might be to define a `Recall` method on the `MailService` type, and invoke that method from `Compensate`. In this manner, the `MailService` can decide the most appropriate action. In some situations, the second email might be sent, but in other cases (perhaps the original email had not been sent because the machine on which the WF program is executing is offline, or the destination mail server supported a recall mechanism for delivered but unread email), use of a full-featured email server might produce a more desirable outcome.

It is important to understand that we are not rolling back the work done by `SendEmail` in a transactional sense. We are logically undoing, or more accurately compensating for, that previously completed work. In the time that elapses between the first email and the second (or the recall), there may well have been side effects due to the visibility of the data. To take a different example, an `Insert` activity that inserts a row into a database could well have a `Compensate` method that deletes that row from the database. But, in the seconds, hours, or weeks that elapse before the row is deleted, any number of applications might have read and acted upon (even updated or deleted) the data in that row. For these reasons, it is important to carefully consider the requirements of the scenarios that are targeted by the compensation logic that you write.

**Compensation Handlers**

Composite activities can also implement `ICompensatableActivity` and provide custom compensation logic. More typically, though, the `Compensate` method of a composite activity will look like the example shown in Listing 4.26.

### Listing 4.26. CompensatableSequence Activity

```csharp
using System;
using System.Workflow.ComponentModel;

namespace EssentialWF.Activities
{
    public class CompensatableSequence : Sequence, ICompensatableActivity
    {
        ActivityExecutionStatus ICompensatableActivity.Compensate(
            ActivityExecutionContext context)
        {
            MailService mailService = context.GetService<MailService>();
            mailService.Send(To, From, "IGNORE:" + Subject);
            return ActivityExecutionStatus.Closed;
        }
    }
}
```
Clearly there is no actual compensation logic here.

However, by implementing `ICompensatableActivity`, the `CompensatableSequence` activity allows the WF program developer to associate a `CompensationHandlerActivity` with any `CompensatableSequence`. When the `CompensatableSequence` transitions to the Closed state from the Compensating state, the WF runtime schedules the execution of the `CompensationHandlerActivity`.

`CompensationHandlerActivity` is similar in purpose to the cancellation handlers and fault handlers discussed earlier. A transition to the Closed state from the Compensating state made by the parent of a `CompensationHandlerActivity` does **not actually occur until the `CompensationHandlerActivity` completes its execution**.

**Listing 4.27** shows how a `CompensationHandlerActivity` is used in the definition of a WF program. Two `CompensatableSequence` activities are executed simultaneously, and each one defines a compensation handler.

**Listing 4.27. Modeling Compensation Logic Using `CompensationHandlerActivity`**

```xml
<Sequence x:Name="s1" xmlns="http://EssentialWF/Activities"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:wf="http://schemas.microsoft.com/winfx/2006/xaml/workflow">
  <Interleave x:Name="i1">
    <CompensatableSequence x:Name="seq1">
      ....
      <wf:CompensationHandlerActivity x:Name="ch1">
        <WriteLine x:Name="w1" Text="In compensation handler 1" />
      </wf:CompensationHandlerActivity>
    </CompensatableSequence>
    <CompensatableSequence x:Name="seq2">
      ....
      <wf:CompensationHandlerActivity x:Name="ch2">
        <WriteLine x:Name="w2" Text="In compensation handler 2" />
      </wf:CompensationHandlerActivity>
    </CompensatableSequence>
  </Interleave>
  <ThrowTypedFault x:Name="throw1" FaultMessage="oops" FaultType="{x:Type System.InvalidOperationException}" />
</Sequence>
```

The two `CompensatableSequence` activities execute, and then a fault occurs. Now it is time to run compensation logic.

**Default Compensation**

**Listing 4.27** illustrates how compensation handlers are defined, but we have not explained how they will be invoked. If we examine **Listing 4.27**, we see that an exception will be thrown by the `THrowTypedFault` activity that follows the `Interleave` containing the two `CompensatableSequence` activities.
It is the default fault handling and compensation logic of Sequence and Interleave, inherited from CompositeActivity, which ensures that the compensation handlers for the two compensatable child activities are run. When the THRowTypedFault activity moves to the Closed state (with a result of Faulted), the exception is propagated to its parent activity, the Sequence. The Sequence does not have a child FaultHandlersActivity, but even so, it will check to see if it contains any child activities that require compensation. Because this is indeed the case, the compensation handlers of the two CompensatableSequence activities will be scheduled. The order in which they are scheduled is the reverse order in which they completed their execution.

When executed, the WF program in Listing 4.27 will output one of the two following results, depending upon which CompensatableSequence finished first:

In compensation handler 1
In compensation handler 2

Or:

In compensation handler 2
In compensation handler 1

Only when the compensation handlers complete their execution will the exception propagate up from the Sequence activity. In this example, the Sequence is the root of the WF program, but if it were not, the exception would be propagated up the tree of activities one composite activity at a time. As the exception propagates up, compensation of any completed, compensatable activities (and cancellation of executing activities) occurs in an orderly (and sequential) fashion until an appropriate FaultHandlerActivity handles the exception or the WF program instance terminates.

Default compensation is also triggered for the child activities of a composite activity if the composite activity transitions from the Canceling state to the Closed state and does not have a CancellationHandlerActivity.

The program in Listing 4.28 illustrates default compensation of our SendEmail activity.

Listing 4.28. Default Compensation of a Primitive Activity

```xml
<Sequence xmlns="http://EssentialWF/Activities"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <SendMail x:Name="send1" From="Bob" To="Dharma" Subject="Hello" />
  <ThrowTypedFault FaultMessage="oops" FaultType="{x:Type System.InvalidOperationException}"/>
</Sequence>
```

The Execute method of SendEmail sends out the email message. After an exception is thrown and propagated by the ThrowTypedFault activity, the Sequence moves to the Faulting state. The Sequence does not have a fault handler to handle the exception, but it will schedule the Compensate method of the SendMail and wait for its Closed event before propagating the exception. The compensation logic of SendMail will send the second email (or the recall). In this scenario, the exception is unhandled by the root of the WF program and hence the WF program instance will terminate.

If a compensatable activity such as SendEmail is executed more than one time (say, as a child activity of a While) then each of the iterations is independently compensatable. Compensation fundamentally relies upon the ability, discussed earlier in this chapter, to save and then retrieve a completed execution context. The compensation of
such an activity entails the compensation (in reverse order of completion) of all the successfully completed iterations (execution contexts).

In the program shown in Listing 4.29, the While activity iterates over a CompensatableSequence three times. Subsequently, the ThrowTypedFault activity throws a fault that is not handled by the program. Just before it is propagated by the Sequence, the compensation of each of the iterations of While takes place.

Listing 4.29. Compensation and AECs

```xml
<Sequence x:Name="s1" xmlns="http://EssentialWF/Activities"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:wf="http://schemas.microsoft.com/winfx/2006/xaml/workflow">
  <While x:Name="while1" >
    <While.Condition>
      <ConstantLoopCondition MaxCount="3" />
    </While.Condition>
    <CompensatableSequence x:Name="body">
      <GetCurrentTime x:Name="t1" />
      <WriteLine x:Name="w1" Text="{wf:ActivityBind t1,Path=Time}" />
      <Wait x:Name="wait1" Duration="00:00:04" />
      <wf:CompensationHandlerActivity x:Name="ch1">
        <WriteLine x:Name="w2" Text="{wf:ActivityBind t1,Path=Time}" />
      </wf:CompensationHandlerActivity>
    </CompensatableSequence>
  </While>
  <ThrowTypedFault x:Name="throw1" FaultMessage="oops" FaultType="{x:Type System.InvalidOperationException}" />
</Sequence>
```

The WF program of Listing 4.29 produces the following output:

```
6/10/2006 7:29:24 PM
6/10/2006 7:29:28 PM
6/10/2006 7:29:32 PM
6/10/2006 7:29:32 PM
6/10/2006 7:29:28 PM
6/10/2006 7:29:24 PM
```

Let's analyze the program in Listing 4.29. The While activity is configured to iterate over a CompensatableSequence three times. During each iteration, a GetCurrentTime activity (t1) is executed, followed by a WriteLine activity (w1), followed by a Wait activity (wait1). GetCurrentTime simply takes a snapshot of the current time in its Execute method (by calling DateTime.Now.ToString()) and sets its Time property to this value. The WriteLine activity's Text property is databound to the GetCurrentTime activity's Time property.

Within the compensation handler of the CompensatableSequence, the Text property of the WriteLine activity w2 is databound to the Text property of the WriteLine activity w1. Because the encompassing AEC of a compensation handler is restored during compensation, the application state that was saved at the time the AEC was persisted is available. Because the two WriteLine activities w1 and w2 reside within the same AEC, instances of w2 (the WriteLine inside the CompensationHandlerActivity) are able to print out values that were produced earlier (by instances of w1).

Because compensation of activities that implement ICompensatableActivity might not need to be triggered in a given program instance, compensatable activities that move to the Closed state with a result of Succeeded are not uninitialized until they are either compensated or until the WF program instance completes.
Custom Compensation

Default compensation unfolds naturally and implicitly as a part of fault handling and cancellation handling in a WF program instance. Of course, no compensation at all will ever happen if a WF program contains no compensatable activities (or, more precisely, if no compensatable activities in a WF program successfully complete their execution).

In some cases, the algorithm used by the WF runtime to implement default compensation does not correspond to the desired compensation logic. For these situations, WF includes an activity type named `CompensateActivity` that allows the WF program developer to exert a finer level of control over the compensation process.

The `CompensateActivity` type is shown in Listing 4.30.

**Listing 4.30. CompensateActivity**

```csharp
namespace System.Workflow.ComponentModel
{
    public sealed class CompensateActivity : Activity
    {
        public string TargetActivityName { get; set; }
        /* *** other members *** */
    }
}
```

The most common usage of `CompensateActivity` occurs when a WF program developer wishes to specify custom compensation logic in a `Compensation-HandlerActivity` and also trigger default compensation as part of that composite activity's compensation logic. To meet this requirement, the `CompensateActivity` is added to a `CompensationHandlerActivity` and the value of its `TargetActivityName` property is set to the name of the associated composite activity. This configuration indicates that when the `CompensateActivity` executes, the default (implicit) compensation algorithm should run. This will schedule the execution of the compensation logic for all appropriate (successfully completed) compensatable activities in the composite activity's subtree.

Listing 4.31 defines explicit compensation for the `CompensatableSequence` that, when all is said and done, does exactly what implicit compensation (a `CompensatableSequence` with no `CompensationHandler` at all) would do, except that it also writes two messages to the console, one on either side of the default compensation logic that is triggered by the execution of the `CompensateActivity`.

**Listing 4.31. Explicit Compensation Using CompensateActivity**

```xml
<CompensatableSequence x:Name="seq1"
xmlns="http://EssentialWF/Activities"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:wf="http://schemas.microsoft.com/winfx/2006/xaml/workflow">
    ...
    <wf:CompensationHandlerActivity x:Name="ch1">
        <WriteLine x:Name="w1" Text="Time to run compensation" />
        <wf:CompensateActivity x:Name="c1" TargetActivityName="seq1" />
        <WriteLine x:Name="w2" Text="Done compensating" />
    </wf:CompensationHandlerActivity>
</CompensatableSequence>
```
More advanced explicit compensation is also possible (for instance, in cases where it is not correct to initiate compensation sequentially in the reverse order of activity completion). For these cases, the `CompensateActivity` can be configured to reference a compensatable child activity of the associated composite activity. In this way, multiple `CompensateActivity` activities can be used to prescribe the order in which compensation unfolds. An example of such compensation logic is shown in Listing 4.32.

**Listing 4.32. Explicit Compensation Using `CompensateActivity`**

```xml
<Sequence x:Name="seq1" xmlns="http://EssentialWF/Activities"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:wf="http://schemas.microsoft.com/winfx/2006/xaml/workflow">
  <CompensatableInterleave x:Name="i1">
    <CompensatableSequence x:Name="cs1">
      <wf:CompensationHandlerActivity x:Name="ch1">
        <WriteLine Text="In compensator ch1" x:Name="w1"/>
      </wf:CompensationHandlerActivity>
    </CompensatableSequence>
    <CompensatableSequence x:Name="cs2">
      <wf:CompensationHandlerActivity x:Name="ch2">
        <WriteLine Text="In compensator ch2" x:Name="w2"/>
      </wf:CompensationHandlerActivity>
    </CompensatableSequence>
    <wf:CompensationHandlerActivity x:Name="ch4">
      <wf:CompensateActivity TargetActivityName="cs2"/>
      <wf:CompensateActivity TargetActivityName="cs1"/>
    </wf:CompensationHandlerActivity>
  </CompensatableInterleave>
  <wf:CompensationHandlerActivity x:Name="ch3"/>
</Sequence>
```

In contrast to the program in Listing 4.29, which used default compensation, the program in Listing 4.32 models compensation explicitly using `CompensateActivity`. In this way, the program in Listing 4.32 controls the order in which the compensation of child activities occurs. The preceding program will output the following:

```
In compensator ch2
In compensator ch1
```

The `CompensatableInterleave` activity used in Listing 4.32 is a simple modification of the Interleave activity: `CompensatableInterleave` needs to implement the `ICompensatableActivity` interface, just like the `CompensatableSequence` of Listing 4.26.

There are a couple of additional things to observe about `CompensateActivity`.

First, this activity can only appear (at any depth) within a fault handler, a cancellation handler, or a compensation handler. The validation logic of `CompensateActivity` also ensures that the `TargetActivityName` property must refer to a compensatable activity.

Second, when `CompensateActivity` executes and its `TargetActivityName` property references a child activity that does not have a result of Succeeded, the `CompensateActivity` does nothing. This logic covers both the case of a target activity that did not compete successfully, as well as the case of a target activity that has already been compensated.

In summary, the examples shown in this section have illustrated how activity compensation can be used as one strategy within the broader implementation of fault handling and cancellation handling for WF programs. It is
important to remember that compensation does not stand alone, but is rather a mechanism that might help you, in certain cases, implement the fault handling and cancellation handling that your WF programs need. If compensation logic is not helpful in this endeavor, it may be safely ignored due to the opt-in model for compensation adopted by the WF programming model.

Where are We?

Chapters 3 and 4 together have presented the complete activity automaton, and the details of the execution model for activities. This asynchronous execution model rests firmly on the ideas of bookmarks and continuations, and goes hand in hand with the episodic execution of WF programs. The embodiment of the WF execution model is the activity automaton, along with a set of constraints, enforced by the WF runtime, that govern the runtime relationships between any composite activity and its child activities.

Because all WF programs are just compositions of activities, we have also come to see how, viewed from the inside, an in-memory WF program instance is a set of CLR objects, which transiently represent activities within various execution contexts.

In the next chapter, we will shift our perspective and take a look at WF programs from the outside—that is, from the vantage point of the application that hosts the WF runtime and manages the execution of WF program instances.