3
Activity Execution

BECAUSE A WF PROGRAM IS JUST an activity (that typically is the root of a tree of activities), the best way to understand how large WF programs execute is to first understand what happens at the level of a single activity.

The WF programming model codifies the lifecycle of an activity in terms of a finite state machine that we will call the **activity automaton**. Chapter 1, "Deconstructing WF," introduced a simple three-state version of this automaton for resumable program statements. The lifecycle of activities also follows this basic pattern but adds several additional states that will be discussed in this chapter and the next.

The execution model for activities is fundamentally asynchronous because it is designed to accommodate activities that perform **episodic execution**—short bursts of execution punctuated by relatively long periods of time spent waiting for external stimulus.

For efficiency reasons, it does not make sense to keep WF program instances in memory while they are idle waiting for data to arrive. When a WF program instance becomes idle, the WF runtime is capable of storing it in a (pluggable) durable storage medium and disposing the program instance. The process of storing the instance state and disposing the instance from memory is called **passivation**. When relevant stimulus arrives from an external entity, perhaps after days spent waiting, the WF runtime automatically **reactivates** the program, bringing it out of durable storage (where it had been passivated) into memory, and resuming its execution. Relative to its logical lifetime, a typical WF program instance lives for a short duration in memory.

Supporting passivation requires serialization of not only the program state but also the execution state (managed by the WF runtime). When WF program instances are passivated, they are captured by the WF runtime as continuations. A passivated program instance may be resumed in a different process or even on a different machine than the one on which it ran prior to passivation. This means that WF programs are thread-agile and process-agile. WF programs do indeed run on CLR threads, but the execution model for activities across resumption points is stackless because it does not rely on the stack associated with a CLR thread. The lifecycle of a WF program instance may, in a physical sense, span processes and machines and is distinctly different from the lifetimes of CLR objects (of type `Activity`) that transiently represent such a program instance while it is in memory.

Scheduling

In general, when an activity executes, it quickly performs some work and then either reports its completion or (having established one or more bookmarks) yields and waits for stimulus. This pattern maps nicely to a conceptual model in which work items are queued and then dispatched, one at a time, each to a target activity.

This pattern, depicted in Figure 3.1, is generally known as **scheduling**, so the component of the WF runtime that encompasses this functionality is known as the **scheduler**. The scheduler dispatches work items one at a time (from a queue), in a first-in-first-out (FIFO) fashion. Additionally, because the WF runtime never intervenes in the processing of a work item that has been dispatched, the scheduler behavior is strictly nonpreemptive.

To distinguish the scheduler's internal queue of work items from WF program queues (which are explicitly created by activity execution logic), we will call the queue that holds scheduler work items the **scheduler work queue**. When its scheduler work queue is empty, a WF program instance is considered idle.

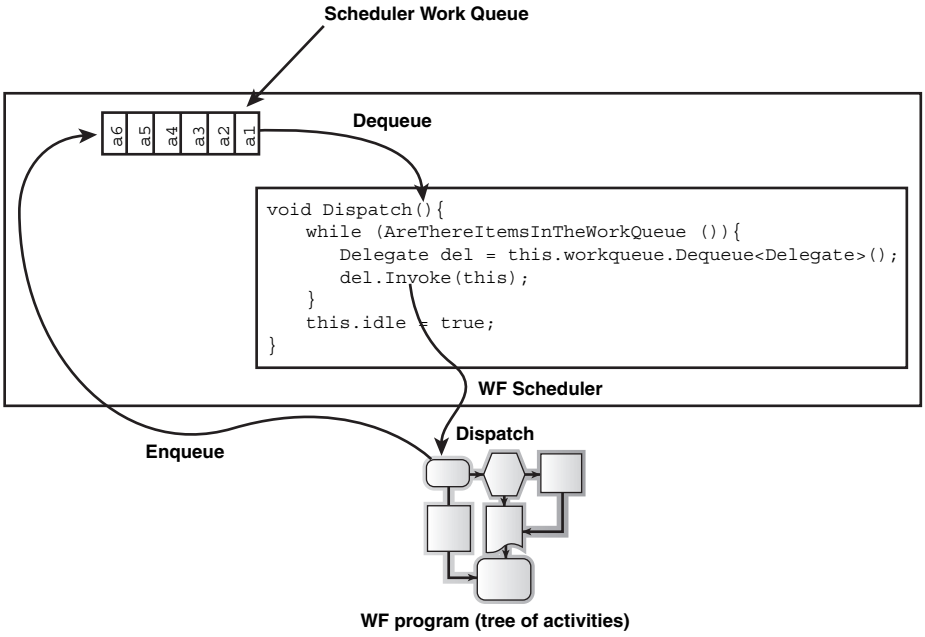

Figure 3.1 WF scheduler

Scheduler Work Items

The work items in the scheduler work queue are delegates. Each work item (delegate) corresponds to a method on an activity in the WF program instance. The activity method that is indicated by a work item (delegate) in the scheduler work queue is known as the **execution handler** of that work item.

Although there is no API to directly manipulate (or view) the scheduler work queue, certain actions taken by activities will cause work items to be enqueued. Delivery of input to a WF program queue can also cause work items to be enqueued.

A given activity's execution may include the invocation of any number of execution handlers. The state on which execution handlers operate is preserved across invocations of execution handlers of the same activity. This state is heap-allocated and is managed independently of the stack that is associated with the currently running CLR thread. The execution of activities across resumption points is stackless.

Scheduling of work items (delegates) is the mechanism by which methods on activities are invoked. This simple machinery drives activity, and WF program, execution. But in order to understand the rules about how and when work items are enqueued, we must understand the lifecycle of an activity, and that is our next topic.

Activity Automaton

The CLR virtualizes the instruction set of machine processors by describing its execution capabilities in terms of a hardware-agnostic instruction set, Microsoft Intermediate Language (MSIL). Programs compiled to MSIL are ultimately translated to machine-specific instructions, but virtualization allows language compilers to target only MSIL and not worry about various processor architectures.

In the WF programming model, the program statements used to build WF programs are classes that derive from `Activity` and `CompositeActivity`. Therefore, unlike MSIL, the "instruction set" supported by WF is not fixed. It is expected that many kinds of activities will be built and used in WF programs, while the WF runtime only relies upon the base classes. An activity developer can choose to implement anything—domain-specific or general-purpose—within the very general boundaries set by the WF runtime. Consequently, the WF runtime is freed from the actual semantics of specific activities.

The WF programming model does codify aspects of the interactions between the WF runtime and activities (such as the dispatch of execution handlers) in terms of an **activity automaton** (a finite state machine), which we will now explore in depth, with examples.

This chapter focuses solely on the normal execution of an activity. An activity that executes normally begins in the ***Initialized*** state, moves to the ***Executing*** state when its work begins, and moves to the ***Closed*** state when its work is completed. This is shown in Figure 3.2. The full activity automaton, shown in Figure 3.3, includes other states that will be discussed in Chapter 4, "Advanced Activity Execution."

Figure 3.2 Basic activity automaton

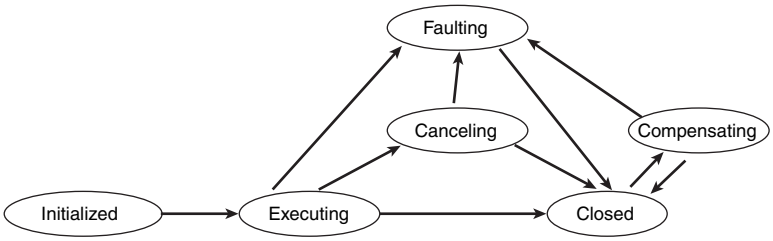

Figure 3.3 Complete activity automaton

The lifecycle of any activity in an executing WF program is captured by the states of the activity automaton and the transitions that exist between these states. Transitions from one state to another are brokered by the WF runtime to ensure the correctness of WF program execution.

Another way to view the activity automaton is as an abstract execution contract that exists between the WF runtime and any activity. It is the responsibility of the WF runtime to enforce that the execution of an activity strictly follows the transitions of the activity automaton. It is the responsibility of the activity to decide when certain transitions should occur. Figure 3.4 depicts the participation of both the WF scheduler and an activity in driving the automaton.

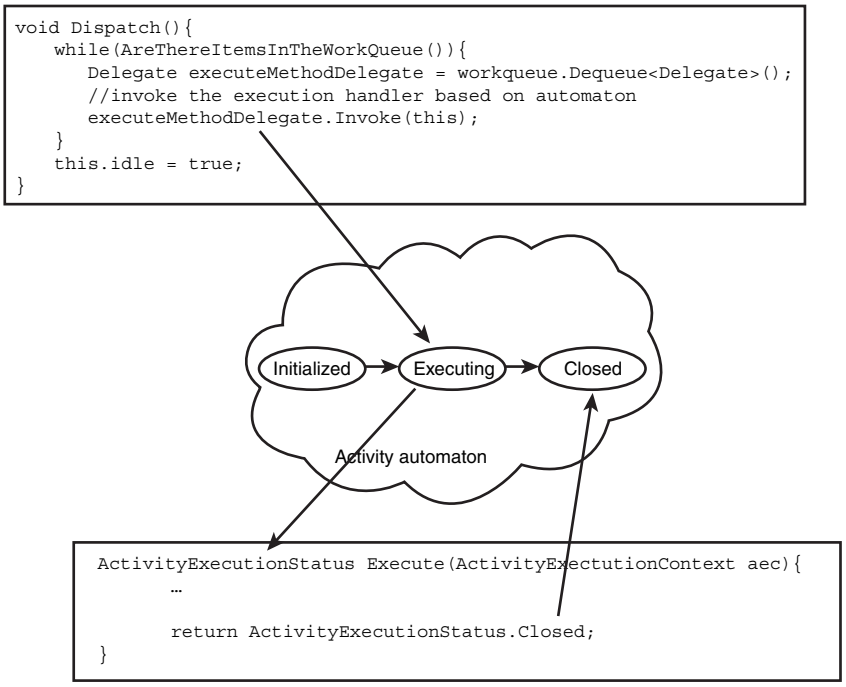

Figure 3.4 The dispatch of execution handlers is governed by the activity automaton.

Activity Execution Status and Result

The `WriteLine` activity shown in Listing 3.1 prints the value of its `Text` property to the console and then reports its completion.

Listing 3.1 `WriteLine` Activity

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

namespace EssentialWF.Activities
{
  public class WriteLine : Activity
  {
    public static readonly DependencyProperty TextProperty
      = DependencyProperty.Register("Text",
        typeof(string), typeof(WriteLine));
```

```
    public string Text
    {
      get { return (string) GetValue(TextProperty); }
      set { SetValue(TextProperty, value); }
    }

    protected override ActivityExecutionStatus Execute(
      ActivityExecutionContext context)
    {
      Console.WriteLine(Text);
      return ActivityExecutionStatus.Closed;
    }
  }
}
```

The execution logic of `WriteLine` is found in its override of the `Execute` method, which is inherited from `Activity`. The `Execute` method is the most important in a set of virtual methods defined on `Activity` that collectively constitute an activity's participation in the transitions of the activity automaton. All activities implement the `Execute` method; the other methods are more selectively overridden.

Listing 3.2 shows members defined by the `Activity` type that we will cover in this chapter and the next.

Listing 3.2 `Activity` Revisited

```
namespace System.Workflow.ComponentModel
{
  public class Activity : DependencyObject
  {
    protected virtual ActivityExecutionStatus Cancel(
      ActivityExecutionContext context);

    protected virtual ActivityExecutionStatus Execute(
      ActivityExecutionContext context);

    protected virtual ActivityExecutionStatus HandleFault(
      ActivityExecutionContext context, Exception fault);

    protected virtual void Initialize(
      IServiceProvider provider);

    protected virtual void Uninitialize(
      IServiceProvider provider);
```

```
    protected virtual void OnExecutionContextLoad(
      IServiceProvider provider);

    protected virtual void OnExecutionContextUnload(
      IServiceProvider provider);

    protected virtual void OnClosed(
      IServiceProvider provider);

    public ActivityExecutionResult ExecutionResult { get; }
    public ActivityExecutionStatus ExecutionStatus { get; }

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

By returning a value of `ActivityExecutionStatus.Closed` from its `Execute` method, the `WriteLine` activity indicates to the WF runtime that its work is done; as a result, the activity moves to the ***Closed*** state.

`Activity` defines a property called `ExecutionStatus`, whose value indicates the current state (in the activity automaton) of the activity. The type of `Activity.ExecutionStatus` is `ActivityExecutionStatus`, which is shown in Listing 3.3.

Listing 3.3 `ActivityExecutionStatus`

```
namespace System.Workflow.ComponentModel
{
  public enum ActivityExecutionStatus
  {
    Initialized,
    Executing,
    Canceling,
    Closed,
    Compensating,
    Faulting
  }
}
```

`Activity` also defines a property called `ExecutionResult`, whose value qualifies an execution status of `ExecutionStatus.Closed`, because that state can be entered from any of five other states. The type of `Activity.ExecutionResult` is `ActivityExecutionResult`, which is shown in Listing 3.4. An activity with an execution status other than ***Closed*** will always have an execution result of ***None***.

Listing 3.4 ActivityExecutionResult

```
namespace System.Workflow.ComponentModel
{
  public enum ActivityExecutionResult
  {
    None,
    Succeeded,
    Canceled,
    Compensated,
    Faulted,
    Uninitialized,
  }
}
```

The values of the `ExecutionStatus` and `ExecutionResult` properties are settable only by the WF runtime, which manages the lifecyle transitions of all activities.

You can determine the current execution status and execution result of an activity by getting the values of its `ExecutionStatus` and `ExecutionResult` properties:

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

public class MyActivity : Activity
{
  protected override ActivityExecutionStatus Execute(
    ActivityExecutionContext context)
  {
    System.Diagnostics.Debug.Assert(
      ActivityExecutionStatus.Executing == ExecutionStatus);

    System.Diagnostics.Debug.Assert(
      ActivityExecutionResult.None == ExecutionResult);

    ...
  }
}
```

The `ExecutionStatus` and `ExecutionResult` properties only have meaning at runtime for activities within a WF program instance.

Activity Execution Context

The `Execute` method has one parameter of type `ActivityExecutionContext`. This object represents the execution context for the currently executing activity.

The `ActivityExecutionContext` type (abbreviated AEC) is shown in Listing 3.5.

Listing 3.5 `ActivityExecutionContext`

```
namespace System.Workflow.ComponentModel
{
   public sealed class ActivityExecutionContext: IDisposable,
     IServiceProvider
  {
    public T GetService<T>();
    public object GetService(Type serviceType);

    public void CloseActivity();

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

AEC has several roles in the WF programming model. The simplest view is that AEC makes certain WF runtime functionality available to executing activities. A comprehensive treatment of AEC will be given in Chapter 4.

The WF runtime manages `ActivityExecutionContext` objects carefully. AEC has only internal constructors, so only the WF runtime creates objects of this type. Moreover, AEC implements `System.IDisposable`. An AEC object is disposed immediately after the return of the method call (such as `Activity.Execute`) in which it is a parameter; if you try to cache an AEC object, you will encounter an `ObjectDisposedException` exception when you access its properties and methods. Allowing AEC objects to be cached could easily lead to violation of the activity automaton:

```
public class MyActivity : Activity
{
  private ActivityExecutionContext cachedContext = null;

  protected override ActivityExecutionStatus Execute(
    ActivityExecutionContext context)
  {
    this.cachedContext = context;
    return ActivityExecutionStatus.Executing;
```

```
  }

  public void UseCachedContext()
  {
    // Next line will throw an ObjectDisposedException
    this.cachedContext.CloseActivity();
  }
}
```

Activity Services

`ActivityExecutionContext` has a role as a provider of services; these services are an activity's gateway to functionality that exists outside of the running WF program instance. AEC implements `System.IServiceProvider` and offers the required `GetService` method plus (for the sake of convenience) a typed `GetService<T>` wrapper over `GetService`. Using these methods, an activity can obtain services that are needed in order to complete its work.

In fact, AEC chains its service provider implementation to that of the WF runtime. This means that an activity can obtain custom services proffered by the application hosting the WF runtime, as shown in Figure 3.5.

Consider a `WriterService` that defines a `Write` method:

```
using System;

namespace EssentialWF.Activities
{
  public abstract class WriterService
  {
    public abstract void Write(string s);
  }
}
```

By defining the writer service abstractly (we could also have used an interface), activities that use the service are shielded from details of how the service is implemented. We can change our implementation of the service over time without affecting activity code.

Here is a simple derivative of `WriterService` that uses the console to print the string that is provided to the `Write` method:

```
using System;
using EssentialWF.Activities;
```

```
namespace EssentialWF.Services
{
  public class SimpleWriterService : WriterService
  {
    public override void Write(string s)
    {
      Console.WriteLine(s);
    }
  }
}
```

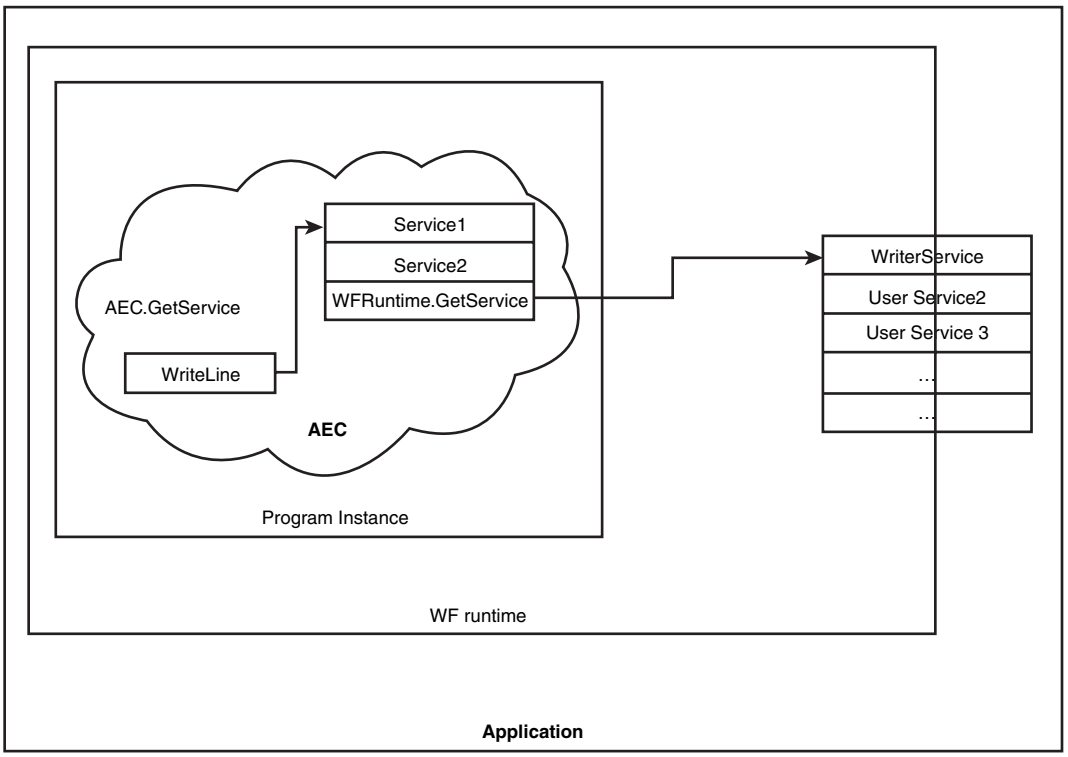

Figure 3.5 Chaining of services

A `SimpleWriterService` object can be added to the WF runtime, which acts as a container of services:

```
using (WorkflowRuntime runtime = new WorkflowRuntime())
{
  runtime.AddService(new SimpleWriterService());
```

```
    ...
  }
```

We can now change the execution logic of `WriteLine` to obtain a `WriterService` and call its `Write` method:

```
public class WriteLine : Activity
{
  // Text property elided for clarity...

  protected override ActivityExecutionStatus Execute(
    ActivityExecutionContext context)
  {
    WriterService writer = context.GetService<WriterService>();
    writer.Write(Text);

    return ActivityExecutionStatus.Closed;
  }
}
```

This change may seem like a small matter, but if `WriterService` is defined as an abstract class (or an interface), it can have multiple implementations. In this way, the application hosting the WF runtime can choose the appropriate writer service without affecting WF program instances that contain `WriteLine` activities (that rely only upon the definition of that service).

In Chapter 6, "Transactions," we will bring transactions into the picture and explore how services used by activities (and activities themselves) can partipate in the transactions that attend the execution of WF program instances.

Bookmarks Revisited

The simplest activities (in terms of execution logic) are like the `WriteLine` activity; they complete their work entirely within their `Execute` method. If all activities did this, you would not be able to build very interesting WF programs. Don't get us wrong; simple activities are useful, and indeed are essential to the definition of most WF programs. Typical duties for such activities include obtaining services and exchanging data with those services, and manipulating the state of the WF program instance.

Most real-world processes, however, reach points in time at which further computational progress cannot be made without stimulus (input) from an external entity. It may be that a WF program waits for a person to make a decision about which branch of execution logic should be taken. Or it may be that an activity delegates some computation to an external entity and then waits for the result of that computation to be returned asynchronously.

In order to understand the mechanics of how this kind of activity executes, we will begin by looking at a contrived example: an activity that delegates work to…itself. Consider the version of `WriteLine` that is shown in Listing 3.6.

Listing 3.6 `WriteLine` Activity That Uses a Bookmark

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

namespace EssentialWF.Activities
{
  public class WriteLine : Activity
  {
    // Text property elided for clarity...

    protected override ActivityExecutionStatus Execute(
      ActivityExecutionContext context)
    {
      base.Invoke(this.ContinueAt, EventArgs.Empty);
      return ActivityExecutionStatus.Executing;
    }

    void ContinueAt(object sender, EventArgs e)
    {
      ActivityExecutionContext context =
        sender as ActivityExecutionContext;

      WriterService writer = context.GetService<WriterService>();
      writer.Write(Text);

      context.CloseActivity();
    }
  }
}
```

Although the example is contrived, there are several things worth looking at here.

By calling `Invoke<T>` (a protected method defined by `Activity`), the `WriteLine` activity creates a bookmark and immediately resumes that bookmark. The bookmark's resumption point is the `WriteLine.ContinueAt` method, and the payload for the resumed bookmark is `EventArgs.Empty`.

The bookmark created by the call to `Invoke<T>` is managed internally by the WF runtime, and because the `Invoke<T>` method also resumes this bookmark, an item is enqueued in the scheduler work queue (corresponding to the `ContinueAt` method).

Because it creates a bookmark (and is awaiting resumption of that bookmark), the `WriteLine` activity can no longer report its completion at the end of the `Execute` method. Instead it returns a value of `ActivityExecutionStatus.Executing`, indicating that although `WriteLine` is yielding the CLR thread by returning from `Execute`, its work is not complete since there is a pending bookmark. The `WriteLine` activity remains in the ***Executing*** state and does not transition (yet) to ***Closed***.

When the scheduler dispatches the work item corresponding to the `ContinueAt` method, it passes an `ActivityExecutionContext` as the `sender` parameter. This allows the `WriteLine` to have access to its current execution context.

The `ContinueAt` method conforms to a standard .NET Framework event handler signature and therefore has a return type of `void`. Because of this, the WF runtime cannot use the return value of `ContinueAt` as the way of determining whether or not the activity should remain in the ***Executing*** state or transition to the ***Closed*** state. The `CloseActivity` method provided by `ActivityExecutionContext` can be used instead. If this method is called, the currently executing activity moves to the ***Closed*** state; if the method is not called, there is no change in the state of the activity. Because `ContinueAt` calls `CloseActivity`, the `WriteLine` activity moves to the ***Closed*** state.

The version of the `WriteLine` activity that uses `Invoke<T>`, though contrived, is still illustrative of the general pattern that you will need to use in many of the activities you develop. Although it is possible for an activity to complete its work within the `Execute` method (as with the version of `WriteLine` that returns `ActivityExecutionStatus.Closed` from its `Execute` method), this is a special case. Just as subroutines are a special, simple case accommodated by the richer concept of a coroutine, activities whose execution logic is embodied in a single `Execute`

method are a special, simple form of episodic computation, in which there is always exactly one episode.

WF Program Execution

Now that we understand the basics of how to write activity execution logic, we can take a closer look at the execution mechanics of a WF program. We will start with a WF program that contains just one activity:

```
<WriteLine Text="hello, world" xmlns="http://EssentialWF/Activities" />
```

Running this program results in the expected output:

```
hello, world
```

In Chapter 2, "WF Programs," we briefly looked at the code that is required to host the WF runtime and run WF programs. We will return to the host-facing side of the WF runtime in Chapter 5, "Applications." For now, it is enough to know the basics: First, the `WorkflowRuntime.CreateWorkflow` method returns a `WorkflowInstance` representing a newly created instance of a WF program; second, the `WorkflowInstance.Start` method tells the WF runtime to begin the execution of that WF program instance.

The call to `WorkflowRuntime.CreateWorkflow` prepares a scheduler (and the accompanying scheduler work queue) for the new WF program instance. When this method returns, all activities in the WF program are in the ***Initialized*** state.

The call to `WorkflowInstance.Start` enqueues one item in the the scheduler work queue—a delegate corresponding to the `Execute` method of the root activity of the WF program. The root activity—in our example, the `WriteLine`—is now in the ***Executing*** state, even though the `Execute` method has not actually been called (the work item has not yet been dispatched). The scheduler work queue is shown in Figure 3.6.

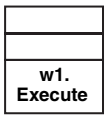

Figure 3.6 Scheduler work queue after WorkflowInstance.Start

Let's assume that we are using the version of `WriteLine` that doesn't call `Invoke<T>`.

When the `Execute` method returns a value of `ActivityExecutionStatus.Closed`, the `WriteLine` activity moves to the ***Closed*** state. In this case, the WF runtime recognizes that the program instance is complete since the root activity in the program instance is complete.

The asynchronous version of `WriteLine` is only slightly more complex. The call to `Invoke<T>` within `Execute` will enqueue a work item in the scheduler work queue (corresponding to the resumption of the internally created bookmark).

Thus, when the `Execute` method (of the version of `WriteLine` that does call `Invoke<T>`) returns, the activity remains in the ***Executing*** state and the scheduler work queue looks as it is shown in Figure 3.7.

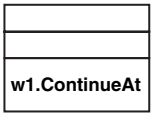

Figure 3.7 Scheduler work queue after WriteLine.Execute

When the `WriteLine.ContinueAt` method returns, the `WriteLine` activity moves to the ***Closed*** state and the program instance completes.

WF Program Queues

Any activity that requires input from an external entity must figure out a way to (a) let that external entity know that it requires input, and (b) receive notification when the input is available. This simple pattern is at the heart of episodic computation, and it is supported in a first-class way by the WF runtime. The plain requirement is that an activity must be able to receive input even if the WF program instance in which it exists is idle and sitting in persistent storage like a SQL Server database table. When the input arrives, the WF program instance must be reactivated and its execution resumed (at the appropriate bookmark).

In Chapter 2, we developed an activity called `ReadLine` (shown again in Listing 3.7), which waits for a string to arrive from an external entity. If you understand how this activity is built and how it executes, you will have the right basis for

understanding and creating higher level communication patterns that are used in WF programs. All such patterns are built on top of the same notion of bookmarks.[1]

Listing 3.7 `ReadLine` Activity

```
using System;
using System.Workflow.ComponentModel;
using System.Workflow.Runtime;

namespace EssentialWF.Activities
{
  public class ReadLine : Activity
  {
    private string text;
    public string Text
    {
      get { return text; }
    }

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

      WorkflowQueue queue =
        qService.CreateWorkflowQueue(this.Name, true);
      queue.QueueItemAvailable += this.ContinueAt;

      return ActivityExecutionStatus.Executing;
    }

    void ContinueAt(object sender, QueueEventArgs e)
    {
      ActivityExecutionContext context =
        sender as ActivityExecutionContext;

      WorkflowQueuingService qService =
        context.GetService<WorkflowQueuingService>();

      WorkflowQueue queue = qService.GetWorkflowQueue(this.Name);
      text = (string) queue.Dequeue();
      qService.DeleteWorkflowQueue(this.Name);
```

[1] Although various communication technologies (such as WCF or ASMX) can be layered upon WF, they all must use the `WorkflowQueuingService` to robustly deliver data to passivated WF program instances.

```
      context.CloseActivity();
    }
  }
}
```

The execution logic of the `ReadLine` activity uses a **WF program queue**. A WF program queue is essentially a named location (a bookmark) where an activity can receive data, even if the WF program instance in which the activity exists is not in memory. A WF program queue is not the same as the WF program instance's scheduler queue, which is managed by the WF runtime. Think of a WF program queue as the data structure in which an explicitly created bookmark holds its payload (to be delivered upon the resumption of the bookmark). It is an addressable location where external entities can deliver data.

The `Execute` method of `ReadLine` obtains the `WorkflowQueuingService` from its `ActivityExecutionContext`. The `WorkflowQueuingService` is asked to create a WF program queue with a name that is the same as that of the activity (`this.Name`). The name of a WF program queue can be any `IComparable` object; usually a `string` will suffice. We are choosing a simple queue naming convention here, but other schemes are possible. Regardless, the external code that provides input to a WF program instance must know the name of the appropriate WF program queue.

The `WorkflowQueuingService` type is shown in Listing 3.8.

Listing 3.8 `WorkflowQueuingService`

```
namespace System.Workflow.Runtime
{
  public class WorkflowQueuingService
  {
    public WorkflowQueue CreateWorkflowQueue(IComparable queueName,
      bool transactional);
    public bool Exists(IComparable queueName);
    public WorkflowQueue GetWorkflowQueue(IComparable queueName);
    public void DeleteWorkflowQueue(IComparable queueName);

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

The same WF program queue name may be used in more than one WF program instance. This just means that if we write a WF program containing a `ReadLine` activity named "r1", we can execute any number of instances of this WF program without any problems. Each instance will create a separate WF program queue with the name "r1". Because data is always enqueued to a specific WF program instance (via `WorkflowInstance.EnqueueItem`), there is no conflict or ambiguity. Another way of stating this is that WF program queues are not shared across WF program instances. This allows us to think of the logical address of a WF program queue as the `WorkflowInstance.InstanceId` identifying the WF program instance that owns the WF program queue, plus the WF program queue name.

A WF program queue acts as a conduit for communication between external entities and an activity in a WF program instance. Code outside of the WF program instance can deposit data into a WF program queue using the `EnqueueItem` method defined on the `WorkflowInstance` class. An activity (and, by extension, a WF program) can create as many distinct WF program queues as it requires.

The `CreateWorkflowQueue` method returns a `WorkflowQueue` object that represents the WF program queue. The `WorkflowQueue` type is shown in Listing 3.9.

Listing 3.9 `WorkflowQueue`

```
namespace System.Workflow.Runtime
{
  public class WorkflowQueue
  {
    public IComparable QueueName { get; }
    public int Count { get; }

    public object Dequeue();
    public object Peek();

    public event EventHandler<QueueEventArgs>
      QueueItemAvailable;

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

The `QueueItemAvailable` event is raised when an item is enqueued into the WF program queue. Under the covers, this is just a bookmark (disguised using C# event syntax).

The `QueueItemAvailable` event is also raised if, when an activity subscribes to this event, there are already (previously enqueued) items present in the WF program queue. This permits a decoupling of the delivery of data to a bookmark and the resumption of that bookmark.

Here is a simple WF program that contains only a single `ReadLine` activity:

```
<ReadLine x:Name="r1" xmlns="http://EssentialWF/Activities"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" />
```

If we save this WF program as a file called "Read.xoml", we can execute it using the console application of Listing 3.10, which hosts the WF runtime and delivers data to the `ReadLine` activity via the WF program queue.

Listing 3.10 A Console Application That Delivers Data to a `ReadLine` Activity

```
using System;
using System.Workflow.ComponentModel.Compiler;
using System.Workflow.Runtime;
using System.Xml;

class Program
{
  static void Main()
  {
    using (WorkflowRuntime runtime = new WorkflowRuntime())
    {
      TypeProvider tp = new TypeProvider(null);
      tp.AddAssemblyReference("EssentialWF.dll");
      runtime.AddService(tp);

      runtime.StartRuntime();

      runtime.WorkflowIdled += delegate(object sender,
        WorkflowEventArgs e)
      {
        Console.WriteLine("WF program instance " +
          e.WorkflowInstance.InstanceId + " is idle");
      };

      runtime.WorkflowCompleted += delegate(object sender,
        WorkflowCompletedEventArgs e)
      {
        Console.WriteLine("WF program instance " +
          e.WorkflowInstance.InstanceId + " completed");
      };
```

```
      WorkflowInstance instance = null;
      using (XmlTextReader reader = new XmlTextReader("Read.xoml"))
      {
        instance = runtime.CreateWorkflow(reader);
        instance.Start();
      }

      string text = Console.ReadLine();
      instance.EnqueueItem("r1", text, null, null);

      // Prevent Main from exiting before
      // the WF program instance completes
      Console.ReadLine();

      runtime.StopRuntime();
    }
  }
}
```

The console application calls `WorkflowRuntime.CreateWorkflow`, which loads the WF program from XAML. It then calls `WorkflowInstance.Start`, which causes the `Execute` method of `ReadLine`—the root activity in the WF program—to be scheduled.

The console application then waits for the user to enter text at the console. Meanwhile, the WF runtime begins the execution of the WF program instance on a thread that is different than the thread on which `Main` is running. The `ReadLine` activity has its `Execute` method invoked. The `ReadLine` activity creates its WF program queue and then waits for data to arrive there. Because there are no other items in the scheduler work queue, the WF program instance is idle.

The console application subscribes for the `WorkflowRuntime.WorkflowIdled` event and, when this event is raised by the WF runtime, writes the `InstanceId` of the WF program instance to the console:

```
WF program instance 631855e5-1958-4ce7-a29a-dc6f8e2a9238 is idle
```

When a line of text is read, the console application calls `EnqueueItem`, passing the text it received from the console as payload associated with the resumption of the bookmark.

The implementation of `WorkflowInstance.EnqueueItem` enqueues (in the scheduler work queue) work items for all activities that are subscribed to this WF program queue's `QueueItemAvailable` event. This is depicted in Figure 3.8.

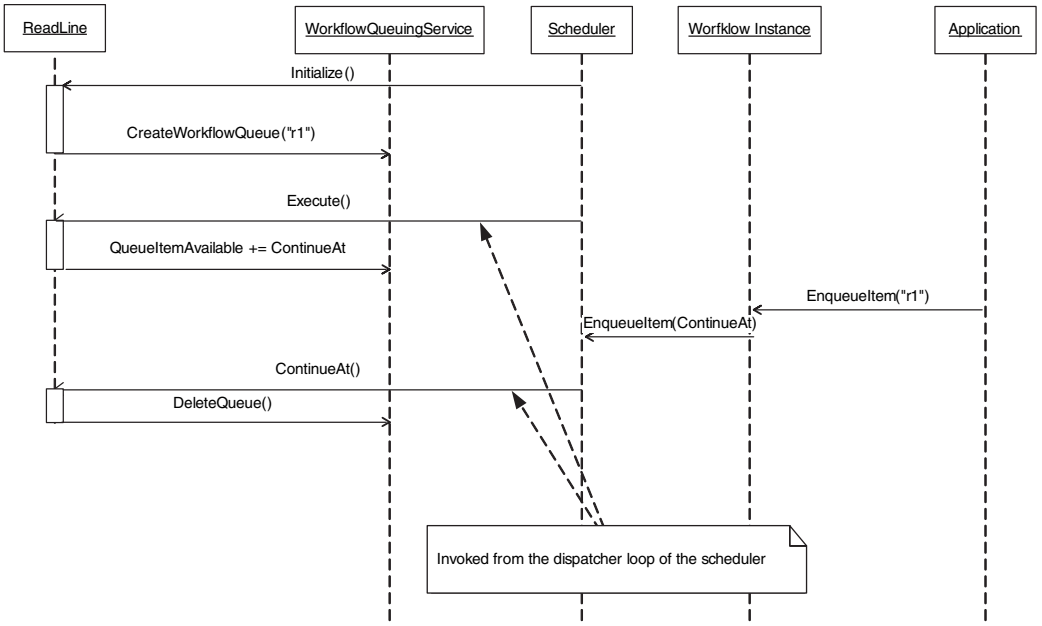

Figure 3.8 Enqueuing data to a WF program queue

In our example, the `ReadLine` activity's callback is called `ContinueAt`. This delegate will be scheduled as a work item and dispatched by the scheduler; if the idle WF program instance had been passivated (not shown in this example), the WF runtime would automatically bring it back into memory.

The `ReadLine` activity will set its `Text` property with the string obtained from the `Dequeue` operation on its WF program queue. In the example, we are doing no error checking to ensure that the object is indeed of type `string`. The `ContinueAt` method informs the WF runtime that it is complete by calling `CloseActivity`. The WF program instance, because it only contains the `ReadLine`, also completes. The console application, which subscribed to the `WorkflowRuntime.WorkflowCompleted` event, prints this fact to the console.

```
WF Program instance 631855e5-1958-4ce7-a29a-dc6f8e2a9238 completed
```

If the console application tries to enqueue data to a WF program queue that does not exist, the `EnqueueItem` method will throw an `InvalidOperationException` indicating that the WF program queue could not be found. In our implementation

of `ReadLine`, the WF program queue is not created until the `ReadLine` activity begins executing. Thus, the following lines of code are problematic:

```
WorkflowInstance instance = runtime.CreateWorkflow(...);
instance.EnqueueItem("r1", "hello", null, null);
```

The preceding code omits the call to `WorkflowInstance.Start`, and because of this the WF program queue named "r1" does not yet exist. In other words, the implementation of `ReadLine` requires that the application doesn't enqueue the data until after the `ReadLine` activity starts to execute. Even the code in the console application of Listing 3.9 presents a race condition because the execution of the WF program instance occurs on a different thread than the execution of the console application. We may be able to work around this race condition quite easily in our contrived example where the WF program is just a single `ReadLine` activity. But in a larger WF program, with many activities managing WF program queues, and executing at different times, this is a lot trickier.

One of the ways to mitigate this problem is to allow activities to create WF program queues during the creation of a WF program instance. This will ensure that, after the call to `WorkflowRuntime.CreateWorkflow`, the WF program instance can immediately receive data (even if it cannot yet process it, which will only begin once `WorkflowInstance.Start` is called). In a later section, we will change `ReadLine` to do exactly this.

Timers

Another example of an activity that cannot complete its execution logic entirely within the `Execute` method is a `Wait` activity that simply waits for a specified amount of time to elapse before completing. The `Wait` activity is shown in Listing 3.11.

Listing 3.11 `Wait` ***Activity***

```
using System;
using System.Workflow.ComponentModel;
using System.Workflow.Runtime;

namespace EssentialWF.Activities
{
  public class Wait : Activity
```

```
  {
    private Guid timerId;

    public static readonly DependencyProperty DurationProperty
      = DependencyProperty.Register("Duration",
        typeof(TimeSpan), typeof(Wait));

    public TimeSpan Duration
    {
      get { return (TimeSpan) GetValue(DurationProperty); }
      set { SetValue(DurationProperty, value); }
    }

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

      timerId = Guid.NewGuid();

      WorkflowQueue queue = qService.CreateWorkflowQueue(
        timerId, true);
      queue.QueueItemAvailable += this.ContinueAt;

      TimerService timerService = context.GetService<TimerService>();
      timerService.SetTimer(timerId, Duration);

      return ActivityExecutionStatus.Executing;
    }

    void ContinueAt(object sender, QueueEventArgs e)
    {
      ActivityExecutionContext context =
        sender as ActivityExecutionContext;

      WorkflowQueuingService qService =
        context.GetService<WorkflowQueuingService>();

      WorkflowQueue queue = qService.GetWorkflowQueue(timerId);
      qService.DeleteWorkflowQueue(timerId);

      context.CloseActivity();
    }
  }
}
```

Listing 3.11 shows the basic implementation of a `Wait` activity that depends upon an implementation of a `TimerService` (see Listing 3.12) for the actual management of the timer. The `Wait` activity, in its `Execute` method, creates a WF program queue providing the bookmark resumption point (`ContinueAt`) and calls `TimerService.SetTimer`, passing a unique identifier representing the timer. The `TimerService` is responsible for managing the actual timers. When the timer is triggered, the timer service resumes the bookmark by enqueuing data in the WF program queue created by the `Wait` activity. When the `ContinueAt` method is invoked by the scheduler (with the AEC as the `sender` argument), the `Wait` activity deletes the WF program queue and transitions to the Closed state.

The `TimerService` defines a `SetTimer` method that allows the activity to specify the duration of the timer as a `TimeSpan`, along with the name of the WF program queue that the `TimerService` will use to deliver a notification using `WorkflowInstance.EnqueueItem` (with a null payload) when the specified amount of time has elapsed.

Listing 3.12 `TimerService` Used by the `Wait` Activity

```
using System;
using System.Workflow.ComponentModel;
using System.Workflow.Runtime;

namespace EssentialWF.Activities
{
   public abstract class TimerService
   {
     public abstract void SetTimer(Guid timerId, TimeSpan duration);
     public abstract void CancelTimer(Guid timerId);
   }
}
```

A simple implementation of the timer service is shown in Listing 3.13.

Listing 3.13 *Implementation of a* `TimerService`

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Workflow.ComponentModel;
using System.Workflow.Runtime;
using EssentialWF.Activities;
```

```
namespace EssentialWF.Services
{
  public class SimpleTimerService : TimerService
  {
    WorkflowRuntime runtime;
    Dictionary<Guid, Timer> timers = new Dictionary<Guid, Timer>();

    public SimpleTimerService(WorkflowRuntime runtime)
    {
      this.runtime = runtime;
    }

    public override void SetTimer(Guid timerId, TimeSpan duration)
    {
      Guid instanceId = WorkflowEnvironment.WorkflowInstanceId;
      Timer timer = new Timer(delegate(object o)
        {
          WorkflowInstance instance = runtime.GetWorkflow(instanceId);
          instance.EnqueueItem(timerId, null, null, null);
        }, timerId, duration, new TimeSpan(Timeout.Infinite));

      timers.Add(timerId, timer);
    }

    public override void CancelTimer(Guid timerId)
    {
      ((IDisposable)timers[timerId]).Dispose();
      timers.Remove(timerId);
    }
  }
}
```

The `SimpleTimerService` mantains a set of `System.Threading.Timer` objects. The `timerId` that is passed as a parameter to the `SetTimer` method serves as the name of the WF program queue created by the `Wait` activity. When a timer fires, the callback (written as an anonymous method) enqueues a (null) item into the appropriate WF program queue, and the `Wait` activity resumes its execution.

In Chapter 6 we will discuss transactions, and we will see how transactional services (such as a durable timer service) can be implemented. Because we have followed the practice of making the `Wait` activity dependent only on the abstract definition of the timer service, we can change the implementation of the timer service without affecting our activities and WF programs.

As mentioned earlier, the WF runtime is a container of services. Custom services that are added to the WF runtime can be obtained by executing activities. An implementation of a `TimerService` can be added to the WF runtime like so:

```
using (WorkflowRuntime runtime = new WorkflowRuntime())
{
  runtime.AddService(new SimpleTimerService(runtime));
  ...
}
```

Executing the `Wait` activity within a simple WF program will cause the program to pause (and potentially passivate) and later, when the timeout occurs, resume the execution. The following program will start and then pause for 5 seconds, and finally resume its execution and complete:

```
<Wait Duration="00:00:05" xmlns="http://EssentialWF/Activities" />
```

Our reason for introducing the `Wait` activity is to illustrate a general pattern.

There is nothing at all special about timers. The `Wait` activity makes a request to a service on which it depends, and indicates to the service where (to which WF program queue) the result of the requested work should be delivered. The service takes some amount of time to complete the requested work. When the work is done, the service returns the result of the work to the activity via the WF program queue.

This bookmarking pattern is the basis for developing WF programs that are "coordinators of work" that is performed outside their boundaries.

Activity Initialization and Uninitialization

In the activity automaton, ***Initialized*** is the start state in which all activities begin their lifecycle. When the `WorkflowRuntime.CreateWorkflow` method returns, all activities in the newly created WF program instance are in the ***Initialized*** state.

Within the implementation of `CreateWorkflow`, the WF runtime calls the `Initialize` method of the root activity in the WF program. There are other interesting details related to the creation of new WF program instances, and they will be covered in Chapter 5; here we will focus only on activity initialization.

Activities can use the `Initialize` method to perform whatever initialization is necessary when a WF program instance is created. Custom services added to the WF runtime (and also the `WorkflowQueuingService`) can be obtained by the activity via the `IServiceProvider` that is passed as a parameter to `Initialize`. `ActivityExecutionContext` is not available because the activity (indeed, the WF program) has not yet begun its execution.

The `CompositeActivity` class overrides `Initialize` and in its implementation invokes the `Initialize` method of all enabled child activities. If you develop a composite activity, or indeed any activity that requires initialization logic, you should always call `base.Initialize` within your implementation of the `Initialize` method to ensure proper initialization of the WF program instance.

The WF runtime's scheduler machinery is not used during initialization to dispatch the calls to `Initialize`. It would be overkill to do so because the WF program instance is not yet running. Because invocation of `Initialize` is synchronous, the WF runtime can guarantee that when the `WorkflowRuntime.CreateWorkflow` method returns, the WF program instance is fully initialized and ready for execution.

If an exception is thrown from any activity's `Initialize` method, the initialization of the WF program instance fails, and the `WorkflowRuntime.CreateWorkflow` method will throw an exception indicating that this has occurred.

So, what can an activity do in its `Initialize` method? `Initialize` carries one parameter of type `System.IServiceProvider`. No execution context exists at this time for the activity, so it is not correct for the WF runtime to provide AEC. Still, the `IServiceProvider` of `Initialize` does the same service chaining that AEC does. Any custom services that you add to the `WorkflowRuntime` are proffered by this service provider so that an activity may do whatever resource initialization is required. The `WorkflowQueuingService` is available too, so that WF program queues may be created.

To summarize, the ***Initialized*** state is the start state of the activity automaton. Activities in this state have not started their execution, and can be said to be in a latent form, but do get a chance to perform initialization logic in their `Initialize` method.

Listing 3.14 updates the `ReadLine` activity so that it creates its WF program queue within its `Initialize` method.

Listing 3.14 The `ReadLine` Activity with Initialization Logic

```
using System;
using System.Workflow.ComponentModel;
using System.Workflow.Runtime;

namespace EssentialWF.Activities
{
  public class ReadLine : Activity
  {
    private string text;
    public string Text
    {
      get { return this.text; }
    }

    protected override void Initialize(
      IServiceProvider provider)
    {
      WorkflowQueuingService qService =
        (WorkflowQueuingService) provider.GetService(
           typeof(WorkflowQueuingService));

      if (!qService.Exists(this.Name))
        qService.CreateWorkflowQueue(this.Name, true);
    }

    protected override ActivityExecutionStatus Execute(
      ActivityExecutionContext context) {

      WorkflowQueuingService qService =
        context.GetService<WorkflowQueuingService>();

      WorkflowQueue queue = qService.GetWorkflowQueue(Name);
      if (queue.Count > 0)
      {
        this.text = (string) queue.Dequeue();
        return ActivityExecutionStatus.Closed;
      }

      queue.QueueItemAvailable += this.ContinueAt;
      return ActivityExecutionStatus.Executing;
    }

    void ContinueAt(object sender, QueueEventArgs e)
    {
      ActivityExecutionContext context =
        sender as ActivityExecutionContext;
```

```
      WorkflowQueuingService qService =
        context.GetService<WorkflowQueuingService>();

      WorkflowQueue queue = qService.GetWorkflowQueue(Name);
      this.text = (string) queue.Dequeue();
      context.CloseActivity();
    }

    protected override void Uninitialize(IServiceProvider provider)
    {
      WorkflowQueuingService qService =
        (WorkflowQueuingService) provider.GetService(
           typeof(WorkflowQueuingService));

      if (qService.Exists(this.Name))
        qService.DeleteWorkflowQueue(this.Name);
    }
  }
}
```

The implementation of `Execute` accounts for the fact that by the time the activity executes, there may already be an item in its WF program queue. If an item is indeed available, there is no need to subscribe to the `QueueItemAvailable` event. The `ReadLine` activity also contains an implementation of the `Uninitialize` method, in which the WF program queue is deleted.

The `Uninitialize` method is the logical counterpart of the `Initialize` method.

`Uninitialize` is called (synchronously, not via a work item in the scheduler work queue) as the final part of an activity's transition to the ***Closed*** state from the ***Executing*** state. It is also called when it is determined by the WF runtime that an activity in the ***Initialized*** state will never be executed. The latter case occurs when the parent of an activity transitions to the ***Closed*** state without having requested the execution of that child activity.

Activities cannot assume that they will always be executed, just as the program statements in all but one branch of a C# `if` statement will be passed over. Any resources created in an activity's `Initialize` method should therefore be cleaned up in its `Uninitialize` method.

As part of an activity's transition to the ***Closed*** state (and just prior to the invocation of `Uninitialize`), the WF runtime synchronously invokes the `OnClosed` method that is defined by `Activity`. In this method, activities can clean up the resources they allocated during their execution (as opposed to during their initialization).

You might wonder why `OnClosed` exists when we also have `Uninitialize`. The simple answer is that `Uninitialize` should clean up resources allocated in `Initialize`, whereas the purpose of `OnClosed` is to clean up resources allocated during the execution of the activity. An executing activity can transition to the ***Closed*** state from several different states (which will be discussed more in the next chapter), and the `OnClosed` method will be called in each of these cases.

To summarize, when we execute a `ReadLine` activity, `ReadLine` has its methods invoked in the following order:

- `Initialize`
- `Execute`
- `ContinueAt`
- `OnClose`
- `Uninitialize`

If a `ReadLine` activity is present in a WF program, but never executes, it will only have its `Initialize` and `Uninitialize` methods called.

Activities as CLR Objects

Because `Activity` implements `System.ComponentModel.IComponent`, which extends `System.IDisposable`, activities are given yet another opportunity to perform cleanup of resources. The `IDisposable.Dispose` method, however (like an activity's constructor), is a practicality necessitated by the fact that a WF program instance is transiently realized as a set of CLR objects when that program instance is in memory. These objects, like any objects, are created and destroyed subject to the rules of the CLR. However, transitions in the CLR object lifecycle are logically unrelated to the execution lifecycle of the WF program instance (and the activities within it). In other words, the calling of the `Activity.Dispose` method reflects the passivation cycles of a WF program instance—every time a WF program instance is passivated, the activity objects that represent the program instance while it is in memory are disposed because they no longer represent the (passivated) program instance.

The WF runtime will call the `Dispose` method on the CLR object representing an activity every time the WF program instance containing the activity is passivated. In contrast, `Initialize` and `Uninitialize` are called exactly once during the logical lifetime of an activity, which can span any number of passivation cycles. In contrast, `Dispose` may be invoked multiple times for an activity during its lifetime.

It is recommended that activities not perform any resource management in their object constructors. CLR objects that transiently represent an activity may be constructed and disposed multiple times during the course of the activity's execution lifetime. The constructor of an activity may be called multiple times even during the creation of a single program instance (or reactivation of an instance). It is crucial to understand that because an activity is an intrinsically resumable entity, its logical lifespan is governed by the activity automaton and not by the lifetime of any CLR object.

In order to provide well-defined points for resource allocation and cleanup, `Activity` defines two additional methods, `OnExecutionContextLoad` and `OnExecutionContextUnload`, which bracket the lifetime of a CLR object representing an activity in a WF instance. You can rely upon the WF runtime to call `OnExecutionContextLoad` during the creation (or reactivation) and `OnExecutionContextUnload` during the passivation of a WF instance. `OnExecutionContextUnload` is essentially just like `Dispose` except that it accepts an `IServiceProvider` as a parameter and therefore has access to runtime services.

`Dispose`, `OnExecutionContextLoad`, and `OnExecutionContextUnload` are side effects of the fact that the WF runtime is layered on top of the CLR, and are related to the management of CLR objects which transiently represent a WF program instance. In contrast, `Initialize`, `Uninitialize`, and `OnClose` are related to the lifetime of the activity as described by the activity automaton. It is crucial to understand this difference between CLR programs and WF programs. From the perspective of the CLR, a CLR program instance is defined by its in-memory existence and lifetime. From the point of view of the WF runtime, a WF program instance is defined on an altogether different plane, and in fact can spend most of its lifetime in persistent storage. Because a WF program instance may passivate and reactivate many times (perhaps on different machines), objects that represent the activities in that instance in memory might need to be constructed and disposed of many times before the WF program instance completes (see Figure 3.9).

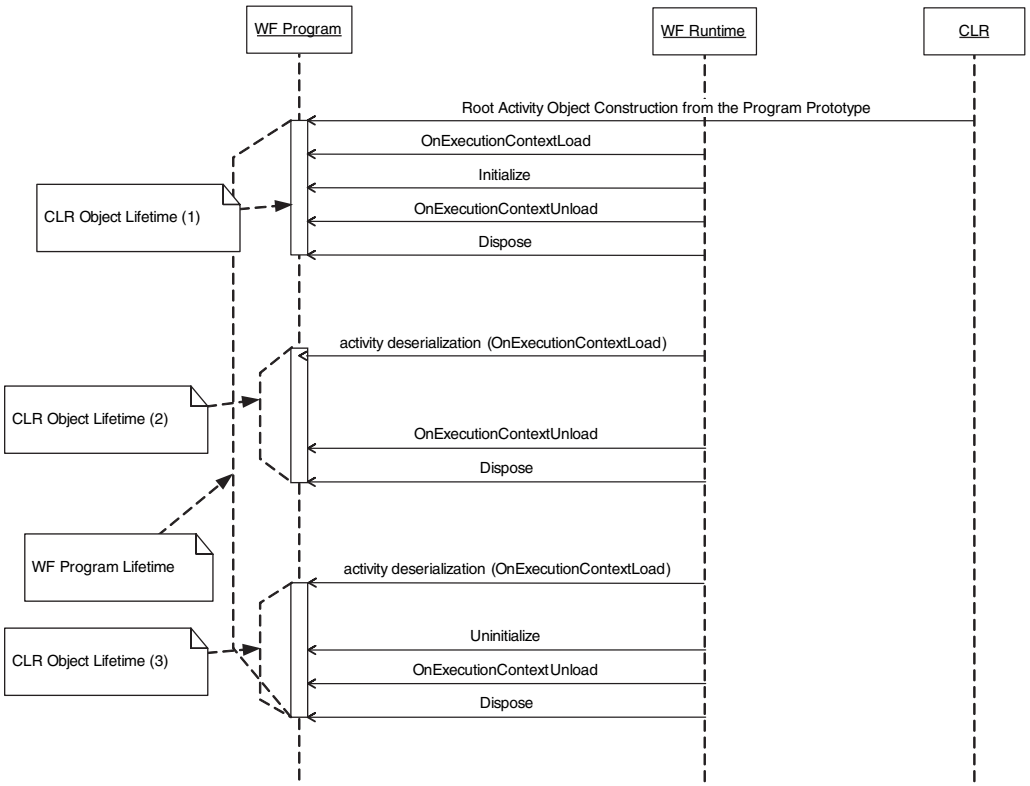

Figure 3.9 Lifecycle of a WF program instance

Many activities require only an empty constructor and `Dispose` method, but it is important nonetheless to know when and why they will be called.

Composite Activity Execution

Enough with WF programs that are only a single activity! It's time to develop some composite activities and then declare and run some more interesting WF programs.

We will begin with the `Sequence` activity of Chapter 2, shown again here:

```
public class Sequence : CompositeActivity
{
  protected override ActivityExecutionStatus Execute(
```

```
  ActivityExecutionContext context)
{
  if (this.EnabledActivities.Count == 0)
    return ActivityExecutionStatus.Closed;

  Activity child = this.EnabledActivities[0];
  child.Closed += this.ContinueAt;
  context.ExecuteActivity(child);

  return ActivityExecutionStatus.Executing;
}

void ContinueAt(object sender,
  ActivityExecutionStatusChangedEventArgs e)
{
  ActivityExecutionContext context =
    sender as ActivityExecutionContext;

  e.Activity.Closed -= this.ContinueAt;
  int index = this.EnabledActivities.IndexOf(e.Activity);

  if ((index + 1) == this.EnabledActivities.Count)
    context.CloseActivity();

  else
  {
    Activity child = this.EnabledActivities[index + 1];
    child.Closed += this.ContinueAt;
    context.ExecuteActivity(child);
  }
}
```

The job of the `Sequence` activity is to emulate a C# `{ }` statement block, and execute its child activities one by one. Only when the final child activity of a `Sequence` finishes can the `Sequence` report that it is complete.

The `Execute` method of `Sequence` first checks to see if there are any child activities at all. If none are present, the method returns `ActivityExecutionStatus.Closed`. The `Sequence` is done because it has nothing to do. It is like an empty C# statement block. If one or more child activities are present, though, the first child activity needs to be scheduled for execution. In order to do this, two lines of code are necessary:

```
child.Closed += ContinueAt;
context.ExecuteActivity(child);
```

These two statements constitute a very simple bookmarking pattern that you will encounter repeatedly in composite activity implementations. The subscripton to the `Closed` event of the child activity sets up a bookmark that is managed internally by the WF runtime. The `Activity.Closed` event is merely syntactic sugar on top of the bookmark management infrastructure. The += results in the creation of a bookmark, and the dispatch of the `Closed` event (the resumption of the bookmark), is brokered via the scheduler.

The invocation of `ActivityExecutionContext.ExecuteActivity` requests that the indicated child activity be scheduled for execution. Specifically, the `Execute` method of the child activity is added as a work item in the scheduler work queue.

In order to enforce the activity automaton, the WF runtime will throw an exception from within `ExecuteActivity` if the child activity is not in the ***Initialized*** state. If the call to `ExecuteActivity` succeeds, an item is added to the scheduler work queue, representing the invocation of the child activity's `Execute` method. A successful call to `ExecuteActivity` also immediately places the child activity in the ***Executing*** state.

The `Sequence` activity's code that schedules the execution of its first child activity and subscribes for this child activity's `Closed` event is analogous to the `ReadLine` activity's logic that creates a WF program queue and subscribes to that queue's `QueueItemAvailable` event. In both cases, the activity is dependent upon some work, outside of its control, and can proceed no further until it is notified that this work has been completed. The code is somewhat different, but the bookmarking pattern is exactly the same.

Of course, for a composite activity like `Sequence`, the pattern must be repeated until all child activities have completed their execution. This is achieved in the `ContinueAt` method, which is scheduled for execution when the currently executing child activity moves to the ***Closed*** state. When it receives notification that a child activity has completed its execution, `Sequence` first removes its subscription for that child activity's `Closed` event. If the child activity that just completed is the last child activity in the `Sequence`, the `Sequence` reports its own completion. Otherwise, the bookmarking pattern is repeated for the next child activity.

There are a couple of crucial aspects to the WF runtime's role as the enforcer of state transitions. If the `Sequence` activity tries to report its completion while a child activity is in the ***Executing*** state, this transition will not be allowed. This fact is the

cornerstone of the WF runtime's *composition-related enforcement* (and is not implied by the activity automaton).

The corollary to this rule is that only an activity's parent is allowed to request that activity's execution. A call to `ActivityExecutionContext.ExecuteActivity` by its parent is the *only* stimulus that will cause an activity to move to the ***Executing*** state.

These simple enforcements play a huge role in establishing the meaning and ensuring the integrity of composite activities and, by extension, WF programs.

Of course, there must be one exception to the rule that only the parent of an activity can schedule its execution, and that is for the root activity, whose `Parent` property is `null`. As we have already seen, it is the application hosting the WF runtime that makes a request to the WF runtime to schedule the execution of the root activity's `Execute` method.

Effectively, as part of the creation of a WF program instance, the WF runtime creates an implicit bookmark whose resumption point is the `Execute` method of the root activity. The invocation of `WorkflowInstance.Start` resumes this bookmark, and begins the execution of the program instance.

It will be instructive to trace the execution of a simple WF program that uses `Sequence`, noting the changes that occur at each step to the scheduler work queue. The XAML in Listing 3.15 is a `Sequence` with a set of `WriteLine` child activities.

Listing 3.15 A WF Program That Uses `Sequence`

```
<Sequence x:Name="s1" xmlns="http://EssentialWF/Activities"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <WriteLine x:Name="w1" Text="One" />
  <WriteLine x:Name="w2" Text="Two" />
  <WriteLine x:Name="w3" Text="Three" />
  <WriteLine x:Name="w4" Text="Four" />
</Sequence>
```

Running an instance of this program will result in the expected output.

```
One
Two
Three
Four
```

When the application hosting the WF runtime calls `WorkflowInstance.Start`, it is telling the WF runtime to resume the initial, implicit bookmark. The result of the call to `Start` is that the scheduler work queue for this instance contains a single item—a work item for the `Execute` method of the root activity.

The root activity—in our example, the `Sequence`—is now in the ***Executing*** state, even though its `Execute` method has not actually been called. Figure 3.10 shows the scheduler work queue, along with the state of the WF program instance (with ***Executing*** activities shown in boldface).

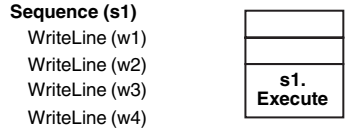

Figure 3.10 WF program instance after `WorkflowInstance.Start`

At this point, the WF runtime's dispatcher logic enters the picture, and invokes the `Sequence` activity's `Execute` method, removing the corresponding item from the scheduler work queue. From this point forward, it is the activities in the WF program that drive the program forward; the WF runtime plays a passive role as the scheduler of work and the enforcer of activity state transitions.

The `Execute` method of `Sequence` will, as we know, schedule its first child activity for execution. When the `Execute` method returns, the scheduler work queue looks as it is shown in Figure 3.11. The first `WriteLine` activity is now in the ***Executing*** state (again, even though its `Execute` method has not been called). The `Sequence` too is in the ***Executing*** state.

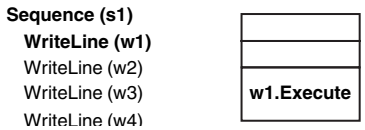

Figure 3.11 WF program instance after `Sequence.Execute`

As we know from the basic pattern used for child activity execution, `Sequence` has, at this point, also subscribed to the `Closed` event of its first child activity. Even though `Closed` (and the other events defined on the `Activity` class) looks like a normal event, under the covers it is an internally managed bookmark.

When the `Execute` method of `WriteLine` returns, the `WriteLine` activity moves to the ***Closed*** state. Because the `Sequence` has subscribed to the event corresponding to this transition, an appropriate work item will be placed in the scheduler work queue. The current state of the program instance is as shown in Figure 3.12; the first `WriteLine` is underlined to indicate that it has completed its execution and is in the ***Closed*** state.

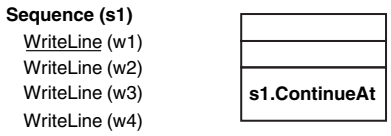

Figure 3.12 WF program instance after first child activity completes

Now the work item for the `ContinueAt` method of `Sequence` is dispatched. As we know, `ContinueAt` will follow the standard pattern for requesting the execution of the second child activity. When `ContinueAt` method returns, the program state is as shown in Figure 3.13, with the second `WriteLine` activity now in the ***Executing*** state.

Figure 3.13 WF program instance after first callback to `Sequence.ContinueAt`

This pattern will continue as the `Sequence` marches through the list of its child activities. When the last child activity reports its completion, the `ContinueAt` method will report the completion of the `Sequence`. The WF runtime will observe this (you can think of the WF runtime as a subscriber to the root activity's `Closed` event), and will do the necessary bookkeeping to complete this WF program instance.

Figure 3.14 summarizes the execution of our WF program as an interaction diagram.

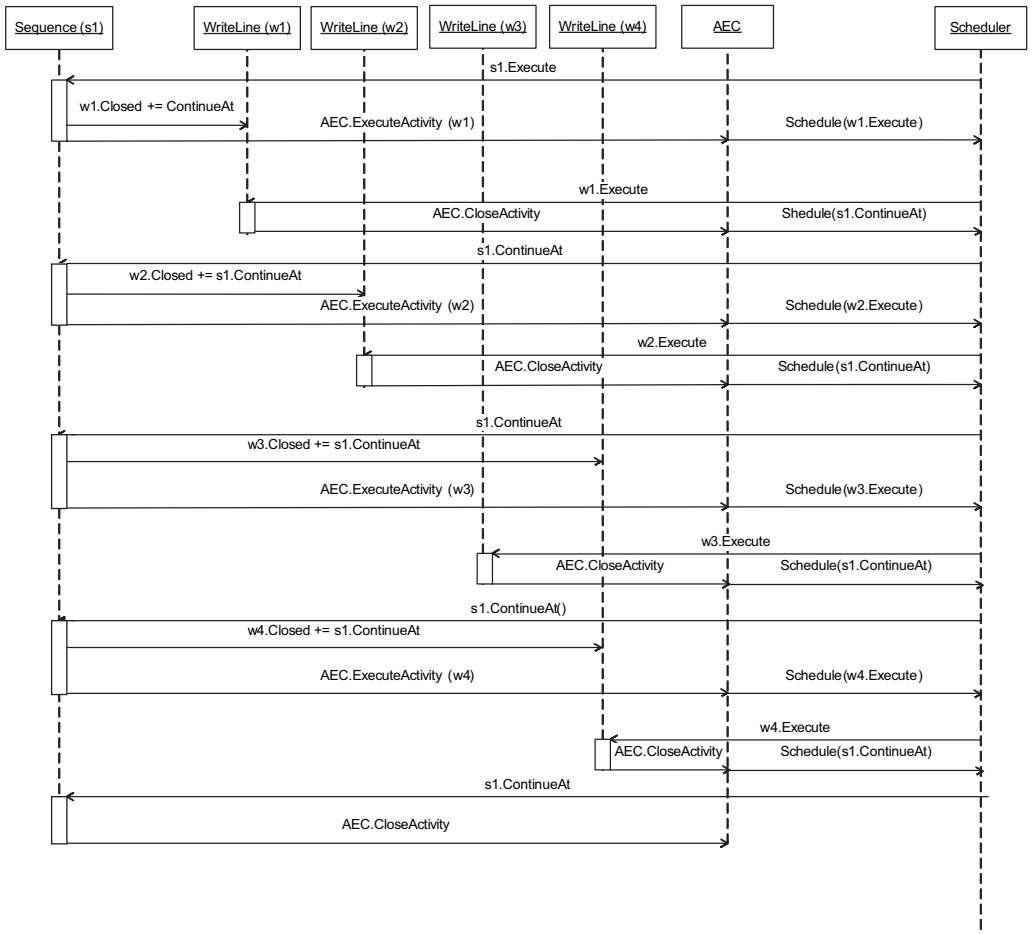

Figure 3.14 Interaction diagram of the execution of Listing 3.15

One crucial point about the `Sequence` activity is that it implemented sequential execution of its child activities using the general-purpose methods and events available on AEC and `Activity`. The WF runtime contains no knowledge of sequential activity execution; it only pays attention to the activity automaton and the containment relationships between activities in its role as enforcer of state transitions.

To see how easy it is to define other forms of control flow as composite activities, let's write a composite activity that executes its child activities in an interleaved manner.

The `Interleave` activity shown in Listing 3.16 implements an `AND` join by first scheduling the execution of all child activities in a single burst and waiting for them all to complete before reporting its own completion.

Listing 3.16 `Interleave` Activity

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

namespace EssentialWF.Activities
{
  public class Interleave : CompositeActivity
  {
    protected override ActivityExecutionStatus Execute(
      ActivityExecutionContext context)
    {
      if (this.EnabledActivities.Count == 0)
        return ActivityExecutionStatus.Closed;

      IList<Activity> shuffled = ShuffleList(EnabledActivities);

      foreach (Activity child in shuffled)
      {
        child.Closed += ContinueAt;
        context.ExecuteActivity(child);
      }

      return ActivityExecutionStatus.Executing;
    }

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

      ActivityExecutionContext context =
        sender as ActivityExecutionContext;

      foreach (Activity child in this.EnabledActivities)
      {
        if ((child.ExecutionStatus !=
              ActivityExecutionStatus.Initialized)
            && (child.ExecutionStatus !=
              ActivityExecutionStatus.Closed))
          return;
      }
```

```
      context.CloseActivity();
    }

    // ShuffleList method elided for clarity
  }
}
```

We will discuss the finer points of the `Interleave` activity's execution (which induces a form of pseudo-concurrency) a bit later in this chapter. First, though, let's look at the mechanics of how `Interleave` executes, just as we did for `Sequence`.

You can see right away that the code for `Interleave` is quite similar to that of `Sequence`. In the `Execute` method, all of the child activities are scheduled for execution, not merely the first one as with `Sequence`. In the implementation of `ContinueAt`, the `Interleave` reports itself as completed only if all child activities are in the `Closed` state.

There is one other interesting line of code:

```
IList<Activity> shuffled = ShuffleList(EnabledActivities);
```

`ShuffleList` is presumed to be a private helper method that simply shuffles the list of child activities into some random order. The `Interleave` activity will work just fine without `ShuffleList`, but we have added it so that users of `Interleave` cannot predict or rely upon the order in which child activities are scheduled for execution.

The XAML in Listing 3.17 is an `Interleave` activity that contains a set of `WriteLine` child activities.

Listing 3.17 Interleaved Execution of `WriteLine` Activities

```
<Interleave x:Name="i1" xmlns="http://EssentialWF/Activities"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <WriteLine x:Name="w1" Text="One" />
  <WriteLine x:Name="w2" Text="Two" />
  <WriteLine x:Name="w3" Text="Three" />
  <WriteLine x:Name="w4" Text="Four" />
</Interleave>
```

Running the program in Listing 3.17 may result in the following output:

```
Four
Two
```

```
Three
One
```

Or the following:

```
Three
One
Four
Two
```

Or the following:

```
One
Two
Three
Four
```

Or any of the other possible orderings of the four strings printed by the four `WriteLine` activities.

Let's trace through the execution of an instance of this program, showing the scheduler work queue and program state. The program is started exactly like the one we developed earlier with `Sequence`; an item is placed in the scheduler work queue representing a call to the `Execute` method of the root activity, and the root activity is placed in the `Executing` state (see Figure 3.15).

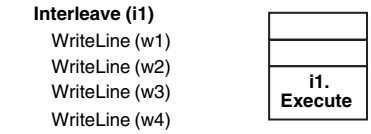

Figure 3.15 WF program instance in Listing 3.17 after `WorkflowInstance.Start`

Let's assume that the call to `ShuffleList` results in the following ordering of child activities: w2, w4, w1, w3. When the `Execute` method of `Interleave` returns, the program state is as shown in Figure 3.16.

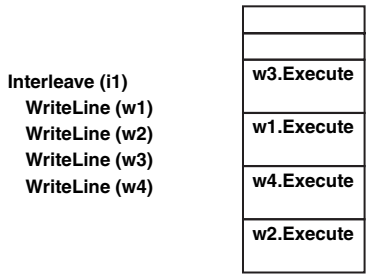

Figure 3.16 WF program instance in Listing 3.17 after `Interleave.Execute`

Now all four child activities are queued for execution and all four child activities of the `Interleave` are in the ***Executing*** state. The dispatcher will pick the item from the front of the queue (`Execute` "w2"). This will cause the `Execute` method of `WriteLine` named "w2" to be invoked. When this method returns, "Two" will have been printed to the console and the state of the program is as shown in Figure 3.17.

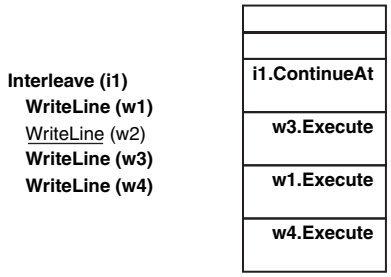

Figure 3.17 WF program instance in Listing 3.17 after `WriteLine` "w2" completes

As expected, because `Interleave` has subscribed to the `Closed` event of each child activity, there is a callback to the `ContinueAt` method present in the scheduler work queue. This item, however, sits behind three other items—the execution handlers for the `Execute` methods of w4, w1, and w3. The process outlined for w2 will therefore repeat three more times, resulting in the state shown in Figure 3.18.

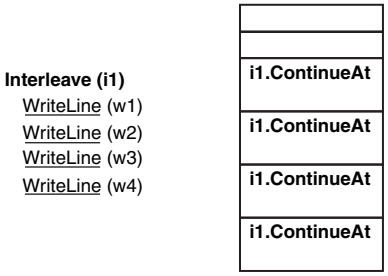

Figure 3.18 WF program instance in Listing 3.17 after `WriteLine` "w4" completes

At this point, all four `WriteLine` activities have completed. The `Interleave` activity, though, has not actually received any notifications because its work items are still in the scheduler work queue. The four work items in the scheduler work queue are all resumptions of the same bookmark. The resumption point is the `ContinueAt` method of `Interleave`; the four work items differ only in the `EventArgs` data that is the payload of each resumed bookmark.

When the first work item is delivered to `Interleave`, the logic of the `ContinueAt` method will determine that all child activities are in the `Closed` state, so the `Interleave` itself is reported as complete. When the other three callbacks are subsequently dispatched, the WF runtime observes that the `Interleave` is already in the ***Closed*** state, so the callbacks are not delivered (they are simply discarded); delivery of these callbacks would violate the activity automaton because the `Interleave` cannot resume execution once it is in the ***Closed*** state.

Now, what we have seen in the execution of this WF program is quite a bit different than what we saw for the WF program that used `Sequence`. Things get even more interesting, though, if each child activity of the `Interleave` is not a simple activity like `WriteLine`, but a `Sequence` (which might contain other `Interleave` activities). Furthermore, it's clearly not very interesting or useful to simply execute `WriteLine` activities in an interleaved manner. It is much more realistic for each branch to be performing work that depends upon external input. In this way, the ordering of the execution of activities is determined, in part, by the timing of `EnqueueItem` operations performed by external code on WF program queues. By modeling these interactions in an `Interleave`, no branch is blocked by any other (because activities use bookmarks when their execution awaits external stimulus) and the execution of the activities within the branches can interleave.

As we know, the `Interleave` activity uses an explicit shuffling technique to decide the ordering in which its child activities are scheduled for execution. The influence of `Interleave`, however, ends there. If a `Sequence` activity is added as a child activity of an `Interleave`, the `Interleave` controls when the `Sequence` executes, but only the `Sequence` controls when its child activities are executed.

The XAML in Listing 3.18 is an `Interleave` with a set of `Sequence` child activities that contain child activities. The name of the WF program queue created by `ReadLine` in its `Initialize` method is the name of the activity. So, four WF program queues will be created during the initialization of a WF program instance, and these WF program queues are named r1, r2, r3, and r4.

Listing 3.18 Interleaved Execution of `Sequence` Activities

```
<Interleave x:Name="i1" 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}" />
    <ReadLine x:Name="r2" />
    <WriteLine x:Name="w2" Text="{wf:ActivityBind r2,Path=Text}" />
  </Sequence>
  <Sequence x:Name="s2">
    <ReadLine x:Name="r3" />
    <WriteLine x:Name="w3" Text="{wf:ActivityBind r3,Path=Text}" />
    <ReadLine x:Name="r4" />
    <WriteLine x:Name="w4" Text="{wf:ActivityBind r4,Path=Text}" />
  </Sequence>
</Interleave>
```

We are not going to go through the execution of an instance of this program step by step—it would take a few pages of diagrams—but we know enough about the execution logic of the `Sequence` and `Interleave` activities to predict what will happen. Assuming that no items are enqueued into any of the WF program queues, the program will reach the state shown in Figure 3.19.

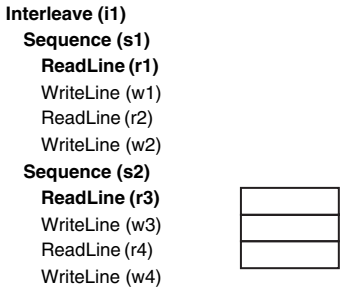

Figure 3.19 WF program instance in Listing 3.18 after reaching `ReadLine` activities

At this point, the program is idle. Both `Sequence` activities have started executing, and each has, in turn, requested the execution of their first child activity (which happens to be a `ReadLine` activity in both cases). Each `ReadLine` activity is stuck waiting for an item to appear in its WF program queue. If the `Interleave` had a third child activity that was a `Sequence` of any number of `WriteLine` activities, then this `Sequence` would run to completion.

If we enqueue the string "hello" into WF program queue "r3", there will be an episode of action. The `ContinueAt` method of the `ReadLine` activity with name "r3" will be scheduled (the name of the WF program queue created by `ReadLine` is the same as its `Name` property). This will cause the `ReadLine` activity to complete, which will schedule notification of its `Closed` event to the enclosing `Sequence` "s2". That `Sequence` will schedule the execution of the `WriteLine` "w3" that follows the just-completed `ReadLine`. The `WriteLine` will get the string received by the `ReadLine` activity (via activity databinding) and write it to the console. The `WriteLine` will complete, again causing a notification to the enclosing `Sequence`. The `Sequence` will then move on to its third child activity, another `ReadLine`, which will now wait until an item is enqueued into its WF program queue.

The series of steps just described will result in the state of the program shown in Figure 3.20.

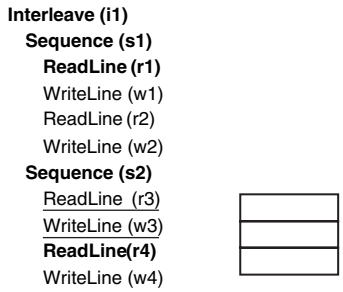

Figure 3.20 WF program instance in Listing 3.18 again in an idle state

This example is typical of the episodic execution we described at the outset of the chapter. As a result of stimulus from the external world, the WF program instance moves forward. And it is truly the composite activities that are driving the program's execution by providing the control flow; the WF runtime is passively dispatching whatever items appear in the scheduler work queue while enforcing adherence to the activity automaton.

Once you understand the activity automaton and the execution-related rules of activity composition, it is easy to model other control flow patterns beyond simple sequential and interleaved execution. This allows your programs to mirror more precisely whatever processes they are trying to coordinate. In the next chapter, we will look at several additional aspects of composite activity development that aid in building different kinds of control flow.

It may be helpful to pause here and consolidate what you've learned from this chapter so far. As an exercise, we suggest writing a custom composite activity. An appropriate choice on which to test your skills is `PrioritizedInterleave`. The `PrioritizedInterleave` activity executes its child activities in priority order. Each child activity has a property, named `Priority`, of type `int`.

When `PrioritizedInterleave` executes, first all child activities with a priority of 1 are executed in an interleaved manner; when those are completed, all child activities with a priority of 2 are executed (also in an interleaved manner). This continues until all child activities have been executed. As you might guess, the execution logic of `PrioritizedInterleave` is something of a combination of the logic we developed for `Sequence` and the logic we developed for `Interleave`.

Listing 3.19 shows an example WF program containing a `PrioritizedInterleave`. The seven child activities of the `PrioritizedInterleave` are grouped into three different sets according to the values of their `Priority` property. The best way to implement the `Priority` property is as an **attached property**, which supports the XAML syntax shown in Listing 3.19. Attached properties are covered in Chapter 7, "Advanced Authoring." You can take a simpler approach and add a `Priority` property to `WriteLine` and then test your `PrioritizedParallel` activity using the modified `WriteLine`.

Listing 3.19 WF Program that Is a `PrioritizedInterleave`

```
<PrioritizedInterleave xmlns="http://EssentialWF/Activities">
  <B PrioritizedInterleave.Priority="1" />
  <C PrioritizedInterleave.Priority="2" />
  <A PrioritizedInterleave.Priority="1" />
  <E PrioritizedInterleave.Priority="2" />
  <F PrioritizedInterleave.Priority="3" />
  <G PrioritizedInterleave.Priority="3" />
  <D PrioritizedInterleave.Priority="2" />
</PrioritizedInterleave>
```

This WF program is depicted in a more readable form in Figure 3.21, which conveys the interleaved execution that occurs within the groupings of child activities.

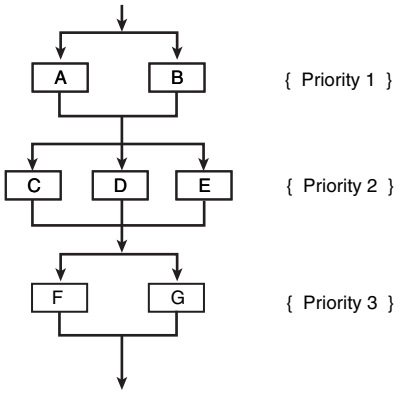

Figure 3.21 `PrioritizedInterleave` **activity containing three groupings**

You might conclude from Figure 3.21 that this WF program could just as easily be built using the `Sequence` and `Interleave` activities we developed previously. True,

but there is another way of looking at things. Both `Sequence` and `Interleave` are nothing but degenerate cases of our `PrioritizedInterleave`, in which the priorities of the child activities are either all different or all the same, respectively. This is a simple but instructive example of the control flow flexibility afforded by the composition model of WF.

WF Threads

From the point of view of activities, the WF runtime makes no guarantee about which CLR thread is used to dispatch a work item in a scheduler work queue. It is possible for any two work items, even consecutively enqueued work items, to be dispatched using different threads.

It is the application hosting the WF runtime that decides how CLR threads are to be allocated to WF program instances (though the WF runtime does impose a limit of one thread at a time for a specific WF program instance). It is also solely the host application that determines when a WF program instance should be passivated. Typically, passivation occurs when a WF program instance becomes idle, but it is possible, as we will see in Chapter 5, for a WF program instance to be passivated even when its scheduler work queue is not empty. As a developer of activities, the safest assumption is that every work item is dispatched on a different thread. Although in practice the same CLR thread will be used to dispatch a set of work items, it is best to not make any assumption about the CLR thread on which activity methods are invoked. This means not storing data in the thread context or call context or, more generally, not relying on `Thread.CurrentThread` in any way.

The WF runtime does guarantee that the scheduler managing the work items for a single WF program instance utilizes *exactly one* CLR thread at a time, for a given episode of the WF program instance. In other words, the scheduler never performs concurrent dispatch of work items in its work queue. Dispatch always occurs one item at a time. Furthermore, the WF runtime never preempts the execution of a dispatched work item. Activities are counted upon to employ bookmarks when they are logically blocked, allowing them to yield the CLR thread while they await stimulus.

The fact that the WF runtime uses a single CLR thread at a time for a given WF program instance is a pragmatic decision. It is possible to imagine concurrent dispatch of work items, but the benefits appear to be outweighed by the drawbacks.

One big advantage of a single-threaded execution model is the simplification of activity development. Activity developers need not worry about concurrent execution of an activity's methods. Locking, preemption, and other aspects of multi-threaded programming are not a part of WF activity development, and these simplifications are important given WF's goal of broad adoption by .NET developers.

Some readers might object to the fact that the threading model of the WF runtime eliminates the possibility of true, or fine-grained, concurrency (the simultaneous use of more than one machine processor). Let's be clear: What is precluded is the possibility of true concurrency *within an instance of a WF program*. In any application where the number of simultaneously executing (non-idle) WF program instances tends to be greater than the number of machine processors, true concurrency would not buy you much. The design-time overhead of a vastly more challenging programming model for activities weighs down this approach, and mightily so in our estimation. Computations that benefit from true concurrency are, for WF programs, best abstracted as features of a service; the service can be made available to activities (using the service chaining techniques we've already described). In this way, the service can be executed in an environment optimized for true concurrency, which may or may not be on the machine on which the WF program instance is running.

True concurrency is a rather simple concept to describe, but the techniques for synchronization that are available in most mainstream programming paradigms are difficult to master and, when not applied properly, are notorious for causing hard-to-find bugs that make programs defective (or, perhaps, in the eyes of their users, capricious). The WF programming model arguably has found a sweet spot, given the types of problems that WF is intended to solve. WF program instances clearly allow interleaved (pseudo-concurrent) execution of activities, and WF makes it *easy to write and use* the constructs that permit interleaving. We have seen an example of such an activity, `Interleave`, and how essentially similar it is to `Sequence`, in both its implementation and its usage in a WF program.

Just like the CLR virtualizes a set of operating system threads, the WF runtime can be said to virtualize CLR threads. The interleaved execution of activities within a WF program instance is therefore not unlike the interleaved execution of CLR threads within an operating system process. Each child activity of an `Interleave` can be thought of as executing on a separate **WF thread**, though in fact this WF thread is a purely conceptual entity and does not have any physical manifestation

in the WF programming model. Figure 3.22 depicts the relationship between these shadowy WF threads and actual CLR threads.

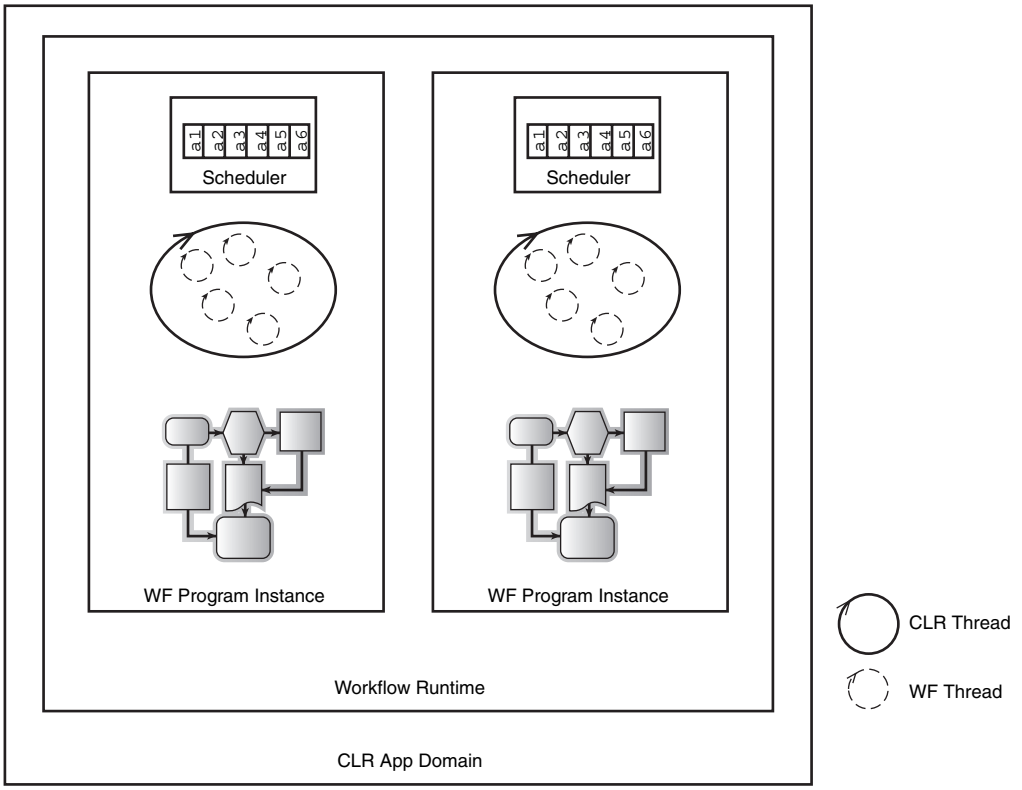

Figure 3.22 WF threads

This pattern of execution is sometimes called pseudo-concurrency or perceived parallelism.

Synchronized Access to State

Given the WF runtime's threading model, it should be clear that the synchronization primitives used in C# programs are not applicable in WF programs.

Synchronization primitives are not aware of the interleaved execution of WF threads. In fact, they can get you into quite a bit of trouble in a WF program and should be generally avoided. For example, if two activities in a WF program refer to

some shared state (for instance, several fields of a third activity, accessed as properties of that activity), then CLR synchronization techniques will not be the right choice for ensuring synchronized access to the shared state. CLR locking primitives are generally not designed to survive and remain valid across passivation cycles of a WF program instance.

Put another way, since the WF programming model virtualizes threads of program execution, it must also carry the burden of providing synchronized access to shared data.

WF provides the ability to synchronize access to state shared by multiple WF threads in terms of a special composite activity defined in the `System.Workflow.ComponentModel` namespace. This activity is named `SynchronizationScopeActivity` and it executes its child activities sequentially.

`SynchronizationScopeActivity` is the WF programming model's synchronization primitive. It allows the developer of a WF program to draw boundaries around synchronization domains of (pseudo)concurrently executing activities, which, conceptually, run on different WF threads.

The `SynchronizationScopeActivity` type is shown in Listing 3.20.

Listing 3.20 `SynchronizationScopeActivity`

```
namespace System.Workflow.ComponentModel
{
  public sealed class SynchronizationScopeActivity : CompositeActivity
  {
    public ICollection<string> SynchronizationHandles { get; set; }

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

As you can see, the `SynchronizationScopeActivity` type carries a property called `SynchronizationHandles` of type `ICollection<string>`. This property holds a set of named **synchronization handles**. A synchronization handle is essentially a locking primitive.

The WF runtime guarantees that occurrences of `SynchronizationScopeActivity` sharing a synchronization handle token will be executed serially without any interleaving of their contained activities. In other words, one `SynchronizationScopeActivity` will complete before the next one (that shares a synchronization

handle with the first) begins. To avoid deadlocks, the WF runtime internally manages virtual locks (not CLR locks) corresponding to the synchronization handles specified by the occurrences of `SynchronizationScopeActivity` in a WF program. These virtual locks survive passivation of the WF program instance.

Before the execution of a `SynchronizationScopeActivity` begins, all of the virtual locks associated with that `SynchronizationScopeActivity` activity's set of synchronization handles are obtained.

Listing 3.21 shows a WF program that uses `SynchronizationScopeActivity` to provide synchronized execution of interleaving activities. Even though there is no actual shared data, the two occurrences of `SynchronizationScopeActivity` require the same virtual lock and therefore execute serially.

Listing 3.21 Synchronization Using `SynchronizationScopeActivity`

```
<Interleave xmlns="http://EssentialWF/Activities"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:wf="http://schemas.microsoft.com/winfx/2006/xaml/workflow" >
  <wf:SynchronizationScopeActivity SynchronizationHandles="h1">
    <WriteLine x:Name="w1" Text="One"/>
    <WriteLine x:Name="w2" Text="Two"/>
  </wf:SynchronizationScopeActivity>
  <wf:SynchronizationScopeActivity SynchronizationHandles="h1">
    <WriteLine x:Name="w3" Text="Three"/>
    <WriteLine x:Name="w4" Text="Four"/>
  </wf:SynchronizationScopeActivity>
</Interleave>
```

This WF program will always produce one of the following outputs:

```
One                 Three
Two                 Four
Three               One
Four                Two
```

There are only two possible outputs for the preceding program. The `Interleave` activity will schedule both of the `SynchronizationScopeActivity` activities. Whichever one is scheduled first acquires the virtual lock that protects the synchronization handle "h1". Once the lock is obtained, the second `SynchronizationScopeActivity` activity is not allowed to execute, even though

it has a work item in the scheduler work queue. Only when the first `SynchronizationScopeActivity` transitions to the ***Closed*** state will the lock be released, and the second `SynchronizationScopeActivity` be permitted to execute.

In the preceding example, there is no interleaving of activity execution across the two occurrences of `SynchronizationScopeActivity`, due to the fact that they require the same lock. If we change the program by altering the value of the `SynchronizationHandles` property for one `SynchronizationScopeActivity` to "h2", then the presence of the two `SynchronizationScopeActivity` activities is meaningless because they are defining different synchronization domains. Interleaved execution of the activities contained within them can and will occur.

`SynchronizationScopeActivity` activities can be nested in a WF program. Each `SynchronizationScopeActivity` acts as a lock manager that is responsible for granting locks to its child activities and managing a wait list of activities waiting to acquire locks (the WF runtime acts as the lock manager for the root activity of the program).

`SynchronizationScopeActivity`, when it begins its execution, collects the locks corresponding to its synchronization handles as well as those for all nested `SynchronizationScopeActivity` activities.

Because a parent `SynchronizationScopeActivity` is guaranteed to start its execution before a `SynchronizationScopeActivity` nested within it, the parent acquires the locks needed for all of its nested child `SynchronizationScopeActivity` instances before executing them, and deadlocks are safely avoided.

For the WF program shown in Listing 3.22, either `SynchronizationScopeActivity` s1 or `SynchronizationScopeActivity` s4 will execute in its entirety before the other one begins executing. In this example, the locks required by s1 and s4 are the same (indicated by the synchronization handles "a", "b", and "c"). In fact, the execution of s1 and s4 will be serialized even if they share a single synchronization handle name in their respective subtrees.

Listing 3.22 Nested `SynchronizationScopeActivity` Declarations

```
<Interleave xmlns="http://EssentialWF/Activities"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:wf="http://schemas.microsoft.com/winfx/2006/xaml/workflow">
  <wf:SynchronizationScopeActivity x:Name="s1"
    SynchronizationHandles="a">
```

```
    <Interleave x:Name="i1">
      <wf:SynchronizationScopeActivity x:Name="s2"
        SynchronizationHandles="b">
        <WriteLine x:Name="w3" Text="One"/>
        <WriteLine x:Name="w4" Text="Two"/>
      </wf:SynchronizationScopeActivity>
      <wf:SynchronizationScopeActivity x:Name="s3"
        SynchronizationHandles="c">
        <WriteLine x:Name="w5" Text="Three"/>
        <WriteLine x:Name="w6" Text="Four"/>
      </wf:SynchronizationScopeActivity>
    </Interleave>
  </wf:SynchronizationScopeActivity>
  <wf:SynchronizationScopeActivity x:Name="s4"
    SynchronizationHandles="c">
    <Interleave x:Name="i2">
      <wf:SynchronizationScopeActivity x:Name="s5"
        SynchronizationHandles="b">
        <WriteLine x:Name="w9" Text="Five"/>
        <WriteLine x:Name="w10" Text="Six"/>
      </wf:SynchronizationScopeActivity>
      <wf:SynchronizationScopeActivity x:Name="s6"
        SynchronizationHandles="a">
        <WriteLine x:Name="w11" Text="Seven"/>
        <WriteLine x:Name="w12" Text="Eight"/>
      </wf:SynchronizationScopeActivity>
    </Interleave>
  </wf:SynchronizationScopeActivity>
</Interleave>
```

`SynchronizationScopeActivity` provides a simple way to synchronize the interleaved execution of WF threads. Effectively, this synchronization technique orders the dispatch of operations in the scheduler work queue in accordance with the synchronization domains that are named by `SynchronizationScopeActivity` activities.

Where Are We?

This chapter introduced the activity automaton, which describes the lifecycle of any activity in a WF program instance. Our foray into custom activity development introduced the service-chaining capabilities of the WF runtime, as well as the use of bookmarks (and associated WF program queues) as a mechanism by which activities can receive stimulus from external entities.

We turned to composite activities next and discussed three—`Sequence`, `Interleave`, and `PrioritizedInterleave`—that provided insights into the flexibility of WF's model for control flow. Control flow patterns are a theme that will be continued in the next chapter.

We discussed the WF execution model and threading model, and specifically examined how the WF scheduler works. By looking at how the execution of various WF programs unfolds, we saw the nuts and bolts of how the scheduler performs its role as an intermediary for the dispatch of resumed bookmarks. As well, we learned that the WF runtime enforces the activity automaton and also protects the integrity of the containment relationships between activities in a WF program. Pseudo-concurrency within a WF program instance is caused by the interleaving of WF threads of execution.

The next chapter is a continuation of this one (feel free to bookmark it now). We will examine the ***Canceling***, ***Faulting***, and ***Compensating*** states of the activity automaton, and also introduce the WF programming model's support for explicit management of activity execution contexts.