Pro WF: Windows Workflow in .NET 3.5

Copyright © 2008 by Bruce Bukovics

ISBN-13 (pbk): 978-1-4302-0975-1

ISBN-13 (electronic): 978-1-4302-0976-8

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Matthew Moodie
Technical Reviewers: Brian Robert Myers, Sylvain Groulx
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell,
Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann,
Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Beth Christmas
Copy Editor: Kim Wimpsett
Associate Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Susan Glinert
Proofreader: Patrick Vincent
Indexer: Julie Grady
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at http://www.apress.com/info/bulksales.

The source code for this book is available to readers at http://www.apress.com. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 1

A Quick Tour of Windows Workflow Foundation

This chapter presents a brief introduction to Windows Workflow Foundation (WF). Instead of diving deeply into any single workflow topic, it provides you with a sampling of topics that are fully presented in other chapters.

You'll learn why workflows are important and why you might want to develop applications using them. You'll then jump right in and implement your very first functioning workflow. Additional hands-on examples are presented that demonstrate other features of Windows Workflow Foundation.

Why Workflow?

As developers, our job is to solve real business problems. The type and complexity of the problems will vary broadly depending on the nature of the business. But regardless of the complexity of any given problem, we tend to solve problems in the same way: we break the problem down into manageable parts. Those parts are further divided into smaller tasks, and so on.

When we've finally reached a point where each task is the right size to understand and manage, we identify the steps needed to accomplish the task. The steps usually have an order associated with them. They represent a sequence of individual instructions that will yield the expected behavior only when they are executed in the correct order.

In the traditional programming model, you implement a task in code using your chosen development language. The code specifies what to do (the execution instructions) along with the sequence of those instructions (the flow of control). You also include code to make decisions (rules) based on the value of variables, the receipt of events, and the current state of the application.

A workflow is simply an ordered series of steps that accomplish some defined purpose according to a set of rules. By that definition, what I just described is a *workflow*.

It might be defined entirely in code, but it is no less a type of workflow. We already use workflows every day we develop software. We might not consider affixing the workflow label to our work, but we do use the concepts even if we are not consciously aware of them.

So why all of this talk about workflows? Why did I write a book about them? Why are you reading it right now?

Workflows Are Different

The workflow definition that I gave previously doesn't tell the whole story, of course. There must be more to it, and there is. To a developer, the word *workflow* typically conjures up images of a highly visual environment where complex business rules and flow of control are declared graphically. It's an environment that allows you to easily visualize and model the activities (steps) that have been declared to solve a problem. And since you can visualize the activities, it's easier to change, enhance, and customize them.

But there is still more to workflows than just the development environment. Workflows represent a different programming model. It's a model that promotes a clear separation between *what* to do and *when* to do it. This separation allows you to change the *when* without affecting the *what*. Workflows generally use a declarative programming model rather than a procedural one. With this model, business logic can be encapsulated in discrete components. But the rules that govern the flow of control between components are declarative.

General-purpose languages such as C# or Visual Basic can obviously be used to solve business problems. But the workflow programming model really enables you to implement your own domain-specific language. With such a language, you can express business rules using terms that are common to a specific problem domain. Experts in that domain are able to view a workflow and easily understand it, since it is declared in terminology that they understand.

For example, if your domain is banking and finance, you might refer to *accounts*, *checks*, *loans*, *debits*, *credits*, *customers*, *tellers*, *branches*, and so on. But if the problem domain is pizza delivery, those entities don't make much sense. Instead, you would model your problems using terms such as *menus*, *specials*, *ingredients*, *addresses*, *phone numbers*, *drivers*, *tips*, and so on. The workflow model allows you to define the problem using terminology that is appropriate for each problem domain.

Workflows allow you to easily model system and human interactions. A *system interaction* is how we as developers would typically approach a problem. You define the steps to execute and write code that controls the sequence of those steps. The code is always in total control.

Human interactions are those that involve real live people. The problem is that people are not always as predictable as your code. For example, you might need to model a mortgage loan application. The process might include steps that must be executed by real people in order to complete the process. How much control do you have over the order of those steps? Does the credit approval always occur first, or is it possible for the appraisal to be done first? What about the property survey? Is it done before or after the appraisal? And what activities must be completed before you can schedule the loan closing? The point is that these types of problems are difficult to express using a purely procedural model because human beings are in control. The exact sequence of steps is not always predictable. The workflow model really shines when it comes to solving human interaction problems.

Why Windows Workflow Foundation?

If workflows are important, then why use Windows Workflow Foundation? Microsoft has provided this foundation in order to simplify and enhance your .NET development. It is not a stand-alone application. It is a software foundation that is designed to enable workflows within your applications. Regardless of the type of application you are developing, there is something in WF that you can leverage.

If you are developing line-of-business applications, you can use WF to orchestrate the business rules. If your application comprises a series of human interactions, you can use a WF state machine workflow to implement logic that can react to those interactions. If you need a highly customizable application, you can use the declarative nature of WF workflows to separate the business logic from the execution flow. This allows customization of the flow of control without affecting the underlying business logic. And if you are looking for a better way to encapsulate and independently test your application logic, implement the logic as discrete custom activities that are executed within the WF runtime environment.

There are a number of good reasons to use WF, and here are a few of them:

- It provides a flexible and powerful framework for developing workflows. You can spend your time and energy developing your own framework, visual workflow designer, and runtime environment. Or you can use a foundation that Microsoft provides and spend your valuable time solving real business problems.
- It promotes a consistent way to develop your applications. One workflow looks very similar to the next. This consistency in the programming model and tools improves your productivity when developing new applications and your visibility when maintaining existing ones.
- It supports *sequential* and *state machine* workflows. Sequential workflows are generally used for system interactions. State machine workflows are well suited to solving problems that focus on human interaction.
- It supports workflow persistence. The ability to save and later reload the state of a running workflow is especially important when modeling human interactions and for other potentially long-running workflows.
- It supports problem solving using a domain-specific model. Microsoft encourages you to develop your own custom activity components. Each custom component addresses a problem that is specific to your problem domain and uses terminology that is common to the domain.
- It provides a complete workflow ecosystem. In addition to the workflow runtime itself, Microsoft also provides a suite of standard activities, workflow persistence, workflow monitoring and tracking, a rules engine, and a workflow designer that is integrated with Visual Studio, which you can also host in your own applications.
- It is infinitely extensible. Microsoft provides a number of extension points that permit you to modify the WF default behavior. For example, if the standard SQL persistence service that is provided with WF doesn't meet your needs, you can implement your own.
- It is included with Visual Studio and available for use in your applications without any additional licensing fees. Because of this, it has become the de facto standard workflow framework for Windows developers. The growing community of WF developers frequently share their ideas, custom activity components, and other code.

Your Development Environment

Windows Workflow Foundation was originally made available as part of .NET 3.0. The development environment was supplied as an add-in to Visual Studio 2005. Visual Studio 2008 now includes built-in support for WF (no add-in required), and .NET 3.5 includes several new WF features such as support for workflow services.

In order to develop applications using Windows Workflow Foundation, you'll need to install a minimum set of software components. The minimum requirements are the following:

- Visual Studio 2008 Professional, Standard, or Team System
- The .NET 3.5 runtime (installed with Visual Studio 2008)

The combination of Visual Studio 2008 and .NET 3.5 will allow you to use all of the latest WF features. And it has the advantage of one simple installation.

Alternatively, you can still use Visual Studio 2005 and .NET 3.0 for WF development. However, if you do so, you will be limited to only those features that shipped with the original version of WF. The minimum requirements for WF development using Visual Studio 2005 and .NET 3.0 are as follows:

- Visual Studio 2005 Enterprise, Professional, or Standard
- The .NET 3.0 runtime
- A designated version of the Windows SDK that supports WF
- The WF add-in to Visual Studio

Please refer to the Microsoft MSDN site for the latest download and installation instructions.

■**Note** Unless otherwise noted, all WF features and classes are available in .NET 3.0 and .NET 3.5. However, any descriptions of the WF development environment refer to Visual Studio 2008. All screenshots in this book were captured using Visual Studio 2008.

Hello Workflow

At this point you are ready to create your first workflow. In the world of technology in which we work, it has become customary to begin any new technical encounter with a "Hello World" example.

Not wanting to break with tradition, I present a "Hello Workflow" example in the pages that follow. If you follow along with the steps as I present them, you will have a really simple yet functional workflow application.

In this example, and in the other examples in this chapter, I present important concepts that are the basis for working with all workflows, regardless of their complexity. If you already have experience working with Windows Workflow Foundation, you might feel compelled to skip over this information. If so, go ahead, but you might want to give this chapter a quick read anyway.

To implement the "Hello Workflow" example, you'll create a sequential workflow project, add one of the standard activities to the workflow, and then add code to display "Hello Workflow" on the console.

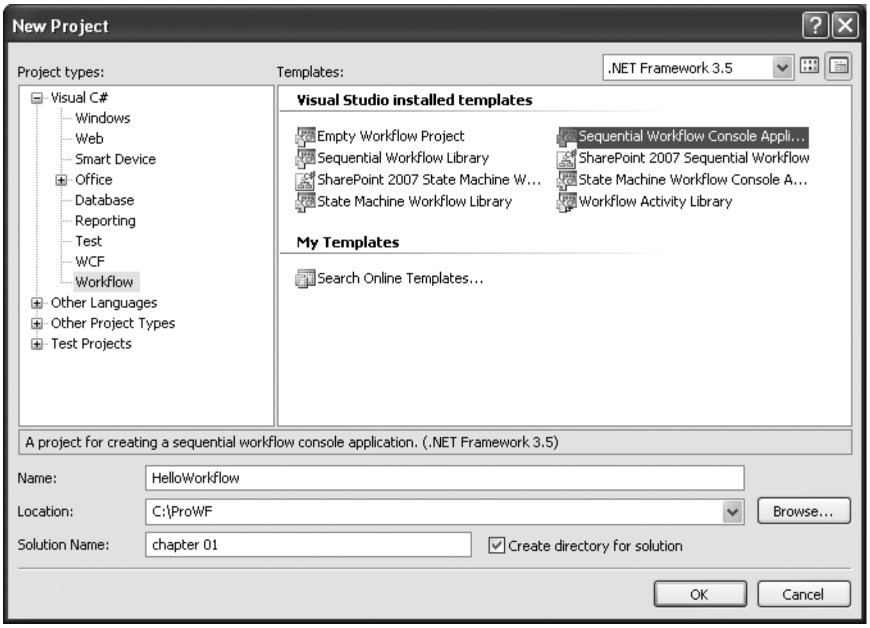
Creating the Workflow Project

Workflow projects are created in the same way as other project types in Visual Studio. After starting Visual Studio 2008, you select File ➤ New ➤ Project. A New Project dialog is presented that should look similar to the one shown in Figure 1-1.

After selecting Visual C# as the language, you'll see Workflow as one of the available project template categories. As shown in Figure 1-1, several workflow project templates are available. For this example, you should choose Sequential Workflow Console Application. This produces a console application that supports the use of Windows Workflow Foundation. A *sequential workflow* is one that executes a series of steps in a defined sequence. That's exactly the type of workflow that we need for this example.

■**Note** Visual Basic developers don't have to worry. A similar set of workflow project templates are also available for Visual Basic if that's your language of choice.

You should now enter a meaningful name for the project, such as **HelloWorkflow**, select a location, and click OK to create the new project.


New Project
Project types:
Templates:
.NET Framework 3.5
Visual C#
Windows
Web
Smart Device
Office
Database
Reporting
Test
WCF
Workflow
Other Languages
Other Project Types
Test Projects
Visual Studio installed templates
Empty Workflow Project
Sequential Workflow Console Appli…
Sequential Workflow Library
SharePoint 2007 Sequential Workflow
SharePoint 2007 State Machine W…
State Machine Workflow Console A…
State Machine Workflow Library
Workflow Activity Library
My Templates
Search Online Templates…
Browse…
OK
Cancel

A project for creating a sequential workflow console application. (.NET Framework 3.5)

Name: HelloWorkflow

Location: C:\ProWF

Solution Name: chapter 01

☑ Create directory for solution

Figure 1-1. *Sequential workflow console application New Project dialog*

■ **Note** In the example shown in Figure 1-1, the project will be added to a new solution named chapter 01. All of the example code for this book is organized into separate solutions for each chapter.

You can see one of the new features of Visual Studio 2008 in the upper-right corner of Figure 1-1. This combo box allows you to select the version of the .NET Framework that you want to target with this project. Options are available for .NET Framework 2.0, 3.0, and 3.5. All examples in this book assume that the target is .NET Framework 3.5.

The choice of target is important since a number of Visual Studio design elements are sensitive to your selection. For example, if .NET Framework 3.5 is selected, the project includes default references to several new .NET 3.5 assemblies (`System.Core` and `System.Xml.Linq`). If you select a different target, those references are omitted. The Visual Studio Toolbox also filters the list of controls to only those that are available in the selected target version of .NET.

After a second or two, the new project is created. The Solution Explorer window shows the source files that are created as part of the project, as shown in Figure 1-2.

Notice that I've expanded the References folder in order to show the assembly references for the project. When you select a workflow project as the template, the assembly references necessary to use WF are added for you. The workflow-related assemblies are the following:

- `System.Workflow.Activities`
- `System.Workflow.ComponentModel`
- `System.Workflow.Runtime`
- `System.WorkflowServices` (.NET 3.5 only)

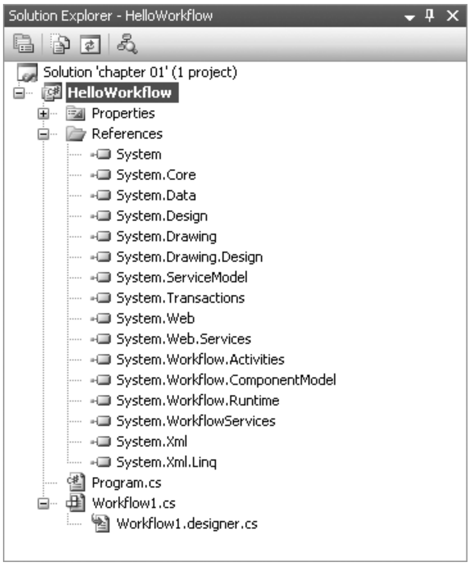


Figure 1-2. *Solution Explorer for the new workflow project*

Within these assemblies, the workflow-related classes are organized into a number of namespaces. In your code, you need to reference only the namespaces that you are actually using.

■ **Note** The `System.WorkflowServices` assembly is new with .NET 3.5 and contains the types that support workflow services (Windows Communication Foundation integration with WF). This assembly is added by default but is not needed for this example.

The project template created a file named `Program.cs`. Since this is a console application, this file contains the `Main` method associated with the application. We'll review the generated code for this file shortly.

Introducing the Workflow Designer

Also generated is a file named `Workflow1.cs`. This file contains the code that defines the workflow and is associated with the visual workflow designer, which is its editor. When this file is opened, as it is when the project is first created, the initial view of the designer looks like Figure 1-3.

The workflow designer is the primary canvas that you will use to define your workflows. You can also define a workflow entirely in code, in much the same way that you can define an entire Windows form or other user interface elements in code. But one of the best features of WF is the designer, and using it will greatly increase your productivity when defining workflows. The designer supports dragging and dropping of activities onto the workflow canvas from the Visual Studio Toolbox.

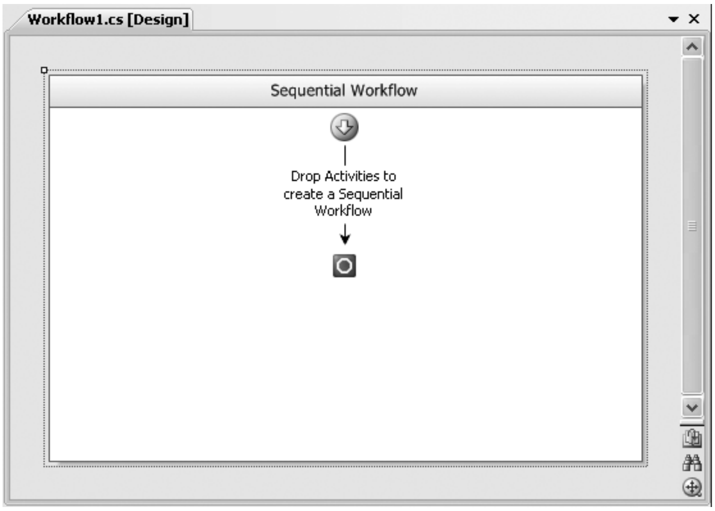


Figure 1-3. *Initial view of the visual workflow designer*

Using Workflow Activities

An *activity* represents a step in the workflow and is the fundamental building block of all WF workflows. All activities either directly or indirectly derive from the base `System.Workflow.ComponentModel.Activity` class. Microsoft supplies a set of standard activities that you can use, but you are encouraged to also develop your own custom activities. Each activity is designed to serve a unique purpose and encapsulates the logic needed to fulfill that purpose.

For a sequential workflow such as this one, the order of the activities in the workflow determines their execution sequence. A sequential workflow has defined beginning and ending points. As shown in Figure 1-3, previously, these points are represented by the arrow at the top of the workflow and the circle symbol at the bottom. What takes place between these two points is yours to define by dropping activities onto the canvas. Once you've dropped a series of activities onto a workflow, you can modify their execution order by simply dragging them to a new location.

Activities wouldn't be very useful if you couldn't change their default behavior. Therefore, most activities provide a set of properties that can be set at design time. The workflow itself also has properties that can be set at design time.

Figure 1-4 shows just a few of the standard activities that are supplied by Microsoft. I review all of the available standard activities in Chapter 3.

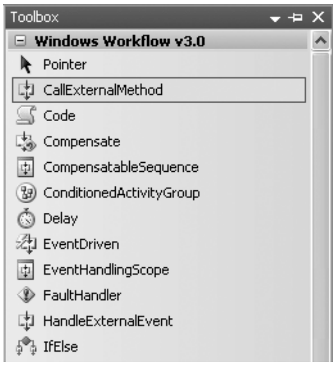


Figure 1-4. *Partial view of standard activities*

The standard activities shown in Figure 1-4 are those that are available in .NET 3.0 (note the Windows Workflow v3.0 title). Activities that are available only with later versions of the framework are segregated into their own section of the Toolbox.

Entering Code

For this simple example, you need to drag and drop the Code activity onto the workflow. The Code activity is a simple way to execute any code that you wish as a step in the workflow. Your workflow should now look like the one shown in Figure 1-5.

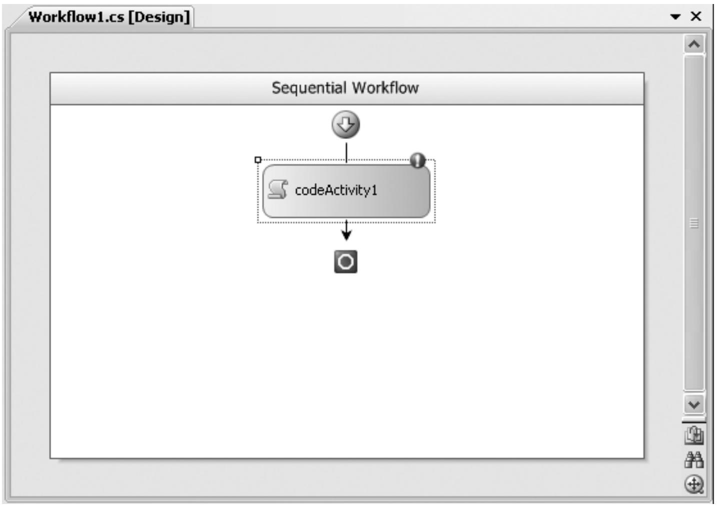


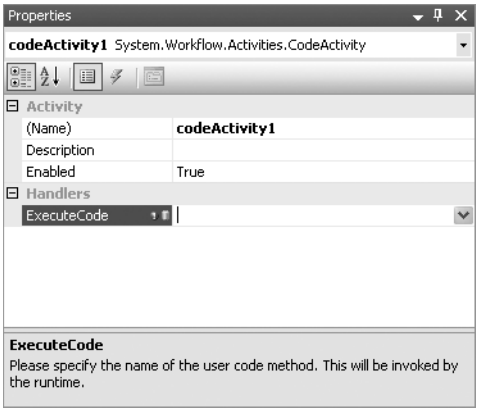
Figure 1-5. *Workflow with single Code activity*

■ **Note** As I discuss in Chapter 3, another way to execute your own code is to implement your own custom activities.

The class name for the Code activity is actually `CodeActivity`. The Toolbox displays a shortened version of the activity name for the standard activities. The single instance of this class is given the default name of `codeActivity1` when it is dropped onto the workflow. Notice the exclamation point at the top of the activity. This is your indication that there are one or more errors associated with this activity. This is typically the case when the activity has required properties that have not been set.

If you click the exclamation point for the `CodeActivity`, you should see an error such as "Property 'ExecuteCode' is not set." To correct this error, you need to set a value for the `ExecuteCode` property. By selecting `codeActivity1` and switching to the Properties window, you'll see the available properties for this activity. The Properties window looks like Figure 1-6.

`ExecuteCode` is actually a public event of the `CodeActivity` class. The property requires the name of an event handler for this event. When the `CodeActivity` is executed as part of a workflow, the `ExecuteCode` event is raised, and the code you place in the handler is executed.



Properties

codeActivity1 System.Workflow.Activities.CodeActivity

Activity	
(Name)	**codeActivity1**
Description	
Enabled	True
Handlers	
ExecuteCode	

ExecuteCode
Please specify the name of the user code method. This will be invoked by the runtime.

Figure 1-6. *Properties window for codeActivity1*

Enter **codeActivity1_ExecuteCode** in the box, and press Enter. You could also double-click the `codeActivity1` in the designer to add a handler for this event. An empty event handler is created for you, and the `Workflow1.cs` file is opened for editing. Listing 1-1 shows the code at this stage. For clarity, I have removed any `using` statements added by the new project template that are not needed for this simple example.

Listing 1-1. *Workflow1.cs File with Empty ExecuteCode Event Handler*

```
using System;
using System.Workflow.Activities;

namespace HelloWorkflow
{
    public sealed partial class Workflow1 : SequentialWorkflowActivity
    {
        public Workflow1()
        {
            InitializeComponent();
        }

        private void codeActivity1_ExecuteCode(object sender, EventArgs e)
        {

        }
    }
}
```

Note As I mentioned previously, `ExecuteCode` is actually an event. This implies that somewhere we should see code that assigns the method `codeActivity1_ExecuteCode` to this event as a handler. That code exists but is not in this source file. Since this class definition includes the `partial` keyword, portions of the class definition can be spread across multiple source files.

If you expand the `Workflow1.cs` file in the Solution Explorer window, you'll see that beneath it there is a source file named `Workflow1.designer.cs`. The `Workflow1.designer.cs` file is the other half of this partial class definition, and it contains the entries placed there by the Visual Studio workflow designer. Included with that code is a statement that assigns the `codeActivity1_ExecuteCode` method as a handler for the `ExecuteCode` event. Microsoft's design allows for a clean separation of designer-controlled code from our code.

Notice that the `Workflow1` class derives from the `SequentialWorkflowActivity` class. This is the base class that is used for sequential workflows. It is interesting to note that `SequentialWorkflowActivity` is indirectly derived from the base `Activity` class. This means that the workflow itself is actually a type of activity. Activities are truly the universal building block of WF.

In order to have this workflow display the obligatory welcome message, you only need to add a call to `Console.WriteLine` like this:

```
private void codeActivity1_ExecuteCode(object sender, EventArgs e)
{
    Console.WriteLine("Hello Workflow!");
}
```

Hosting the Workflow Runtime

Now that the workflow has been implemented, let's turn our attention to the `Program.cs` file included in the project. Listing 1-2 shows the original code generated for this file.

Listing 1-2. *Original Program.cs Generated Code*

```
#region Using directives

using System;
using System.Threading;
using System.Workflow.Runtime;

#endregion

namespace HelloWorkflow
{
    class Program
    {
        static void Main(string[] args)
        {
            using (WorkflowRuntime workflowRuntime = new WorkflowRuntime())
            {
                AutoResetEvent waitHandle = new AutoResetEvent(false);
                workflowRuntime.WorkflowCompleted += delegate(
                    object sender, WorkflowCompletedEventArgs e)
                    {
                        waitHandle.Set();
                    };
                workflowRuntime.WorkflowTerminated += delegate(
                    object sender, WorkflowTerminatedEventArgs e)
                    {
                        Console.WriteLine(e.Exception.Message);
                        waitHandle.Set();
                    };

                WorkflowInstance instance = workflowRuntime.CreateWorkflow(
                    typeof(HelloWorkflow.Workflow1));
                instance.Start();

                waitHandle.WaitOne();
            }
        }
    }
}
```

The purpose of this code is to host the workflow runtime and execute the workflow that you just defined. When you create this project from the Sequential Workflow Console Application template, Microsoft is nice enough to generate this boilerplate code. Without making any changes to the code, you can build and run this project and (hopefully) see the correct results.

But before we do that, let's review this code and make one minor addition. The code starts by creating an instance of the `WorkflowRuntime` class. As the name implies, this is the all-important class that is actually in charge of running the workflow. The class also provides a number of events and methods that permit you to monitor and control the execution of any workflow. The instance of the `WorkflowRuntime` class is wrapped in a `using` code block. This ensures that the instance is properly disposed of when the code exits the scope of this block.

Next, an instance of the `AutoResetEvent` class is created. This is a thread synchronization class that is used to release a single waiting thread. Thread synchronization you say? Exactly what threads do you need to synchronize? When a workflow executes, it does so in a separate thread that is created and managed by the workflow runtime. This makes sense since the workflow runtime is capable of handling multiple workflows at the same time. In this example, the two threads that must be synchronized are the workflow thread and the main thread of the host console application.

In order for the host console application to know when the workflow has completed, the code subscribes to two events of the `WorkflowRuntime` class: `WorkflowCompleted` and `WorkflowTerminated`. It uses the anonymous delegate syntax for this (added in .NET 2.0), implementing the event handler code inline rather than referencing a separate method.

When a workflow completes normally, the `WorkflowCompleted` event is raised, and this code is executed:

```
waitHandle.Set();
```

This signals the `AutoResetEvent` object, which releases the console application from its wait.

If an error occurs, the `WorkflowTerminated` event is raised, and this code is executed:

```
Console.WriteLine(e.Exception.Message);
waitHandle.Set();
```

This displays the error message from the exception and then releases the waiting thread. There are other events that can be used to monitor the status of a workflow, but these are the only two that we need in this example.

Once the workflow runtime has been prepared, an instance of the workflow is created and started with this code:

```
WorkflowInstance instance
    = workflowRuntime.CreateWorkflow(typeof(HelloWorkflow.Workflow1));
instance.Start();
```

The `CreateWorkflow` method has several overloaded versions, but this one simply requires the `Type` of the workflow that you want to create as its only parameter. When a workflow is created, it doesn't begin executing immediately. Instead, a `WorkflowInstance` object is returned from this method and used to start the execution of the workflow.

Finally, the console application suspends execution of the current thread and waits until the `AutoResetEvent` object is signaled with this code:

```
waitHandle.WaitOne();
```

■ **Caution** This workflow is trivial and will execute very quickly. But real-world workflows will require much more time to execute. It is vitally important that the host application waits until the workflow has finished. Otherwise, the host application might terminate before the workflow has had a chance to complete all of its activities.

Running the Application

Before you build and execute this application, add these lines of code to the very end of the `Main` method:

```
Console.WriteLine("Press any key to exit");
Console.ReadKey();
```

This will make it easier to see the results when you execute the project. Without these lines, the console application will execute and then immediately finish, not giving you a chance to see the results.

Now it's time to build and execute the project. You can execute the project by selecting Start Debugging from the Debug menu. Or, if you have the default Visual Studio key mappings still in place, you can simply press F5. If all goes well, you should see these results displayed on the console:

```
Hello Workflow!
Press any key to exit
```

Congratulations! Your first encounter with Windows Workflow Foundation was successful.

Passing Parameters

Workflows would have limited usefulness without the ability to receive parameter input. Passing parameters to a workflow is one of the fundamental mechanisms that permit you to affect the outcome of the workflow.

The preceding example writes a simple string constant to the console. Let's now modify that example so that it uses input parameters to format the string that is written. The parameters will be passed directly from the host console application.

Declaring the Properties

Input parameters can be passed to a workflow as normal .NET Common Language Runtime (CLR) properties. Therefore, the first step in supporting input parameters is to declare the local variables and input properties in the workflow class. You can start with the `Workflow1.cs` file from the previous example and add the following code shown in bold to the `Workflow1` class:

```
public sealed partial class Workflow1 : SequentialWorkflowActivity
{
    /// <summary>
    /// The target of the greeting
    /// </summary>
    public String Person { get; set; }

    /// <summary>
    /// The greeting message
    /// </summary>
    public String Message { get; set; }

    public Workflow1()
    {
        InitializeComponent();
    }

    private void codeActivity1_ExecuteCode(object sender, EventArgs e)
    {
        Console.WriteLine("Hello Workflow!");
    }
}
```

These two properties define the input parameters that are needed for this example. As shown here, I'm using autoimplemented properties that were introduced with C# 2008 and Visual Studio 2008. A private backing field for each property is automatically created by the compiler. Using autoimplemented properties is a convenience, not a requirement. If you prefer (or if you are using Visual Studio 2005), you can manually implement the private backing field and the `get` and `set` accessors.

Next, you can modify the `Console.WriteLine` statement in the `codeActivity1_ExecuteCode` method shown previously to use the new variables like this:

```
private void codeActivity1_ExecuteCode(object sender, EventArgs e)
{
    //display the variable greeting
    Console.WriteLine("Hello {0}, {1}", Person, Message);
}
```

Passing Values at Runtime

The `Program.cs` file requires a few changes in order to pass the new parameters to the workflow. Listing 1-3 shows the revised code for `Program.cs` in its entirety. The additions and changes from the previous example are highlighted in bold.

Listing 1-3. *Program.cs Code to Pass Runtime Parameters*

```
#region Using directives

using System;
using System.Collections.Generic;
using System.Threading;
using System.Workflow.Runtime;

#endregion

namespace HelloWorkflow
{
    class Program
    {
        static void Main(string[] args)
        {
            using (WorkflowRuntime workflowRuntime = new WorkflowRuntime())
            {
                AutoResetEvent waitHandle = new AutoResetEvent(false);
                workflowRuntime.WorkflowCompleted += delegate(
                    object sender, WorkflowCompletedEventArgs e)
                {
                    waitHandle.Set();
                };
                workflowRuntime.WorkflowTerminated += delegate(
                    object sender, WorkflowTerminatedEventArgs e)
                {
                    Console.WriteLine(e.Exception.Message);
                    waitHandle.Set();
                };

                //create a dictionary with input arguments
                Dictionary<String, Object> wfArguments
                    = new Dictionary<string, object>();
                wfArguments.Add("Person", "Bruce");
                wfArguments.Add("Message", "did the workflow succeed?");

                WorkflowInstance instance
                    = workflowRuntime.CreateWorkflow(
                        typeof(HelloWorkflow.Workflow1), wfArguments);

                instance.Start();

                waitHandle.WaitOne();

                Console.WriteLine("Press any key to exit");
                Console.ReadKey();
            }
        }
    }
}
```

Input parameters are passed to a workflow as a generic `Dictionary` object. The `Dictionary` must be keyed by a `String` and use an `Object` as the value for each parameter entry. Once the `Dictionary` object is created, you can use the Add method to specify the value for each parameter that the workflow expects.

Notice that the key to each parameter in the Dictionary is an exact match to one of the workflow public properties. This is not an accident. In order for the parameters to make their way into the workflow, the names must match exactly, including their case. Likewise, the Type of the parameter value must match the expected Type of the property. In this example, both properties are of type String; therefore, a string value must be passed in the Dictionary.

Note If you do manage to misspell a parameter name, you'll receive an ArgumentException during execution. For example, if you attempt to pass a parameter named person (all lowercase) instead of Person, you'll receive an ArgumentException with the message "The workflow 'Workflow1' has no public writable property named 'person'." If the parameter name is correct but the value is of the incorrect type, the message will be "Invalid workflow parameter value."

The only other required change is to actually pass the Dictionary object to the workflow. This is accomplished with an overloaded version of the CreateWorkflow method, as shown here:

```
WorkflowInstance instance
    = workflowRuntime.CreateWorkflow(
        typeof(HelloWorkflow.Workflow1), wfArguments);
```

When this example application is executed, the results indicate that the parameters are successfully passed to the workflow:

```
Hello Bruce, did the workflow succeed?
Press any key to exit
```

Making Decisions

As you've already seen, workflows use a declarative programming model. Much of your time defining a workflow will be occupied with organizing activities into their proper sequence and setting properties. The real business logic is still there in one form or another. But you can now declare the flow of control between individual activities in the workflow instead of in your code. This separation between your business logic and the rules that control it make workflows a flexible tool for defining complex business problems.

In this next example, you'll see one way to implement simple decisions in a workflow.

Creating a Workflow Library

This example implements a simple calculator using a workflow. The user interface is a Windows Forms application, and the workflow is implemented in a separate assembly.

Note The purpose of this example isn't to demonstrate the best way to implement a calculator. You could easily develop a self-contained calculator application without a workflow in less code. Instead, this example illustrates the use of a workflow to declare the flow of control and to make simple decisions.

Let’s start by creating the workflow project. You can follow the same steps that you used to create the project for the previous examples. However, this time select the Sequential Workflow Library project template. This creates a DLL assembly that supports workflows. Name the project SimpleCalculatorWorkflow. Once the project has been created, you should see the same initial workflow canvas shown in Figure 1-3.

Adding Workflow Properties

This calculator application is extremely simple, but it does support the basic operations of add (+), subtract (-), multiply (x), and divide (/). One of these operations is performed each time the calculator workflow is executed. Each operation works with two integers that are passed in to the workflow. The result of the operation is returned as a Double.

The first step in implementing this workflow is to define these parameters as public properties of the Workflow1 class. The code in Listing 1-4 shows the contents of the Workflow1.cs file after the addition of these properties.

Listing 1-4. *Workflow1.cs File with Calculator Workflow Properties*

```
using System;
using System.Workflow.Activities;

namespace SimpleCalculatorWorkflow
{
    public sealed partial class Workflow1 : SequentialWorkflowActivity
    {
        public String Operation { get; set; }
        public Int32 Number1 { get; set; }
        public Int32 Number2 { get; set; }
        public Double Result { get; set; }

        public Workflow1()
        {
            InitializeComponent();
        }

    }

}
```

The Operation property is used to identify the type of operation and must be one of the valid operation strings (+, -, x, /). As you’ll see next, this property is used as the basis for some simple decisions within the workflow. The other properties are the two numbers and the result used by each calculation.

Adding IfElse Activities

The workflow must make one very simple decision. It must determine the type of arithmetic operation that is requested and then perform it. To accomplish this, you can use the IfElseActivity.

Open the workflow designer for the Workflow1.cs file, and drag the activity labeled IfElse from the Toolbox. After dropping it onto the workflow, the designer should look like Figure 1-7.

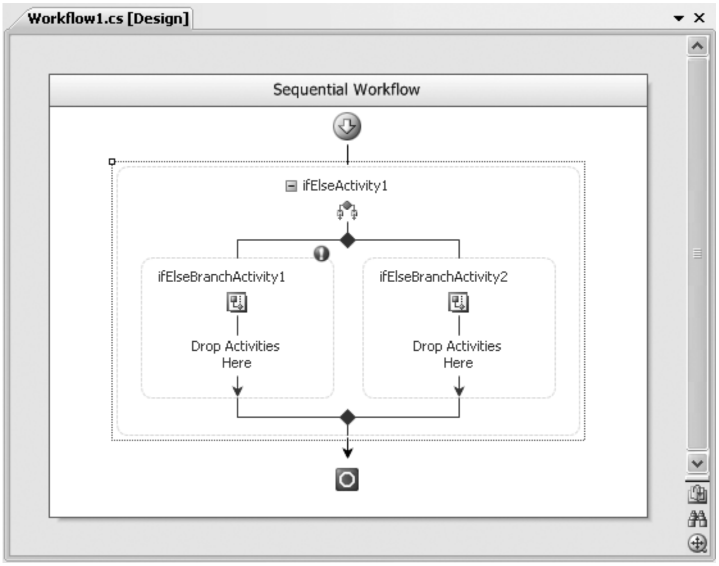


Figure 1-7. *Initial view of IfElse activity*

The `IfElseActivity` is actually a container for other activities. It contains a set of two or more `IfElseBranchActivity` instances. Each one represents a possible branch in a decision tree. The order of the branches is important since the first branch with a `true` condition is the one that is followed. The evaluation of branches starts with the leftmost branch.

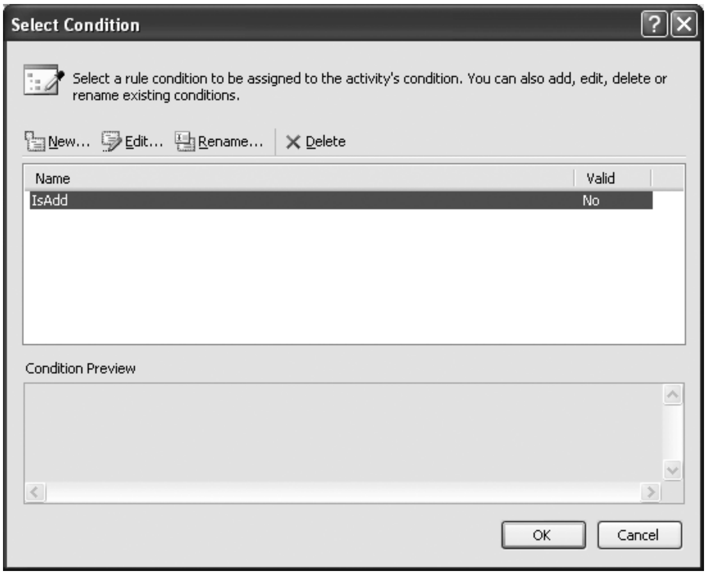
As shown in Figure 1-7, the first two branches are already added for you. As you'll soon see, you're not limited to just those first two branches. You can add as many branches as your workflow requires.

The first branch represents the add operation of the calculator, and it will check the `Operation` property for the value +. To enter the first condition, highlight the leftmost `IfElseBranchActivity` class, and switch to the Properties window. First, change the `Name` property of the activity to something meaningful such as `ifElseBranchActivityIsAdd`. Although renaming the activity isn't a strict requirement, a descriptive name makes it easier to quickly identify an activity in the workflow designer.

There are two ways to define a condition. You can use a Code Condition or a Declarative Rule Condition. The Code Condition is similar to the `CodeActivity` that you've already seen. It works by writing code within an event handler that evaluates to a `Boolean` result. The Declarative Rule Condition must also evaluate to a `Boolean`, but it is defined separately from the workflow code. One benefit of rule conditions is that they can be modified at runtime without the need to recompile any code. Since code conditions are just that, code, they require a recompile in order to make any modifications.

For this example, select Declarative Rule Condition from the list of available values in the `Condition` property.

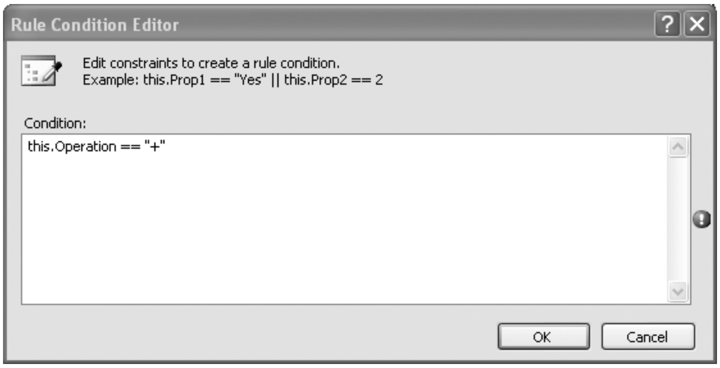
After selecting Declarative Rule Condition, expand the `Condition` property to reveal additional properties for the rule. Enter a `ConditionName` of `IsAdd` as a descriptive name for this rule. Each rule must have a unique name. Next, after selecting the `ConditionName` property, click the ellipsis. This presents the Select Condition dialog shown in Figure 1-8.


Select Condition
Select a rule condition to be assigned to the activity's condition. You can also add, edit, delete or rename existing conditions.
New...
Edit...
Rename...
Delete
Condition Preview
OK
Cancel

Name	Valid
IsAdd	No

Figure 1-8. *Select Condition dialog*

From this dialog, you can select, edit, rename, or delete an existing Declarative Rule Condition or create a new one. In this case, a new empty rule condition has already been created because you provided a condition name of `IsAdd` before entering this dialog. You now need to click the Edit button to modify this empty rule condition. The Rule Condition Editor is then presented, as shown in Figure 1-9.


Rule Condition Editor
Edit constraints to create a rule condition.
Example: this.Prop1 == "Yes" || this.Prop2 == 2
OK
Cancel

Condition:

```
this.Operation == "+"
```

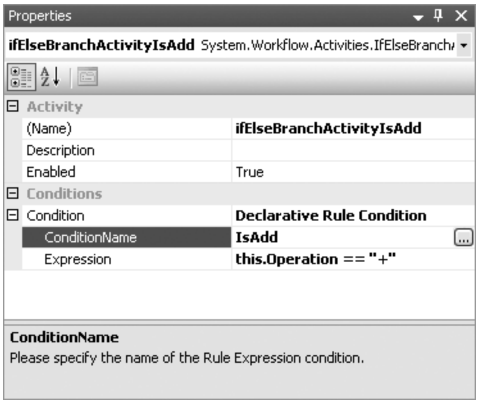
Figure 1-9. *Rule Condition Editor with the first rule*

This dialog is where you enter your `Boolean` expression. As shown in Figure 1-9, I've already entered this rule condition:

```
this.Operation == "+"
```

The Rule Condition Editor has IntelliSense support; therefore, you can enter `this` followed by a period, and a list of the available workflow properties will be presented.

This expression will be evaluated when the workflow is executed and a simple `true`/`false` result returned. If the result is `true`, then any other activities you add to this `IfElseBranchActivity` will be executed. If `false` is returned from the rule, the next `IfElseBranchActivity` is evaluated, and so on. After you click OK, the dialog is closed, and you are returned to the Select Condition dialog. Click OK once again to return to the Properties window, which should look like Figure 1-10.


Properties
ifElseBranchActivityIsAdd System.Workflow.Activities.IfElseBranch
Activity

(Name)	ifElseBranchActivityIsAdd
Description	
Enabled	True
Conditions	
Condition	Declarative Rule Condition
ConditionName	IsAdd
Expression	this.Operation == "+"

ConditionName
Please specify the name of the Rule Expression condition.

Figure 1-10. *IfElseBranchActivity rule condition properties*

You'll have a total of four rule conditions to add, one for each possible arithmetic operation that the workflow will support. You've just added the `IsAdd` condition to check for `+`. Now repeat the process and add a rule condition to the second `IfElseBranchActivity` to check for the subtraction operation (`-`). This is also a good time to rename the second `IfElseBranchActivity` to `ifElseBranchActivityIsSubtract`. Enter `IsSubtract` as the `ConditionName`, and enter the expression like this:

```
this.Operation == "-"
```

You need to define three additional `IfElseBranch` activities: two for multiply and divide plus one additional branch. To add a new `IfElseBranchActivity`, right-click the `ifElseActivity1` object in the designer and select Add Branch. This adds a new branch to the right of any existing branches. Do this three times. If the rightmost branch doesn't include a conditional expression, it is executed if none of the other branch conditions evaluates to `true`. As such, it acts as the final `else` in your decision tree.

■ **Note** You can add a new `IfElseBranchActivity` in two other ways. After selecting the `ifElseActivity1` object, you can select Add Branch from the top-level Workflow menu. If you switch to the Properties window, you can also select Add Branch as one of the available shortcut options at the bottom of the window. If you don't see the commands at the bottom of the Properties window, it is likely they have been switched off. To enable them, right-click anywhere in the Properties window, and make sure Commands is checked.

Once you've added the branches, add the multiply and divide rule conditions. The rule expression for multiply looks like this:

```
this.Operation == "x"
```

The expression for divide looks like this:

```
this.Operation == "/"
```

Adding Calculation Logic

Each IfElseBranchActivity in the workflow represents a separate arithmetic operation. When one of the rule conditions that you just entered is true, any activities within that IfElseBranchActivity are executed. The branches are evaluated one at a time, starting at the left side.

To implement the code for an operation, you can drag and drop a CodeActivity from the Toolbox to the first IfElseBranchActivity container. Drop it in the empty area in the container labeled Drop Activities Here. Now switch to the Properties window for the new CodeActivity and enter a name of AddOperation in the ExecuteCode property. After pressing Enter, an event handler with a name of AddOperation is added to the workflow. Add a line of code to the handler that performs an addition operation on the two numbers. The modified code looks like this:

```
private void AddOperation(object sender, EventArgs e)
{
    Result = Number1 + Number2;
}
```

Repeat these steps for each of the other operations: subtract, multiply, and divide. You also want to throw an exception if an unknown operation is requested. To do this, add a CodeActivity to the final branch, set its ExecuteCode property to UnknownOperation, and add code that looks like this:

```
private void UnknownOperation(object sender, EventArgs e)
{
    throw new ArgumentException(String.Format(
        "Invalid operation of {0} requested", Operation));
}
```

If you followed all of the steps correctly, the workflow should look like Figure 1-11.

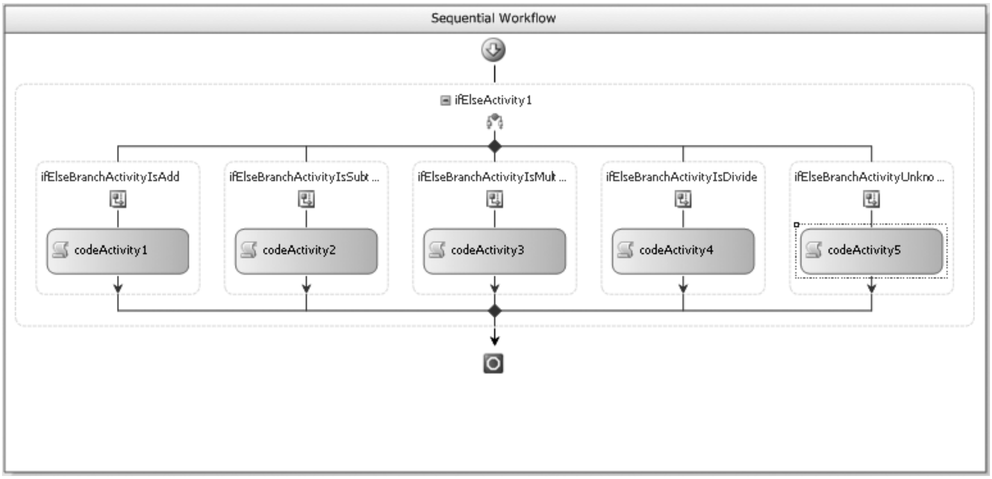

Sequential Workflow
ifElseActivity1
ifElseBranchActivityIsAdd
ifElseBranchActivityIsSubt ...
ifElseBranchActivityIsMult ...
ifElseBranchActivityIsDivide
ifElseBranchActivityUnkno ...
codeActivity1
codeActivity2
codeActivity3
codeActivity4
codeActivity5

Figure 1-11. *Workflow with multiple IfElseBranchActivity branches*

Listing 1-5 shows the completed code for Workflow1.cs.

Listing 1-5. *Complete Workflow1.cs File*

```
using System;
using System.Workflow.Activities;

namespace SimpleCalculatorWorkflow
{
    public sealed partial class Workflow1 : SequentialWorkflowActivity
    {
        public String Operation { get; set; }
        public Int32 Number1 { get; set; }
        public Int32 Number2 { get; set; }
        public Double Result { get; set; }

        public Workflow1()
        {
            InitializeComponent();
        }

        /// <summary>
        /// Add the numbers
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void AddOperation(object sender, EventArgs e)
        {
            Result = Number1 + Number2;
        }

        /// <summary>
        /// Subtract the numbers
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void SubtractOperation(object sender, EventArgs e)
        {
            Result = Number1 - Number2;
        }

        /// <summary>
        /// Multiply the numbers
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void MultiplyOperation(object sender, EventArgs e)
        {
            Result = Number1 * Number2;
        }

        /// <summary>
        /// Divide the numbers
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void DivideOperation(object sender, EventArgs e)
        {
            if (Number2 != 0)
            {
                Result = (Double)Number1 / (Double)Number2;
            }
            else
            {
                Result = 0;
            }
        }

        /// <summary>
        /// Handle invalid operation
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void UnknownOperation(object sender, EventArgs e)
        {
            throw new ArgumentException(String.Format(
                "Invalid operation of {0} requested", Operation));
        }
    }
}
```

Creating the Calculator Client

Now that the workflow is complete, you need a client application that uses it. Create a new project as you've done in the past. This time, choose Windows Forms Application as the project type, and name the application `SimpleCalculator`.

Add a project reference to the `SimpleCalculatorWorkflow` project containing the workflow. Since this Windows application will be hosting the workflow runtime, you'll need to also add these assembly references:

- `System.Workflow.Activities`
- `System.Workflow.ComponentModel`
- `System.Workflow.Runtime`

■ **Note** You need to manually add these assembly references since you started with a Windows Forms Application project instead of one of the workflow project templates. The workflow project templates would have added these assembly references for you.

Add a `Button` object to the form for each number (0 to 9) and each operation (+, -, ×, /). Also include a `Button` for clear (C) and equals (=). You'll also need a `TextBox` to display the numbers as they are entered and to display the result. The final application should look something like Figure 1-12.

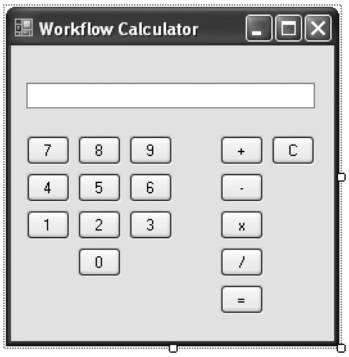


Figure 1-12. *Calculator client application*

Next, you'll need to attach event handlers to the `Click` event of each button. However, there really are only four different event handlers that you'll need to add:

- `NumericButton_Click`: Assign this handler to all of the number buttons (0 to 9).
- `OperationButton_Click`: Assign this handler to all of the operation buttons (+, -, ×, /).
- `Equals_Click`: Assign this handler to the equals button (=).
- `Clear_Click`: Assign this handler to the clear button (C).

Listing 1-6 shows the completed code for the `Form1.cs` file.

Listing 1-6. *Form1.cs Windows Calculator Application*

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Windows.Forms;
using System.Workflow.Runtime;

namespace SimpleCalculator
{
    public partial class Form1 : Form
    {
        private WorkflowRuntime _workflowRuntime;
        private AutoResetEvent _waitHandle = new AutoResetEvent(false);
        private String _operation = String.Empty;
        private Int32 _number1;
        private Int32 _number2;
        private Double _result;

        public Form1()
        {
            InitializeComponent();

            //start up the workflow runtime. must be done
            //only once per AppDomain
            InitializeWorkflowRuntime();
        }
```

```
/// <summary>
/// Start the workflow runtime
/// </summary>
private void InitializeWorkflowRuntime()
{
    _workflowRuntime = new WorkflowRuntime();
    _workflowRuntime.WorkflowCompleted
        += delegate(object sender, WorkflowCompletedEventArgs e)
        {
            _result = (Double)e.OutputParameters["Result"];
            _waitHandle.Set();
        };
    _workflowRuntime.WorkflowTerminated
        += delegate(object sender, WorkflowTerminatedEventArgs e)
        {
            MessageBox.Show(String.Format(
                "Workflow Terminated: {0}", e.Exception.Message),
                "Error in Workflow");
            _waitHandle.Set();
        };
}
```

During construction of the form, the `InitializeWorkflowRuntime` method is executed. Within this method the workflow runtime is initialized in a similar manner to the previous examples.

One additional requirement for this example is to retrieve the `Result` property when the workflow completes. This property contains the result of the requested calculation. The code to retrieve this value is in the `WorkflowCompleted` event handler for the `WorkflowRuntime` object. When this event is raised by the workflow runtime, the `Result` property is saved in a form member variable. This member variable is later used to update the user interface with the result.

```
private void NumericButton_Click(object sender, EventArgs e)
{
    txtNumber.AppendText(((Button)sender).Text.Trim());
}

private void Clear_Click(object sender, EventArgs e)
{
    Clear();
}

/// <summary>
/// An operation button was pressed
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void OperationButton_Click(object sender, EventArgs e)
{
    try
    {
        _number1 = Int32.Parse(txtNumber.Text);
        _operation = ((Button)sender).Text.Trim();
        txtNumber.Clear();
    }
```

```
            catch (Exception exception)
            {
                MessageBox.Show(String.Format(
                    "Operation_Click error: {0}",
                    exception.Message));
            }
        }

        /// <summary>
        /// The equals button was pressed. Invoke the workflow
        /// that performs the calculation
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void Equals_Click(object sender, EventArgs e)
        {
            try
            {
                _number2 = Int32.Parse(txtNumber.Text);

                //create a dictionary with input arguments
                Dictionary<String, Object> wfArguments
                    = new Dictionary<string, object>();
                wfArguments.Add("Number1", _number1);
                wfArguments.Add("Number2", _number2);
                wfArguments.Add("Operation", _operation);

                WorkflowInstance instance
                    = _workflowRuntime.CreateWorkflow(
                        typeof(SimpleCalculatorWorkflow.Workflow1),
                            wfArguments);
                instance.Start();

                _waitHandle.WaitOne();

                //display the result
                Clear();
                txtNumber.Text = _result.ToString();
            }
            catch (Exception exception)
            {
                MessageBox.Show(String.Format(
                    "Equals error: {0}", exception.Message));
            }
        }
```

The `Equals_Click` event handler is where the workflow is created and started. As you saw in Listing 1-3, the input parameters are passed to the workflow as a generic `Dictionary` object. When the object referenced by the `_waitHandle` variable signals that the workflow has completed, the result that is saved in the `WorkflowCompleted` event handler is used to update the user interface.

```
        private void Clear()
        {
            txtNumber.Clear();
            _number1 = 0;
            _number2 = 0;
            _operation = String.Empty;
        }

        /// <summary>
        /// Executed during app shutdown
        /// </summary>
        /// <param name="e"></param>
        protected override void OnFormClosing(FormClosingEventArgs e)
        {
            base.OnFormClosing(e);
            if (_workflowRuntime != null)
            {
                _workflowRuntime.StopRuntime();
                _workflowRuntime.Dispose();
                _workflowRuntime = null;
            }
        }
    }
}
```

The override for the `OnFormClosing` method is executed when the form is closing and the application is shutting down. The code in this method stops the workflow runtime in an orderly way, first calling the `StopRuntime` method and then calling `Dispose`. Calling these methods isn't absolutely required, but it does force an immediate release of any resources held by the workflow runtime. If you don't call `Dispose` yourself, the garbage collector will free the resources eventually.

■**Note** Remember that starting with .NET 2.0 and Visual Studio 2005, Windows Forms applications make use of partial classes to separate your code from the code that is maintained by the forms designer. For this reason, you won't see the code that creates the visual elements for the form (`Button` and `TextBox`). You also won't see the code that adds the event handlers to the `Button.Click` event. All of that code is in the `Form1.Designer.cs` file, not in the `Form1.cs` file shown in Listing 1-6.

Testing and Debugging the Calculator

To test the workflow calculator, you build and run the `SimpleCalculator` project. All operations work in the expected way. You enter a number, select an operation (+, -, ×, /), enter the second number, and then click the equals (=) button to see the result. If everything has been implemented correctly, you should see correct results for each type of arithmetic operation.

For example, if you enter **123 + 456 =**, you should see the result shown in Figure 1-13.

If you want to step into the code during execution, you can do so in the normal Visual Studio way. You place a breakpoint on a line of code, and the Visual Studio debugger will break execution at that point.

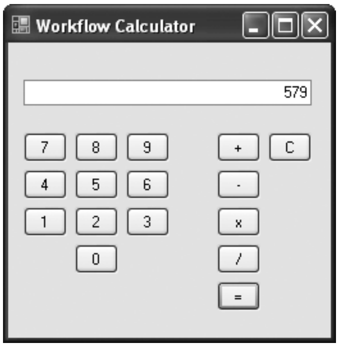


Figure 1-13. *Working workflow calculator*

In addition to the standard way to set a breakpoint in code, you can also set a breakpoint within the workflow designer. To see this, open the designer view of the `Workflow1.cs` file, and make sure the entire workflow is selected. Then press F9 to set a breakpoint for the entire workflow. You can also add a breakpoint from the Debug menu or right-click the workflow and select Breakpoint.

Now when you run the `SimpleCalculatorWorkflow` project in Debug (F5), the debugger should break when execution of the workflow begins. You can step over workflow activities (F10) or step into (F11) any code in the workflow such as the code activities. Either way, the debugger provides you with a good way to determine which workflow branches are executed.

USING THE WORKFLOW DEBUGGER

The workflow debugger works only if the project that you start is the one containing the workflow. Normally, when testing an application such as this simple calculator, you would start the Windows application, not one of the referenced assemblies. Starting the Windows application doesn't support use of the workflow debugger.

To see the workflow debugger in action, set the `SimpleCalculatorWorkflow` project as your startup project in Visual Studio. Next, open the project properties, and select the Debug tab. The default Start Action is Start Project. Change this to Start External Program, and select `SimpleCalculator.exe` as the startup program. An ellipsis is available that allows you to browse directly to the folder containing this EXE.

Now when you start the `SimpleCalculatorWorkflow` project in Debug mode, the simple calculator application is launched. The workflow debugger is now enabled and will stop execution on any breakpoints that you've set.

One other interesting and useful feature of the workflow designer is its ability to enable and disable individual activities. For example, right-click the leftmost `IfElseBranchActivity` in the workflow, and select Disable from the context menu. The activity is now shown with a shaded background as a visual cue that it is disabled. This is illustrated in Figure 1-14.

If you run the `SimpleCalculator` project now, what results should you expect? The activity that you disabled is the one that handled the addition operation (+). Therefore, if you try to add two numbers, you'll receive the error dialog shown in Figure 1-15.

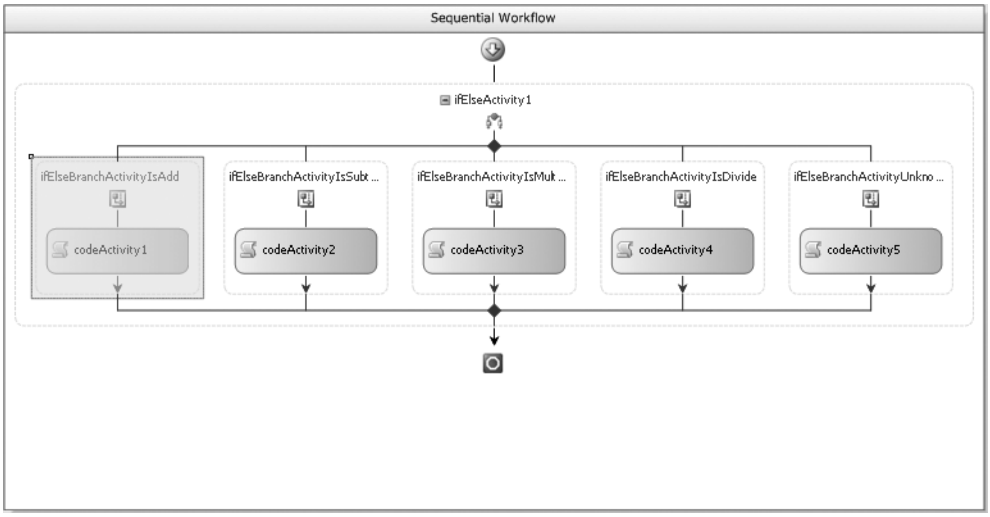
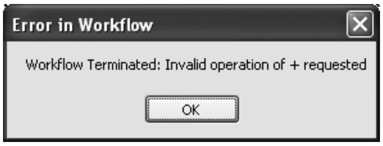

Sequential Workflow
ifElseActivity1
ifElseBranchActivityIsAdd
ifElseBranchActivityIsSubt ...
ifElseBranchActivityIsMult ...
ifElseBranchActivityIsDivide
ifElseBranchActivityUnkno ...
codeActivity1
codeActivity2
codeActivity3
codeActivity4
codeActivity5

Figure 1-14. *Workflow with disabled activity*



Workflow Terminated: Invalid operation of + requested

Figure 1-15. *Error dialog when activity is disabled*

Since the branch that checked for the addition operation is now disabled, the only branch that can execute is the rightmost branch that acts as a catchall. The `CodeActivity` in that branch is responsible for throwing the exception shown in Figure 1-15. If you perform any other operation, the calculator still works correctly because those branches are enabled. If you enable the addition branch again, all operations will be back to normal.

This ability to selectively enable and disable individual activities illustrates some of the flexibility you have with a workflow. You've moved the decision tree for this application out of the code and into a workflow. This technique of disabling activities comes in handy during the initial testing and debugging of your workflow applications.

Summary

The purpose of this chapter was to provide you with a quick tour of Windows Workflow Foundation. You started by implementing your first workflow application. This simple example introduced you to the visual workflow designer, workflow activities, the `CodeActivity`, and the workflow runtime. You then saw how to pass parameters to a workflow in the second example. Finally, you developed a simple calculator application that declares its decision tree using the `IfElseActivity` and implements its calculation logic in a workflow. Along the way you were also introduced to rule conditions, output parameters, and the workflow debugger.

In the next chapter, you'll learn about the major components in Windows Workflow Foundation and see how they work together.