Pro WF

Windows Workflow in .NET 4

Bruce Bukovics

Apress®

Pro WF: Windows Workflow in .NET 4

ISBN-13 (pbk): 978-1-4302-2721-2

ISBN-13 (electronic): 978-1-4302-2722-9

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

President and Publisher: Paul Manning
Lead Editor: Ewan Buckingham, Matt Moodie
Technical Reviewer: Matt Milner
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Coordinating Editor: Jim Markham
Copy Editor: Kim Wimpsett
Production Support: Patrick Cunningham
Indexer: Brenda Miller
Artist: April Milne
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media, LLC., 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 `www.springeronline.com`.

For information on translations, please e-mail `rights@apress.com`, or visit `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 `www.apress.com/info/bulksales`.

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

For Teresa and Brennan

Contents at a Glance

Contents

About the Author

Bruce Bukovics has been a working developer for more than 25 years. During this time, he has designed and developed applications in such widely varying areas as banking, corporate finance, credit card processing, payroll processing, and retail systems.

He has firsthand developer experience with a variety of languages, including C, C++, Delphi, Java, Visual Basic, and C#. His design and development experience began back in the mainframe days and includes client/server, distributed n-tier, and service-oriented applications.

He considers himself a pragmatic programmer and test-driven development evangelist. He doesn't always stand on formality and is willing to look at alternate or unorthodox solutions to a problem if that's what it takes.

He is currently employed at Radiant Systems Inc. in Alpharetta, Georgia, as a senior software architect in the central technology group.

About the Technical Reviewer

■ **Matt Milner** is a member of the technical staff at Pluralsight, where he focuses on connected systems technologies (WCF, Windows WF, BizTalk, AppFabric, and Windows Azure). Matt is also an independent consultant specializing in Microsoft .NET application design and development. As a writer, Matt has contributed to several journals and magazines, including MSDN Magazine where he authored the workflow content for the *Foundations* column. Matt regularly shares his love of technology by speaking at local, regional, and international conferences such as TechEd. Microsoft has recognized Matt as an MVP for his community contributions around connected systems technology.

Acknowledgments

As usual, a number of people deserve my appreciation. At the top of the list are my wife, Teresa, and my son, Brennen. While I was spending every available hour working on this book, you were both going about your day-to-day lives without me. I'm sorry about that. Thank you for being patient with me and supporting me while I finished this project. I love you both very much.

A big thank-you also goes out to Matt Milner, the technical reviewer for this book. Matt's job was basically to keep me honest. He reviewed each chapter and had the tedious job of executing all of my example code to ensure that it ran correctly. Matt also directed my attention to areas that I might have missed and provided valuable suggestions that improved the quality of this book.

The Apress team once again did an outstanding job—this is my fourth book with them. Matthew Moodie was the editor for my last book and once again stepped in to work with me on this one. He did another superb job providing guidance and suggestions that improved the overall quality of this book. Thanks also go to Ewan Buckingham who was there to provide his editorial guidance at just the right moments in the project.

James Markham was the coordinating editor on the project. That means he was the traffic cop who managed the schedules and directed files to and from the rest of the team. Great job, James; thank you for your work on this book. Thanks also go to Fran Parnell who served as the original coordinating editor before James transitioned to the team. I was very fortunate to have Kim Wimpsett as my copy editor once again. She worked on my last book, and I requested her early in this project. Thank you, Kim, for an excellent job. You once again corrected my many errors without dramatically changing my written voice. Production has my appreciation for their fine formatting work on this book.

For this book, I was fortunate to have access to additional Microsoft development resources that were not available to me for my previous books. Foremost among these resources was Ed Hickey. Ed was the Microsoft program coordinator for the Connect program that I joined and my central contact for all things Microsoft. On more than one occasion I contacted Ed with a problem, and he always followed through by contacting just the right Microsoft developer. Thank you, Ed.

I also need to thank the Microsoft (and non-Microsoft) folks who frequented the private Microsoft WF 4 forum. These folks addressed my questions, comments, suggestions, and bug reports and patiently tried to explain how WF 4 really worked without the benefit of any formal public documentation. I'm sure I've missed some names, but these folks went the extra mile to make sure that my questions were addressed: Scott Mason, Justin Brown, Nate Talbert, Ed Pinto, Matt Winkler, and Dave Cliffe. And thanks also go out to Maurice de Beijer who often pointed me in the right direction went I went astray. On more than one occasion, it seemed as though Maurice and I had the forum to ourselves and were asking similar questions.

I continue to receive many positive comments from readers of my previous WF books. Many of you write to me with questions that I try to answer, but others simply write to let me know how much they enjoyed one of my books. Thank you for your continued support. I hope you enjoy this latest edition.

Introduction

I started working with Windows Workflow Foundation (WF) in 2006 during the early beta and Community Technology Preview (CTP) stages. WF became a shipping Microsoft product named .NET Framework 3.0 in November 2006 along with Windows Presentation Foundation (WPF) and Windows Communication Foundation (WCF). I actually started to learn and use all three of these foundations at the same time in my day job.

While I was impressed with the flexibility and capabilities of WPF and WCF, I was somehow inexplicably drawn to Windows Workflow Foundation (WF). WF isn't just a new way to implement a user interface or a new way to communicate between applications and services. WF represents a completely new way to develop applications. It is declarative, visual, and infinitely flexible. It promotes a model that cleanly separates *what* to do from *when* to do it. This separation allows you to change the *when* without affecting the *what*. Business logic is implemented as a set of discrete, testable components that are assembled into workflows like building blocks.

Workflow isn't a new concept. But when Microsoft spends years developing a workflow foundation and provides it to us without cost, it is an event worth noting. Other workflow frameworks exist, but since it is included in the .NET Framework, WF is the de facto standard workflow framework for Windows applications.

This is the third edition of this book. The first two editions targeted the version of WF that shipped with the .NET Framework 3.0 and 3.5, respectively. This book targets the all-new version 4 of WF, which has been completely redesigned and rewritten. If you are using the 3.*x* version of WF, this is not the book for you—you need my book *Pro WF: Windows Workflow in .NET 3.5*, also published by Apress.

I originally wrote the first edition of this book because I was excited about WF. I was excited about the opportunities that it held for application developers like us. I'm even more excited today, since Microsoft has listened to the feedback and given us a completely new and greatly improved workflow framework.

My hope is that this book will help you use WF to build an exciting new generation of workflow-enabled applications.

Who Should Read This Book

This book is for all .NET developers who want to learn how to use Windows Workflow Foundation version 4 in their own applications. This book is not a primer on .NET or the C# language. To get the most out of the examples that I present in this book, you need a good working knowledge of the .NET Framework. All of the examples are presented in C#, so you should be proficient with C#.

An Overview of This Book

The material in this book is a WF 4 tutorial presented in 18 chapters, with each chapter building upon the ones before it. I've tried to organize the material so that you don't have to jump ahead in order to

understand how something works. But since the chapters build upon each other, I do assume that you have read each chapter in order and understand the material that has already been presented.

The short sections that follow provide a brief summary of each chapter.

Chapter 1: A Quick Tour of Windows Workflow Foundation

This chapter provides a brief introduction to WF. In this chapter, you jump right in and develop your first workflow ("Hello Workflow"). You are introduced to some of the fundamental concepts of WF, such as how to pass parameters to a workflow and how to make decisions within a workflow.

Chapter 2: Foundation Overview

The goal of this chapter is to provide a high-level overview of WF in its entirety. This chapter doesn't teach you how to use each individual WF feature, but it does acquaint you with the design-time and runtime features that are available with WF. This chapter is a road map for the material that is covered in the remainder of the book.

Chapter 3: Activities

Activities are the building blocks of WF and where you place the business logic that is specific to your particular problem domain. In this chapter, you will learn how to develop your own custom activities using the base classes that ship with WF. This chapter also provides a high-level review of the standard activities that are provided with WF.

Chapter 4: Workflow Hosting

WF is not a stand-alone application. It is a framework for building your own workflow-enabled applications. This chapter demonstrates how to host and execute workflows in your own application. It describes how to use each of the hosting classes that are supplied with WF.

Chapter 5: Procedural Flow Control

WF includes support for two different workflow modeling styles out of the box: procedural and flowchart. The modeling style determines how the flow of control between individual activities is modeled. The focus of this chapter is the procedural modeling style. It uses familiar programming constructs to control the flow of execution.

Chapter 6: Collection-Related Activities

This chapter focuses on the activities that enable you to work with collections of data. WF includes standard activities that iterates over each element in a collection, executing the same activity for each element. Also included in WF are a set of activities that allow you to manipulate collections, adding and removing elements and so on.

Chapter 7: Flowchart Modeling Style

The other workflow modeling style that is supported by WF is the flowchart modeling style. This style of modeling workflows enables you to use direct links between activities to control the flow of execution. In this chapter, I review the activities that are provided with WF to support this modeling style. After explaining how to model a workflow using this style, I revisit several examples that were presented in earlier chapters. This is done to contrast how the two modeling styles (procedural and flowchart) can be used to solve similar business problems.

Chapter 8: Host Communication

This chapter focuses on direct communication between the host application and a workflow instance. The chapter provides an overview of long-running workflows and the bookmark mechanism used to implement them. The use of workflow extensions for sending data to a host application is also discussed. The classes that support a general-purpose callback mechanism are also demonstrated.

Chapter 9: Workflow Services

This chapter focuses on the Windows Communication Foundation (WCF) support that is provided with WF. Included with this support is the ability to declaratively author WCF services using WF as well as to invoke WCF services from within a workflow.

Chapter 10: Workflow Services Advanced Topics

This chapter continues coverage of the WCF support that is provided by WF. The chapter expands on this basic example from Chapter 9 by implementing additional workflow services that are consumed by the original workflow. Context-based and Content-based correlation is demonstrated, along with the duplex message exchange pattern. The chapter concludes with a discussion of exception and fault processing, flowing transactions into a workflow service, and the use of standard WF behaviors to fine-tune workflow service performance.

Chapter 11: Workflow Persistence

An important capability of WF is the ability to persist workflow instances (save and reload them at a later time). The chapter focuses on how to enable workflow persistence in your applications. The built-in support for persistence to a SQL Server database is demonstrated in this chapter.

Chapter 12: Customizing Workflow Persistence

This chapter focuses on ways to extend or customize workflow persistence and continues the discussion that began in Chapter 11. The chapter also provides an example that implements a custom instance store that persists workflow instances to the file system rather than to a database.

Chapter 13: Transactions, Compensation, and Exception Handling

This chapter focuses on the mechanisms provided by WF to support the handling of exceptions and to ensure the consistency of work that is performed within a workflow. Exception handling techniques, transactions and compensation are all demonstrated.

Chapter 14: Workflow Tracking

Workflow tracking is a built-in mechanism that automatically instruments your workflows. By simply adding a tracking participant to the workflow runtime, you are able to track and record status and event data related to each workflow and each activity within a workflow. This chapter shows you how to use the built-in support for tracking and how to use tracking profiles to filter the type of tracking data that is produced. The chapter also demonstrates how to develop your own custom tracking participants to process the tracking data.

Chapter 15: Enhancing the Design Experience

In this chapter, you learn how to create custom activity designers. These designer components provide the visible representation of an activity on the workflow designer canvas. The chapter also demonstrates several ways to implement validation logic for activities.

Chapter 16: Advanced Custom Activities

This chapter focuses on several advanced custom activity scenarios. Most of these scenarios are related to the execution of one or more children. The chapter demonstrates how to develop your own custom activities that execute one or more child activities or invoke a callback delegate. Also demonstrated are the techniques for providing the metadata that WF requires for each activity. The chapter concludes with an example that demonstrates the use of execution properties and bookmark options.

Chapter 17: Hosting the Workflow Designer

The workflow designer is not limited to use only within the Visual Studio environment. WF provides the classes necessary to host this same designer within your applications. This chapter is all about hosting this designer. After a brief overview of the major workflow designer components, you will implement a simple application that hosts the workflow designer. In subsequent sections, you will build upon the application, adding new functionality with each section.

Chapter 18: WF 3.*x* Interop and Migration

This chapter focuses on strategies for dealing with existing WF 3.0 or 3.5 applications (WF 3.*x*). The chapter begins with an overview of the migration strategies that are available to you followed by a demonstration of the `Interop` activity. This activity enables you to execute some WF 3.*x* activities within the WF 4 runtime environment.

Appendix A: Glossary

This is a glossary of commonly used WF terms.

Appendix B: Comparing WF 3.*x* to WF 4

This appendix highlights major differences between the previous version of WF (3.*x*) and WF 4.

What You Need to Use This Book

To execute the examples presented in this book, you'll need to install a minimum set of software components on a supported OS. The minimum requirements are the following:

- Visual Studio 2010 Professional, Premium, or Ultimate.
- The .NET 4 runtime (installed with Visual Studio 2010).
- SQL Server 2005 or 2008 Express edition. If you have one of the full licensed versions of SQL Server, that will work fine. SQL Server 2008 Express is installed with Visual Studio 2010.

Check with Microsoft for a current list of supported operating systems. The Microsoft .NET Framework Development Center (`http://msdn.microsoft.com/en-us/netframework/default.aspx`) is a good starting point to locate any miscellaneous files that you need.

Obtaining This Book's Source Code

I have found that the best way to learn and retain a new skill is through hands-on examples. For this reason, this book contains a lot of example source code. I've been frustrated on more than one occasion with technical books that don't print all of the source code in the book. The code may be available for download, but then you need to have a computer handy while you are reading the book. That doesn't work well at the beach. So, I've made it a point to present all of the code that is necessary to actually build and execute the examples.

When you are ready to execute the example code, you don't have to enter it yourself. You can download all of the code presented in this book from the Apress site at `www.apress.com`; go to the Source Code/Download section to find it. I've organized all of the downloadable code into separate folders and Visual Studio solutions for each chapter. I suggest that you use the same approach as you work through the examples in this book.

How to Reach Me

If you have questions or comments about this book or Windows Workflow, I'd love to hear from you. Just send your email to workflow@bukovics.com. To make sure your mail makes it past any spam filters, you might want to include the text *ProWF4* somewhere in the subject line.

CHAPTER 1

A Quick Tour of Windows Workflow Foundation

This chapter introduces you to Windows Workflow Foundation (WF). Instead of diving deeply into any single workflow topic, it provides you with a brief 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 in which we work. But regardless of the complexity of any given problem, we tend to solve problems in the same way: we break the problem down into identifiable and manageable tasks. Those tasks 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 individual 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 procedural programming model, you implement a task in code using your chosen development language. First and foremost, the code performs some small unit of useful work. It might execute a query or update statement on a database, enforce validation rules, determine the next page to show to a user, queue a message on another system, and so on. But in addition to implementing the real work of the task, you also need to implement the "glue" code that determines the sequence of the individual steps. You need to make branching and looping decisions, check the value of variables, validate inputs, and produce outputs. And when the smallest of tasks are combined into larger composite tasks, there's even more code needed to control how all of those tasks will work together to accomplish some greater purpose.

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

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 graphical environment where complex business rules are declared visually rather than entirely in code. Individual tasks are organized into an appropriate sequence, and branching and looping decisions are declared to control the flow of execution between tasks. It's an environment that allows you to easily visualize and model the tasks to solve a problem. And since you can visualize the tasks, it's easier to understand and change them.

But there's much more to workflows than just a visual development environment. Workflows represent an entirely different programming model—a declarative one. In a traditional procedural programming model, the code that performs the real work for a task may be entwined with the code that determines when to execute the task. If you need to change the conditions under which a task should execute, you may have to slog your way past a lot of code that is unrelated to your change. And when you apply your simple flow control change, you need to avoid inadvertent changes to the part of the code that performs the real work.

In contrast with this, a declarative workflow model promotes a clear separation between *what* to do (the real work that you're trying to accomplish) and *when* to do it. This separation allows you to change the *when* without affecting the *what*. With this model, the real work of the task is encapsulated in discrete activities. You can think of an activity as a unit of work with a defined set of inputs and outputs. In WF, you have the option of implementing activities in code or declaratively assembling them from other activities. But regardless of how the individual activities are authored, they are assembled like building blocks into complete workflows. The job of the workflow is to coordinate the work of one or more activities. Branching and looping decisions that control the flow of execution between activities are declared within the workflow—they are not hard-coded within each activity. With the workflow model, if you need to make that same flow control change, you simply change the workflow declaration. The activity code that performs the real work remains untouched.

General-purpose languages such as C# or Visual Basic can obviously be used to solve business problems. But one additional advantage of the workflow programming model is that it enables you to implement your own domain-specific language. In WF, this is accomplished by developing custom activities that model the problem domain. 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 the activities that are declared within it. They can 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 and timing 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, and a

large amount of code is required just to manage all of the possible execution paths. A human interaction problem such as this can typically be expressed more naturally and clearly in a workflow model.

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 the use of a declarative workflow model within your own applications. Regardless of the type of application you are developing, you can likely leverage something in WF.

If you are developing line-of-business applications, you can use WF to implement the business rules as a set of custom activities and workflows. If your application comprises a series of human interactions, you can model long-running workflows that are capable of coordinating the work that is done by humans with the application.

If you need a highly customizable application, you can use the declarative nature of WF to allow end-user customization of the workflows. And if you are just looking for a better way to encapsulate and organize your application logic, you can implement the logic as discrete custom activities. Since each activity has a defined set of inputs and outputs, it is easy to independently test each activity before it is assembled into a workflow.

The previous were all good reasons to use WF, and here are a few more of them:

- WF 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.
- WF 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 improves your visibility when maintaining existing ones.
- WF supports multiple modeling styles. You can choose to model a workflow using familiar procedural constructs such as `if` statements and `while` loops. Or you can choose to model a workflow that uses flowchart concepts where looping and branching decisions are declared as direct links between activities. And for the ultimate in flexibility, you can even mix and match both styles within the same workflow.
- WF provides tight integration with Windows Communication Foundation (WCF). Standard activities are provided that enable you to consume WCF services from within a workflow or expose a workflow as a WCF service endpoint.
- WF 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.

- WF provides a complete workflow ecosystem. Microsoft provides the workflow runtime, a suite of standard activities, base classes for building your own activities, workflow persistence, and workflow tracking. Tooling support is also provided in the form of a workflow designer that is integrated with Visual Studio, which you can also host in your own applications.
- WF is included with .NET and Visual Studio and available for use in your applications without any additional licensing fees.

Your Development Environment

Windows Workflow Foundation was originally made available as part of .NET 3.0 and later enhanced in .NET 3.5. The tooling support for WF (workflow designer, templates, and debugger support) was originally provided as an add-in to Visual Studio 2005 and later built in to Visual Studio 2008.

Visual Studio 2010 and .NET 4 are the delivery vehicle for WF 4, which is the topic of this book. All of the examples in this book target WF 4, and all of the screen shots were captured from Visual Studio 2010. The one exception is the chapter on interop with WF 3.*x*. That chapter uses some activities that target the WF 3.*x* environment to demonstrate the interop capabilities in WF 4.

To run the examples in this book, you'll need the following:

- Visual Studio 2010 Professional, Premium, or Ultimate
- The .NET 4 runtime (installed with Visual Studio 2010)
- SQL Server Express 2008 (installed with Visual Studio 2010)

WF 4 represents a significant break from previous versions of WF. Microsoft has listened to the feedback that it received for WF 3.*x* and decided to take a clean-slate approach to improve WF 4. The result is that the entire framework has been rewritten from the ground up.

The good news is that WF 4 is a monumental improvement over its predecessors. Microsoft has gone to great lengths to simplify the development model, improve the performance, and reduce many of the pain points that were present with the previous versions of the framework.

The bad news is that the 3.*x* and 4 versions are not compatible with each other. A custom activity written for WF 3.*x* won't run under WF 4 unless it is first wrapped in an interop activity. But the entire WF 3.*x* framework continues to be shipped with .NET 4. And Visual Studio 2010 includes all of the designer, debugger, and template support for the WF 3.*x* version of the framework. So if you've already made a substantial investment in WF 3.*x*, you can continue to support and enhance those applications using the latest Microsoft offerings.

■ **Note** This book targets the 4 version of WF. If you're looking for a book on WF 3.0 or 3.5, please consider my previous workflow book, *Pro WF Windows Workflow in .NET 3.5*, published by Apress.

The Workflow Workflow

Before I begin the first example, I want to present what I call the *workflow workflow*. This is the mental checklist of steps that you follow when developing workflow applications. I'll present enhanced or more

specialized revisions of this list throughout the book to add other steps that you should consider. But in its simplest form, developing workflow applications using WF can be summarized by these steps:

1. Select or implement the activities to perform some work.
2. Declare a workflow that coordinates the work of one or more activities.
3. Develop a workflow host application.

The first step is to select the activities that you need in order to perform some useful work. The work could be updating a database, sending a WCF message to another system, performing a calculation, and so on. The work to perform is entirely up to you and depends on the problem that you are attempting to solve. You might be able to use an out-of-the-box activity (or combination of activities) that is provided with WF to perform the work. Or, more than likely, you will need to implement your own custom activities that perform the work. Once your custom activities have been developed and tested, they form a library of reusable building blocks that can be used to build multiple workflows.

After you identify (or implement) the activities that do the real work, you can turn your attention to the workflow itself. Remember that the job of the workflow is to coordinate work, not to necessarily perform the work itself. During this step you declare the structure of the workflow by adding activities and flow control elements. The flow control elements may be other activities that define loops and make branching decisions. Or they may take the form of direct connections between activities that define the execution sequence.

Finally, you need a host application that can execute instances of your new workflow. Since WF is a set of foundation classes rather than a finished application, you are able to leverage WF in a wide variety of application types. You can use WF from WinForms, Windows Presentation Foundation (WPF), ASP.NET, Windows services, or even lowly console applications. You may need to expose your workflow as a Windows Communication Foundation service and consume it from other client applications. If so, you can host the workflow using a Microsoft-provided hosting environment such as Internet Information Services (IIS) or Windows Process Activation Service (WAS). Or you can choose to develop your own self-hosting application. Regardless of the type of application, the most basic job of the host is to start an instance of the workflow and wait until it completes. In subsequent chapters, you will learn about additional duties that the host application can perform.

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.

Here are the steps you will follow to implement the "Hello Workflow" example:

1. Create a new Workflow Console Application.
2. Add a `Sequence` activity to the workflow.
3. Add a `WriteLine` activity as a child of the `Sequence` activity.
4. Set the `Text` property of the `WriteLine` activity to the message that you'd like to display.
5. Review the boilerplate code that runs the workflow.

In this example, and in the other examples in this chapter, I present fundamental 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 this information. If so, go ahead, but you might want to give this chapter a quick read anyway.

Creating the Project

You create workflow projects in the same way as other project types in Visual Studio. After starting Visual Studio, select File ➤ New ➤ Project. A New Project dialog is presented that allows you to enter project parameters and to select the template to use for the new project.

After selecting Visual C# as the language, you'll see Workflow as one of the available project template categories. Visual Studio is capable of creating projects that target different versions of the .NET Framework. The.NET Framework to target is set at the top of the New Project dialog. The New Project dialog is target-aware, meaning that it presents only the templates that are available for the selected version of the framework. The Visual Studio Toolbox also filters the list of controls to those that are available in the selected version of the .NET Framework. You should select .NET Framework 4 for all of the examples in this book. The one exception will be when you are creating 3.5 workflow components in Chapter 18.

For this example, select Workflow Console Application as the template to use for the new project. This creates a Windows console application that includes an empty workflow definition file and the necessary boilerplate code to execute it. Enter `HelloWorkflow` as the project name, and also enter a solution name of `chapter 01`. Figure 1-1 shows the New Project dialog after I've entered all of the necessary information to create the new project. Click OK once you are ready to create the new project.

■ **Note** In the example code that accompanies this book, I use a separate solution for each chapter. All of the example projects for a chapter are added to the solution for that chapter. You might want to adopt the same strategy when you are entering these examples.

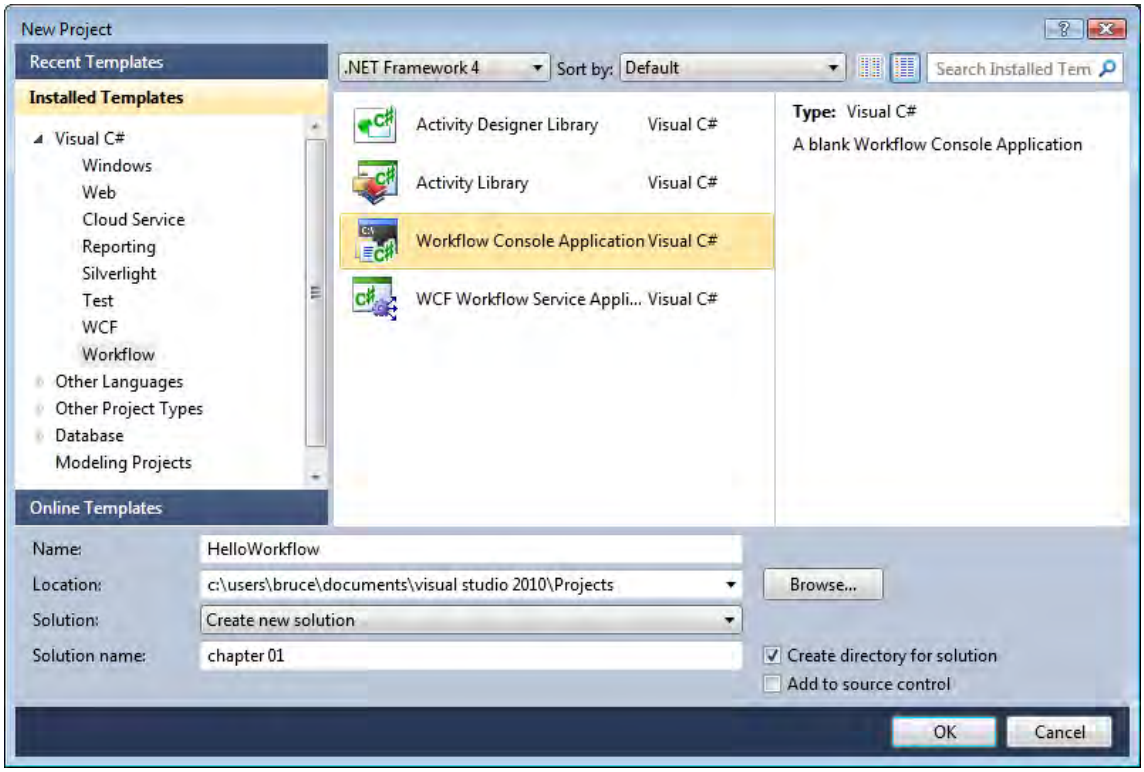

Figure 1-1. *New Project dialog*

The new project template creates two files that you'll need to modify in the steps that follow. The `Workflow1.xaml` file is the workflow definition, and the `Program.cs` file contains the code needed to run an instance of the workflow. The `Workflow1.xaml` file is an XML-based declarative definition of the workflow that is compiled into a Common Language Runtime (CLR) type during the build process.

The new project also includes references to the .NET assemblies that you need to execute a simple workflow. Foremost in the list of assembly references that you need is `System.Activities`. Within this assembly, the workflow-related classes and activities are organized into a number of namespaces. In your code, you need to reference only the namespaces that you are actually using.

■ **Note** The Workflow Console Application template creates a project that targets the .NET Framework 4 Client Profile. This is a subset of the full .NET Framework that omits server-oriented assemblies in order to reduce its size. If necessary, the target framework can be changed from the project properties page. This subset of the full framework is fine for most of the examples in this book. I'll draw your attention to the examples that require the full framework.

Declaring the Workflow

I said previously that workflows coordinate the work of one or more activities. So, declaring a workflow requires that you identify the activities to execute and arrange them in some logical sequence. You can do that entirely in code, or you can accomplish the same thing declaratively using the workflow designer. The designer supports dragging and dropping of activities onto the workflow canvas from the Visual Studio Toolbox.

In this project, the `Workflow1.xaml` file that was added for you contains a declarative XML-based representation of the workflow that can be maintained by the workflow designer. If it is not already open, double-click the `Workflow1.xaml` file in Solution Explorer to open it in the designer now. Figure 1-2 shows the workflow in the designer.

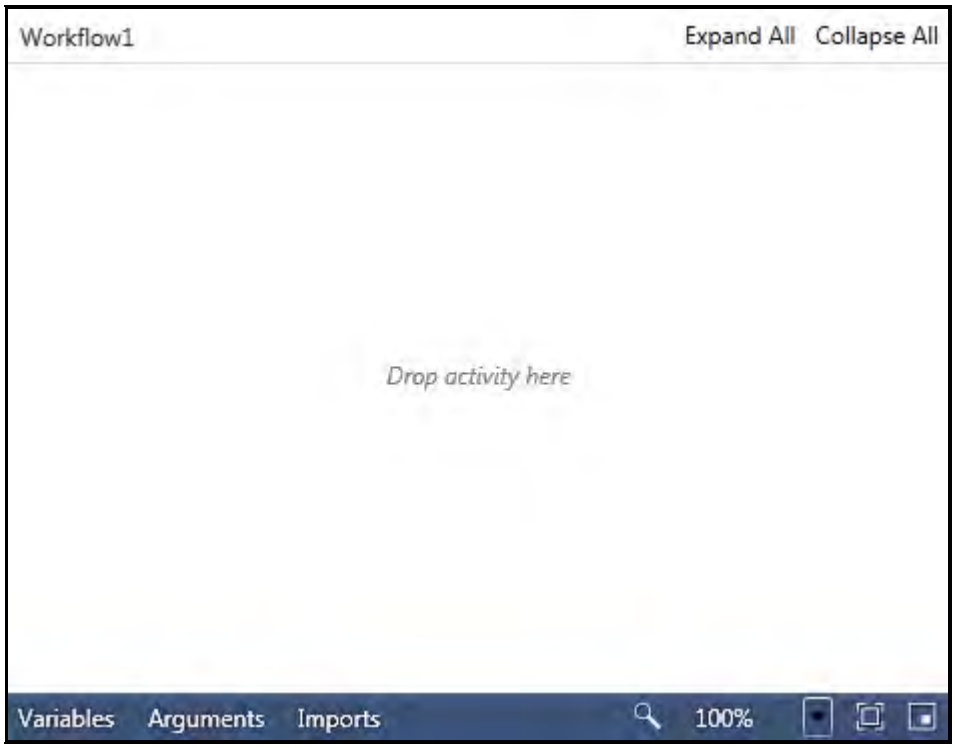

Figure 1-2. *Empty Workflow1.xaml opened in the workflow designer*

As you can see from Figure 1-2, the workflow is initially empty. At this point it represents an empty canvas, ready to accept workflow activities that you drag and drop from the Visual Studio Toolbox.

An activity represents a step in the workflow and is the fundamental building block of all WF workflows. Microsoft supplies a set of standard activities that you can use, but you will need to develop your own custom activities in order to accomplish any really meaningful work. Each activity is designed to serve a unique purpose and encapsulates the logic needed to fulfill that purpose.

Since most workflows require more than one activity to accomplish a task, you will usually begin by adding a container for those activities. The first activity that you add to a workflow (known as the *topmost* or *root* activity) is frequently the `Sequence` or `Flowchart` activity. These are two standard control flow activities that are provided with WF. They are both composite activities that allow you to add other activities as children. They both serve the same purpose: to provide a simple way to determine the sequence in which any child activities execute.

The examples in this chapter (and the next few chapters) are based on the `Sequence` activity, which uses a procedural style of flow control. The `Flowchart` activity represents a different and unique way to author workflows and is discussed in Chapter 7.

Adding the Sequence Activity

Begin the declaration of this workflow by dragging and dropping a `Sequence` activity from the Visual Studio Toolbox to the empty `Workflow1.xaml` file in the designer. The standard activities provided with WF are organized into several different categories according to the purpose for each activity. You should find the `Sequence` activity in the Control Flow category of the Toolbox, as shown in Figure 1-3.

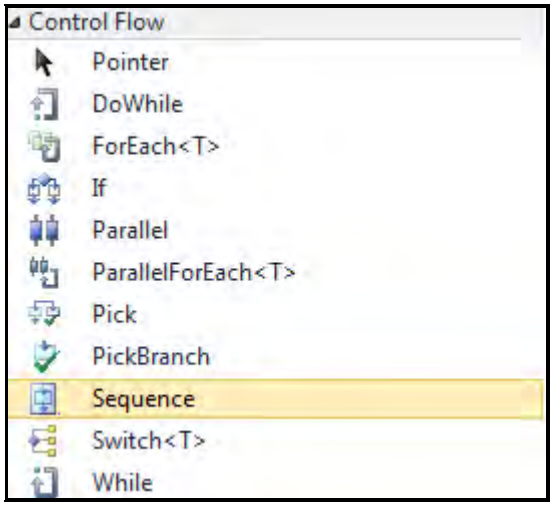

Figure 1-3. *Toolbox with Sequence activity*

A `Sequence` activity is considered a composite activity because it is a container for other child activities. In general, the primary responsibility of any composite activity is to run any child activities contained within it in some prescribed order. In the case of the `Sequence` activity, the prescribed order is a simple sequence: run the first child activity, followed by the second child activity, and so on. After you have added a series of activities to a `Sequence` activity, you can modify their execution order by simply dragging them to a new location relative to the other activities.

While you're taking a look at the workflow designer for the first time, you might want to also make note of several of its features. The Variables button allows you to define local variables to maintain state within the workflow or to pass data between activities. The Arguments button allows you to define input or output arguments for the workflow. This first example doesn't require the use of either of these features.

Also included is an Imports button. This allows you to add namespaces that you will frequently reference within the workflow. This is similar to adding a `using` statement in your C# code. It allows you to reference the class name without the need to specify the entire namespace-qualified name.

The lower-right corner of the designer includes controls that let you modify the designer display. You can zoom in or out to change the set of activities that are visible at one time. A Mini map control allows you to navigate large workflows using a scrollable thumbnail view of the entire workflow.

The designer also includes Expand and Collapse options located at the upper-right corner of the design surface. These options allow you to further refine the view of the activities shown within the designer by expanding or collapsing additional detail for each activity. You won't need to use these options for these first simple examples, but they are helpful when you are working with larger workflows.

Adding the WriteLine Activity

The objective of this example is to write the message "Hello Workflow" on the console. To accomplish that objective, you can use one of the standard activities named `WriteLine`. As the name implies, this activity can write any text that you want to the console. Optionally, it can write text to a `TextWriter` instead of directly to the console, but you don't need that functionality for this example. To add this activity to the workflow, open the Toolbox again, and find the `WriteLine` activity. It should be located in the Primitives category of the Toolbox.

Once you've located it, drag the `WriteLine` activity to the workflow designer, and drop it on the open `Sequence` activity. The workflow should now look like Figure 1-4.

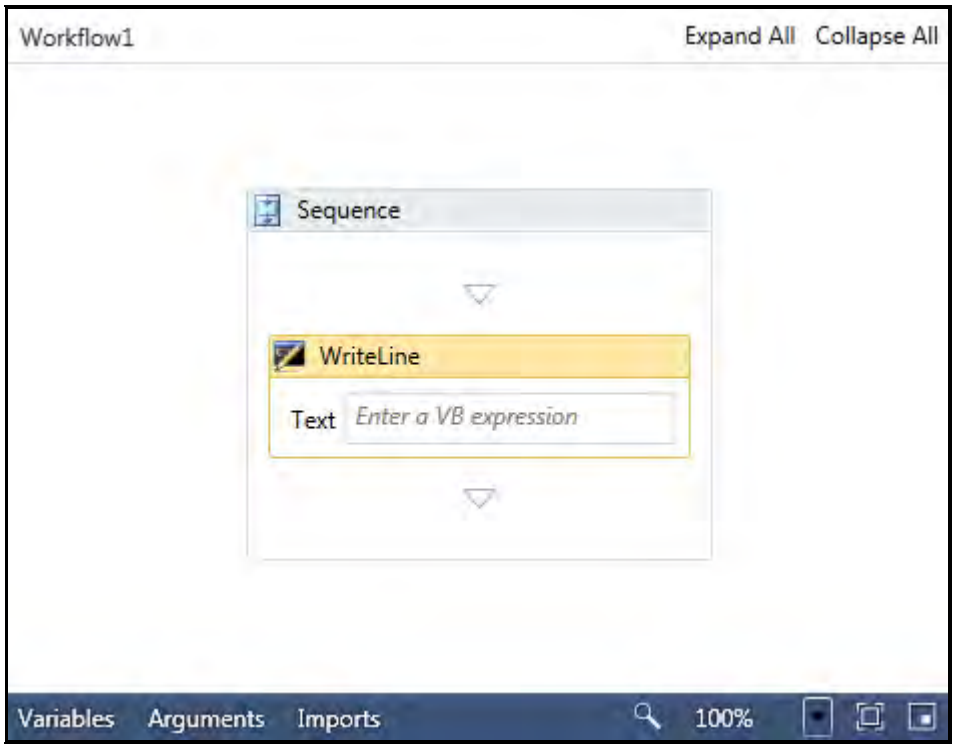

Figure 1-4. *Workflow with WriteLine activity*

To complete the workflow definition for this example, you need to set the `Text` property of the `WriteLine` activity. To accomplish this, highlight the `WriteLine` activity and then press F4 (assuming you still have the default key mappings) to open the Properties window. Enter the string literal **"`Hello Workflow`"** (including the double quotes) as the value for the `Text` property.

■ **Caution** String literals must be entered within double quotes. The `Text` property is actually expecting you to define a workflow expression that returns a string. An expression can be a simple string literal or something much more complex such as a call to a method. If you enter `Hello World` with no double quotes, the Expression Editor won't know that it's a string literal and will attempt to parse and interpret the string as a Visual Basic expression.

Figure 1-5 shows the completed Properties window for the `WriteLine` activity.

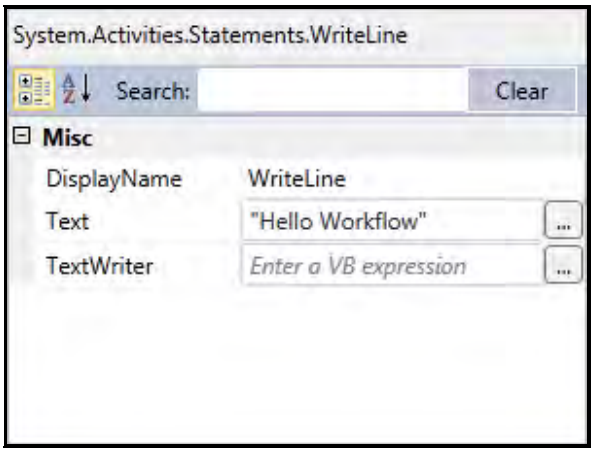

Figure 1-5. *Properties window for WriteLine activity*

As shown in Figure 1-5, the `WriteLine` activity also includes a property named `DisplayName`. This property is common to all activities and determines the name that is shown for the activity within the designer. Most of the time you can use the default name, but it is helpful to provide a more meaningful name when you are working with a larger workflow, especially when the workflow includes multiple instances of the same activity. The `DisplayName` property can also be changed directly within the designer.

Also note that the `Text` property for the `WriteLine` activity can be set directly within the designer. This eliminates the need to use the Properties window to set the properties that are used most often. Not all activities support this kind of property editing, but most try to support the most common properties directly in the designer.

Save all of your changes to the `Workflow1.xaml` file if you haven't already done so.

■ **Tip** This particular example really doesn't require the `Sequence` activity since it contains only a single `WriteLine` activity. You could have added the `WriteLine` activity directly to the empty `Workflow1.xaml` file, and the workflow would execute correctly with the same results. However, in most cases, you will declare workflows that require many activities, and you will need a container activity such as `Sequence` to organize and control their execution. For this reason, it is a good habit to always start with a `Sequence` or `Flowchart` activity.

Hosting the Workflow

Now that you've declared the example workflow, open the `Program.cs` file, and turn your attention to the code that runs an instance of the workflow. For the most part, you can use the boilerplate code that was produced by the new project template.

In the following code, I've made a few minor changes and reformatted the code to fit the format of this book. Since this is a console application, all of the code is contained within the static `Main` method.

```
using System;
using System.Activities;

namespace HelloWorkflow
{
    class Program
    {
        static void Main(string[] args)
        {
            WorkflowInvoker.Invoke(new Workflow1());

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

■ **Note** The boilerplate code that was generated for you will likely contain additional `using` statements that are not shown here. To avoid confusion, I show only those `using` statements that are absolutely necessary. I try to follow this practice for all the examples in this book.

As you will see in later chapters of this book, there are a number of ways to execute a workflow. What you see here is absolutely the simplest way to execute a workflow. The `WorkflowInvoker` class provides a way to execute a workflow with the simplicity of calling a method. The static `Invoke` method requires an instance of the workflow that you want to execute. When the `Invoke` method is used like this, the workflow executes synchronously on the current thread. In this sense, the `Invoke` method is a blocking call that completes only once the workflow has completed.

■ **Note** Please refer to Chapter 4 for an in-depth discussion of other workflow hosting options.

Whenever I'm working with console applications, I add a couple of final calls to the `Console` class. These calls display a message on the console and pause the application until a key has been pressed. Without these lines, a console application that is run in the debugger (F5) will execute and then immediately finish, not providing you with a chance to see the results.

Running the Application

After building the project, you should be ready to test it. You can press F5 (or Ctrl-F5 to start without debugging). This assumes the default C# key mappings. Use the appropriate key combination for your development environment, or select Start Debugging from the Debug menu.

If everything works correctly, you should see these results on the console:

```
Hello Workflow

Press any key to exit
```

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

Exploring the Xaml

Before moving on to the next example, you should take a few minutes to examine the `Workflow1.xaml` file that was used in the previous example. This is the file that contains the workflow definition that you modified via the workflow designer.

If you still have this file open in the designer, close it. Double-clicking the file opens it in the default view, which is the workflow designer. Instead, right-click the file and open it in Code View. This should open it using the XML editor. I've reformatted the contents of the file to fit the format of this book, and I've removed a few entries that are used internally by the debugger and designer. Your file should look similar to this:

```
<Activity mc:Ignorable="sap" x:Class="HelloWorkflow.Workflow1"
  mva:VisualBasic.Settings=
    "Assembly references and imported namespaces serialized as XML namespaces"
  xmlns="http://schemas.microsoft.com/netfx/2009/xaml/activities"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:mv="clr-namespace:Microsoft.VisualBasic;assembly=System"
  xmlns:mva=
    "clr-namespace:Microsoft.VisualBasic.Activities;assembly=System.Activities"
  xmlns:s="clr-namespace:System;assembly=mscorlib"
  xmlns:s1="clr-namespace:System;assembly=System"
  xmlns:s2="clr-namespace:System;assembly=System.Xml"
  xmlns:s3="clr-namespace:System;assembly=System.Core"
  xmlns:sad="clr-namespace:System.Activities.Debugger;assembly=System.Activities"
  xmlns:sap="http://schemas.microsoft.com/netfx/2009/xaml/activities/presentation"
  xmlns:scg="clr-namespace:System.Collections.Generic;assembly=System"
  xmlns:scg1=
    "clr-namespace:System.Collections.Generic;assembly=System.ServiceModel"
  xmlns:scg2="clr-namespace:System.Collections.Generic;assembly=System.Core"
  xmlns:scg3="clr-namespace:System.Collections.Generic;assembly=mscorlib"
  xmlns:sd="clr-namespace:System.Data;assembly=System.Data"
  xmlns:sd1="clr-namespace:System.Data;assembly=System.Data.DataSetExtensions"
  xmlns:sl="clr-namespace:System.Linq;assembly=System.Core"
  xmlns:st="clr-namespace:System.Text;assembly=mscorlib"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Sequence>
    <WriteLine Text="Hello Workflow" />
  </Sequence>
</Activity>
```

This file uses Extensible Application Markup Language (Xaml) to declare the workflow. Xaml is a serialization format that specifies object instances as XML elements and properties of those objects as XML attributes.

Most of this small Xaml file is occupied with Microsoft namespace definitions. These namespaces provide access to the schemas that define the various parts of the Xaml structure. During the build process, this file is compiled into a new CLR type. In the `Program.cs` file that we just reviewed, you saw that you were able to create an instance of this compiled workflow type like this:

```
WorkflowInvoker.Invoke(new Workflow1());
```

In this case, `Workflow1` is a new class that is defined by this Xaml file. The `Class` attribute on the first line of the Xaml file is what determines the fully qualified namespace and class name of the new compiled type:

```
<Activity mc:Ignorable="sap" x:Class="HelloWorkflow.Workflow1"
```

The root tag of `Activity` is significant, since it indicates that the base class for this new type is `Activity`. The `Activity` class is a base type provided with WF that all activities and workflows must ultimately derive from.

■ **Note** You will learn more about the `Activity` class hierarchy in Chapter 3.

This workflow used a `Sequence` activity that was the container for a single `WriteLine` activity. The `WriteLine` activity had a default `Text` property that you set to an appropriate message to write to the console. It's easy to see how the remaining XML nodes map to this structure that you declared using the workflow designer:

```
<Sequence>
  <WriteLine Text="Hello Workflow" />
</Sequence>
```

Please note that the workflow is fully declared within this Xaml file. Many of the other designers in Visual Studio create separate code-beside files that separate the designer-maintained code from the code that you maintain. Examples that use this mechanism include WinForms, WPF, and the 3.*x* version of WF. But with WF 4, the workflows are entirely declarative—there is no code-beside file. Unlike previous versions of WF, no procedural code is allowed within the workflow definition. All C# code has been pushed into custom activities where it really belongs.

Most of the time, you'll want to use the workflow designer to declare and maintain your workflows. It just makes sense to use the development tools that will make you most productive, and the designer provides that nice drag-and-drop development experience. But there will be times when you find it easier to manually edit the Xaml files directly. There's absolutely no problem in doing that. The designer doesn't add any magic entries that you can't do yourself. It simply provides a very nice visual way to manipulate these files.

To see this in action for yourself, you can modify the `HelloWorkflow` project by modifying the Xaml file directly. For a quick example of this, duplicate the `WriteLine` element, and change the message like this:

```
<Sequence>
  <WriteLine Text="Hello Workflow" />
  <WriteLine Text="I added this activity via Xaml" />
</Sequence>
```

If you haven't made a really dreadful cut-and-paste error, you should be able to build the project and run it to see these results:

```
Hello Workflow

I added this activity via Xaml

Press any key to exit
```

Passing Parameters

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

The preceding example writes a constant string literal to the console. To see how parameters are passed to a workflow, you'll implement another similar example that uses input parameters to format the string before it is written.

Here are the steps you will follow to implement this example:

1. Add a new workflow to the `HelloWorkflow` project.
2. Define a workflow input argument
3. Add a `Sequence` activity to the workflow.
4. Add a `WriteLine` activity to the `Sequence` activity.
5. Set the `Text` property to a message that includes the input argument.
6. Pass the input argument to the workflow when you are executing it.

■ **Note** Throughout the remainder of this book, I'll assume that you already know the basics of creating new projects and don't require the step-by-step commentary. However, I will let you know when there is something significant that you need to be aware of when creating the new project or adding a new item to an existing project.

Declaring the Workflow

For this example, you will add a new workflow to the existing `HelloWorkflow` project. Select the Add New Item option for the project, and then select the Workflow category. Select the Activity template, and name the new workflow `HelloWorkflowParameters`. The new item should open in designer view.

■ **Tip** Don't be confused by the terminology that is used here. You are adding a new workflow, but since all workflows are actually activities, the Add New Item template uses the term *Activity*.

The purpose of this example is to demonstrate how to pass data to the workflow. Any data that you want to pass to a workflow must first be declared as a workflow argument. To accomplish this, click the Arguments button to open the Argument Editor. You can find this button on the lower-left side of the designer. The list of arguments is initially empty, so you should select the Create Argument option to add a new one. Arguments have a name, direction (in, out, or both), argument type, and optional default value. You need to add a single argument with these parameters:

Name	Direction	Argument Type
ArgFirstName	In	String

The direction of In indicates that this is data that will be passed as an input argument to the workflow. Figure 1-6 shows the Argument Editor after I've entered the required input argument. Click the Arguments button once again closes the Argument Editor.

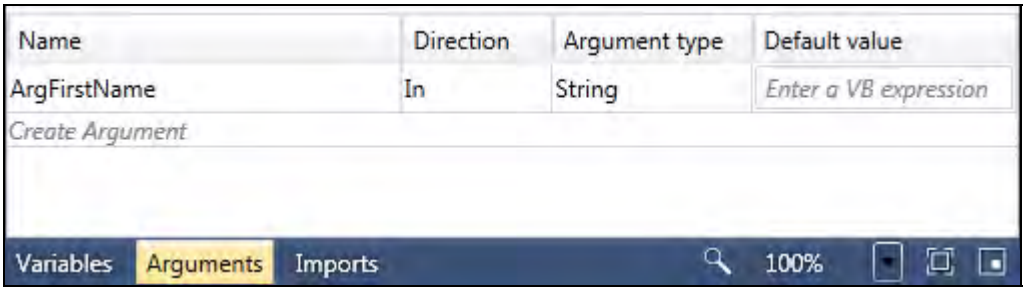

Figure 1-6. *Argument Editor with required argument*

■ **Note** You may notice that I included the *Arg* prefix in front of the argument name. This isn't a WF requirement or an attempt on my part to talk like a pirate (not that there's anything wrong with that). I simply did this to make it easier to spot the arguments as you proceed with the workflow declaration. In this example, you're working with only a single argument. However, in much larger workflows, you will be using a combination of input and output arguments and workflow variables. It's convenient to be able to look at the name and immediately know that it's an argument.

Now that the input argument has been defined, please follow these steps to finish the declaration of the workflow:

1. Add a `Sequence` activity to the empty workflow.
2. Add a `WriteLine` activity to the Sequence activity.
3. Set the `WriteLine.Text` property to a meaningful message. This time, instead of using only a string literal, you need to reference the `ArgFirstName` argument in the expression. After selecting the `WriteLine` activity, you can enter the string expression directly in the Properties window as you did in the previous example. Or you can click the ellipsis button next to the `Text` property to open the Expression Editor. The Expression Editor provides a bit more real estate for entering your expressions.

Figure 1-7 shows the Expression Editor after I've entered the string expression.

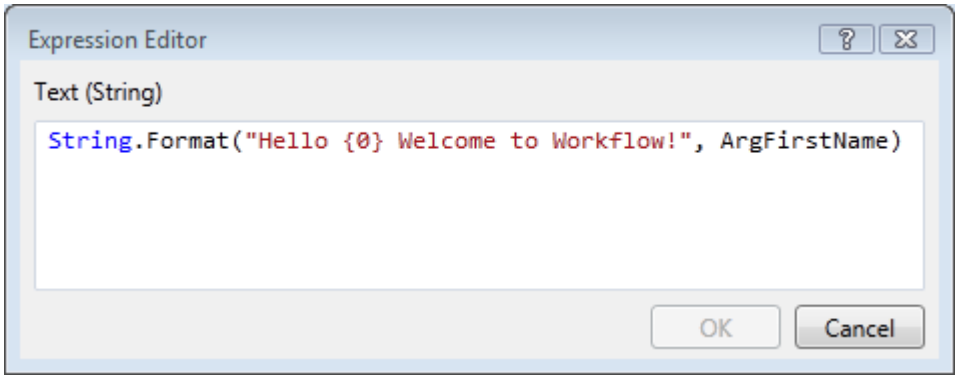

Figure 1-7. *Expression Editor with string expression*

As shown in Figure 1-7, I entered an expression that uses the `String.Format` method to concatenate several strings, including the value from the `ArgFirstName` argument:

```
String.Format("Hello {0} Welcome to Workflow!", ArgFirstName)
```

In this expression, `ArgFirstName` is not double-quoted, so it references the input argument that you defined instead of a string literal.

When you are entering an expression, you can use the IntelliSense support that is provided by the Expression Editor. For example, once you enter `String.` (including the period), you should see an IntelliSense window pop up to assist you in selecting the correct member from the `String` class. Likewise, when you start to enter `Arg`, you should be presented with a list of everything that is in scope that begins with those letters.

■ **Tip** When using IntelliSense within the Expression Editor, remember to use the Tab key to select the highlighted item in the list. Don't use the Enter key. Pressing the Enter key will add a carriage return, causing a line break after the selected item. Workflow expressions are entered using Visual Basic (VB) syntax, so the extra carriage return causes errors in the expression. If you really do need to continue an expression on multiple lines, you'll need to include the VB continuation character (_, an underscore) at the end of the line you want to continue.

When entering an expression, you actually have quite a bit of freedom to decide how it will be entered. In this particular case, the `Text` property of the `WriteLine` activity expects an expression that returns a string. It really doesn't matter how you build that string, as long as it resolves to a string. For example, you could have concatenated the string yourself instead of using the `String.Format` method.

■ **Note** For now, you really only have to be aware that expressions are entered in VB syntax, so you don't need to enter trailing semicolons and other bits of C# syntax. I provide more information on expressions in Chapter 2.

This completes the workflow definition. Structurally, the workflow should look like the previous example shown in Figure 1-4.

Hosting the Workflow

To complete this example, open the `Program.cs` file, and make just a few minor modifications to pass the input argument and to execute this new workflow. Here is a revised copy of the `Program.cs` file with the necessary changes:

```
using System;
using System.Activities;
using System.Collections.Generic;

namespace HelloWorkflow
{
    class Program
    {
        static void Main(string[] args)
        {
            WorkflowInvoker.Invoke(new HelloWorkflowParameters(),
                new Dictionary<String, Object>
                {
                    {"ArgFirstName", "Bruce"}
                });
```

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

Parameters are passed to a workflow in the form of a generic dictionary of objects keyed by a string (`Dictionary<String, Object>`). The string key must exactly match the argument name (including case), and the type of the object must match the expected argument type. This workflow requires only the single `ArgFirstName` argument, but multiple arguments would be passed in this same dictionary if they were required. An overload of the `WorkflowInvoker.Invoke` method is used that accepts the dictionary of parameters.

■ **Note** If you do manage to misspell a parameter name, you won't know it until you actually run the workflow. At that time, a `System.ArgumentException` will be raised, informing you that you tried to set a nonexistent argument. Likewise, if you pass data of an incorrect type (such as passing an integer when the argument is expecting a string), the same exception will be raised with a slightly different message.

Running the Application

After building the application, you should be able to run it and see these results, proving that the parameter correctly made its way into the workflow:

```
Hello Bruce Welcome to Workflow!

Press any key to exit
```

Using Argument Properties

Using a `Dictionary` to pass input arguments to a workflow is a flexible way to provide input. However, it doesn't provide you with much feedback during development. In particular, you have no compile-time checking of argument names or types.

To remedy this, the compiled workflow class also exposes any input arguments as public properties. The properties are defined as `InArgument<T>` for input arguments, where the generic parameter identifies the underlying type that you defined for the argument. The `InArgument<T>` class is one of a series of related classes used to define arguments (`InArgument<T>`, `OutArgument<T>`, `InOutArgument<T>`). To make it easier to set the value for an argument, the `InArgument<T>` class defines an assignment operator that allows you to directly assign a value of type `T` to the argument.

What all of this means is that a much more type-safe way to pass arguments to a workflow is to use these generated properties like this:

```
using System;
using System.Activities;
using System.Collections.Generic;

namespace HelloWorkflow
{
    class Program
    {
        static void Main(string[] args)
        {
            HelloWorkflowParameters wf = new HelloWorkflowParameters();
            wf.ArgFirstName = "Bruce";
            WorkflowInvoker.Invoke(wf);

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

Now you no longer have to wait until you run the workflow to determine whether the argument name is correct and that the value is of the correct type. If you run this revised code, you should see the same results as the previous example that used a `Dictionary` for argument input.

■ **Note** Although using the argument properties of a workflow does solve the problem of type safety, it does raise a potential performance problem. In the previous example that does not use argument properties, the workflow definition and the input arguments were passed to the `WorkflowInvoker` class as separate arguments. This means that you could potentially create a single instance of the workflow definition and use it over and over again to start multiple workflow instances. Each workflow instance could easily have a different set of arguments since they are passed separately from the definition. If you use argument properties, you would have to create a new instance of the workflow definition for each instance that you want to execute. With argument properties, you lose some of the performance advantage of reusing the workflow definition.

Making Decisions

This next example demonstrates a number of other WF concepts. You'll learn how to develop a simple custom activity, how to define and use workflow variables, how to obtain a result value from the workflow, and one way to make simple branching decisions. In the process, you'll be introduced to the `Assign`, `Switch<T>`, and `Throw` activities, which are three of the standard activities included with WF.

The sample application for this example is a simple command-line calculator. The goal is to be able to enter an expression such as `1 + 1` and have the workflow return the correct result. To accomplish this,

you will implement a new custom activity to parse the expression into its respective parts (two numbers and an operation). You will then declare a calculator workflow that uses the new activity along with several standard activities to perform the calculation. The result is returned as an output argument of the workflow.

Here are the steps you will follow to implement this example:

1. Create a new Workflow Console Application.
2. Implement a custom activity to parse the arithmetic expression.
3. Define input and output workflow arguments.
4. Define workflow variables.
5. Add the custom activity to the workflow.
6. Add `Switch<T>` and `Assign` activities to perform the requested calculation.
7. Write the host application code to accept user input, and execute the workflow.

Creating the Project

Create a new project using the Workflow Console Application template. Name the project `Calculator`, and add it to the solution for this chapter. You can delete the `Workflow1.xaml` file that was generated for the project since it won't be used.

Add a new workflow to the project using the Activity template, naming it `Calculator`. Add a `Sequence` activity to the empty `Calculator` workflow.

Implementing a Custom Activity

To use this application, the user will enter an expression such as `1 + 1` or `3 * 5` that is passed to the workflow as a single string. In this step, you will develop a custom activity that will parse the string into its respective parts (two numbers and an operation string). Later, when you declare the workflow, the three parsed values will be assigned to workflow variables and used by other activities.

To create a new activity, select the Add New Item option for the `Calculator` project, and select the Code Activity template. You can find this new item template in the Workflow category, and it is used to create a new activity that is implemented in code rather than assembled from existing activities. Name the file for the new activity `ParseCalculatorArgs.cs`. The complete implementation for this activity is shown here:

```
using System;
using System.Activities;

namespace Calculator
{
```

The base class for this activity is `CodeActivity`. WF provides several base classes that you can derive from to implement your own activities. Your choice of base class depends on the requirements of the custom activity.

■ **Note** Chapter 3 discusses the activity base classes in greater detail.

`CodeActivity` is the simplest of the base classes and is the one that you want in this case. It provides minimal access to the workflow runtime and is designed as a simple way to execute your own code within the workflow model.

The activity defines a total of four arguments that are exposed as public properties: one input and three output. Arguments are defined using the generic `InArgument<T>` or `OutArgument<T>` class (depending on the intended direction). In both cases, the type parameter to the generic class specifies the type of the argument. The arguments determine the shape of this activity, just as the input parameters and the return value determine the shape of a C# method. They are the contract with any other WF classes that need to interact with this activity.

I've added `RequiredArgumentAttribute` to the input argument. This identifies that argument as being required and is used to produce an error indicator in the designer if this argument is not set. Adding this attribute isn't a requirement, but it is a good practice to identify required arguments. Anything that helps the consumers of your custom activities to avoid errors is worth the minimal effort.

```
public sealed class ParseCalculatorArgs : CodeActivity
{
    [RequiredArgument]
    public InArgument<String> Expression { get; set; }
    public OutArgument<Double> FirstNumber { get; set; }
    public OutArgument<Double> SecondNumber { get; set; }
    public OutArgument<String> Operation { get; set; }
```

The `Execute` method is where the real work of this activity takes place. This method is defined in the base class as virtual and is overridden here. The `CodeActivityContext` passed to the `Execute` method provides access to the execution context for this activity. The execution context determines the runtime environment that is available to this activity while it executes. This includes the set of variables in scope and that can be safely referenced by the activity.

Before doing anything else, the three output arguments are set to default values. The `FirstNumber` and `SecondNumber` arguments represent the two numbers that will be parsed from the expression and are set to zero. The `Operation` argument will be later set to the operation (+, -, *, /). The `Set` method is called when setting the value of an argument. This method requires you to pass the context object because it determines the current scope of the argument. You aren't simply setting the value for a global argument that you directly manage. You're setting the value for a single instance of an argument that is in scope during a single execution of this activity.

```
    protected override void Execute(CodeActivityContext context)
    {
        FirstNumber.Set(context, 0);
        SecondNumber.Set(context, 0);
        Operation.Set(context, "error");
```

In a similar manner, the value for the `Expression` argument is retrieved and stored in a local variable. This argument is the single string that contains the entire arithmetic expression to be parsed. The `Get` method of the argument is called to retrieve the value, passing the execution context object as was done with the `Set` method.

The remainder of the code is simple C# parsing logic that splits the expression into its respective parts. As the three parts of the expression are identified, the values are used to set the output arguments.

```
            String line = Expression.Get(context);
            if (!String.IsNullOrEmpty(line))
            {
                String[] arguments = line.Split(' ');
                if (arguments.Length == 3)
                {
                    Double number = 0;
                    if (Double.TryParse(arguments[0], out number))
                    {
                        FirstNumber.Set(context, number);
                    }
                    Operation.Set(context, arguments[1]);
                    if (Double.TryParse(arguments[2], out number))
                    {
                        SecondNumber.Set(context, number);
                    }
                }
            }
        }
    }
}
```

At this point you should build the project to ensure that the code for this activity was entered correctly. Building the project also adds this activity to the Toolbox, making it available to you when declaring the workflow. Since the `Workflow1.xaml` file that was generated with the new project was deleted, you'll need to comment out or remove one line in the `Program.cs` file in order to successfully build the project.

Defining Arguments

Now that you've implemented the custom activity, it's time to declare the workflow. Open the `Calculator.xaml` file in the workflow designer if it isn't already open. The first order of business is to define the input and output arguments for the workflow. You do this just as you did in the previous example by clicking the Arguments button to open the Argument Editor. Enter these two arguments:

Name	Direction	Argument Type
ArgExpression	In	String
Result	Out	Double

The `ArgExpression` argument is the arithmetic expression that was entered by the user, and the `Result` is the return value from the workflow containing the result of the calculation. Figure 1-8 shows the Argument Editor after I've entered these arguments.

Name	Direction	Argument type	Default value
ArgExpression	In	String	*Enter a VB expression*
Result	Out	Double	*Default value not supported*
Create Argument			

Variables | Arguments | Imports | 100%

Figure 1-8. *Argument Editor with input and output arguments*

■ **Note** You may have noticed that I didn't add the *Arg* prefix to the `Result` argument. That's because WF uses the argument name `Result` in certain types of activities as the name for a single output argument. I just happen to like that convention, so I kept it as `Result`. I also generally don't add the *Arg* prefix to arguments that I define in code, such as in this custom activity. I reserve the prefix for use within the workflow. These are my conventions—you should feel free to define ones that make sense for you.

Defining Variables

You saw that arguments define the shape of an activity and define its contract with other WF classes. And the arguments that you define for a workflow serve the same purpose on a larger scale. But what do you do when you need to maintain internal state within a workflow? That's a job for workflow variables. You define workflow variables when you have data that must be maintained throughout the life of the workflow or when you have transient data that is passed between activities. Variables are always internal to the workflow, in much the same way that private or local variables are internal to a C# class or method.

Workflow variables are entered by pressing the Variables button located in the lower-left corner of the workflow designer. Clicking this button opens the Variable Editor, which operates in a similar fashion as the Argument Editor. When defining variables, you don't need to define a direction since they are always internal. However, you do need to define the scope for each variable. All variables are scoped to a single activity. Just as in traditional C# programming, the scope of a variable determines its visibility, which in this case means which activities can reference the variable. If you decide to scope a variable at the root (topmost) activity of a workflow, it is visible to all activities. But you can also define variables for other composite activities in the workflow that are designed to accept variables. For example, you can define variables for the `Sequence` activity, but you can't define variables for the custom `ParseCalculatorArgs` that you developed earlier in this example.

The scoping of variables is important since it not only determines their visibility but also determines when they can be disposed of and garbage collected. Just as in a traditional C# program, a local variable that is defined within a method can be freed and garbage collected as soon as the method ends and the variable goes out of scope. In like manner, the memory used by workflow variables can be freed once the activity that defines them goes out of scope.

■ **Note** The scoping of workflow variables is one of the major conceptual changes between WF 3.*x* and WF 4. In WF 3.*x*, workflow variables were always scoped at the root workflow level. Although this made it easy to share variables between activities, it also led to problems since all data was essentially global. It was sometimes difficult to identify all the activities that referenced a particular variable and determine how their interactions modified the data. Since the global data was always in scope, all workflow variables had to be persisted whenever the workflow was persisted (written to disk when it was idle). The introduction of variable scope into the WF 4 model makes it much clearer how the variables are being used and by which activities. It also improves the performance when persisting workflows since only those variables that are currently in scope need to be persisted.

For this workflow, you need to define the variables listed here. You need these variables as a temporary storage location for the output arguments of the `ParseCalculatorArgs` activity. The variables will then be used as input to subsequent activities defined within the workflow.

Name	Variable Type	Scope
FirstNumber	Double	Sequence
SecondNumber	Double	Sequence
Operation	String	Sequence

To set the scope of a variable, you can either select the activity first or select it from the list in the Variable Editor. The Variable Editor should look like Figure 1-9 after you've entered the variables.

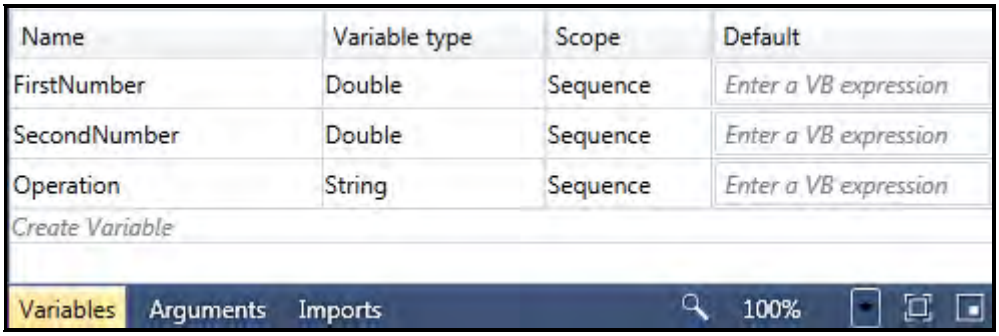

Figure 1-9. *Variable Editor*

Adding the Custom Activity

Add an instance of the custom `ParseCalculatorArgs` activity that you developed earlier to the open `Sequence` activity. You should see this activity at the top of the Toolbox just like the standard activities that ship with WF. Figure 1-10 is the top of the Toolbox showing this custom activity.

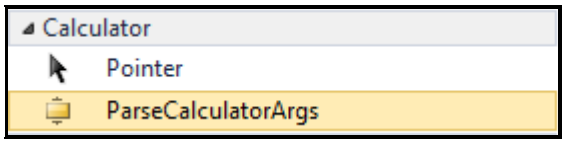

Figure 1-10. *Toolbox showing custom activity*

After dropping a single instance of this activity onto the `Sequence` activity, open the Properties window for the `ParseCalculatorArgs` activity. The four arguments that you defined for this activity are shown here as properties. Your job now is to wire up these properties to the workflow arguments and variables that you defined in the previous steps. Here are the property assignments that you need to make:

Property Name	Value	Description
Expression	ArgExpression	Assigned to the workflow InArgument
FirstNumber	FirstNumber	Assigned to the workflow Variable
Operation	Operation	Assigned to the workflow Variable
SecondNumber	SecondNumber	Assigned to the workflow Variable

In this particular case, most of the property names just happen to be the same as the argument or variable that you are assigning to each property. But that won't always be the case. This assigns the `InArgument` named `Expression` to the `InArgument` of the workflow named `ArgExpression`. The other properties are each defined as an `OutArgument` of the activity and are assigned to workflow variables. Those three variables will be used by subsequent steps in the workflow.

As you enter these argument or variable names, keep in mind that you are really entering workflow expressions. For example, the `Expression` property (an `InArgument` of the activity) is defined as a string. Since you are entering an expression, you can assign any value to this property as long as it resolves to a string. You could enter a literal string, or in this case, you are assigning the property to the value of the workflow argument named `ArgExpression`. When the activity executes and the code retrieves the value for the `InArgument` named `Expression` (using the `Get` method), the workflow expression that you entered here is evaluated. Since the expression is a reference to the workflow `InArgument`, the value of that argument is returned.

In a similar way, the other properties are assigned workflow expressions that will evaluate to the named workflow variables. Since the other properties are each defined as an `OutArgument` of the activity, you can't assign a literal value to them. They must be assigned to a workflow variable or `OutArgument`.

Figure 1-11 is the Properties window after I have entered expressions for all the activity properties.

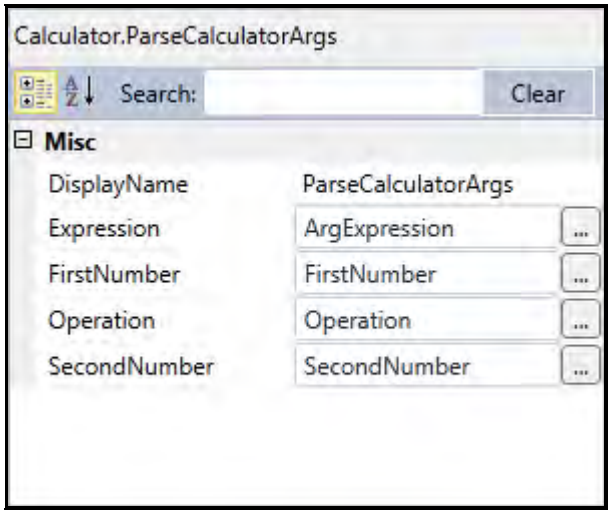

Figure 1-11. *ParseCalculatorArgs properties*

Adding the Switch<T> and Assign Activities

Now that the workflow has parsed the arithmetic expression, it's time to make decisions and perform some basic arithmetic. The only real decision that must be made is what type of arithmetic operation the user wants to perform.

You can make that decision in several ways within the workflow model, but the one that I've chosen to use for this example is the `Switch<T>` activity. This generic activity operates just like a `switch` statement in C#, allowing you to execute a different set of activities based on the value of a single expression.

Drag and drop an instance of the `Switch<T>` activity to the location just below the `ParseCalculatorArgs` activity. When you do, you will be prompted to select the type for this generic activity. The generic type must match the type of the expression that will be evaluated for branching. In this example, the branching decisions will be made on the value of the `Operation` workflow variable, which is a string. So, select String when you are prompted to select a generic type. Figure 1-12 is the type selection dialog after I made my selection.

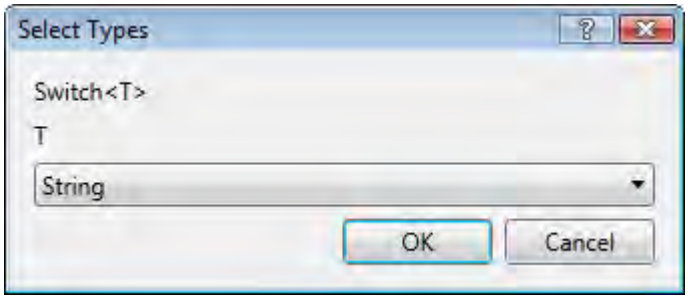

Figure 1-12. *Generic type selection dialog*

The `Switch<T>` activity should already be expanded for your use, but you can double-click it to expand it if necessary. The `Expression` property is where you supply the workflow expression that you want to use for branching. Enter the `Operation` variable name for this property. You can enter it directly within the workflow designer, or you can switch to the Properties window and enter it there.

The `Switch<T>` activity allows you to enter any number of cases, each one containing the activities that you want to execute when that case is true. In this example, each case is one of the supported arithmetic operations (+, -, *, /). There is also a placeholder for a default case that is executed when none of the other cases is true.

To add the first case to support addition, click "Add new case" at the bottom of the activity. The left side of the case is where you enter the value to match, and the right side is the container for the activities to execute when the case is true. For this first case, enter **+** on the left side (where it is labeled "Input case here").

Since this first case is for addition, you need to add the `FirstNumber` and `SecondNumber` variables together and assign the sum to the `Result` argument of the workflow. You can accomplish this with another one of the standard activities included with WF, the `Assign` activity (found in the Primitives Toolbox category). This activity assigns an expression to a variable or argument. Drag and drop an instance of the `Assign` activity to the open addition case that you just added. The `Assign` activity has two properties. The `Assign.To` property is the variable or argument that receives the assignment. The `Assign.Value` property is the expression that will be assigned to the `Assign.To` property. Enter `Result` for the `Assign.To` property name. This is the workflow `OutArgument` that is used to return the result of the arithmetic operation. Enter this expression in the `Assign.Value` property:

```
FirstNumber + SecondNumber
```

This expression adds the two numbers and places the sum in the `Result` argument of the workflow. Your `Switch<T>` activity should look like Figure 1-13 at this point.

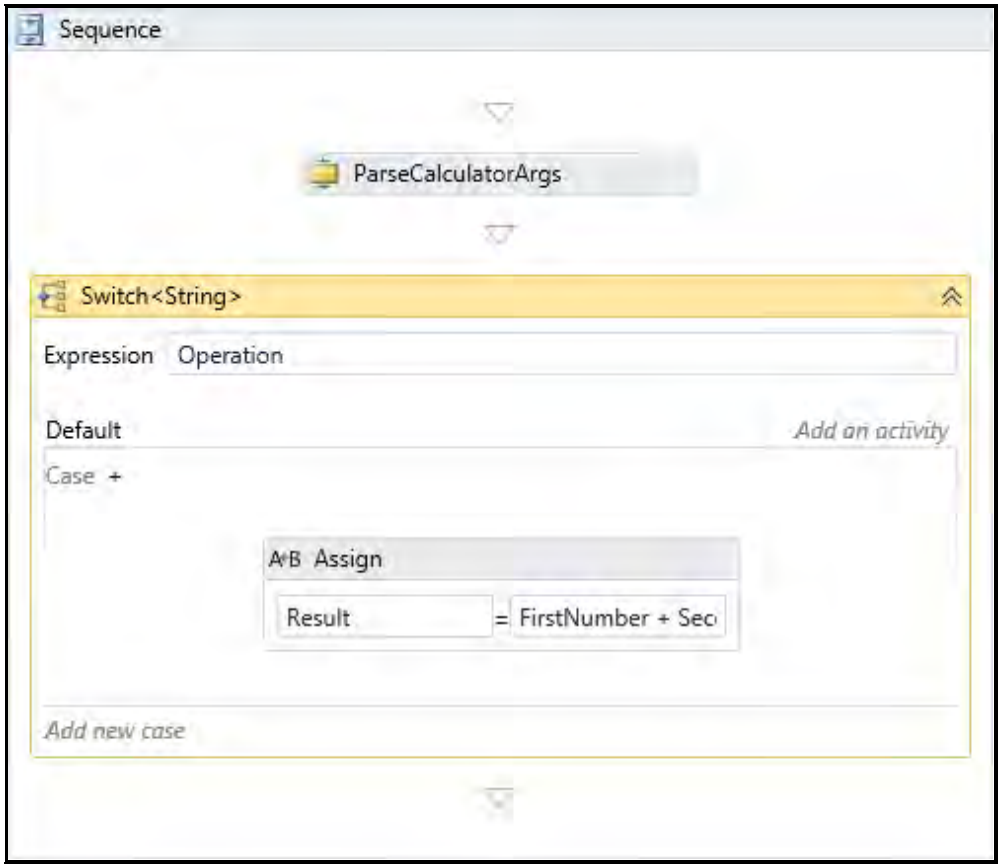

Figure 1-13. *Switch activity with the addition case*

Add three additional cases to the `Switch<T>` activity to handle subtraction, multiplication, and division. Add an `Assign` activity to each one to perform the requested operation. Here are the parameters that you should enter. Even though the case for addition has already been entered, the parameters for this case are included in this list for completeness.

Case	Assign To	Assign Value
+	Result	FirstNumber + SecondNumber
-	Result	FirstNumber - SecondNumber
*	Result	FirstNumber * SecondNumber
/	Result	FirstNumber / SecondNumber

■ **Note** In this example, you added a single `Assign` activity to each case within the `Switch<T>`. However, you're not limited to executing a single activity for each case. You could have added a `Sequence` activity to the case and then added multiple child activities to the `Sequence` activity. In this way, the workflow model allows you to compose multiple layers of child activities that are executed at the proper time.

You need to add one final activity to this workflow to complete it. The Default case of the `Switch<T>` activity is executed when none of the other cases is true. Instead of ignoring the error, go ahead and drop a `Throw` activity onto the right side of the Default case. The `Throw` activity is used to throw a .NET exception. Enter this expression in the `Exception` property of the `Throw` activity to raise an exception if the user enters an invalid operation:

```
New InvalidOperationException("Operation Invalid")
```

The `Switch<T>` activity should now look like Figure 1-14.

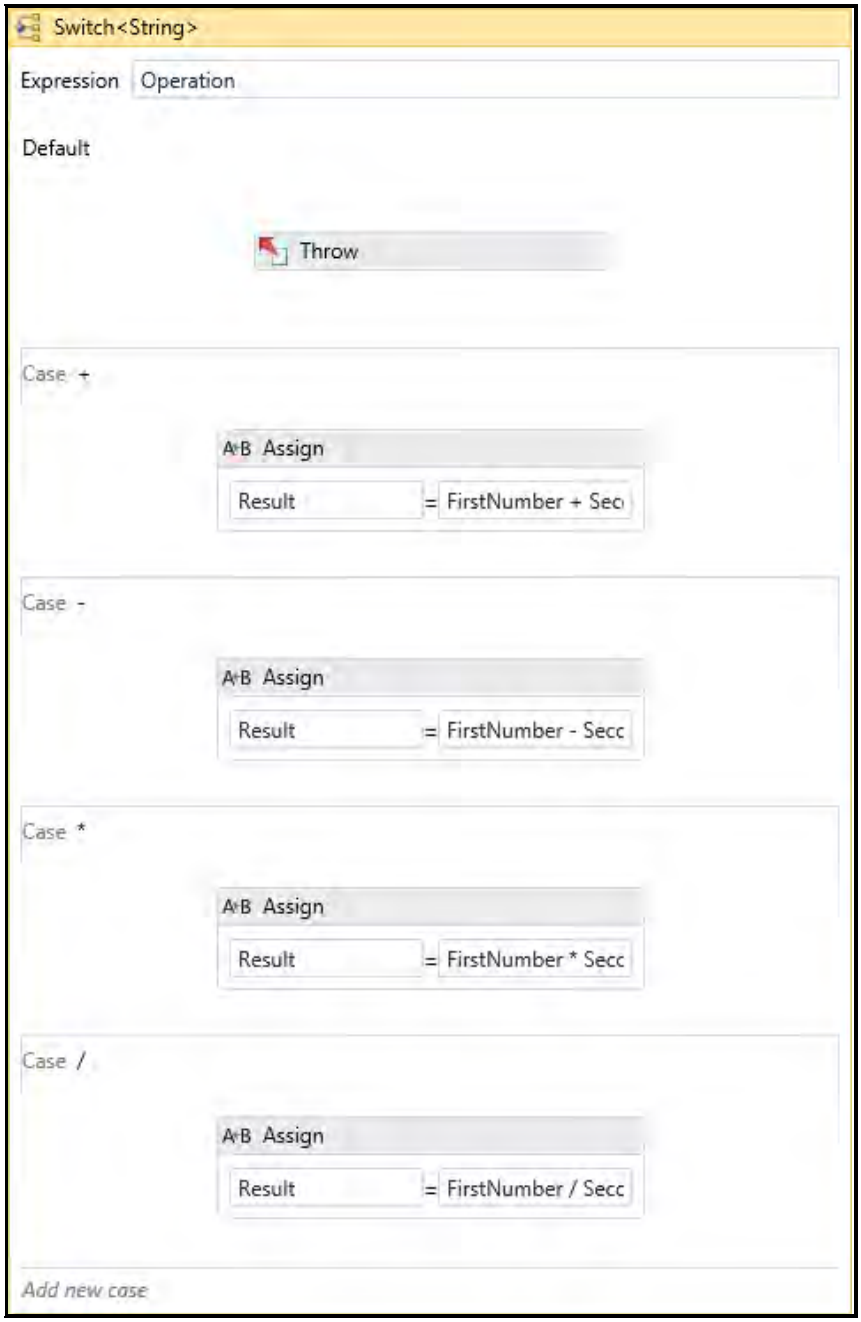

Figure 1-14. *Completed Switch activity*

When you're done working in the expanded view of the `Switch<T>` activity, you can return to the parent activity using the breadcrumb navigation at the top of the designer. Figure 1-15 shows the navigation bar that you should see at this point. Clicking Sequence will return you to the parent activity. Clicking Calculator will return all the way to the root activity of the workflow.

Figure 1-15. *Breadcrumb navigation*

Hosting the Workflow

The code that you need to add to the `Program.cs` file is shown here. The code to start the workflow is similar to what you've already seen in the previous examples. The major difference from the previous hosting code is that the entire process of starting a workflow has been put into a `while` loop. At the top of the loop, the user is asked to enter an arithmetic expression or to enter the literal "quit" to exit the program. The string that is accepted from the console is passed to the workflow as the expression to solve.

```
using System;
using System.Activities;
using System.Collections.Generic;

namespace Calculator
{
    class Program
    {
        static void Main(string[] args)
        {
            while (true)
            {
                Console.WriteLine("Enter an expression or 'quit' to exit");
                String expression = Console.ReadLine();
                if (!String.IsNullOrEmpty(expression))
                {
                    if (expression.Trim().ToLower() == "quit")
                    {
                        Console.WriteLine("Exiting program");
                        return;
                    }
                }

                Calculator wf = new Calculator();
                wf.ArgExpression = expression;
```

Output from the workflow is returned by the `Invoke` method in the form of a `IDictionary<String, Object>`. The output value for an argument is retrieved from the dictionary using its name. Only workflow arguments defined with a direction of out or in/out are returned in this collection. This code also wraps the `Invoke` method in a `try/catch` block in order to catch the exception that might be thrown by the workflow if an invalid expression is entered.

```
                try
                {
                    IDictionary<String, Object> results =
                        WorkflowInvoker.Invoke(wf);
                    Console.WriteLine("Result = {0}", results["Result"]);
                }
                catch (InvalidOperationException exception)
                {
                    Console.WriteLine(exception.Message.ToString());
                }
            }
        }
    }
}
```

■ **Note** In this example, the work of polling the user for input and looping until they are done is handled by the host application. It could have also been handled within the workflow itself. However, at this point in the discussion I wanted to keep the example as simple as possible and not prematurely introduce additional workflow concepts related to host communication. Chapter 8 provides additional information on host communication and the interactions between the host application and the workflow instance.

Running the Application

After building the project, you should be ready to take it out for a test-drive. Make sure you set the `Calculator` project as the startup project. When you first run the program, you should be prompted to enter an arithmetic expression to solve. If you enter a valid expression such as `1 + 1`, you should receive the correct answer of "2" and be prompted to enter another expression:

```
Enter an expression or 'quit' to exit

1 + 1

Result = 2

Enter an expression or 'quit' to exit
```

Try all of the other operations (-, *, /) to make sure that they all work correctly. If you enter an invalid operation or an expression that is not in the correct format (two numbers and an operation separated by spaces), the Default case in the `Switch<T>` activity is executed, and an exception is thrown. The exception is caught by the host application, and the exception message is displayed:

```
Enter an expression or 'quit' to exit

bad expr

Operation Invalid

Enter an expression or 'quit' to exit
```

When you're done testing this application, you can enter the literal "quit" to exit the program. Here's a representative example of the results that you should see:

```
Enter an expression or 'quit' to exit

5 + 3

Result = 8

Enter an expression or 'quit' to exit

100 - 75

Result = 25

Enter an expression or 'quit' to exit

8 * 7.56

Result = 60.48

Enter an expression or 'quit' to exit

123 / 2

Result = 61.5

Enter an expression or 'quit' to exit

1+1
```

```
Operation Invalid

Enter an expression or 'quit' to exit

quit

Exiting program

Press any key to continue . . .
```

Debugging the Application

Before you leave this example, this is a good time to familiarize yourself with the debugging support provided by the development environment. As you might expect, all of the normal C# debugging is still available. You can set a breakpoint anywhere within the C# code, and the debugger will stop when it reaches that breakpoint. For example, you can set a breakpoint within the custom `ParseCalculatorArgs` activity and step through the code as the activity executes.

But the WF support in Visual Studio also provides additional ways to debug your workflows. You can also set breakpoints directly on an activity while you are in the workflow designer. For example, Figure 1-16 shows the workflow after I set a breakpoint on the `Switch<T>` activity.

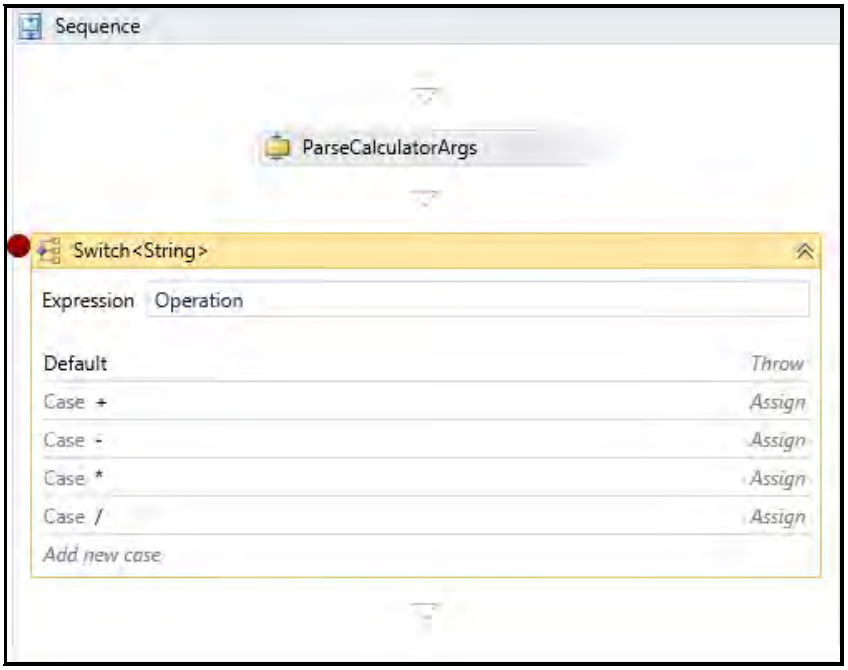

Figure 1-16. *Workflow with breakpoint set*

When I run the application with debugging (F5), execution breaks just before the activity begins execution. This is shown in Figure 1-17.

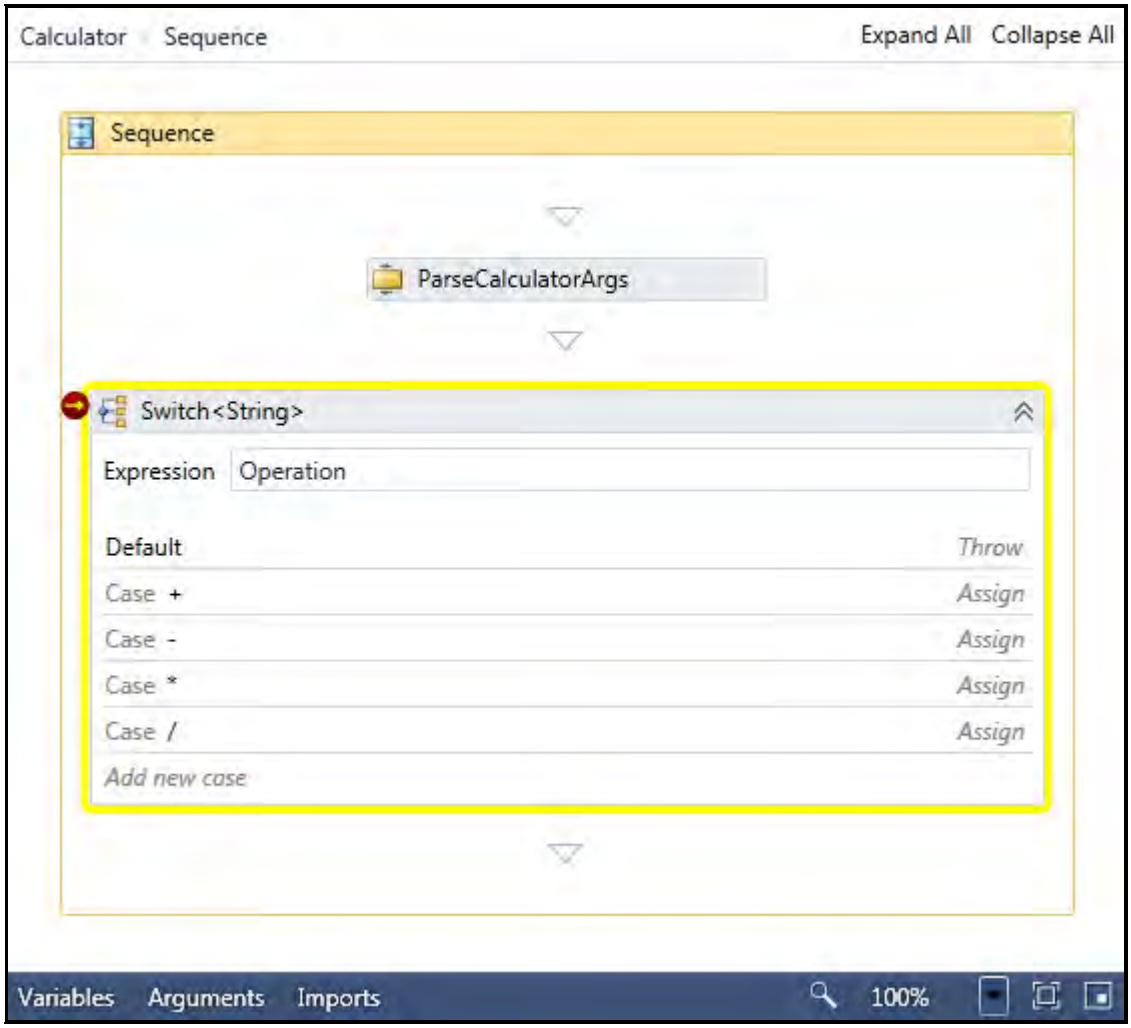

Figure 1-17. *Workflow with breakpoint hit*

Once the breakpoint is hit and execution has stopped, you have the usual set of debugging actions such as "step into" or "step over." You can also use the debugger when you are viewing the workflow Xaml in Code View. You can set breakpoints directly in the Xaml and use debugger actions to step through the activities once the breakpoint has been hit. In fact, breakpoints set in one view (for example in the workflow designer) are carried forward when you view the same workflow in the other view.

Unit Testing

One of the benefits of the workflow model is that it forces you to think about encapsulating your business logic into discrete components. It's good practice to do this even when you are developing nonworkflow applications. But everything in WF is built around the idea that you are coordinating units of work (encapsulated in activities) that have (ideally) already been tested. That's where unit testing comes into the picture. The same design features of WF that make it easy to coordinate these separate units of work also make it easy to test them.

Before completely leaving the calculator example, I want to briefly implement a few unit tests for the application. Unit tests in Visual Studio live in their own separate projects, so to begin, add a new project to the solution named `CalculatorTest`. Use the project template named Test Project that is in the Test category. The project template creates a sample `UnitTest1.cs` file for you. You can delete this file since it won't be needed.

■ **Note** The goal of these tests is to demonstrate how you can implement unit tests for your WF activities and workflows. My assumption is that you are already familiar with the unit testing support within Visual Studio. If you are interested in a tutorial or reference on unit testing, I suggest one of the excellent books on the subject.

The unit test project you added already has a reference to the Microsoft unit test framework, but you need to add an assembly reference to `System.Activities` to provide access to the WF classes. You should also add a project reference to the `Calculator` project since it contains the classes that you need to test. That project should be in the same solution as this test project.

Testing the Custom Activity

The first tests are for the custom `ParseCalculatorArgs` activity. Select Add New Test for the test project, and select Basic Unit Test as the item template. Name the new test class `ParseCalculatorArgsTest`. The code for this class follows:

```
using System;
using System.Activities;
using System.Collections.Generic;
using Calculator;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace CalculatorTest
{
    [TestClass]
    public class ParseCalculatorArgsTest
    {
```

This first method tests for valid results using a valid parameter that is passed to the activity. The second method tests the negative condition by passing an invalid argument.

This test code illustrates one important new concept: you can execute individual activities just as easily as you can execute a complete workflow. To the workflow runtime, a workflow is really just another activity that has one or more child activities.

```
        [TestMethod]
        public void ValidExpressionTest()
        {
            Dictionary<String, Object> parameters
                = new Dictionary<string, object>();
            parameters.Add("Expression", "1 + 2");

            IDictionary<String, Object> outputs = WorkflowInvoker.Invoke(
                new ParseCalculatorArgs(), parameters);

            Assert.IsNotNull(outputs, "outputs should not be null");
            Assert.AreEqual(3, outputs.Count, "outputs count is incorrect");
            Assert.AreEqual((Double)1, outputs["FirstNumber"],
                "FirstNumber is incorrect");
            Assert.AreEqual((Double)2, outputs["SecondNumber"],
                "SecondNumber is incorrect");
            Assert.AreEqual("+", outputs["Operation"],
                "Operation is incorrect");
        }

        [TestMethod]
        public void InvalidExpressionTest()
        {
            Dictionary<String, Object> parameters
                = new Dictionary<string, object>();
            parameters.Add("Expression", "badexpression");

            IDictionary<String, Object> outputs = WorkflowInvoker.Invoke(
                new ParseCalculatorArgs(), parameters);

            Assert.IsNotNull(outputs, "outputs should not be null");
            Assert.AreEqual(3, outputs.Count, "outputs count is incorrect");
            Assert.AreEqual("error", outputs["Operation"],
                "Operation is incorrect");
        }
    }
}
```

After building the project, you should be ready to execute the tests. There are several ways to do this, but I find that the easiest is to start the test from within the current source file. You can scroll up to the top of the file and right-click the class name (`ParseCalculatorArgsTest`). One of the options should be Run Tests, which runs all of the tests that are currently in scope. Since you selected the class name, it runs all of the tests for the class. Figure 1-18 shows the Test Results panel after I run these tests:

Test run completed Results: 2/2 passed; Item(s) checked: 0

Result	Test Name	Project	Error Message
Passed	InvalidExpressionTest	CalculatorTest	
Passed	ValidExpressionTest	CalculatorTest	

Figure 1-18. *Successful unit tests for ParseCalculatorArgs*

Testing the Workflow

The previous tests exercised only the `ParseCalculatorArgs` activity. Now that you have some reassurance that the activity works by itself, you can add a set of tests for the calculator workflow. Add another unit test class to the same project, and name it `CalculatorTest`. Here is the code for the `CalculatorTest.cs` file:

```
using System;
using System.Activities;
using System.Collections.Generic;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace CalculatorTest
{
    [TestClass]
    public class CalculatorTest
    {
        [TestMethod]
        public void AddTest()
        {
            Dictionary<String, Object> parameters
                = new Dictionary<string, object>();
            parameters.Add("ArgExpression", "111 + 222");

            IDictionary<String, Object> outputs = WorkflowInvoker.Invoke(
                new Calculator.Calculator(), parameters);

            Assert.IsNotNull(outputs, "outputs should not be null");
            Assert.AreEqual(1, outputs.Count, "outputs count is incorrect");
            Assert.AreEqual((Double)333, outputs["Result"],
                "Result is incorrect");
        }
```

```
        [TestMethod]
        public void SubtractTest()
        {
            Dictionary<String, Object> parameters
                = new Dictionary<string, object>();
            parameters.Add("ArgExpression", "333 - 222");

            IDictionary<String, Object> outputs = WorkflowInvoker.Invoke(
                new Calculator.Calculator(), parameters);

            Assert.IsNotNull(outputs, "outputs should not be null");
            Assert.AreEqual(1, outputs.Count, "outputs count is incorrect");
            Assert.AreEqual((Double)111, outputs["Result"],
                "Result is incorrect");
        }

        [TestMethod]
        public void MultiplyTest()
        {
            Dictionary<String, Object> parameters
                = new Dictionary<string, object>();
            parameters.Add("ArgExpression", "111 * 5");

            IDictionary<String, Object> outputs = WorkflowInvoker.Invoke(
                new Calculator.Calculator(), parameters);

            Assert.IsNotNull(outputs, "outputs should not be null");
            Assert.AreEqual(1, outputs.Count, "outputs count is incorrect");
            Assert.AreEqual((Double)555, outputs["Result"],
                "Result is incorrect");
        }

        [TestMethod]
        public void DivideTest()
        {
            Dictionary<String, Object> parameters
                = new Dictionary<string, object>();
            parameters.Add("ArgExpression", "555 / 5");

            IDictionary<String, Object> outputs = WorkflowInvoker.Invoke(
                new Calculator.Calculator(), parameters);

            Assert.IsNotNull(outputs, "outputs should not be null");
            Assert.AreEqual(1, outputs.Count, "outputs count is incorrect");
            Assert.AreEqual((Double)111, outputs["Result"],
                "Result is incorrect");
        }
    }
}
```

This code is very similar to the previous tests. In this case, the focus is on testing the entire workflow rather than a single activity, so the `WorkflowInvoker` executes the workflow instead of the custom activity.

I've included separate test methods for each of the possible operations (add, subtract, multiply, and divide).

When you're ready, you can execute all the tests for the entire project from the Test menu. Select the Run option and then All Tests in Solution. If all goes well, your results should look like Figure 1-19.

Test run completed Results: 6/6 passed; Item(s) checked: 0

Result	Test Name	Project	Error Message
Passed	InvalidExpressionTest	CalculatorTest	
Passed	ValidExpressionTest	CalculatorTest	
Passed	MultiplyTest	CalculatorTest	
Passed	AddTest	CalculatorTest	
Passed	DivideTest	CalculatorTest	
Passed	SubtractTest	CalculatorTest	

Figure 1-19. *Successful unit tests for the solution*

■ **Note** In this example, I chose to fully implement the custom activity and the workflow first and perform the initial testing from the console application. The unit tests were developed afterward. The test-first school of thought is that you should develop your tests first before you have fully implemented the classes that you want to test. Initially the tests will fail, but that changes after the classes are fully implemented.

You can implement your applications and unit tests either way. Because workflow applications are built from a number of discrete activities, it is fairly easy to follow the test-first methodology if that is your choice. I chose to fully implement the application first because the focus of this book is on demonstrating WF concepts, not on unit testing. And quite frankly, it's more interesting to see a working application rather than a list of green check marks.

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 workflow designer, workflow activities, the `Sequence` and `WriteLine` activities, and the `WorkflowInvoker` class that you used to execute a workflow. This first example also introduced you to the Xaml format that is used to declare workflows. In the second example, you learned how to pass parameters using workflow arguments.

The calculator example demonstrated how to construct a simple custom activity and one way to declare branching decisions within a workflow. You learned how to define input and output arguments for the custom activity and how to use workflow variables to pass values between activities. This example also introduced you to the `Switch<T>`, `Assign`, and `Throw` standard activities.

After demonstrating some of the additional debugger fweatures that are available for WF, the chapter concluded with a set of unit tests for the calculator example.

In the next chapter, you'll learn more about the major components in Windows Workflow Foundation.