Craig McMurtry
Marc Mercuri
Nigel Watling
Matt Winkler

Windows® Communication Foundation 3.5

UNLEASHED

SAMS

Windows Communication Foundation 3.5 Unleashed

ISBN-13: 978-0-672-33024-7
ISBN-10: 0-672-33024-5

Library of Congress Cataloging-in-Publication Data:

Windows Communication Foundation 3.5 unleashed / Craig McMurtry ... [et al.]. – 2nd ed.

p. cm.

Includes bibliographical references and index.

ISBN 978-0-672-33024-7

1. Application software–Development. 2. Electronic data processing–Distributed processing. 3. Microsoft Windows (Computer file) 4. Web services. I. McMurtry, Craig.

QA76.76.A65W59 2009

005.4'46–dc22

2008038773

Printed in the United States of America

First Printing October 2008

Trademarks

Warning and Disclaimer

Bulk Sales

Editor-in-Chief
Karen Gettman

Executive Editor
Neil Rowe

Acquisitions Editor
Brook Farling

Development Editor
Mark Renfrow

Managing Editor
Patrick Kanouse

Project Editor
SanDee Phillips

Copy Editor
Mike Henry

Indexer
Ken Johnson

Proofreaders
Kathy Ruiz
Leslie Joseph

Technical Editor
John Lambert

Publishing Coordinator
Cindy Teeters

Cover and Interior Designer
Gary Adair

Composition
Mark Shirar

Introduction

The Windows Communication Foundation, which was code-named Indigo, is a technology that allows pieces of software to communicate with one another. There are many other such technologies, including the Component Object Model (COM) and Distributed Component Object Model (DCOM), Remote Method Invocation (RMI), Microsoft Message Queuing (MSMQ), and WebSphere MQ. Each of those works well in a particular scenario, not so well in others, and is of no use at all in some cases. The Windows Communication Foundation is meant to work well in any circumstance in which a Microsoft .NET assembly must exchange data with any other software entity. In fact, the Windows Communication Foundation is meant to always be the very best option. Its performance is at least on par with that of any other alternative and is usually better; it offers at least as many features and probably several more. It is certainly always the easiest solution to program.

Concretely, the Windows Communication Foundation consists of a small number of .NET libraries with several new sets of classes that it adds to the Microsoft .NET Framework class library, for use with version 2.0 and later of the .NET Common Language Runtime. It also adds some facilities for hosting Windows Communication Foundation solutions to the 5.1 and later versions of Internet Information Services (IIS), the web server built into Windows operating systems.

The Windows Communication Foundation is distributed free of charge as part of a set that includes several other technologies, including the Windows Presentation Foundation, which was code-named Avalon, Windows CardSpace, which was code-named InfoCard, and the

Windows Workflow Foundation. Prior to its release, that group of technologies was called WinFX, but it was renamed the .NET Framework 3.0 in June 2006. Despite that name, the .NET Framework 3.0 and 3.5 is still primarily just a collection of classes added to the .NET Framework 2.0 for use with the 2.0 version of the .NET Common Language Runtime, along with some enhancements to the Windows operating system, as shown in Figure I.1.

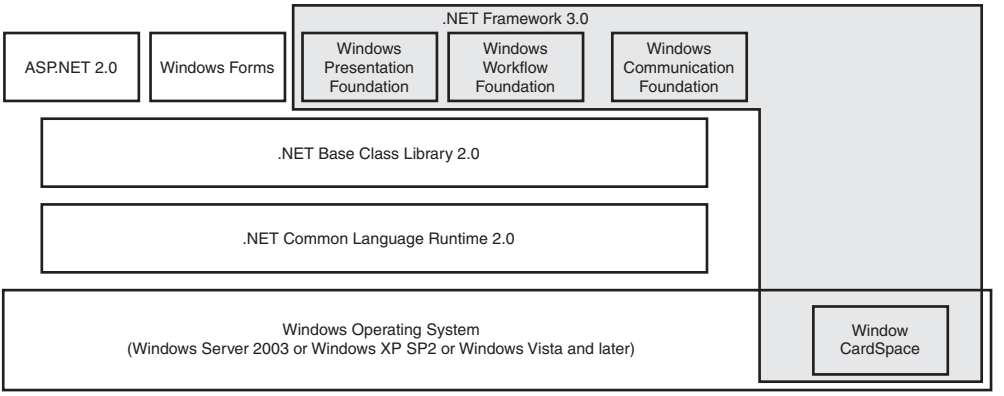

FIGURE I.1 The .NET Framework 3.0.

You can install the .NET Framework 3.0 and 3.5 on Windows XP Service Pack 2, Windows Server 2003, and Windows Server 2003 R2. The runtime components are preinstalled on Windows Vista. On Windows Server 2008 you can add the .NET Framework 3.0 via the Application Server Foundation role service. Only a very small number of features of the .NET Framework 3.0 are available exclusively on Windows Vista and later operating systems.

The .NET Framework 3.5 builds incrementally on top of .NET Framework 3.0. Features relevant to this book include web protocol support for building Windows Communication Foundation services, including AJAX, JSON, REST, POX, RSS and ATOM, workflow-enabled services and full tooling support in Visual Studio 2008. During development, .the NET Framework 3.5 was factored into “red” bits and “green” bits. The red bits were features from .NET Framework 3.0 and the goal was to provide Service Pack levels of compatibility. All the code that worked in 3.0 will work in 3.5. The green bits provide new, additional functionality. Again, the addition of an assembly containing new functionality should have no effect on existing code. The bottom line is that all the code in this book will work with .NET Framework 3.5 and all the code in this book (except the new features introduced in .NET Framework 3.5) should work in .NET Framework 3.0.

This book does not serve as an encyclopedic reference to the Windows Communication Foundation. Instead, it provides the understanding and knowledge required for most practical applications of the technology.

The book explains the Windows Communication Foundation while showing how to use it. So, typically, each chapter provides the precise steps for building a solution that demonstrates a particular aspect of the technology, along with a thorough explanation of each step. Readers who can program in C#, and who like to learn by doing, will be able to follow the steps. Those who prefer to just read will get a detailed account of the features of the Windows Communication Foundation and see how to use them.

To follow the steps in the chapters, you should have installed any version of Visual Studio 2005 or 2008 that includes the C# compiler. Free copies are available at http://msdn.microsoft.com/vstudio/express/. You should also have IIS, ASP.NET, and MSMQ installed.

The .NET Framework 3.0 or 3.5 is required, as you might expect. You can download them from http://www.microsoft.com/downloads/. The instructions in the chapters assume that all the runtime and developer components of the .NET Framework 3.0 or 3.5 have been installed. It is the runtime components that are preinstalled on Windows Vista and that can be added via the Server Manager on Windows Server 2008. The developer components consist of a Software Development Kit (SDK) and two enhancements to Visual Studio 2005. The SDK provides documentation, some management tools, and a large number of very useful samples. The enhancements to Visual Studio 2005 augment the support provided by IntelliSense for editing configuration files, and provide a visual designer for Windows Workflow Foundation workflows. These features are included in Visual Studio 2008.

To fully utilize Windows CardSpace, which is also covered in this book, you should install Internet Explorer 7. Internet Explorer 7 is also available from http://www.microsoft.com/downloads.

Starting points for the solutions built in each of the chapters are available for download from the book's companion page on the publisher's website, as well as from http://www.cryptmaker.com/WindowsCommunicationFoundationUnleashed. To ensure that Visual Studio does not complain about the sample code being from a location that is not fully trusted, you can, after due consideration, right-click the downloaded archive, choose Properties from the context menu, and click on the button labeled Unblock, shown in Figure I.2, before extracting the files from the archive.

Note that development on the Vista operating system is supported for Visual Studio 2008 and for Visual Studio 2005 with the Visual Studio 2005 Service Pack 1 Update for Windows Vista. This update is also available from http://www.microsoft.com/downloads. Developers working with an earlier version of Visual Studio 2005 on the Vista operating system should anticipate some compatibility issues. To minimize those issues, they can do two things. The first is to disable Vista's User Account Protection feature. The second is to always start Visual Studio 2005 by right-clicking on the executable or the shortcut, selecting Run As from the context menu that appears, and selecting the account of an administrator from the Run As dialog.

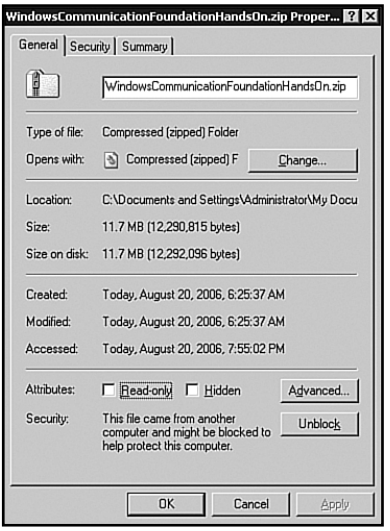

FIGURE I.2 Unblocking a downloaded source code archive.

As with the .NET Framework 3.5 when compared to the .NET Framework 3.0, this book is very similar to its predecessor. Changes include the addition of Visual Studio 2008 support and Chapter 3, "Data Representation and Durable Services," now covers durable services. The chapters on Windows CardSpace show the updated user interface and cover new features. Chapter 18, "Representational State Transfer and Plain XML Services," on REST and POX, includes details on the new syndication and JSON APIs. Perhaps the most significant change is a complete rewrite of Chapter 6, "Using the Windows Communication Foundation and the Windows Workflow Foundation Together," covering the much improved integration between Windows Workflow Foundation and the Windows Communication Foundation.

Many people contributed to this book. The authors would like to thank Joe Long, Eric Zinda, Angela Mills, Omri Gazitt, Steve Swartz, Steve Millet, Mike Vernal, Doug Purdy, Eugene Osvetsky, Daniel Roth, Ford McKinstry, Craig McLuckie, Alex Weinert, Shy Cohen, Yasser Shohoud, Kenny Wolf, Anand Rajagopalan, Jim Johnson, Andy Milligan, Steve Maine, Ram Pamulapati, Ravi Rao, Mark Garbara, Andy Harjanto, T. R. Vishwanath, Doug Walter, Martin Gudgin, Marc Goodner, Giovanni Della-Libera, Kirill Gavrylyuk, Krish Srinivasan, Mark Fussell, Richard Turner, Ami Vora, Ari Bixhorn, Steve Cellini, Neil Hutson, Steve DiMarco, Gianpaolo Carraro, Steve Woodward, James Conard, Nigel Watling, Vittorio Bertocci, Blair Shaw, Jeffrey Schlimmer, Matt Tavis, Mauro Ottoviani, John Frederick, Mark Renfrow, Sean Dixon, Matt Purcell, Cheri Clark, Mauricio Ordonez, Neil Rowe, Donovan Follette, Pat Altimore, Tim Walton, Manu Puri, Ed Pinto, Erik Weiss, Suwat Chitphakdibodin, Govind Ramanathan, Ralph Squillace, John Steer, Brad Severtson, Gary Devendorf, Kavita Kamani, George Kremenliev, Somy Srinivasan, Natasha Jethanandani, Ramesh Seshadri, Lorenz Prem, Laurence Melloul, Clemens Vasters, Joval Lowy, John Justice, David Aiken, Larry Buerk, Wenlong Dong, Nicholas Allen, Carlos

Figueira, Ram Poornalingam, Mohammed Makarechian, David Cliffe, David Okonak, Atanu Banerjee, Steven Metsker, Antonio Cruz, Steven Livingstone, Vadim Meleshuk, Elliot Waingold, Yann Christensen, Scott Mason, Jan Alexander, Johan Lindfors, Hanu Kommalapati, Steve Johnson, Tomas Restrepo, Tomasz Janczuk, Garrett Serack, Jeff Baxter, Arun Nanda, Luke Melton, and Al Lee.

A particular debt of gratitude is owed to John Lambert for reviewing the drafts. No one is better qualified to screen the text of a book on a programming technology than an experienced professional software tester. Any mistakes in the pages that follow are solely the fault of the writers, however.

The authors are especially grateful for the support of their wives. They are Marta MacNeill, Kathryn Mercuri, Sylvie Watling, and Libby Winkler. Matt, the only parent so far, would also like to thank his daughter, Grace.

CHAPTER 1
Prerequisites

IN THIS CHAPTER

- Partial Types
- Generics
- Nullable Value Types
- The Lightweight Transaction Manager
- Role Providers

To properly understand and work effectively with the Windows Communication Foundation, you should be familiar with certain facilities of the 2.0 versions of the .NET Framework and the .NET common language runtime. This chapter introduces them: partial types, generics, nullable value types, the Lightweight Transaction Manager, and role providers. The coverage of these features is not intended to be exhaustive, but merely sufficient to clarify their use in the chapters that follow.

Partial Types

Microsoft Visual C# 2005 allows the definition of a type to be composed from multiple partial definitions distributed across any number of source code files for the same module. That option is made available via the modifier `partial`, which can be added to the definition of a class, an interface, or a struct. Therefore, this part of the definition of a class

```
public partial MyClass
{
      private string myField = null;

      public string MyProperty
      {
            get
            {
                  return this.myField;
            }
      }
}
```

and this other part

```
public partial MyClass
{
      public MyClass()
      {
      }

      public void MyMethod()
      {
        this.myField = "Modified by my method.";
      }
}
```

can together constitute the definition of the type `MyClass`. This example illustrates just one use for partial types, which is to organize the behavior of a class and its data into separate source code files.

Generics

"Generics are classes, structures, interfaces, and methods that have placeholders for one or more of the types they store or use" (Microsoft 2006). Here is an example of a generic class introduced in the `System.Collections.Generic` namespace of the .NET Framework 2.0 Class Library:

```
public class List<T>
```

Among the methods of that class is this one:

```
public Add(T item)
```

Here, `T` is the placeholder for the type that an instance of the generic class `System.Collections.Generic.List<T>` will store. In defining an instance of the generic class, you specify the actual type that the instance will store:

```
List<string> myListOfStrings = new List<string>();
```

Then you can use the `Add()` method of the generic class instance like so:

```
myListOfStrings.Add("Hello, World");
```

Generics enable the designer of the `List<T>` class to define a collection of instances of the same unspecified type—in other words, to provide the template for a type-safe collection. A user of `List<T>` can employ it to contain instances of a type of the user's choosing, without the designer of `List<T>` having to know which type the user might choose. Note as well that whereas a type derived from a base type is meant to derive some of the functionality it requires from the base, with the remainder still having to be programmed, `List<string>` comes fully equipped from `List<T>`.

The class, `System.Collections.Generic.List<T>`, is referred to as a *generic type definition*. The placeholder, `T`, is referred to as a *generic type parameter*. Declaring

```
List<string> myListOfStrings;
```

yields `System.Collections.Generic.List<string>` as a *constructed type*, and `string` as a *generic type argument*.

Generics can have any number of generic type parameters. For example, `System.Collections.Generic.Dictionary<TKey,TValue>` has two.

The designer of a generic may use constraints to restrict the types that can be used as generic type arguments. This generic type definition

```
public class MyGenericType<T> where T: IComparable, new()
```

constrains the generic type arguments to types with a public, parameter-less constructor that implements the `IComparable` interface. This less restrictive generic type definition

```
public class MyGenericType<T> where T: class
```

merely constrains generic type arguments to reference types. Note that `T: class` includes both classes and interfaces.

Both generic and nongeneric types can have generic methods. Here is an example of a nongeneric type with a generic method:

```
using System;

public class Printer
{
    public void Print<T>(T argument)
    {
        Console.WriteLine(argument.ToString());
    }

    static void Main(string[] arguments)
    {
        Printer printer = new Printer();
        printer.Print<string>("Hello, World");
        Console.WriteLine("Done");
        Console.ReadKey();
    }
}
```

In programming a generic, it is often necessary to determine the type of generic argument that has been substituted for a generic type parameter. This revision to the preceding example shows how you can make that determination:

```
public class Printer
{
    public void Print<T>(T argument)
    {
        if(typeof(T) == typeof(string))
        {
            Console.WriteLine(argument);
        }
        else
        {
            Console.WriteLine(argument.ToString());
        }
    }

    static void Main(string[] arguments)
    {
        Printer printer = new Printer();
        printer.Print<string>("Hello, World");
        Console.WriteLine("Done");
        Console.ReadKey();
    }
}
```

A generic interface may be implemented by a generic type or a nongeneric type. Also, both generic and nongeneric types may inherit from generic base types.

```
public interface IMyGenericInterface<T>
{
    void MyMethod(T input);
}

public class MyGenericImplementation<T>: IMyGenericInterface<T>
{
    public void MyMethod(T input)
    {
    }
}

public class MyGenericDescendant<T> : MyGenericImplementation<T>
{
}

public class MyNonGenericImplementation : IMyGenericInterface<string>
{
    public void MyMethod(string input)
    {
```

```
    }
}

public class MyNonGenericDescendant : MyGenericImplementation<string>
{
}
```

Nullable Value Types

According to the Common Language Infrastructure specification, there are two ways of representing data in .NET: by a value type or by a reference type (Ecma 2006, 18). Although instances of value types are usually allocated on a thread's stack, instances of reference types are allocated from the managed heap, and their values are the addresses of the allocated memory (Richter 2002, 134–5).

Whereas the default value of a reference type variable is `null`, indicating that it has yet to be assigned the address of any allocated memory, a value type variable always has a value of the type in question and can never have the value `null`. Therefore, although you can determine whether a reference type has been initialized by checking whether its value is `null`, you cannot do the same for a value type.

However, there are two common circumstances in which you would like to know whether a value has been assigned to an instance of a value type. The first is when the instance represents a value in a database. In such a case, you would like to be able to examine the instance to ascertain whether a value is indeed present in the database. The other circumstance, which is more pertinent to the subject matter of this book, is when the instance represents a data item received from some remote source. Again, you would like to determine from the instance whether a value for that data item was received.

The .NET Framework 2.0 incorporates a generic type definition that provides for cases like these in which you want to assign `null` to an instance of a value type, and test whether the value of the instance is `null`. That generic type definition is `System.Nullable<T>`, which constrains the generic type arguments that may be substituted for `T` to value types. Instances of types constructed from `System.Nullable<T>` can be assigned a value of `null`; indeed, their values are `null` by default. Thus, types constructed from `System.Nullable<T>` are referred to as *nullable value types*.

`System.Nullable<T>` has a property, `Value`, by which the value assigned to an instance of a type constructed from it can be obtained if the value of the instance is not `null`. Therefore, you can write

```
System.Nullable<int> myNullableInteger = null;
myNullableInteger = 1;
if (myNullableInteger != null)
{
    Console.WriteLine(myNullableInteger.Value);
}
```

The C# programming language provides an abbreviated syntax for declaring types constructed from `System.Nullable<T>`. That syntax allows you to abbreviate

```
System.Nullable<int> myNullableInteger;
```

to

```
int? myNullableInteger;
```

The compiler will prevent you from attempting to assign the value of a nullable value type to an ordinary value type in this way:

```
int? myNullableInteger = null;
int myInteger = myNullableInteger;
```

It prevents you from doing so because the nullable value type could have the value `null`, which it actually would have in this case, and that value cannot be assigned to an ordinary value type. Although the compiler would permit this code,

```
int? myNullableInteger = null;
int myInteger = myNullableInteger.Value;
```

the second statement would cause an exception to be thrown because any attempt to access the `System.Nullable<T>.Value` property is an invalid operation if the type constructed from `System.Nullable<T>` has not been assigned a valid value of `T`, which has not happened in this case.

One proper way to assign the value of a nullable value type to an ordinary value type is to use the `System.Nullable<T>.HasValue` property to ascertain whether a valid value of `T` has been assigned to the nullable value type:

```
int? myNullableInteger = null;
if (myNullableInteger.HasValue)
{
    int myInteger = myNullableInteger.Value;
}
```

Another option is to use this syntax:

```
int? myNullableInteger = null;
int myInteger = myNullableInteger ?? -1;
```

by which the ordinary integer `myInteger` is assigned the value of the nullable integer `myNullableInteger` if the latter has been assigned a valid integer value; otherwise, `myInteger` is assigned the value of `-1`.

The Lightweight Transaction Manager

In computing, a transaction is a *discrete* activity—an activity that is completed in its entirety or not at all. A resource manager ensures that if a transaction is initiated on some

resource, the resource is restored to its original state if the transaction is not fully completed. A distributed transaction is one that spans multiple resources and therefore involves more than a single resource manager. A manager for distributed transactions has been incorporated into Windows operating systems for many years. It is the *Microsoft Distributed Transaction Coordinator*.

.NET Framework versions 1.0 and 1.1 provided two ways of programming transactions. One way was provided by ADO.NET. That technology's abstract `System.Data.Common.DbConnection` class defined a `BeginTransaction()` method by which you could explicitly initiate a transaction controlled by the particular resource manager made accessible by the concrete implementation of `DbConnection`. The other way of programming a transaction was provided by Enterprise Services. It provided the `System.EnterpriseServices.Transaction` attribute that could be added to any subclass of `System.EnterpriseServices.ServicedComponent` to implicitly enlist any code executing in any of the class's methods into a transaction managed by the Microsoft Distributed Transaction Coordinator.

ADO.NET provided a way of programming transactions explicitly, whereas Enterprise Services allowed you to do it declaratively. However, in choosing between the explicit style of programming transactions offered by ADO.NET and the declarative style offered by Enterprise Services, you were also forced to choose how a transaction would be handled. With ADO.NET, transactions were handled by a single resource manager, whereas with Enterprise Services, a transaction incurred the overhead of involving the Microsoft Distributed Transaction Coordinator, regardless of whether the transaction was actually distributed.

.NET 2.0 introduced the Lightweight Transaction Manager, `System.Transactions.TransactionManager`. As its name implies, the Lightweight Transaction Manager has minimal overhead: "...[p]erformance benchmarking done by Microsoft with SQL Server 2005, comparing the use of a [Lightweight Transaction Manager transaction] to using a native transaction directly found no statistical differences between using the two methods" (Lowy 2005, 12). If only a single resource manager is enlisted in the transaction, the Lightweight Transaction Manager allows that resource manager to manage the transaction and the Lightweight Transaction Manager merely monitors it. However, if the Lightweight Transaction Manager detects that a second resource manager has become involved in the transaction, the Lightweight Transaction Manager has the original resource manager relinquish control of the transaction and transfers that control to the Distributed Transaction Coordinator. Transferring control of a transaction in progress to the Distributed Transaction Coordinator is called *promotion of the transaction*.

The `System.Transactions` namespace allows you to program transactions using the Lightweight Transaction Manager either explicitly or implicitly. The explicit style uses the `System.Transactions.CommitableTransaction` class:

```
CommitableTransaction transaction = new CommittableTransaction();
using(SqlConnection myConnection = new SqlConnection(myConnectionString))
```

```
{
        myConnection.Open();

        myConnection.EnlistTransaction(tx);

        //Do transactional work

        //Commit the transaction:
        transaction.Close();

}
```

The alternative, implicit style of programming, which is preferable because it is more flexible, uses the `System.Transactions.TransactionScope` class:

```
using(TransactionScope scope = new TransactionScope)
{
        //Do transactional work:
        //...
        //Since no errors have occurred, commit the transaction:
        scope.Complete();
}
```

This style of programming a transaction is implicit because code that executes within the `using` block of the `System.Transactions.TransactionScope` instance is implicitly enrolled in a transaction. The `Complete()` method of a `System.Transactions.TransactionScope` instance can be called exactly once, and if it is called, the transaction will commit.

The `System.Transactions` namespace also provides a means for programming your own resource managers. However, knowing the purpose of the Lightweight Transaction Manager and the implicit style of transaction programming provided with the `System.Transactions.TransactionScope` class will suffice for the purpose of learning about the Windows Communication Foundation.

Role Providers

Role providers are classes that derive from the abstract class `System.Web.Security.RoleProvider`. That class has the interface shown in Listing 1.1. It defines ten simple methods for managing roles, including ascertaining whether a given user has been assigned a particular role. Role providers, in implementing those abstract methods, will read and write a particular store of role information. For example, one of the concrete implementations of `System.Web.Security.RoleProvider` included in the .NET Framework 2.0 is `System.Web.Security.AuthorizationStoreRoleProvider`, which uses an Authorization Manager Authorization Store as its repository of role information. Another concrete implementation, `System.Web.Security.SqlRoleProvider`, uses a SQL Server database as its store. However, because the `System.Web.Security.RoleProvider` has

such a simple set of methods for managing roles, if none of the role providers included in the .NET Framework 2.0 is suitable, you can readily provide your own implementation to use whatever store of role information you prefer. Role providers hide the details of how role data is stored behind a simple, standard interface for querying and updating that information. Although `System.Web.Security.RoleProvider` is included in the `System.Web` namespaces of ASP.NET, role providers can be used in any .NET 2.0 application.

LISTING 1.1 `System.Web.Security.RoleProvider`

```
public abstract class RoleProvider : ProviderBase
{
    protected RoleProvider();

    public abstract string ApplicationName { get; set; }

    public abstract void AddUsersToRoles(
                string[] usernames, string[] roleNames);
    public abstract void CreateRole(
                string roleName);
    public abstract bool DeleteRole(
                string roleName, bool throwOnPopulatedRole);
    public abstract string[] FindUsersInRole(
                string roleName, string usernameToMatch);
    public abstract string[] GetAllRoles();
    public abstract string[] GetRolesForUser(
                string username);
    public abstract string[] GetUsersInRole(
                string roleName);
    public abstract bool IsUserInRole(
                string username, string roleName);
    public abstract void RemoveUsersFromRoles(
                string[] usernames, string[] roleNames);
    public abstract bool RoleExists(string roleName);
}
```

The static class, `System.Web.Security.Roles`, provides yet another layer of encapsulation for role management. Consider this code snippet:

```
if (!Roles.IsUserInRole(userName, "Administrator"))
{
  [...]
}
```

Here, the static `System.Web.Security.Roles` class is used to inquire whether a given user has been assigned to the Administrator role. What is interesting about this snippet is that the inquiry is made without an instance of a particular role provider having to be created

first. The static `System.Web.Security.Roles` class hides the interaction with the role provider. The role provider it uses is whichever one is specified as being the default in the configuration of the application. Listing 1.2 is a sample configuration that identifies the role provider named `MyRoleProvider`, which is an instance of the `System.Web.Security.AuthorizationStoreRoleProvider` class, as the default role provider.

LISTING 1.2 Role Provider Configuration

```
<configuration>
  <connectionStrings>
    <add name="AuthorizationServices"
    `connectionString="msxml://~\App_Data\SampleStore.xml" />
  </connectionStrings>
  <system.web>
    <roleManager defaultProvider="MyRoleProvider"
      enabled="true"
      cacheRolesInCookie="true"
      cookieName=".ASPROLES"
      cookieTimeout="30"
      cookiePath="/"
      cookieRequireSSL="false"
      cookieSlidingExpiration="true"
      cookieProtection="All" >
      <providers>
        <clear />
          <add
            name="MyRoleProvider"
            type="System.Web.Security.AuthorizationStoreRoleProvider"
            connectionStringName="AuthorizationServices"
            applicationName="SampleApplication"
            cacheRefreshInterval="60"
            scopeName="" />
      </providers>
    </roleManager>
  </system.web>
</configuration>
```

Summary

This chapter introduced some programming tools that were new in .NET 2.0 and that are prerequisites for understanding and working effectively with the Windows Communication Foundation:

- The new `partial` keyword in C# allows the definitions of types to be composed from any number of parts distributed across the source code files of a single module.

- Generics are templates from which any number of fully preprogrammed classes can be created.
- Nullable value types are value types that can be assigned a value of `null` and checked for `null` values.
- The Lightweight Transaction Manager ensures that transactions are managed as efficiently as possible. An elegant new syntax has been provided for using it.
- Role providers implement a simple, standard interface for managing the roles to which users are assigned that is independent of how the role information is stored.

References

Ecma International. 2006. ECMA-335: Common Language Infrastructure (CLI) Partitions I–VI. Geneva: Ecma.

Lowy, Juval. *Introducing System.Transactions*. http://www.microsoft.com/downloads/details.aspx?familyid=11632373-BC4E-4C14-AF25-0F32AE3C73A0&displaylang=en. Accessed July 27, 2008.

Microsoft 2006. *Overview of Generics in the .NET Framework*. http://msdn2.microsoft.com/en-us/library/ms172193.aspx. Accessed August 20, 2006.

Richter, Jeffrey. 2002. *Applied Microsoft .NET Framework Programming*. Redmond, WA: Microsoft Press.

Index

A

B

C

D

E

F

G - H

I

J - K - L

M

N

O

P

Q - R

S

T

U - V

W

X - Y - Z