Microsoft®

MCTS EXAM
70-536

# Microsoft® .NET Framework–Application Development Foundation

2
SECOND EDITION

*Fully Updated and Revised*

Tony Northrup

SELF-PACED
Training Kit

# How to access your CD files

The print edition of this book includes a CD. To access the CD files, go to http://aka.ms/626195/files, and look for the Downloads tab.

Note: Use a desktop web browser, as files may not be accessible from all ereader devices.

Questions? Please contact: mspinput@microsoft.com

Microsoft Press

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2008935429

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9   QWT   3 2 1 0 9 8

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

**Acquisitions Editor:** Ken Jones
**Developmental Editor:** Laura Sackerman
**Project Editor:** Carol Vu
**Editorial Production:** S4Carlisle Publishing Services
**Technical Reviewer:** Kurt Meyer; Technical Review services provided by Content Master, a member of CM Group, Ltd.
**Cover:** Tom Draper Design

Body Part No. X15-12470

*In loving memory of Chelsea Knowles*

## About the Author

## Tony Northrup

In the mid-1980s, Tony Northrup, MCTS, MCSE, CISPP, and MVP, learned to program in BASIC on a ZX-81 personal computer built from a kit. Later, he mastered 68000 assembly and ANSI C on the Motorola VERSAdos operating system before beginning to write code for MS-DOS. After a brief time with the NEXTSTEP operating system, Tony returned to a Microsoft platform because he was impressed by the beta version of Microsoft Windows NT 3.1. Although he has dabbled in other operating systems, Tony has since focused on Windows development in Microsoft Visual C++, Microsoft Visual Basic, C#, and Perl (for automation projects). Tony now develops almost exclusively for the .NET Framework.

Tony started writing in 1997 and has since published more than a dozen technology books on the topics of development and networking. In addition, Tony has written dozens of articles at *http://www.microsoft.com,* covering topics ranging from securing ASP.NET applications to designing firewalls to protect networks and computers. Tony spends his spare time hiking through the woods near his Phillipston, Massachusetts, home. He's rarely without his camera, and in the past six years has created what might be the largest and most popular publicly accessible database of nature and wildlife photographs on the Internet. Tony lives with his dog, Sandi, and his cat, Sam. For more information about Tony, visit *http://www.northrup.org*.

# Contents at Glance

# Table of Contents

# Acknowledgments

The author's name appears on the cover of a book, but I am only one member of a much larger team. First of all, thanks to Ken Jones at Microsoft for allowing me to update the first edition of this book. During the writing process, I worked most closely with Carol Vu, Laura Sackerman, and Susan McClung. Carol, Laura, and Sue, thanks for your patience with me, and for making this a great book. Kurt Meyer was my technical reviewer, and he was far more committed to the project than any reviewer I've worked with in the past. Each of my editors contributed significantly to this book and I hope to work with them all in the future.

Many other people helped with this book, albeit a bit more indirectly, by keeping me sane throughout the writing process. Lori Hendrickson introduced me to Cacique in Costa Rica. Nisha Rajasekaran helped me buy clothes. Tara Banks, Eric Parucki, and Stephanie Wunderlich improved my vocabulary by repeatedly beating me at Scrabble. Chris and Diane Geggis trusted me with Remy. Jennie Lozier drank my Chardonnay. Eric and Alyssa Faulkner, with the help of Amy Gilvary, threw an Independence Day party (at my house, oddly). Finally, Diane and Franklin Glenn made some incredible chocolate cake. Thanks, guys.

# Introduction

This training kit is designed for developers who plan to take Microsoft Certified Technical Specialist (MCTS) exam 70-536, as well as for developers who need to know how to develop applications using the Microsoft .NET Framework. Before you begin using this kit, you should have a working knowledge of Microsoft Windows and Microsoft Visual Basic or C#.

By using this training kit, you'll learn how to do the following:

- Develop applications that use system types and collections
- Implement service processes, threading, and application domains to enable application isolation and multithreading
- Create and deploy manageable applications
- Create classes that can be serialized to enable them to be easily stored and transferred
- Create hardened applications that are resistant to attacks and restrict access based on user and group roles
- Use interoperability and reflection to leverage legacy code and communicate with other applications
- Write applications that send e-mail messages
- Create applications that can be used in different regions with different languages and cultural conventions
- Draw charts and create images, and either display them as part of your application or save them to files

## Hardware Requirements

The following hardware is required to complete the practice exercises:

- A computer with a 1.6 GHz or faster processor (2.2 GHz recommended)
- 512 megabytes (MB) of RAM or more (1 GB recommended)
- 2 gigabytes (GB) of available hard disk space
- A DVD-ROM drive

- 1,024 x 768 or higher resolution display with 256 or higher colors (1280 x 1024 recommended)
- A keyboard and Microsoft mouse, or compatible pointing device

## Software Requirements

The following software is required to complete the practice exercises:

- One of the following operating systems, using either a 32-bit or 64-bit architecture:
  - Windows XP
  - Windows Server 2003
  - Windows Vista
- Visual Studio 2008 (A 90-day evaluation edition of Visual Studio 2008 Professional Edition is included on DVD with this book.)

## Using the CD and DVD

A companion CD and an evaluation software DVD are included with this training kit. The companion CD contains the following:

- **Practice tests** You can reinforce your understanding of how to create .NET Framework applications by using electronic practice tests you customize to meet your needs from the pool of Lesson Review questions in this book. Or you can practice for the 70-536 certification exam by using tests created from a pool of 200 realistic exam questions, which is enough to give you many different practice exams to ensure that you're prepared.
- **Code** Each chapter in this book includes sample files associated with the lab exercises at the end of every lesson. For most exercises, you will be instructed to open a project prior to starting the exercise. For other exercises, you will create a project on your own and be able to reference a completed project on the CD in the event you experience a problem following the exercise. A few exercises do not involve sample files. To install the sample files on your hard disk, run Setup.exe in the Code folder on the companion CD. The default installation folder is \Documents\Microsoft Press\MCTS Self-Paced Training Kit Exam 70-536_2E.
- **An eBook** An electronic version (eBook) of this book is included for times when you don't want to carry the printed book with you. The eBook is in Portable Document Format (PDF), and you can view it by using Adobe Acrobat or Adobe Reader.

The evaluation software DVD contains a 90-day evaluation edition of Visual Studio 2008 Professional Edition, in case you want to use it with this book.

**Digital Content for Digital Book Readers:** If you bought a digital-only edition of this book, you can enjoy select content from the print edition's companion CD.
Visit *http://www.microsoftpressstore.com/title/9780735626195* to get your downloadable content. This content is always up-to-date and available to all readers.

## How to Install the Practice Tests

To install the practice test software from the companion CD to your hard disk, do the following:

1. Insert the companion CD into your CD drive, and accept the license agreement. A CD menu appears.

**NOTE If the CD Menu Doesn't Appear**

If the CD menu or the license agreement doesn't appear, AutoRun might be disabled on your computer. Refer to the Readme.txt file on the CD-ROM for alternate installation instructions.

2. On the CD menu click the Practice Tests item, and follow the instructions on the screen.

## How to Use the Practice Tests

To start the practice test software, follow these steps:

1. Click Start, select All Programs, and then select Microsoft Press Training Kit Exam Prep. A window appears that shows all the Microsoft Press training kit exam prep suites installed on your computer.
2. Double-click the lesson review or practice test you want to use.

**NOTE Lesson Reviews vs. Practice Tests**

Select the (70-536) Microsoft .NET Framework—Application Development Foundation Lesson Review to use the questions from the "Lesson Review" sections of this book. Select the (70-536) Microsoft .NET Framework—Application Development Foundation *practice test* to use a pool of questions similar to those in the 70-536 certification exam.

## Lesson Review Options

When you start a lesson review, the Custom Mode dialog box appears so that you can configure your test. You can click OK to accept the defaults, or you can customize the number of questions you want, how the practice test software works, which exam objectives you want the questions to relate to, and whether you want your lesson review to be timed. If you're retaking a test, you can select whether you want to see all the questions again or only those questions you missed or didn't answer.

After you click OK, your lesson review starts, as follows:

- To take the test, answer the questions and use the Next, Previous, and Go To buttons to move from question to question.
- After you answer an individual question, if you want to see which answers are correct—along with an explanation of each correct answer—click Explanation.
- If you'd rather wait until the end of the test to see how you did, answer all the questions and then click Score Test. You'll see a summary of the exam objectives you chose and the percentage of questions you got right overall and per objective. You can print a copy of your test, review your answers, or retake the test.

## Practice Test Options

When you start a practice test, you choose whether to take the test in Certification Mode, Study Mode, or Custom Mode, as follows:

- **Certification Mode** Closely resembles the experience of taking a certification exam. The test has a set number of questions, it's timed, and you can't pause and restart the timer.
- **Study Mode** Creates an untimed test in which you can review the correct answers and the explanations after you answer each question.
- **Custom Mode** Gives you full control over the test options so that you can customize them as you like.

In all modes, the user interface you see when taking the test is basically the same, but with different options enabled or disabled depending on the mode. The main options are discussed in the previous section, "Lesson Review Options."

When you review your answer to an individual practice test question, a "References" section is provided that lists where in the training kit you can find the information that relates to that question and provides links to other sources of information. After you click Test Results to score your entire practice test, you can click the Learning Plan tab to see a list of references for every objective.

### How to Uninstall the Practice Tests

To uninstall the practice test software for a training kit, use the Add Or Remove Programs option in the Control Panel.

## Microsoft Certified Professional Program

The Microsoft certifications provide the best method to prove your command of current Microsoft products and technologies. The exams and corresponding certifications are developed to validate your mastery of critical competencies as you design and develop, or implement and support, solutions with Microsoft products and technologies. Computer professionals who become Microsoft-certified are recognized as experts and are sought after industry-wide. Certification brings a variety of benefits to the individual and to employers and organizations.

---

**MORE INFO All the Microsoft Certifications**

For a full list of Microsoft certifications, go to *www.microsoft.com/learning/mcp/default.asp*.

---

## Technical Support

Every effort has been made to ensure the accuracy of this book and the contents of the companion CD. If you have comments, questions, or ideas regarding this book or the companion CD, please send them to Microsoft Press by using either of the following methods:

E-mail: *tkinput@microsoft.com*

Postal Mail:

Microsoft Press
Attn: *MCTS Self-Paced Training Kit (Exam 70-536): Microsoft .NET Framework–Application Development Foundation, Second Edition* Editor
One Microsoft Way
Redmond, WA 98052–6399

For additional support information regarding this book and the CD-ROM (including answers to commonly asked questions about installation and use), visit the Microsoft Press Technical Support Web site at *www.microsoft.com/learning/support/books/*. To connect directly to the Microsoft Knowledge Base and enter a query, visit *support.microsoft.com/search/*. For support information regarding Microsoft software, please connect to *support.microsoft.com*.

## Evaluation Edition Software Support

The 90-day evaluation edition provided with this training kit is not the full retail product and is provided only for the purposes of training and evaluation. Microsoft and Microsoft Technical Support do not support this evaluation edition.

Information about any issues relating to the use of this evaluation edition with this training kit is posted to the Support section of the Microsoft Press Web site (*www.microsoft.com/learning/support/books/*). For information about ordering the full version of any Microsoft software, please call Microsoft Sales at (800) 426-9400 or visit *www.microsoft.com*.

Chapter 4

# Collections and Generics

Developers often need to store groups of related objects. For example, an e-mail inbox would contain a group of messages, a phone book would contain a group of phone numbers, and an audio player would contain a group of songs.

The .NET Framework provides the *System.Collections* namespace to allow developers to manage groups of objects. Different collections exist to provide performance benefits in different scenarios, flexible sorting capabilities, support for different types, and dictionaries that pair keys and values.

**Exam objectives in this chapter:**

- Manage a group of associated data in a .NET Framework application by using collections.
- Improve type safety and application performance in a .NET Framework application by using generic collections.
- Manage data in a .NET Framework application by using specialized collections.

**Lessons in this chapter:**

## Before You Begin

This book assumes that you have at least two to three years of experience developing Web-based, Microsoft Windows–based, or distributed applications using the .NET Framework. Candidates should have a working knowledge of Microsoft Visual Studio. Before you begin, you should be familiar with Microsoft Visual Basic or C# and be comfortable with the following tasks:

- Creating console and Windows Presentation Foundation (WPF) applications in Visual Studio using Visual Basic or C#
- Adding namespaces and system class library references to a project
- Running a project in Visual Studio, setting breakpoints, stepping through code, and watching the values of variables

## Lesson 1: Collections and Dictionaries

The *System.Collections* and *System.Collections.Specialized* namespaces contain a number of classes to meet varying requirements for storing groups of related objects. To use them most efficiently, you need to understand the benefits of each class. This lesson describes each collection and dictionary type and shows you how to use them.

**After this lesson, you will be able to:**

- Use collections and choose the best collection class for different requirements
- Use dictionaries and choose the best dictionary class for different requirements

**Estimated lesson time: 30 minutes**

## Collections

A collection is any class that allows for gathering items into lists and for iterating through those items. The .NET Framework includes the following collection classes:

- ***ArrayList*** A simple collection that can store any type of object. *ArrayList* instances expand to any required capacity.
- ***Queue*** A first-in, first-out (FIFO) collection. You might use a *Queue* on a messaging server to store messages temporarily before processing or to track customer orders that need to be processed on a first-come, first-serve basis.
- ***Stack*** A last-in, first-out (LIFO) collection. You might use a *Stack* to track changes so that the most recent change can be undone.
- ***StringCollection*** Like *ArrayList*, except values are strongly typed as strings, and *StringCollection* does not support sorting.
- ***BitArray*** A collection of boolean values.

### *ArrayList*

Use the *ArrayList* class (in the *System.Collections* namespace) to add objects that can be accessed directly using a zero-based index or accessed in a series using a *foreach* loop. The capacity of an *ArrayList* expands as required. The following example shows how to use the *ArrayList.Add* method to add different types of objects to a single array, and then access each object using a *foreach* loop:

```
' VB
Dim al As New ArrayList()
al.Add("Hello")
al.Add("World")
al.Add(5)
al.Add(New FileStream("delemete", FileMode.Create))

Console.WriteLine("The array has " + al.Count.ToString + " items:")

For Each s As Object In al
    Console.WriteLine(s.ToString())
Next

// C#
ArrayList al = new ArrayList();
al.Add("Hello");
al.Add("World");
al.Add(5);
al.Add(new FileStream("delemete", FileMode.Create));

Console.WriteLine("The array has " + al.Count + " items:");

foreach (object s in al)
    Console.WriteLine(s.ToString());
```

This console application displays the following:

```
The array has 4 items:
Hello
World
5
System.IO.FileStream
```

In practice, you generally add items of a single type to an *ArrayList*. This allows you to call the *Sort* method to sort the objects using their *IComparable* implementation. You can also use the *Remove* method to remove an object you previously added and use the *Insert* method to add an element at the specified location in the zero-based index. The following code sample demonstrates this:

```
' VB
Dim al As New ArrayList()
al.Add("Hello")
al.Add("World")
al.Add("this")
al.Add("is")
al.Add("a")
al.Add("test")

al.Remove("test")
al.Insert(4, "not")

al.Sort()

For Each s As Object In al
    Console.WriteLine(s.ToString())
Next
```

```
// C#
ArrayList al = new ArrayList();
al.Add("Hello");
al.Add("World");
al.Add("this");
al.Add("is");
al.Add("a");
al.Add("test");

al.Remove("test");
al.Insert(4, "not");

al.Sort();

foreach (object s in al)
    Console.WriteLine(s.ToString());
```

This code sample results in the following display. Notice that the items are sorted alphabetically (using the string *IComparable* implementation) and "test" has been removed:

```
A
Hello
is
not
this
World
```

**IMPORTANT** Using *StringCollection*

You could also use *StringCollection* in place of *ArrayList* in the previous example. However, *StringCollection* does not support sorting, described next. The primary advantage of *StringCollection* is that it's strongly typed for string values.

You can also create your own custom *IComparer* implementations to control sort order. While the *IComparable.CompareTo* method controls the default sort order for a class, *IComparer.Compare* can be used to provide custom sort orders. For example, consider the following simple class, which only implements *IComparer*:

```
' VB
Public Class reverseSort
    Implements IComparer
    Private Function Compare(ByVal x As Object, ByVal y As Object) _
        As Integer Implements IComparer.Compare
        Return ((New CaseInsensitiveComparer()).Compare(y, x))
    End Function
End Class
```

```
// C#
public class reverseSort : IComparer
{
    int IComparer.Compare(Object x, Object y)
    {
        return ((new CaseInsensitiveComparer()).Compare(y, x));
    }
}
```

Given that class, you could pass an instance of the class to the *ArrayList.Sort* method. The following code sample demonstrates this and also demonstrates using the *ArrayList.AddRange* method, which adds each element of an array as a separate element to the instance of *ArrayList*:

```
' VB
Dim al As New ArrayList()
al.AddRange(New String() {"Hello", "world", "this", "is", "a", "test"})

al.Sort(New reverseSort())

For Each s As Object In al
    Console.WriteLine(s.ToString())
Next

// C#
ArrayList al = new ArrayList();
al.AddRange(new string[] {"Hello", "world", "this", "is", "a", "test"});

al.Sort(new reverseSort());

foreach (object s in al)
    Console.WriteLine(s.ToString());
```

This code displays the following:

```
world
this
test
is
Hello
A
```

You can also call the *ArrayList.Reverse* method to reverse the current order of items in the *ArrayList*.

To locate a specific element, call the *ArrayList.BinarySearch* method and pass an instance of the object you are searching for. *BinarySearch* returns the zero-based index of the item. For example, the following code sample displays 2 because the string "this" is in the third position, and the first position is 0:

```
' VB
Dim al As New ArrayList()
al.AddRange(New String() {"Hello", "world", "this", "is", "a", "test"})
Console.WriteLine(al.BinarySearch("this"))

// C#
ArrayList al = new ArrayList();
al.AddRange(new string[] {"Hello", "world", "this", "is", "a", "test"});
Console.WriteLine(al.BinarySearch("this"));
```

Similarly, the *ArrayList.Contains* method returns *true* if the *ArrayList* instance contains the specified object and *false* if it does not contain the object.

## *Queue* and *Stack*

The *Queue* and *Stack* classes (in the *System.Collections* namespace) store objects that can be retrieved and removed in a single step. *Queue* uses a FIFO sequence, while *Stack* uses a LIFO sequence. The *Queue* class uses the *Enqueue* and *Dequeue* methods to add and remove objects, while the *Stack* class uses *Push* and *Pop*. The following code demonstrates the differences between the two classes:

```
' VB
Dim q As New Queue()
q.Enqueue("Hello")
q.Enqueue("world")
q.Enqueue("just testing")

Console.WriteLine("Queue demonstration:")
For i As Integer = 1 To 3
    Console.WriteLine(q.Dequeue().ToString())
Next

Dim s As New Stack()
s.Push("Hello")
s.Push("world")
s.Push("just testing")

Console.WriteLine("Stack demonstration:")
For i As Integer = 1 To 3
    Console.WriteLine(s.Pop().ToString())
Next

// C#
Queue q = new Queue();
q.Enqueue("Hello");
q.Enqueue("world");
q.Enqueue("just testing");
```

```
Console.WriteLine("Queue demonstration:");
for (int i = 1; i <= 3; i++)
    Console.WriteLine(q.Dequeue().ToString());

Stack s = new Stack();
s.Push("Hello");
s.Push("world");
s.Push("just testing");

Console.WriteLine("Stack demonstration:");
for (int i = 1; i <= 3; i++)
    Console.WriteLine(s.Pop().ToString());
```

The application produces the following output:

```
Queue demonstration:
Hello
world
just testing
Stack demonstration:
just testing
world
Hello
```

You can also use *Queue.Peek* and *Stack.Peek* to access an object without removing it from the stack. Use *Queue.Clear* and *Stack.Clear* to remove all objects from the stack.

### *BitArray* and *BitVector32*

*BitArray* is an array of boolean values, where each item in the array is either true or false. While *BitArray* can grow to any size, *BitVector32* (a structure) is limited to exactly 32 bits. If you need to store boolean values, use *BitVector32* anytime you require 32 or fewer items, and use *BitArray* for anything larger.

## Dictionaries

Dictionaries map keys to values. For example, you might map an employee ID number to the object that represents the employee, or you might map a product ID to the object that represents the product. The .NET Framework includes the following dictionary classes:

- ***Hashtable*** A dictionary of name/value pairs that can be retrieved by name or index
- ***SortedList*** A dictionary that is sorted automatically by the key
- ***StringDictionary*** A hashtable with name/value pairs implemented as strongly typed strings

- ***ListDictionary*** A dictionary optimized for a small list of objects with fewer than 10 items
- ***HybridDictionary*** A dictionary that uses a *ListDictionary* for storage when the number of items is small and automatically switches to a *Hashtable* as the list grows
- ***NameValueCollection*** A dictionary of name/value pairs of strings that allows retrieval by name or index

*SortedList* (in the *System.Collections* namespace) is a dictionary that consists of key/value pairs. Both the key and the value can be any object. *SortedList* is sorted automatically by the key. For example, the following code sample creates a *SortedList* instance with three key/value pairs. It then displays the definitions for *Queue*, *SortedList*, and *Stack*, in that order:

```
' VB
Dim sl As New SortedList()
sl.Add("Stack", "Represents a LIFO collection of objects.")
sl.Add("Queue", "Represents a FIFO collection of objects.")
sl.Add("SortedList", "Represents a collection of key/value pairs.")

For Each de As DictionaryEntry In sl
    Console.WriteLine(de.Value)
Next

// C#
SortedList sl = new SortedList();
sl.Add("Stack", "Represents a LIFO collection of objects.");
sl.Add("Queue", "Represents a FIFO collection of objects.");
sl.Add("SortedList", "Represents a collection of key/value pairs.");

foreach (DictionaryEntry de in sl)
    Console.WriteLine(de.Value);
```

Notice that *SortedList* is an array of *DictionaryEntry* objects. As the previous code sample demonstrates, you can access the objects you originally added to the *SortedList* using the *DictionaryEntry.Value* property. You can access the key using the *DictionaryEntry.Key* property.

You can also access values directly by accessing the *SortedList* as a collection. The following code sample (which builds upon the previous code sample) displays the definition for *Queue* twice. *Queue* is the first entry in the zero-based index because the *SortedList* instance automatically sorted the keys alphabetically:

```
' VB
Console.WriteLine(sl("Queue"))
Console.WriteLine(sl.GetByIndex(0))
```

```
// C#
Console.WriteLine(sl["Queue"]);
Console.WriteLine(sl.GetByIndex(0));
```

The *ListDictionary* class (in the *System.Collections.Specialized* namespace) also provides similar functionality, and is optimized to perform best with lists of fewer than 10 items. *HybridDictionary* (also in the *System.Collections.Specialized* namespace) provides the same performance as *ListDictionary* with small lists, but it scales better when the list is expanded.

While *SortedList* can take an object of any type as its value (but only strings as keys), the *StringDictionary* class (in the *System.Collections.Specialized* namespace) provides similar functionality, without the automatic sorting, and requires both the keys and the values to be strings.

*NameValueCollection* also provides similar functionality, but it allows you to use either a string or an integer index for the key. In addition, you can store multiple string values for a single key. The following code sample demonstrates this by displaying two definitions for the terms *stack* and *queue:*

```
' VB
Dim sl As New NameValueCollection()
sl.Add("Stack", "Represents a LIFO collection of objects.")
sl.Add("Stack", "A pile of pancakes.")
sl.Add("Queue", "Represents a FIFO collection of objects.")
sl.Add("Queue", "In England, a line.")
sl.Add("SortedList", "Represents a collection of key/value pairs.")

For Each s As String In sl.GetValues(0)
    Console.WriteLine(s)
Next

For Each s As String In sl.GetValues("Queue")
    Console.WriteLine(s)
Next

// C#
NameValueCollection sl = new NameValueCollection();
sl.Add("Stack", "Represents a LIFO collection of objects.");
sl.Add("Stack", "A pile of pancakes.");
sl.Add("Queue", "Represents a FIFO collection of objects.");
sl.Add("Queue", "In England, a line.");
sl.Add("SortedList", "Represents a collection of key/value pairs.");

foreach (string s in sl.GetValues(0))
    Console.WriteLine(s);

foreach (string s in sl.GetValues("Queue"))
    Console.WriteLine(s);
```

## Lab: Creating a Shopping Cart

In this lab, you create a simple shopping cart that can be sorted by the price of the items.

### Exercise: Using *ArrayList*

In this exercise, you use an *ArrayList* and a custom class to create a shopping cart with basic functionality.

1. Using Visual Studio, create a new Console Application project. Name the project ShoppingCart.
2. Add a simple class to represent a shopping cart item, containing properties for the item name and price. The following code sample shows one way to do this:

```
' VB
Public Class ShoppingCartItem
    Public itemName As String
    Public price As Double

    Public Sub New(ByVal _itemName As String, ByVal _price As Double)
        Me.itemName = _itemName
        Me.price = _price
    End Sub
End Class

// C#
public class ShoppingCartItem
{
    public string itemName;
    public double price;

    public ShoppingCartItem(string _itemName, double _price)
    {
        this.itemName = _itemName;
        this.price = _price;
    }
}
```

3. Add the *System.Collections* namespace to your project.
4. In the *Main* method create an instance of *ArrayList*, and then add four shopping cart items with different names and prices. Display the items on the console using a *foreach* loop. The following code sample demonstrates this:

```
' VB
Dim shoppingCart As New ArrayList()
shoppingCart.Add(New ShoppingCartItem("Car", 5000))
shoppingCart.Add(New ShoppingCartItem("Book", 30))
shoppingCart.Add(New ShoppingCartItem("Phone", 80))
shoppingCart.Add(New ShoppingCartItem("Computer", 1000))

For Each sci As ShoppingCartItem In shoppingCart
    Console.WriteLine(sci.itemName + ": $" + sci.price.ToString())
Next

// C#
ArrayList shoppingCart = new ArrayList();
shoppingCart.Add(new ShoppingCartItem("Car", 5000));
shoppingCart.Add(new ShoppingCartItem("Book", 30));
shoppingCart.Add(new ShoppingCartItem("Phone", 80));
shoppingCart.Add(new ShoppingCartItem("Computer", 1000));

foreach (ShoppingCartItem sci in shoppingCart)
    Console.WriteLine(sci.itemName + ": $" + sci.price.ToString());
```

5. Build and run your application and verify that it works correctly.
6. Now, implement the *IComparable* interface for the *ShoppingCartItem* class to sort the items by price. The following code should replace the existing class definition for *ShoppingCartItem*:

```
' VB
Public Class ShoppingCartItem
    Implements IComparable
    Public itemName As String
    Public price As Double

    Public Sub New(ByVal _itemName As String, ByVal _price As Double)
        Me.itemName = _itemName
        Me.price = _price
    End Sub

    Public Function CompareTo(ByVal obj As Object) _
        As Integer Implements System.IComparable.CompareTo
        Dim otherItem As ShoppingCartItem = _
            DirectCast(obj, ShoppingCartItem)
        Return Me.price.CompareTo(otherItem.price)
    End Function
End Class

// C#
public class ShoppingCartItem : IComparable
{
    public string itemName;
    public double price;

    public ShoppingCartItem(string _itemName, double _price)
    {
        this.itemName = _itemName;
        this.price = _price;
    }
```

```
        public int CompareTo(object obj)
        {
            ShoppingCartItem otherItem = (ShoppingCartItem)obj;
            return this.price.CompareTo(otherItem.price);
        }
    }
```

7. Now, write code to sort the shopping cart collection from most to least expensive. The simplest way is to add two lines of code just before the *foreach* loop:

```
' VB
shoppingCart.Sort()
shoppingCart.Reverse()

// C#
shoppingCart.Sort();
shoppingCart.Reverse();
```

8. Build and run your application again and verify that the shopping cart is sorted from most to least expensive.

## Lesson Summary

- You can use the *ArrayList*, *Queue*, and *Stack* collection classes to create collections using any class. *ArrayList* allows you to iterate through items and sort them. *Queue* provides FIFO sequencing, while *Stack* provides LIFO sequencing. *BitArray* and *BitVector32* are useful for boolean values.
- Dictionaries organize instances of objects in key/value pairs. The *HashTable* class can meet most of your requirements. If you want the dictionary to be sorted automatically by the key, use the *SortedDictionary* class. *ListDictionary* is designed to perform well with fewer than 10 items.

## Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 1, "Collections and Dictionaries." The questions are also available on the companion CD if you prefer to review them in electronic form.

**NOTE Answers**

Answers to these questions and explanations of why each answer choice is right or wrong are located in the "Answers" section at the end of the book.

1. You create an instance of the *Stack* class. After adding several integers to it, you need to remove all objects from the *Stack*. Which method should you call?

   A. *Stack.Pop*

   B. *Stack.Push*

   C. *Stack.Clear*

   D. *Stack.Peek*

2. You need to create a collection to act as a shopping cart. The collection will store multiple instances of your custom class, *ShoppingCartItem*. You need to be able to sort the items according to price and time added to the shopping cart (both properties of the *ShoppingCartItem*). Which class should you use for the shopping cart?

   A. *Queue*

   B. *ArrayList*

   C. *Stack*

   D. *StringCollection*

3. You create an *ArrayList* object and add 200 instances of your custom class, *Product*. When you call *ArrayList.Sort*, you receive an *InvalidOperationException*. How should you resolve the problem? (Choose two. Each answer forms part of the complete solution.)

   A. Implement the *IComparable* interface.

   B. Create a method named *CompareTo*.

   C. Implement the *IEnumerable* interface.

   D. Create a method named *GetEnumerator*.

## Lesson 2: Generic Collections

Collections like *ArrayList*, *Queue*, and *Stack* use the *Object* base class to allow them to work with any type. However, accessing the collection usually requires you to cast from the base *Object* type to the correct type. Not only does this make development tedious and more error-prone, but it hurts performance.

Using generics, you can create strongly typed collections for any class, including custom classes. This simplifies development within the Visual Studio editor, helps ensure appropriate use of types, and can improve performance by reducing the need to cast.

**After this lesson, you will be able to:**

- Explain why you should use generic collections
- Use the *SortedList* generic collection
- Use generics with custom classes
- Use the *Queue* and *Stack* collection generically
- Use the generic *List* collection

**Estimated lesson time: 30 minutes**

### Generics Overview

Many of the collections in the .NET Framework support adding objects of any type, such as *ArrayList*. Others, like *StringCollection*, are strongly typed. Strongly typed classes are easier to develop with because the Visual Studio designer can list and validate members automatically. In addition, you do not need to cast classes to more specific types, and you are protected from casting to an inappropriate type.

Generics provide many of the benefits of strongly typed collections, but they can work with any type that meets the requirements. In addition, using generics can improve performance by reducing the number of casting operations required. Table 4-1 lists the most useful generic collection classes and the corresponding nongeneric collection type.

**Table 4-1 Generic Collection Classes**

| Generic Class | Comparable Nongeneric Classes |
|---|---|
| *List<T>* | *ArrayList*, *StringCollection* |
| *Dictionary<T,U>* | *Hashtable*, *ListDictionary*, *HybridDictionary*, *OrderedDictionary*, *NameValueCollection*, *StringDictionary* |
| *Queue<T>* | *Queue* |

**Table 4-1 Generic Collection Classes**

| Generic Class | Comparable Nongeneric Classes |
|---|---|
| *Stack<T>* | *Stack* |
| *SortedList<T,U>* | *SortedList* |
| *Collection<T>* | *CollectionBase* |
| *ReadOnlyCollection<T>* | *ReadOnlyCollectionBase* |

## Generic *SortedList<T,U>* Collection

The following code sample creates a generic *SortedList<T,U>* using strings as the keys and integers as the values. As you type this code into the Visual Studio editor, notice that it prompts you to enter string and integer parameters for the *SortedList.Add* method as if *SortedList.Add* were strongly typed:

```
' VB
Dim sl As New SortedList(Of String, Integer)()
sl.Add("One", 1)
sl.Add("Two", 2)
sl.Add("Three", 3)

For Each i As Integer In sl.Values
    Console.WriteLine(i.ToString())
Next

// C#
SortedList<string, int> sl = new SortedList<string,int>();
sl.Add("One", 1);
sl.Add("Two", 2);
sl.Add("Three", 3);

foreach (int i in sl.Values)
    Console.WriteLine(i.ToString());
```

In Visual Basic, specify the type arguments for the generic class using the constructor parameters by specifying the *Of* keyword. In C#, specify the type arguments using angle brackets before the constructor parameters.

**Real World**

*Tony Northrup*

You can get the job done by working with a collection that accepts objects, such as *ArrayList*. However, using generics to create strongly typed collections makes development easier in many ways. First, you won't ever forget to cast something, which will reduce the number of bugs in your code (and I've had some really odd bugs when working with the base *Object* class). Second, development is easier because the Visual Studio editor prompts you to provide the correct type as you type the code. Finally, you don't suffer the performance penalty incurred when casting.

## Using Generics with Custom Classes

You can use generics with custom classes as well. Consider the following class declaration:

```
' VB
Public Class person
    Private firstName As String
    Private lastName As String

    Public Sub New(ByVal _firstName As String, ByVal _lastName As String)
        firstName = _firstName
        lastName = _lastName
    End Sub

    Public Overloads Overrides Function ToString() As String
        Return firstName + " " + lastName
    End Function
End Class

// C#
public class person
{
    string firstName;
    string lastName;

    public person(string _firstName, string _lastName)
    {
        firstName = _firstName;
        lastName = _lastName;
    }

    override public string ToString()
    {
        return firstName + " " + lastName;
    }
}
```

You can use the *SortedList<T,U>* generic class with the custom class exactly as you would use it with an integer, as the following code sample demonstrates:

```
' VB
Dim sl As New SortedList(Of String, person)()
sl.Add("One", New person("Mark", "Hanson"))
sl.Add("Two", New person("Kim", "Akers"))
sl.Add("Three", New person("Zsolt", "Ambrus"))

For Each p As person In sl.Values
    Console.WriteLine(p.ToString())
Next

// C#
SortedList<string, person> sl = new SortedList<string,person>();
sl.Add("One", new person("Mark", "Hanson"));
sl.Add("Two", new person("Kim", "Akers"));
sl.Add("Three", new person("Zsolt", "Ambrus"));

foreach (person p in sl.Values)
    Console.WriteLine(p.ToString());
```

## Generic *Queue<T>* and *Stack<T>* Collections

Similarly, the following code sample demonstrates using the generic versions of both *Queue* and *Stack* with the *person* class:

```
' VB
Dim q As New Queue(Of person)()
q.Enqueue(New person("Mark", "Hanson"))
q.Enqueue(New person("Kim", "Akers"))
q.Enqueue(New person("Zsolt", "Ambrus"))

Console.WriteLine("Queue demonstration:")
For i As Integer = 1 To 3
    Console.WriteLine(q.Dequeue().ToString())
Next

Dim s As New Stack(Of person)()
s.Push(New person("Mark", "Hanson"))
s.Push(New person("Kim", "Akers"))
s.Push(New person("Zsolt", "Ambrus"))

Console.WriteLine("Stack demonstration:")
For i As Integer = 1 To 3
    Console.WriteLine(s.Pop().ToString())
Next

// C#
Queue<person> q = new Queue<person>();
q.Enqueue(new person("Mark", "Hanson"));
q.Enqueue(new person("Kim", "Akers"));
q.Enqueue(new person("Zsolt", "Ambrus"));

Console.WriteLine("Queue demonstration:");
for (int i = 1; i <= 3; i++)
    Console.WriteLine(q.Dequeue().ToString());
```

```
Stack<person> s = new Stack<person>();
s.Push(new person("Mark", "Hanson"));
s.Push(new person("Kim", "Akers"));
s.Push(new person("Zsolt", "Ambrus"));

Console.WriteLine("Stack demonstration:");
for (int i = 1; i <= 3; i++)
    Console.WriteLine(s.Pop().ToString());
```

## Generic *List<T>* Collection

Some aspects of generic collections might require specific interfaces to be implemented by the type you specify. For example, calling *List.Sort* without any parameters requires the type to support the *IComparable* interface. The following code sample expands the *person* class to support the *IComparable* interface and the required *CompareTo* method and allows it to be sorted in a *List<T>* generic collection using the person's first and last name:

```
' VB
Public Class person
    Implements IComparable
    Private firstName As String
    Private lastName As String

    Public Function CompareTo(ByVal obj As Object) _
        As Integer Implements System.IComparable.CompareTo
        Dim otherPerson As person = DirectCast(obj, person)
        If Me.lastName <> otherPerson.lastName Then
            Return Me.lastName.CompareTo(otherPerson.lastName)
        Else
            Return Me.firstName.CompareTo(otherPerson.firstName)
        End If
    End Function

    Public Sub New(ByVal _firstName As String, ByVal _lastName As String)
        firstName = _firstName
        lastName = _lastName
    End Sub

    Public Overrides Function ToString() As String
        Return firstName + " " + lastName
    End Function
End Class

// C#
public class person : IComparable
{
    string firstName;
    string lastName;
    public int CompareTo(object obj)
    {
        person otherPerson = (person)obj;
        if (this.lastName != otherPerson.lastName)
            return this.lastName.CompareTo(otherPerson.lastName);
        else
            return this.firstName.CompareTo(otherPerson.firstName);
    }

    public person(string _firstName, string _lastName)
    {
        firstName = _firstName;
        lastName = _lastName;
    }

    override public string ToString()
    {
        return firstName + " " + lastName;
    }
}
```

After adding the *IComparable* interface to the *person* class, you now can sort it in a generic *List<T>*, as the following code sample demonstrates:

```
' VB
Dim l As New List(Of person)()
l.Add(New person("Mark", "Hanson"))
l.Add(New person("Kim", "Akers"))
l.Add(New person("Zsolt", "Ambrus"))

l.Sort()

For Each p As person In l
    Console.WriteLine(p.ToString())
Next

// C#
List<person> l = new List<person>();
l.Add(new person("Mark", "Hanson"));
l.Add(new person("Kim", "Akers"));
l.Add(new person("Zsolt", "Ambrus"));

l.Sort();

foreach (person p in l)
    Console.WriteLine(p.ToString());
```

With the *IComparable* interface implemented, you could also use the *person* class as the key in a generic *SortedList<T,U>* or *SortedDictionary<T,U>* class.

## Lab: Creating a Shopping Cart with a Generic *List<T>*

In this lab, you update a simple WPF application to manage a shopping cart.

### Exercise: Using *List<T>*

In this exercise, you update a pre-made user interface to display a list with multiple sorting options.

1. Navigate to the <*InstallHome*>\Chapter04\Lesson2\Exercise1\Partial folder from the companion CD to your hard disk, and open either the C# version or the Visual Basic .NET version of the solution file. Notice that a basic user interface for the WPF application already exists.
2. This application should allow the user to add shopping cart items to a shopping cart and display the items in the *ListBox* control. First, create a class declaration for *ShoppingCartItem* that includes name and price properties and override the *ToString* method to display both properties, as shown here:

```
' VB
Public Class ShoppingCartItem
    Public itemName As String
    Public price As Double

    Public Sub New(ByVal _itemName As String, ByVal _price As Double)
        Me.itemName = _itemName
        Me.price = _price
    End Sub

    Public Overrides Function ToString() As String
        Return Me.itemName + ": " + Me.price.ToString("C")
    End Function
End Class

// C#
public class ShoppingCartItem
{
    public string itemName;
    public double price;

    public ShoppingCartItem(string _itemName, double _price)
    {
        this.itemName = _itemName;
        this.price = _price;
    }

    public override string ToString()
    {
        return this.itemName + ": " + this.price.ToString("C");
    }
}
```

3. Next, create an instance of a generic collection to act as the shopping cart. The shopping cart object should be strongly typed to allow only *ShoppingCartItem* instances. The following example shows how to do this with the *List<T>* class:

```
' VB
Dim shoppingCart As New List(Of ShoppingCartItem)()

// C#
List<ShoppingCartItem> shoppingCart = new List<ShoppingCartItem>();
```

4. Bind the *shoppingCartList.ItemSource* property to the *shoppingCart*. While there are several ways to do this, the following code demonstrates how to do it from within the *Window_Loaded* event handler:

```
' VB
shoppingCartList.ItemsSource = shoppingCart

// C#
shoppingCartList.ItemsSource = shoppingCart;
```

5. Now, add a handler for the *addButton.Click* event that reads the data that the user has typed into the *nameTextBox* and *priceTextBox*, creates a new *ShoppingCartItem*, adds it to the *shoppingCart*, and then refreshes the *shoppingCartList*:

```
' VB
Try
    shoppingCart.Add(New ShoppingCartItem(nameTextBox.Text, _
        Double.Parse(priceTextBox.Text)))
    shoppingCartList.Items.Refresh()
    nameTextBox.Clear()
    priceTextBox.Clear()
Catch ex As Exception
    MessageBox.Show("Please enter valid data: " + ex.Message)
End Try

// C#
try
{
    shoppingCart.Add(new ShoppingCartItem(nameTextBox.Text,
        double.Parse(priceTextBox.Text)));
    shoppingCartList.Items.Refresh();
    nameTextBox.Clear();
    priceTextBox.Clear();
}
catch (Exception ex)
{
    MessageBox.Show("Please enter valid data: " + ex.Message);
}
```

6. Build and run your application. Verify that you can add items to the shopping cart and that they are displayed in the *ListBox*.

7. Now, add functionality to the *ShoppingCartItem* class so that you can sort the shopping cart by price or item name, as the following code sample demonstrates:

```
' VB
Public Shared Function SortByName(ByVal item1 As ShoppingCartItem, _
    ByVal item2 As ShoppingCartItem) As Integer
    Return item1.itemName.CompareTo(item2.itemName)
End Function

Public Shared Function SortByPrice(ByVal item1 As ShoppingCartItem, _
    ByVal item2 As ShoppingCartItem) As Integer
    Return item1.price.CompareTo(item2.price)
End Function

// C#
public static int SortByName(ShoppingCartItem item1,
    ShoppingCartItem item2)
{
    return item1.itemName.CompareTo(item2.itemName);
}

public static int SortByPrice(ShoppingCartItem item1,
    ShoppingCartItem item2)
{
    return item1.price.CompareTo(item2.price);
}
```

8. After adding those two methods, update the *sortNameButton.Click* and *sortPriceButton.Click* event handlers to sort the *shoppingCart* and then refresh the *shoppingCartList* as follows:

```
' VB
Sub sortNameButton_Click(ByVal sender As Object, _
    ByVal e As RoutedEventArgs)
    shoppingCart.Sort(AddressOf ShoppingCartItem.SortByName)
    shoppingCartList.Items.Refresh()
End Sub

Sub sortPriceButton_Click(ByVal sender As Object, _
    ByVal e As RoutedEventArgs)
    shoppingCart.Sort(AddressOf ShoppingCartItem.SortByPrice)
    shoppingCartList.Items.Refresh()
End Sub

// C#
private void sortNameButton_Click(object sender, RoutedEventArgs e)
{
    shoppingCart.Sort(ShoppingCartItem.SortByName);
    shoppingCartList.Items.Refresh();
}
```

```
private void sortPriceButton_Click(object sender, RoutedEventArgs e)
{
    shoppingCart.Sort(ShoppingCartItem.SortByPrice);
    shoppingCartList.Items.Refresh();
}
```

9. Build and run your application. Add several items to the shopping cart with different names and prices. Click each of the sorting buttons and verify that the shopping cart is re-sorted.

## Lesson Summary

- Generic collections allow you to create strongly typed collections for any class.
- The *SortedList<T,U>* generic collection automatically sorts items.
- You can use generics with custom classes. However, to allow the collection to be sorted without providing a comparer, the custom class must implement the *IComparable* interface.
- The *Queue* and *Stack* collections have both generic and nongeneric implementations.
- The *List<T>* collection provides a generic version of *ArrayList*.

## Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 2, "Generic Collections." The questions are also available on the companion CD if you prefer to review them in electronic form.

**NOTE Answers**

Answers to these questions and explanations of why each answer choice is right or wrong are located in the "Answers" section at the end of the book.

1. You are creating a collection that will act as a database transaction log. You need to be able to add instances of your custom class, *DBTransaction*, to the collection. If an error occurs, you need to be able to access the most recently added instance of *DBTransaction* and remove it from the collection. The collection must be strongly typed. Which class should you use?

   A. *HashTable*

   B. *SortedList*

   C. *Stack*

   D. *Queue*

2. You are creating a custom dictionary class. You want it to be type-safe, using a string for a key and your custom class *Product* as the value. Which class declaration meets your requirements?

   A.

   ```
   ' VB
   Public Class Products2
       Inherits StringDictionary
   End Class

   // C#
   public class Products2 : StringDictionary
   { }
   ```

   B.

   ```
   ' VB
   Class Products
       Inherits Dictionary(Of String, Product)
   End Class

   // C#
   class Products : Dictionary<string, Product>
   { }
   ```

   C.

   ```
   ' VB
   Class Products
       Inherits StringDictionary(Of String, Product)
   End Class

   // C#
   class Products : StringDictionary<string, Product>
   { }
   ```

   D.

   ```
   ' VB
   Class Products
       Inherits Dictionary
   End Class

   // C#
   class Products : Dictionary
   { }
   ```

3. You create an instance of the *SortedList* collection, as shown here:

   ```
   ' VB
   Dim sl As New SortedList(Of Product, string)()

   // C#
   SortedList<Product, string> sl = new SortedList<Product, string>();
   ```

Which declaration of the *Product* class works correctly?

**A.**

```
' VB
Public Class Product
    Implements IComparable
    Public productName As String

    Public Sub New(ByVal _productName As String)
        Me.productName = _productName
    End Sub

    Public Function CompareTo(ByVal obj As Object) As Integer _
        Implements System.IComparable.CompareTo
        Dim otherProduct As Product = DirectCast(obj, Product)
        Return Me.productName.CompareTo(otherProduct.productName)
    End Function
End Class

// C#
public class Product : IComparable
{
    public string productName;

    public Product(string _productName)
    {
        this.productName = _productName;
    }

    public int CompareTo(object obj)
    {
        Product otherProduct = (Product)obj;
        return this.productName.CompareTo(otherProduct.productName);
    }
}
```

**B.**

```
' VB
Public Class Product
    Public productName As String

    Public Sub New(ByVal _productName As String)
        Me.productName = _productName
    End Sub

    Public Function CompareTo(ByVal obj As Object) As Integer _
        Implements System.IComparable.CompareTo
        Dim otherProduct As Product = DirectCast(obj, Product)
        Return Me.productName.CompareTo(otherProduct.productName)
    End Function
End Class
```

```
// C#
public class Product
{
    public string productName;

    public Product(string _productName)
    {
        this.productName = _productName;
    }

    public int CompareTo(object obj)
    {
        Product otherProduct = (Product)obj;
        return this.productName.CompareTo(otherProduct.productName);
    }
}
```

C.

```
' VB
Public Class Product
    Implements IEquatable
    Public productName As String

    Public Sub New(ByVal _productName As String)
        Me.productName = _productName
    End Sub

    Public Function Equals(ByVal obj As Object) As Integer _
        Implements System.IEquatable.Equals
        Dim otherProduct As Product = DirectCast(obj, Product)
        Return Me.productName.Equals(otherProduct.productName)
    End Function
End Class

// C#
public class Product : IEquatable
{
    public string productName;

    public Product(string _productName)
    {
        this.productName = _productName;
    }

    public int Equals(object obj)
    {
        Product otherProduct = (Product)obj;
        return this.productName.Equals(otherProduct.productName);
    }
}
```

D.

```
' VB
Public Class Product
    Public productName As String

    Public Sub New(ByVal _productName As String)
        Me.productName = _productName
    End Sub

    Public Function Equals(ByVal obj As Object) As Integer
        Dim otherProduct As Product = DirectCast(obj, Product)
        Return Me.productName.Equals(otherProduct.productName)
    End Function
End Class

// C#
public class Product
{
    public string productName;

    public Product(string _productName)
    {
        this.productName = _productName;
    }

    public int Equals(object obj)
    {
        Product otherProduct = (Product)obj;
        return this.productName.Equals(otherProduct.productName);
    }
}
```

## Chapter Review

To practice and reinforce the skills you learned in this chapter further, you can do the following:

- Review the chapter summary.
- Review the list of key terms introduced in this chapter.
- Complete the case scenarios. These scenarios set up real-word situations involving the topics of this chapter and ask you to create a solution.
- Complete the suggested practices
- Take a practice test

## Chapter Summary

- Collections store groups of related objects. *ArrayList* is a simple collection that can store any object and supports sorting. *Queue* is a FIFO collection, while *Stack* is a LIFO collection. Dictionaries provide key/value pairs for circumstances that require you to access items in an array using a key.
- Whenever possible, you should use generic collections over collections that use the *Object* base class. Generic collections are strongly typed and offer better performance.

## Key Terms

Do you know what these key terms mean? You can check your answers by looking up the terms in the glossary at the end of the book.

- Collection
- Generic

## Case Scenarios

In the following case scenarios you apply what you've learned about how to plan and use collections. You can find answers to these questions in the "Answers" section at the end of this book.

## Case Scenario 1: Using Collections

You are an application developer for Contoso, Inc. You are creating a WPF application that correlates unsolved crimes with behaviors of known convicts. You create classes called *Crime*, *Evidence*, *Convict*, and *Behavior*.

## Questions

Answer the following questions for your manager:

1. Each *Crime* will have multiple *Evidence* objects, and each *Convict* will have multiple *Behavior* objects. How can you enable this?
2. You need to be able to sort the *Evidence* and *Behavior* collections to allow investigators to identify the most relevant results. Investigators should be able to sort the collections using multiple methods. What type of collection should you use?
3. How can you provide different sorting algorithms?

## Case Scenario 2: Using Collections for Transactions

You are an application developer working for Fabrikam, Inc., a financial services company. You are creating an application that will handle financial transactions.

Your application receives incoming transactions from a Web service and must process the transactions in the order they arrive. Each transaction can involve multiple debits and credits. For example, transferring money from account A to account B requires a debit from account A and a credit to account B. If any credit or debit involved in a transaction fails, all credits and debits must be rolled back, starting with the most recently completed transactions.

## Questions

Answer the following questions for your manager:

1. Transactions might come in faster than you can process them. How can you store the transactions and ensure that you process them in the correct sequence?
2. How can you track the debits and credits you have performed so they can be rolled back if required?
3. Should you use generic classes?

## Suggested Practices

To master the system types and collections exam objective, complete the following tasks.

### Manage a Group of Associated Data in a .NET Framework Application by Using Collections

For this task, you should complete at least Practices 1 and 2 to gain experience using collections. For a better understanding of the performance implications of using the *BitArray* collection instead of the *BitVector32* structure, complete Practice 3 as well.

- **Practice 1** Create an instance of *ArrayList* and add several instances of your own custom class to it. Next, sort the array in at least two different ways.
- **Practice 2** Create a console application that creates instances of each of the different dictionary classes. Populate the dictionaries and access the items both directly and by iterating through them using a *foreach* loop.
- **Practice 3** Write a simple console application that adds 20 boolean values to an instance of the *BitArray* class and then iterates through each of them using a *foreach* loop. Repeat the process 100,000 times using a *for* loop. Time how long the entire process takes by comparing *DateTime.Now* before and after the process. Next, perform the same test using *BitVector32*. Determine which is faster and whether the performance impact is significant.

### Improve Type Safety and Application Performance in a .NET Framework Application by Using Generic Collections

For this task, you should complete at least Practices 1 and 2 to gain experience using generic collections. For a better understanding of the performance implications of using generic collections, complete Practice 3 as well.

- **Practice 1** Write an application that creates an instance of each of the built-in generic collection classes, adds items to each of the collections, and then displays them using a *foreach* loop.
- **Practice 2** Using a custom class that you created for real-world use, create a class that acts as a collection of your custom class objects and is derived from the generic *Dictionary<T,U>* class.

- **Practice 3** Write a simple console application that performs hundreds of thousands of *Push* and *Pop* operations with the nongeneric and generic versions of the *Stack* class. Time how long it takes for both the nongeneric and generic versions and determine whether the generic version is actually faster.

## Manage Data in a .NET Framework Application by Using Specialized Collections

For this task, you should complete at least Practice 1. For a better understanding of the performance implications of using specialized collections, complete Practice 2 as well.

- **Practice 1** Write an application that creates an instance of each of the built-in specialized collection classes, adds items to each of the collections, and then displays them using a *foreach* loop.
- **Practice 2** Write a simple console application that adds hundreds of thousands of strings to an instance of the *StringCollection* class and then iterates through each of them using a *foreach* loop. Time how long the process takes by comparing *DateTime.Now* before and after it completes. Next, perform the same process using the generic version of *List<T>*, typed for the *string* class. Determine which is faster and whether the performance impact is significant.

## Take a Practice Test

The practice tests on this book's companion CD offer many options. For example, you can test yourself on just the content covered in this chapter, or you can test yourself on all the 70-536 certification exam content. You can set up the test so that it closely simulates the experience of taking a certification exam, or you can set it up in study mode so that you can look at the correct answers and explanations after you answer each question.

**MORE INFO Practice tests**

For details about all the practice test options available, see the section "How to Use the Practice Tests," in the Introduction of this book.

Chapter 10

# Logging and Systems Management

Real-world applications, especially those deployed in IT environments, must be manageable. Making an application manageable involves allowing systems administrators to monitor and troubleshoot the application. The .NET Framework provides the *Systems.Diagnostics* namespace to allow you to write events to the event log, create debug and trace information, and provide performance counters.

IT departments also regularly need internal tools that analyze computer status or respond to changes in the operating system. Windows Management Instrumentation (WMI) provides these capabilities, and the .NET Framework provides a useful WMI interface.

**Exam objectives in this chapter:**

- Manage an event log by using the *System.Diagnostics* namespace.
- Manage system processes and monitor the performance of a .NET Framework application by using the diagnostics functionality of the .NET Framework.
- Debug and trace a .NET Framework application by using the *System.Diagnostics* namespace.
- Embed management information and events into a .NET Framework application.

**Lessons in this chapter:**

## Before You Begin

This book assumes that you have at least two to three years of experience developing Web-based, Microsoft Windows–based, or distributed applications using the .NET Framework. Candidates should have a working knowledge of Microsoft Visual Studio.

Before you begin, you should be familiar with Microsoft Visual Basic or C# and be comfortable with the following tasks:

- Creating Console and Windows Presentation Foundation (WPF) applications in Visual Studio using Visual Basic or C#
- Adding namespaces and system class library references to a project
- Running a project in Visual Studio

## Lesson 1: Logging Application State

Systems administrators rely heavily on the *Windows event log*, a central repository for information about operating system and application activities and errors. For example, Windows adds events each time the operating system starts or shuts down. Applications typically add events when users log on or off, when users change important settings, or when serious errors occur.

By taking advantage of the Windows event log (rather than creating a text-based log file), you allow systems administrators to use their existing event management infrastructure. Most enterprise IT departments have software in place to monitor event logs for important events and forward those events to a central help desk for further processing. Using the Windows event log saves you from writing custom code to support these capabilities.

This lesson describes how to add events, read the event log, and create custom event logs.

**After this lesson, you will be able to:**

- Read and write events
- Log debugging and trace information

**Estimated lesson time: 45 minutes**

## Reading and Writing Events

Systems administrators use the Windows event log to monitor and troubleshoot the operating system. By adding events to the event log, you can provide systems administrators with useful details about the inner workings of your application without directly displaying the information to the user. Because many IT departments have an event management infrastructure that aggregates events, the simple act of adding events to the event log can allow your application to be monitored in enterprise environments.

### How to View the Event Logs

Use the Event Viewer snap-in to view event logs. You can open the Event Viewer snap-in by following these steps in Windows Vista:

1. Click Start, right-click Computer, and then click Manage. Respond to the User Account Control (UAC) prompt if it appears.
2. Expand the Computer Management, System Tools, and Event Viewer nodes.
3. Browse the subfolders to select an event log.

Recent versions of Windows include the following three event logs (among other less frequently used event logs, depending on the version of Windows and the components installed), located within Event Viewer\Windows Logs in Windows Vista:

- **System** Stores all non-security-related operating system events.
- **Security** Stores auditing events, including user logons and logoffs. If nonstandard auditing is enabled, the Security event log can store events when users access specific files or registry values. Applications cannot write to the Security event log.
- **Application** Originally intended to store all events from all applications that do not create an application-specific event log.

## How to Register an Event Source

Events always include a source, which identifies the application that generated the event. Before you log events, you must register your application as a source.

Adding an event source requires administrative privileges. Because Windows Vista does not run programs with administrative privileges by default, adding an event source is best done during the setup process (which typically does have administrative privileges).

If your application is not running as an administrator, you can register an event source manually by following these steps:

1. Log on as an administrator to the application server.
2. Start the registry editor by running Regedit.exe.
3. Locate the following registry subkey:

   HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog\Application
4. Right-click the Application subkey, click New, and then click Key.
5. Type the name of your event source for the key name (for example, **My Application**), and then press Enter.
6. Close the registry editor.

To create an event log source programmatically, call the static *EventLog.CreateEventSource* method with administrative privileges. You can then create events with the registered source. The following code sample determines whether a source already exists and registers the event source with the Application event log if the source does not yet exist:

```
' VB
If Not EventLog.SourceExists("My Application") Then
   EventLog.CreateEventSource("My Application", "Application")
End If

// C#
if (!EventLog.SourceExists("My Application"))
   EventLog.CreateEventSource("My Application", "Application");
```

You can also use *EventLog.CreateEventSource* to create a custom event log, simply by specifying the name. For example, the following code sample creates an event log named My Log and registers a source named My App:

```
' VB
If Not EventLog.Exists("My Log") Then
   EventLog.CreateEventSource("My App", "My Log")
End If

// C#
if (!EventLog.Exists("My Log") )
   EventLog.CreateEventSource("My App", "My Log");
```

In the Windows Vista Event Viewer snap-in, the custom event log appears under Applications And Services Logs. Because calling *EventLog.CreateEventSource* requires administrative privileges, you should call it during your application's setup procedure.

## How to Log Events

Once your application is registered as a source, you can add an event by using an instance of the *EventLog* class (in the *System.Diagnostics* namespace), defining the *EventLog.Source* property, and then calling the *EventLog.WriteEntry* method. *EventLog .WriteEntry* supports the following parameters:

- **message** A text message that should describe the condition as thoroughly as possible.
- **type** The *EventLogEntryType* enumeration, which can be *Information*, *Warning*, *Error*, *FailureAudit* (used when a user is denied access to a resource), or *SuccessAudit* (used when a user is allowed access to a resource).
- **eventID** A number that uniquely identifies the event type. Administrators might use this to search for specific events. You can create your own application-specific event IDs.

- **category** A number that identifies the event category. Like the event ID, this is application-specific.
- **rawData** A byte array that you can provide if you want to give administrators more information about the event.

The following code sample adds an event to the Application event log, assuming that the source "My Application" has already been registered with the Application event log:

```
' VB
Dim myLog As New EventLog("Application")
myLog.Source = "My Application"
myLog.WriteEntry("Could not connect", EventLogEntryType.Error, 1001, 1S)

// C#
EventLog myLog = new EventLog("Application");
myLog.Source = "My Application";
myLog.WriteEntry("Could not connect", EventLogEntryType.Error, 1001, 1);
```

## How to Read Events

To read events, create an *EventLog* instance. Then, access the *EventLog.Entries* collection. The following application displays all Application events to the console:

```
' VB
Dim myLog As New EventLog("Application")
For Each entry As EventLogEntry In myLog.Entries
   Console.WriteLine(entry.Message)
Next

// C#
EventLog myLog = new EventLog("Application");
foreach (EventLogEntry entry in myLog.Entries)
   Console.WriteLine(entry.Message);
```

### Real World

*Tony Northrup*

Whether you're a developer, systems administrator, or user, you've been frustrated by ambiguous error messages at some point. For example, I have this error message in my Application event log: "Faulting application, version, faulting module, version 0.0.0.0, fault address 0x00000000". Good luck fixing the problem based on that!

To avoid this frustration and to facilitate troubleshooting, good developers provide very detailed error messages. Although this is a very user-friendly practice, it can also weaken the security of your application if you list confidential information like usernames, passwords, or connection strings.

## Logging Debugging and Trace Information

Often, during the development process, developers write messages to the console or display dialog boxes to track the application's processes. Although this information can be useful, you wouldn't want it to appear in a production application. To add debug-only code that will not run in release builds, you can use the *System.Diagnostics .Debug* class.

Use the static *Debug.Indent* method to cause all subsequent debugging output to be indented. Use the static *Debug.Unindent* method to remove an indent. Set the *Debug .IndentSize* property to specify the number of spaces with each indent (the default is four), and set the *Debug.IndentLevel* property to specify an indentation level.

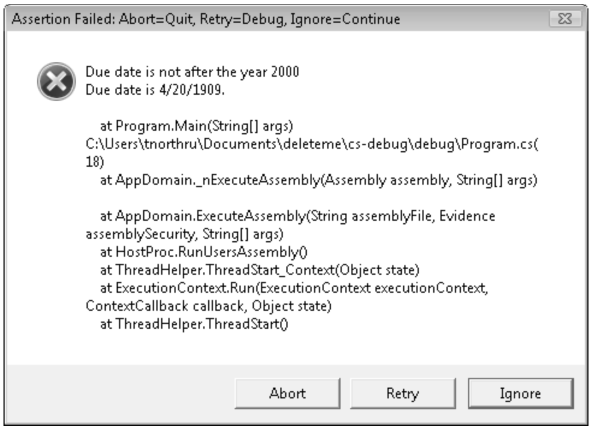
The following code sample demonstrates how to use the *Debug* class to mark the beginning and end of an application. If you build this code in Visual Studio with the build type set to Debug, you will see the Starting Application and Ending Application messages. If you build this code with the build type set to Release, you will not see those messages. However, you will still see the "Hello, world!" message:

```
'VB
Debug.Listeners.Add(New ConsoleTraceListener())
Debug.AutoFlush = True
Debug.Indent()
Debug.WriteLine("Starting application")
Console.WriteLine("Hello, world!")
Debug.WriteLine("Ending application")
Debug.Unindent()

//C#
Debug.Listeners.Add(new ConsoleTraceListener());
Debug.AutoFlush = true;
Debug.Indent();
Debug.WriteLine("Starting application");
Console.WriteLine("Hello, world!");
Debug.WriteLine("Ending application");
Debug.Unindent();
```

*Debug.Write* and *Debug.WriteLine* function exactly the same as *Console.Write* and *Console.WriteLine*. To reduce the amount of code that you need to write, the *Debug* class adds the *WriteIf* and *WriteLineIf* methods, each of which accepts a boolean value as the first parameter and writes the output only if the value is *True*.

*Debug.Assert* also accepts a boolean condition. In general, you should use assertions to verify something that you know should *always* be true. For example, in a financial application, you might use an assertion to verify that the due date of a bill is after the year 2000. If an assertion fails, the Common Language Runtime (CLR) stops program execution and displays a dialog box similar to that shown in Figure 10-1. You should not use asserts in production applications.


Assertion Failed: Abort=Quit, Retry=Debug, Ignore=Continue

```
Due date is not after the year 2000
Due date is 4/20/1909.

   at Program.Main(String[] args)
C:\Users\tnorthru\Documents\deleteme\cs-debug\debug\Program.cs(
18)
   at AppDomain._nExecuteAssembly(Assembly assembly, String[] args)

   at AppDomain.ExecuteAssembly(String assemblyFile, Evidence
assemblySecurity, String[] args)
   at HostProc.RunUsersAssembly()
   at ThreadHelper.ThreadStart_Context(Object state)
   at ExecutionContext.Run(ExecutionContext executionContext,
ContextCallback callback, Object state)
   at ThreadHelper.ThreadStart()
```

**Figure 10-1** A failed call to *Debug.Assert*

*Debug.AutoFlush* determines whether debug output is written immediately. If you always want *Debug* output to be displayed immediately (the most common option), set *Debug.AutoFlush* to *True*. If you want to store *Debug* output and display it all at once (such as when an application exits), set *Debug.AutoFlush* to *False* and call *Debug.Flush* to write the output.

## Using Trace

The *Trace* class functions almost identically to the *Debug* class. However, calls to *Trace* are executed in both Debug and Release builds. Therefore, use the *Debug* class to write messages only in the Debug build, and use the *Trace* class to write messages regardless of the build type. For example, consider the following code sample:

```
'VB
Debug.Listeners.Add(New ConsoleTraceListener())
Debug.AutoFlush = True
Debug.Indent()
Debug.WriteLine("Debug: Starting application")
Trace.WriteLine("Trace: Starting application")

Console.WriteLine("Hello, world!")

Debug.WriteLine("Debug: Ending application")
Trace.WriteLine("Trace: Ending application")

//C#
Debug.Listeners.Add(new ConsoleTraceListener());
Debug.AutoFlush = true;
Debug.Indent();

Debug.WriteLine("Debug: Starting application");
Trace.WriteLine("Trace: Starting application");

Console.WriteLine("Hello, world!");

Debug.WriteLine("Debug: Ending application");
Trace.WriteLine("Trace: Ending application");
```

This code sample generates the following output in a Debug build:

```
    Debug: Starting application
    Trace: Starting application
Hello, world!
    Debug: Ending application
    Trace: Ending application
```

The code sample generates the following, shorter output in a Release build. Notice that the output is not indented because the call to *Debug.Indent* was not executed:

```
Trace: Starting application
Hello, world!
Trace: Ending application
```

Properties that you configure for the *Debug* class also apply to the *Trace* class. For example, if you add a listener to the *Debug* class, you do not need to add the same listener to the *Trace* class.

## Using Listeners

By default, *Debug* and *Trace* write to the Output window in Visual Studio (if you are running the application directly from Visual Studio) because they have a default listener: *DefaultTraceListener*. This allows you to view trace output without directly affecting the application's user interface.

If viewing debug and trace output in the Output window is not sufficient, you can also add the following listeners to the *Debug.Listeners* collection:

- **ConsoleTraceListener** Sends output to the console or the standard error stream.
- **TextWriterTraceListener** Sends output to a text file or a stream. Use *Console.Out* to write output to the console.
- **XmlWriterTraceListener** Sends output to an Extensible Markup Language (XML) file using a *TextWriter* or *Stream* instance. This is useful for creating log files.
- **EventSchemaListener** Sends output to an XML schema–compliant log file. This is useful only if you need output to comply to an existing schema.
- **DelimitedListTraceListener** Sends output to a delimited text file. You can configure the delimiter using the *DelimitedListTraceLister.Delimiter* property.
- **EventLogTraceListener** Writes output to the event log. Each time output is flushed, a separate event is generated. To avoid generating large numbers of events (and possibly affecting performance) set *Debug.AutoFlush* to *False*.

## Configuring Debugging Using a .config File

Often, it's useful to allow users to view trace output. For example, they might be able to use the trace output to isolate a problem or to provide detailed information to you about the internal workings of the application in a production environment. To allow users to enable tracing, add the <*system.diagnostics*> section to your application's .config file.

The following .config file configures a console trace listener and provides instructions for users that allow them to enable tracing selectively:

```
<configuration>
 <system.diagnostics>
   <trace autoflush="false" indentsize="4">
       <listeners>
           <add name="configConsoleListener"
               type="System.Diagnostics.ConsoleTraceListener" />
       </listeners>
   </trace>
   <switches>
       <!-- This switch controls data messages. In order to receive
          data trace messages, change value="0" to value="1" -->
       <add name="DataMessagesSwitch" value="0" />
       <!-- This switch controls general messages. In order to
          receive general trace messages change the value to the
          appropriate level. "1" gives error messages, "2" gives
          errors and warnings, "3" gives more detailed error
          information, and "4" gives verbose trace information -->
```

```
      <add name="TraceLevelSwitch" value="0" />
    </switches>
  </system.diagnostics>
<configuration>
```

The following .config file directs tracing output to a text file and removes the default listener. Notice that you use the *initializeData* attribute when adding the listener to specify the output file—this is true for other listeners that require a filename as well:

```
<configuration>
  <system.diagnostics>
    <trace autoflush="false" indentsize="4">
      <listeners>
        <add name="TextTraceListener"
          type="System.Diagnostics.TextWriterTraceListener"
          initializeData="output.txt" />
        <remove name="Default" />
      </listeners>
    </trace>
  </system.diagnostics>
<configuration>
```

## Lab: Working with Event Logs

In this lab, you will create a WPF application that adds events to a custom event log.

### Exercise 1: Create an Event Log and Log an Event

In this exercise, you must create a solution that includes three projects: a WPF application, a class derived from *Installer*, and a Setup project. You must create a Setup project to add the custom event log during the setup process because the user typically has administrative credentials only during setup.

1. Use Visual Studio to create a new WPF Application project named LoggingApp in either Visual Basic.NET or C#.
2. In the Extensible Application Markup Language (XAML) for the LoggingApp window, add an event handler for the *Loaded* event. The XAML now resembles the following:

```
<Window x:Class="LoggingApp.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Loaded="Window_Loaded"
   Title="Window1" Height="300" Width="300">
   <Grid>

   </Grid>
</Window>
```

3. In the code file, add the *System.Diagnostics* namespace. Then implement the *Loaded* event handler to add an event indicating that the application has started. Add the event to a custom event log named LoggingApp Log with a source of LoggingApp. The following code demonstrates how to do this:

```
' VB
Dim myLog As New EventLog("LoggingApp Log")
myLog.Source = "LoggingApp"
myLog.WriteEntry("LoggingApp started!", _
   EventLogEntryType.Information, 1001)

// C#
EventLog myLog = new EventLog("LoggingApp Log");
myLog.Source = "LoggingApp";
myLog.WriteEntry("LoggingApp started!",
   EventLogEntryType.Information, 1001);
```

4. Add a new project to the solution using the Class Library template, and name it LoggingAppInstaller.
5. In the *LoggingAppInstaller* namespace, derive a custom class named *InstallLog* from the *Installer* class. As described in Lesson 3 of Chapter 9, "Installing and Configuring Applications," implement the *Install, Rollback,* and *Uninstall* methods to add and remove an event log named LoggingApp and a source named Logging AppSource. Note that you need to add a reference to the System.Configuration .Install dynamic-link library (DLL). The following code sample demonstrates how to write the code:

```
' VB
Imports System.Diagnostics
Imports System.Configuration.Install
Imports System.ComponentModel
Imports System.Collections

   <RunInstaller(True)> _
   Public Class InstallLog
       Inherits Installer
       Public Sub New()
           MyBase.New()
       End Sub

       Public Overrides Sub Commit( _
           ByVal mySavedState As IDictionary)
           MyBase.Commit(mySavedState)
       End Sub

       Public Overrides Sub Install( _
           ByVal stateSaver As IDictionary)
           MyBase.Install(stateSaver)
```

```
            If Not EventLog.Exists("LoggingApp Log") Then
                EventLog.CreateEventSource("LoggingApp", "LoggingApp Log")
            End If
        End Sub

        Public Overrides Sub Uninstall( _
            ByVal savedState As IDictionary)
            MyBase.Uninstall(savedState)
            RemoveLog()
        End Sub

        Public Overrides Sub Rollback( _
            ByVal savedState As IDictionary)
            MyBase.Rollback(savedState)
            RemoveLog()
        End Sub

        Public Sub RemoveLog()
            If EventLog.Exists("LoggingApp Log") Then
                EventLog.DeleteEventSource("LoggingApp")
                EventLog.Delete("LoggingApp Log")
            End If
        End Sub
    End Class
```

```
// C#
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Diagnostics;
using System.Configuration.Install;
using System.ComponentModel;
using System.Collections;

namespace LoggingAppInstaller
{
   [RunInstaller(true)]
   public class InstallLog : Installer
   {
       public InstallLog()
           : base()
       {
       }

       public override void Commit(IDictionary mySavedState)
       {
           base.Commit(mySavedState);
       }
```

```
        public override void Install(IDictionary stateSaver)
        {
            base.Install(stateSaver);
            if (!EventLog.Exists("LoggingApp Log"))
                EventLog.CreateEventSource(
                    "LoggingApp", "LoggingApp Log");
        }

        public override void Uninstall(IDictionary savedState)
        {
            base.Uninstall(savedState);
            RemoveLog();
        }

        public override void Rollback(IDictionary savedState)
        {
            base.Rollback(savedState);
            RemoveLog();
        }

        public void RemoveLog()
        {
            if (EventLog.Exists("LoggingApp Log"))
            {
                EventLog.DeleteEventSource("LoggingApp");
                EventLog.Delete("LoggingApp Log");
            }
        }
    }
}
```

6. Add a Setup project to your solution named LoggingApp Setup.
7. Right-click Application Folder in the left pane, click Add, and then click Project Output. In the Add Project Output Group dialog box, click Primary Output for the LoggingApp project and then click OK.
8. Right-click Primary Output From LoggingApp, and then click Create Shortcut To Primary Output From LoggingApp. Name the shortcut LoggingApp and then drag it to the User's Programs Menu folder.
9. Add custom actions to the Setup Project to call the appropriate *InstallLog* methods. Right-click LoggingApp Setup in Solution Explorer, click View, and then click Custom Actions.
10. Right-click Install and then click Add Custom Action. In the Select Item In Project dialog box, double-click Application Folder and then click Add Output. In the Add Project Output Group dialog box, click the Project drop-down list, click LoggingAppInstaller, select Primary Output, and then click OK. Then, click OK again. Accept the default name, and notice that the *InstallerClass* property for the primary output is set to *True*.

11. Right-click Rollback and then click Add Custom Action. In the Select Item In Project dialog box, double-click Application Folder, click Primary Output From LoggingAppInstaller, and then click OK.
12. Repeat step 11 to add the LoggingAppInstaller DLL for the Uninstall custom action.
13. Build your solution. Right-click LoggingApp Setup in Solution Explorer, and then click Build.
14. Open the LoggingApp Setup build destination folder and double-click LoggingApp Setup.msi to start the installer. Accept the default settings to install the application. If you are running Windows Vista, respond appropriately when User Account Control (UAC) prompts you for administrative credentials.
15. Click Start, click All Programs, and then click LoggingApp to start the program. Then, close the window.
16. Open Event Viewer. In Windows Vista, you can do this by clicking Start, right-clicking Computer, and then clicking Manage. Respond to the UAC prompt, expand System Tools, and select Event Viewer.
17. Navigate to Event Viewer, Applications And Services Logs, and LoggingApp Log to verify that the new event log exists. Notice the single event in the event log, indicating that LoggingApp started.
18. Uninstall LoggingApp using the Programs And Features tool in Control Panel.
19. Close and reopen Event Viewer. Notice that LoggingApp Log has been removed.

## Lesson Summary

- Before you can add events, you must register an event source by calling *EventLog.CreateEventSource*. You can then call *EventLog.WriteEntry* to add events. Read events by creating an instance of the *EventLog* class and accessing the *EventLog.Entries* collection.
- Use the *Debug* and *Trace* classes to log the internal workings of your application for troubleshooting purposes. *Debug* functions only in Debug releases. *Trace* can function with any release type. Users can configure listeners for *Debug* and *Trace* using the .config files.

## Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 1, "Logging Application State." The questions are also available on the companion CD if you prefer to review them in electronic form.

**NOTE Answers**

Answers to these questions and explanations of why each answer choice is right or wrong are located in the "Answers" section at the end of the book.

1. You are creating a custom installer for an application that needs to add events to the Application event log. Which of the following do you need to do during the setup process?

   A. Call *EventLog.CreateEventSource*

   B. Call *EventLog.Create*

   C. Call *EventLog.GetEventLogs*

   D. Call *EventLog.WriteEntry*

2. You are creating a custom tool for your IT department that analyzes failure audits generated by the operating system. Which event log should you examine?

   A. Application

   B. System

   C. Security

   D. Setup

3. When running a Debug build of an application, you want to display a dialog box if the result of a calculation (stored in the *result* integer) is less than zero. Which of the following methods does this correctly?

   A. Debug.Assert(result >= 0, "Result error")

   B. Trace.Assert(result >= 0, "Result error")

   C. Debug.WriteIf(result >=  0, "Result error")

   D. Trace.WriteIf(result >= 0, "Result error")

4. You are creating a Console application, and you want *Debug* and *Trace* output displayed directly to the console. Which code sample does this correctly?

   A.

```
'VB
Debug.Listeners.Add(New DefaultTraceListener())
Debug.AutoFlush = True

//C#
Debug.Listeners.Add(new DefaultTraceListener ());
Debug.AutoFlush = true;
```

B.

```
'VB
Debug.Listeners.Add(New ConsoleTraceListener())
Debug.AutoFlush = True

//C#
Debug.Listeners.Add(new ConsoleTraceListener());
Debug.AutoFlush = true;
```

C.

```
'VB
Debug.Listeners.Add(New EventLogTraceListener())
Debug.AutoFlush = True

//C#
Debug.Listeners.Add(new EventLogTraceListener());
Debug.AutoFlush = true;
```

D.

```
'VB
Debug.Listeners.Add(New XmlWriterTraceListener())
Debug.AutoFlush = True

//C#
Debug.Listeners.Add(new XmlWriterTraceListener());
Debug.AutoFlush = true;
```

## Lesson 2: Working with Performance Counters

For years, administrators have used performance counters to monitor the performance of computers, networks, and applications. Developers have also used performance counters to help identify bottlenecks in their application's performance.

With the *System.Diagnostics* namespace in the .NET Framework, you can add custom performance counters and update the performance data from within your application. Then, you or an administrator can monitor any aspect of your application's performance, which can be useful for performance tuning and troubleshooting.

This lesson describes how to monitor standard and custom performance counters and how to add and update custom performance counters.

**After this lesson, you will be able to:**

- Monitor performance counters
- Add custom performance counters
- Provide performance counter data

**Estimated lesson time: 25 minutes**

### Monitoring Performance Counters

Windows includes hundreds of performance counters that allow you to monitor the operating system's activities in real time. You can view these counters using the Performance snap-in. In Windows Vista, you can access the Performance snap-in from within the Computer Management console by following these steps:

1. Click Start, right-click Computer, and then click Manage. Respond to the UAC prompt if it appears.
2. In the Computer Management console, expand System Tools, Reliability And Performance, and Monitoring Tools, and then select Performance Monitor.
3. On the Performance Monitor toolbar, click the button marked with a green plus sign to add a counter.

   The Add Counters dialog box appears, as shown in Figure 10-2.
4. In the Available Counters list, expand a category name and then click a counter. If required, select an instance and click Add.

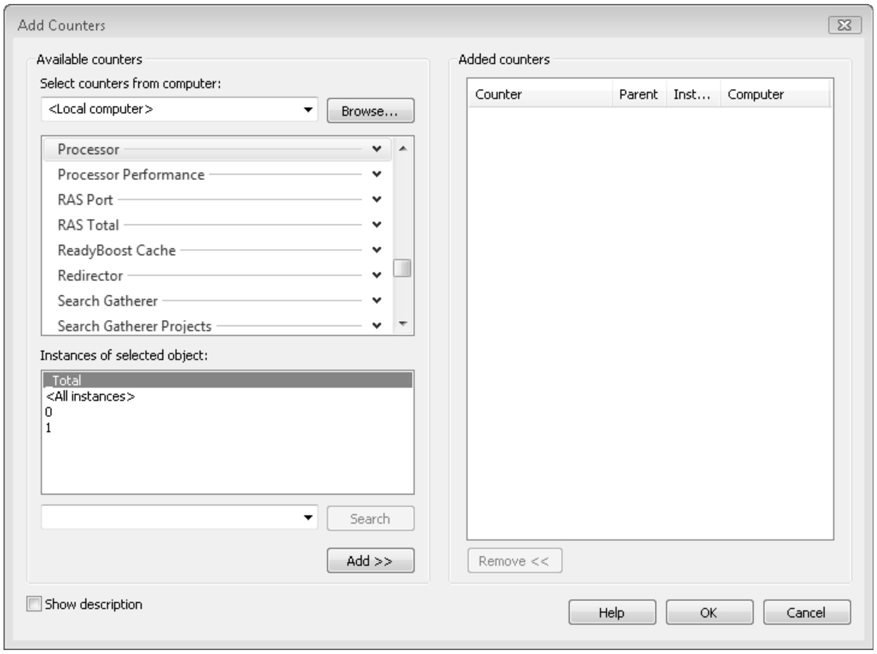


**Figure 10-2** Adding a performance counter

5. Repeat step 4 to add more counters.
6. Click OK to begin monitoring the counters in real time.

   The Performance snap-in displays the values for the counters you selected.

To monitor performance counters within a program, create an instance of *Performance-Counter* by specifying the performance object, counter, and (if required) the instance. You can determine the names of these parameters, and whether an instance is required, by using the Performance snap-in. Then call the *PerformanceCounter.NextValue* method to reset the counter. Make a second call to *PerformanceCounter.NextValue* to retrieve the performance data. Depending on the counter, the performance data might be averaged over the time passed between calls to *PerformanceCounter.NextValue*.

The following code sample, which requires both the *System.Diagnostics* and *System .Threading* namespaces, displays the current processor utilization averaged over a period of 1 second:

```
'VB
' Create a PerformanceCounter object that measures processor time
Dim pc As New PerformanceCounter("Processor", "% Processor Time", "_Total")
```

```
' Reset the performance counter
pc.NextValue()

' Wait one second
Thread.Sleep(1000)

' Retrieve the processor usage over the past second
Console.WriteLine(pc.NextValue())
```

```
//C#
// Create a PerformanceCounter object that measures processor time
PerformanceCounter pc = new
    PerformanceCounter("Processor", "% Processor Time", "_Total");

// Reset the performance counter
pc.NextValue();

// Wait one second
Thread.Sleep(1000);

// Retrieve the processor usage over the past second
Console.WriteLine(pc.NextValue());
```

The first call to *PerformanceCounter.NextValue* always returns zero; therefore, it is always meaningless. Only subsequent calls contain useful data. The following code illustrates this by showing the datagrams sent per second:

```
'VB
Dim pc As New PerformanceCounter("IPv4", "Datagrams/sec")
For i As Integer = 0 To 9

    Console.WriteLine(pc.NextValue())
Next
```

```
//C#
PerformanceCounter pc = new PerformanceCounter("IPv4", "Datagrams/sec");

for (int i = 0; i < 10; i++)
{
    Console.WriteLine(pc.NextValue());
}
```

The output resembles the following, showing that the network interface was receiving 100 to 220 datagrams per second:

```
0
136.4877
213.3919
210.881
106.4458
186.9752
```

```
208.2334
172.8078
127.5594
219.6767
```

Because the IPv4\Datagrams/sec counter is averaged over 1 second, you can query it repeatedly and always retrieve the previous second's average. If you queried the Processor\% Processor Time_Total counter repeatedly, the results would resemble the following because repeatedly querying the value results in the instantaneous utilization. In the case of a computer processor, in any given instant, the processor is either idle or fully utilized—values between 0 and 100 occur only when examining the utilization over a period of time:

```
0
100
100
100
100
100
0
0
100
100
```

## Adding Custom Performance Counters

If you want to provide performance data generated by your application, you should create a custom performance counter category and then add the counters to that category. You can't add performance counters to a built-in category.

To add a custom performance counter category and a single counter, call the static *PerformanceCounterCategory.Create* method. Provide the category name, a description of the category, a name for the counter, and a description of the counter. The following code sample demonstrates this:

```
'VB
PerformanceCounterCategory.Create("CategoryName", "CounterHelp", _
    PerformanceCounterCategoryType.MultiInstance, "CounterName", _
    "CounterHelp")

//C#
PerformanceCounterCategory.Create("CategoryName", "CounterHelp",
    PerformanceCounterCategoryType.MultiInstance, "CounterName",
    "CounterHelp");
```

Note the third parameter: the *PerformanceCounterCategoryType* enumeration. You should specify *SingleInstance* if the counter definitely has only one instance. Specify

*MultiInstance* if the counter might have multiple instances. For example, because computers might have two or more processors, counters that display processor status are always *MultiInstance.*

If you want to add multiple counters to a single category, create an instance of *CounterCreationDataCollection* and add multiple *CounterCreationData* objects to the collection. The following code sample demonstrates this:

```
'VB
Dim counters As CounterCreationDataCollection = New CounterCreationDataCollection
counters.Add(New CounterCreationData("Sales", _
    "Number of total sales", PerformanceCounterType.NumberOfItems64))
counters.Add(New CounterCreationData("Active Users", _
    "Number of active users", PerformanceCounterType.NumberOfItems64))
counters.Add(New CounterCreationData("Sales value", _
    "Total value of all sales", PerformanceCounterType.NumberOfItems64))
PerformanceCounterCategory.Create("MyApp Counters", _
    "Counters describing the performance of MyApp", _
    PerformanceCounterCategoryType.SingleInstance, counters)

//C#
CounterCreationDataCollection counters = new CounterCreationDataCollection();
counters.Add(new CounterCreationData("Sales",
    "Number of total sales", PerformanceCounterType.NumberOfItems64));
counters.Add(new CounterCreationData("Active Users",
    "Number of active users", PerformanceCounterType.NumberOfItems64));
counters.Add(new CounterCreationData ("Sales value",
    "Total value of all sales", PerformanceCounterType.NumberOfItems64));
PerformanceCounterCategory.Create("MyApp Counters",
    "Counters describing the performance of MyApp",
    PerformanceCounterCategoryType.SingleInstance, counters);
```

To check whether a category already exists, use the *PerformanceCounterCategory.Exists* method. To remove an existing category, call *PerformanceCounterCategory.Delete.*

You should add performance counters during an application's setup process for two reasons. First, adding performance counters requires administrative privileges. Second, the operating system requires a few moments to refresh the list of performance counters. Therefore, they might not be accessible the moment you add the counters. However, the typical delay between installing an application and running the application is generally sufficient.

## Providing Performance Counter Data

After you create a custom performance counter, you can update the data as needed. You don't need to update it constantly—just when the value changes. Performance counter data is sampled only every 400 milliseconds, so if you update the value more frequently than that, it won't improve the accuracy significantly.

To update a performance counter, create a *PerformanceCounter* object just as you would for reading a performance counter value. However, you must set the *ReadOnly* property to *false*. You can do this using the overloaded *PerformanceCounter* constructor that takes a boolean parameter, as shown here, or you can set the *ReadOnly* property after creating the object:

```
'VB
Dim pc As PerformanceCounter = New PerformanceCounter( _
    "MyApp Counters", "Sales", False)

//C#
PerformanceCounter pc = new PerformanceCounter(
    "MyApp Counters", "Sales", false);
```

After creating the *PerformanceCounter* object, you can set the value directly by defining the *RawValue* property. Alternatively, you can call the thread-safe *Decrement*, *Increment*, and *IncrementBy* methods to adjust the value relative to the current value. The following code sample demonstrates how to use each of these methods:

```
'VB
Dim pc As PerformanceCounter = New PerformanceCounter( _
    "MyApp Counters", "Sales", False)
pc.RawValue = 7
pc.Decrement
pc.Increment
pc.IncrementBy(3)

//C#
PerformanceCounter pc = new PerformanceCounter(
    "MyApp Counters", "Sales", false);
pc.RawValue = 7;
pc.Decrement();
pc.Increment();
pc.IncrementBy(3);
```

*PerformanceCounter.Increment* and *PerformanceCounter.Decrement* are thread-safe, but they're much slower than simply updating *PerformanceCounter.RawValue*. Therefore, you should use *PerformanceCounter.Increment* and *PerformanceCounter.Decrement* only when multiple threads might update the performance counter simultaneously.

## Lab: Providing Performance Data

In this lab, you will create an application that provides performance data that systems administrators can use to monitor the application's performance.

## Exercise 1: Create and Update Performance Counters

In this exercise, you will create a solution that includes three projects: a WPF application, a class derived from *Installer*, and a Setup project. You must create a Setup project to add the custom performance counter during the setup process because the user typically has administrative credentials only during setup. The application that you create will record the number of times the user has clicked a button in a custom performance counter.

1. Use Visual Studio to create a new WPF Application project named PerfApp in either Visual Basic .NET or C#.
2. Add a single *Label* control named *counterLabel* and a single *Button* control named *counterButton* to the form. Double-click *counterButton* to edit the *Click* event handler.
3. In the code file, add the *System.Diagnostics* namespace. Then write code in the *counterButton.Click* event handler to increment the *PerfApp\Clicks* counter and display the current value in *counterLabel*. The following code demonstrates how to do this:

```
' VB
Dim pc As New PerformanceCounter("PerfApp", "Clicks", False)
pc.Increment()
counterLabel.Content = pc.NextValue().ToString()

// C#
PerformanceCounter pc = new PerformanceCounter("PerfApp", "Clicks", false);
pc.Increment();
counterLabel.Content = pc.NextValue().ToString();
```

4. Add a new project to the solution using the Class Library template. Name the project PerfAppInstaller.
5. In the *PerfAppInstaller* namespace, derive a custom class named *InstallCounter* from the *Installer* class. As described in Lesson 3 of Chapter 9, implement the *Install, Rollback,* and *Uninstall* methods to add and remove a performance category named PerfApp and a counter named Clicks. You need to add a reference to the System.Configuration.Install DLL. The following code sample demonstrates how to write the code:

```
' VB
Imports System.Configuration.Install
Imports System.ComponentModel

<RunInstaller(True)> _
       Public Class InstallCounter
   Inherits Installer
   Public Sub New()
       MyBase.New()
   End Sub
   Public Overloads Overrides Sub Commit( _
       ByVal mySavedState As IDictionary)
       MyBase.Commit(mySavedState)
   End Sub

   Public Overloads Overrides Sub Install( _
       ByVal stateSaver As IDictionary)
       MyBase.Install(stateSaver)
       If Not PerformanceCounterCategory.Exists("PerfApp") Then
           PerformanceCounterCategory.Create("PerfApp", _
               "Counters for PerfApp", _
               PerformanceCounterCategoryType.SingleInstance, _
               "Clicks", "Times the user has clicked the button.")
       End If
   End Sub

   Public Overloads Overrides Sub Uninstall( _
       ByVal savedState As IDictionary)
       MyBase.Uninstall(savedState)
       If PerformanceCounterCategory.Exists("PerfApp") Then
           PerformanceCounterCategory.Delete("PerfApp")
       End If
   End Sub

   Public Overloads Overrides Sub Rollback( _
       ByVal savedState As IDictionary)
       MyBase.Rollback(savedState)
       If PerformanceCounterCategory.Exists("PerfApp") Then
           PerformanceCounterCategory.Delete("PerfApp")
       End If
   End Sub
End Class

// C#
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Diagnostics;
using System.Configuration.Install;
using System.ComponentModel;
using System.Collections;

namespace PerfAppInstaller
{
   [RunInstaller(true)]
   public class InstallCounter : Installer
   {
       public InstallCounter()
           : base()
       {
       }
```

```
        public override void Commit(IDictionary mySavedState)
        {
            base.Commit(mySavedState);
        }

        public override void Install(IDictionary stateSaver)
        {
            base.Install(stateSaver);
            if (!PerformanceCounterCategory.Exists("PerfApp"))
                PerformanceCounterCategory.Create("PerfApp",
                    "Counters for PerfApp",
                    PerformanceCounterCategoryType.SingleInstance,
                    "Clicks",
                    "Times the user has clicked the button.");
        }

        public override void Uninstall(IDictionary savedState)
        {
            base.Uninstall(savedState);
            if (PerformanceCounterCategory.Exists("PerfApp"))
                PerformanceCounterCategory.Delete("PerfApp");
        }

        public override void Rollback(IDictionary savedState)
        {
            base.Rollback(savedState);
            if (PerformanceCounterCategory.Exists("PerfApp"))
                PerformanceCounterCategory.Delete("PerfApp");
        }
    }
}
```

6. Add a Setup project named PerfApp Setup to your solution.
7. Right-click Application Folder in the left pane, click Add, and then click Project Output. In the Add Project Output Group dialog box, click Primary Output for the PerfApp project and click OK.
8. Right-click Primary Output From PerfApp and then click Create Shortcut To Primary Output From PerfApp. Name the shortcut PerfApp and then drag it to the User's Programs Menu folder.
9. Add custom actions to the Setup project to call the appropriate *InstallLog* methods. Right-click PerfApp Setup in Solution Explorer, click View, and then click Custom Actions.
10. Right-click Install and then click Add Custom Action. In the Select Item In Project dialog box, double-click Application Folder and then click Add Output. In the Add Project Output Group dialog box, click the Project drop-down list

and then click PerfAppInstaller. Select Primary Output, click OK and then click OK again. Accept the default name, and notice that the InstallerClass property for the primary output is set to True.

11. Right-click Rollback and then click Add Custom Action. In the Select Item In Project dialog box, double-click Application Folder. Click Primary Output From PerfAppInstaller and then click OK.
12. Repeat step 11 to add the PerfAppInstaller DLL for the Uninstall custom action.
13. Build your solution. Right-click PerfApp Setup in Solution Explorer and then click Build.
14. Open the PerfApp Setup build destination folder and double-click PerfApp Setup.msi to start the installer. Accept the default settings to install the application. If you are running Windows Vista, respond appropriately when UAC prompts you for administrative credentials.
15. Open the Performance snap-in and add the *PerfApp\Clicks* counter to monitor it in real time.
16. Leave the Performance snap-in running. Click Start, click All Programs, and then click PerfApp to start the program. Click the button several times to increment the counter. Notice that *counterLabel* displays the number of clicks and the Performance snap-in shows the value in real time.
17. Uninstall PerfApp using the Programs And Features tool in Control Panel.
18. Close and reopen the Performance snap-in. Notice that the *PerfApp* counter category has been removed.

## Lesson Summary

- To monitor performance counters programmatically, create an instance of *PerformanceCounter*. Then call the *PerformanceCounter.NextValue* method to reset the counter. Make subsequent calls to *PerformanceCounter.NextValue* to retrieve the performance data.
- To add custom performance counters, call the static *PerformanceCounterCategory .Create* method. Provide the category name, a description of the category, a name for the counter, and a description of the counter.
- To provide performance counter data, create a *PerformanceCounter* object and set the *ReadOnly* property to *false*. You can then set the value directly by defining the *RawValue* property or by calling the thread-safe *Decrement*, *Increment*, and *IncrementBy* methods.

## Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 2, "Working with Performance Counters." The questions are also available on the companion CD if you prefer to review them in electronic form.

**NOTE Answers**

Answers to these questions and explanations of why each answer choice is right or wrong are located in the "Answers" section at the end of the book.

1. You are creating a multithreaded application. You create an instance of *PerformanceCounter* named *pc* that might be referenced from multiple threads simultaneously. Which of the following calls is thread-safe? (Choose all that apply.)
   - A. *pc.RawValue = pc.RawValue + 32*
   - B. *pc.Increment()*
   - C. *pc.Decrement()*
   - D. *pc.Increment(12)*
2. You want to add a performance counter category with multiple counters programmatically. Which class should you use to specify the counters?
   - A. *PerformanceCounterCategory*
   - B. *CounterSample*
   - C. *CounterCreationDataCollection*
   - D. *CounterCreationData*

## Lesson 3: Managing Computers

Applications often need to examine aspects of a computer, such as currently running processes and locally attached storage devices. In addition, it's often useful to respond to changes in the system status, such as a new process starting or a newly attached storage device. You can use the *Process* class and Windows Management Instrumentation (WMI) to accomplish these tasks with the .NET Framework.

**After this lesson, you will be able to:**

- Examine processes
- Access WMI information and respond to WMI events

**Estimated lesson time: 20 minutes**

### Examining Processes

You can use the *Process.GetProcesses* static method to retrieve a list of current processes. The following code sample lists the process ID (PID) and process name of all processes visible to the assembly. Processes run by other users might not be visible:

```
' VB
For Each p As Process In Process.GetProcesses()
   Console.WriteLine("{0}: {1}", p.Id.ToString(), p.ProcessName)
Next

// C#
foreach (Process p in Process.GetProcesses())
   Console.WriteLine("{0}: {1}", p.Id, p.ProcessName);
```

To retrieve a specific process by ID, call *Process.GetProcessById*. To retrieve a list of processes with a specific name, call *Process.GetProcessesByName*. To retrieve the current process, call *Process.GetCurrentProcess*.

Once you create a *Process* instance, you can access a list of the modules loaded by that process using *Process.Modules* (if you have sufficient privileges). If you lack sufficient privileges (which vary depending on the process), the CLR throws a *Win32Exception*. The following code sample demonstrates how to list all processes and modules when sufficient privileges are available:

```
' VB
For Each p As Process In Process.GetProcesses()
   Console.WriteLine("{0}: {1}", p.Id.ToString(), p.ProcessName)
   Try
       For Each pm As ProcessModule In p.Modules
```

```
            Console.WriteLine("    {0}: {1}", pm.ModuleName, _
                pm.ModuleMemorySize.ToString())
        Next
    Catch ex As System.ComponentModel.Win32Exception
        Console.WriteLine("    Unable to list modules")
    End Try
Next

// C#
foreach (Process p in Process.GetProcesses())
{
    Console.WriteLine("{0}: {1}", p.Id.ToString(), p.ProcessName);
    try
    {
        foreach (ProcessModule pm in p.Modules)
            Console.WriteLine("    {0}: {1}", pm.ModuleName,
                pm.ModuleMemorySize.ToString());
    }
    catch (System.ComponentModel.Win32Exception ex)
    {
        Console.WriteLine("    Unable to list modules");
    }
}
```

The first time you reference any *Process* property, the *Process* class retrieves and caches values for all *Process* properties. Therefore, property values might be outdated. To retrieve updated information, call the *Process.Refresh* method.

The following are some of the most useful *Process* properties:

- **BasePriority** The priority of the process.
- **ExitCode** After a process terminates, the instance of the *Process* class populates the *ExitCode* and *ExitTime* properties. The meaning of the *ExitCode* property is defined by the application, but typically zero indicates a nonerror ending, and any nonzero value indicates the application ended with an error.
- **ExitTime** The time the process ended.
- **HasExited** A boolean value that is *true* if the process has ended.
- **Id** The PID.
- **MachineName** The name of the computer on which the process is running.
- **Modules** A list of modules loaded by the process.
- **NonpagedMemorySize64** The amount of nonpaged memory allocated to the process. Nonpaged memory must be stored in RAM.

- **PagedMemorySize64** The amount of paged memory allocated to the process. Paged memory can be moved to the paging file.
- **ProcessName** The name of the process, which is typically the same as the executable file.
- **TotalProcessorTime** The total amount of processing time the process has consumed.

To start a new process, call the *Process.Start* static method and specify the name of the executable file. If you want to pass the process parameters (such as command-line parameters), pass those as a second string. The following code sample shows how to start Notepad and have it open the C:\Windows\Win.ini file:

```
' VB
Process.Start("Notepad.exe", "C:\windows\win.ini")

// C#
Process.Start("Notepad.exe", @"C:\windows\win.ini");
```

## Accessing Management Information

Windows exposes a great deal of information about the computer and operating system through WMI. WMI information is useful when you need to examine the computer to determine how to set up your application, or when creating tools for systems management or inventory.

First, define the management scope by creating a new *ManagementScope* object and calling *ManagementScope.Connect*. Typically, the management scope is \\<*computer_name*>\root\cimv2. The following code sample, which requires the *System.Management* namespace, demonstrates how to create the management scope:

```
' VB
Dim scope As New ManagementScope("\\localhost\root\cimv2")
scope.Connect()

// C#
ManagementScope scope =
   new ManagementScope(@"\\localhost\root\cimv2");
scope.Connect();
```

You also need to create a WMI Query Language (WQL) query using an instance of *ObjectQuery*, which will be executed within the scope you specified. WQL is a subset of Structured Query Language (SQL) with extensions to support WMI event notification and other WMI-specific features. The following code sample demonstrates how to query all objects in the *Win32_OperatingSystem* object. However, there are many different WMI objects. For a complete list, refer to WMI Classes at *http://msdn.microsoft.com/en-us/library/aa394554.aspx*.

```
' VB
Dim query As New ObjectQuery( _
    "SELECT * FROM Win32_OperatingSystem")

// C#
ObjectQuery query = new ObjectQuery(
   "SELECT * FROM Win32_OperatingSystem");
```

With the scope and query defined, you can execute your query by creating a *ManagementObjectSearcher* object and then calling the *ManagementObjectSearcher.Get* method to create a *ManagementObjectCollection* object.

```
' VB
Dim searcher As New ManagementObjectSearcher(scope, query)
Dim queryCollection As ManagementObjectCollection = searcher.Get()

// C#
ManagementObjectSearcher searcher = new ManagementObjectSearcher(scope, query);
ManagementObjectCollection queryCollection = searcher.Get();
```

Alternatively, you can use the overloaded *ManagementObjectSearcher* constructor to specify the query without creating separate scope or query objects, as the following example demonstrates:

```
' VB
Dim searcher As New ManagementObjectSearcher( _
    "SELECT * FROM Win32_LogicalDisk")
Dim queryCollection As ManagementObjectCollection = searcher.Get()

// C#
ManagementObjectSearcher searcher =
   new ManagementObjectSearcher(
       "SELECT * FROM Win32_LogicalDisk");
ManagementObjectCollection queryCollection = searcher.Get();
```

Finally, you can iterate through the *ManagementObject* objects in the *ManagementObjectCollection* and directly access the properties. The following loop lists several properties from the *ManagementObject* defined in the *Win32_OperatingSystem* example shown earlier:

```
' VB
For Each m As ManagementObject In queryCollection
    Console.WriteLine("Computer Name : {0}", m("csname"))
    Console.WriteLine("Windows Directory : {0}", m("WindowsDirectory"))
    Console.WriteLine("Operating System: {0}", m("Caption"))
    Console.WriteLine("Version: {0}", m("Version"))
    Console.WriteLine("Manufacturer : {0}", m("Manufacturer"))
Next

// C#
foreach (ManagementObject m in queryCollection)
{
    Console.WriteLine("Computer Name : {0}", m["csname"]);
    Console.WriteLine("Windows Directory : {0}", m["WindowsDirectory"]);
    Console.WriteLine("Operating System: {0}", m["Caption"]);
    Console.WriteLine("Version: {0}", m["Version"]);
    Console.WriteLine("Manufacturer : {0}", m["Manufacturer"]);
}
```

The following code sample demonstrates how to query the local computer for operating system details:

```
'VB
' Perform the query
Dim searcher As New ManagementObjectSearcher( _
    "SELECT * FROM Win32_OperatingSystem")
Dim queryCollection As ManagementObjectCollection = searcher.Get()

' Display the data from the query
For Each m As ManagementObject In queryCollection
    ' Display the remote computer information
    Console.WriteLine("Computer Name : {0}", m("csname"))
    Console.WriteLine("Windows Directory : {0}", m("WindowsDirectory"))
    Console.WriteLine("Operating System: {0}", m("Caption"))
    Console.WriteLine("Version: {0}", m("Version"))
    Console.WriteLine("Manufacturer : {0}", m("Manufacturer"))
Next

//C#
// Perform the query
ManagementObjectSearcher searcher =
   new ManagementObjectSearcher(
       "SELECT * FROM Win32_OperatingSystem");
ManagementObjectCollection queryCollection = searcher.Get();

// Display the data from the query
foreach (ManagementObject m in queryCollection)
{
   // Display the remote computer information
   Console.WriteLine("Computer Name : {0}", m["csname"]);
   Console.WriteLine("Windows Directory : {0}", m["WindowsDirectory"]);
   Console.WriteLine("Operating System: {0}", m["Caption"]);
   Console.WriteLine("Version: {0}", m["Version"]);
   Console.WriteLine("Manufacturer : {0}", m["Manufacturer"]);
}
```

Similarly, the following code lists all disks connected to the local computer:

```
'VB
' Create a scope to identify the computer to query
Dim scope As New ManagementScope("\\localhost\root\cimv2")
scope.Connect()

' Create a query for operating system details
Dim query As New ObjectQuery("SELECT * FROM Win32_LogicalDisk")

' Perform the query
Dim searcher As New ManagementObjectSearcher(scope, query)
Dim queryCollection As ManagementObjectCollection = searcher.Get()

' Display the data from the query
For Each m As ManagementObject In queryCollection
    ' Display the remote computer information
    Console.WriteLine("{0} {1}", m("Name").ToString(), _
        m("Description").ToString())
Next

//C#
// Create a scope to identify the computer to query
ManagementScope scope = new ManagementScope(@"\\localhost\root\cimv2");
scope.Connect();

// Create a query for operating system details
ObjectQuery query =
   new ObjectQuery("SELECT * FROM Win32_LogicalDisk");

// Perform the query
ManagementObjectSearcher searcher =
   new ManagementObjectSearcher(scope, query);
ManagementObjectCollection queryCollection = searcher.Get();

// Display the data from the query
foreach (ManagementObject m in queryCollection)
{
   // Display the remote computer information
   Console.WriteLine("{0} {1}", m["Name"].ToString(),
       m["Description"].ToString());
}
```

**Exam Tip** The number of WMI Classes is immense. Fortunately, you don't have to be able to list them for the 70-536 exam. Instead, familiarize yourself conceptually with how to write WMI queries and retrieve the results. For a complete reference, refer to WMI Classes at *http://msdn.microsoft.com/en-us/library/aa394554.aspx*.

## Waiting for WMI Events

You can also respond to WMI events, which are triggered by changes to the operating system status by creating an instance of *WqlEventQuery*. To create an instance of *WqlEventQuery*, pass the constructor an event class name, a query interval, and a query condition. Then, use the *WqlEventQuery* to create an instance of *ManagementEventWatcher*.

You can then use *ManagementEventWatcher* to either create an event handler that will be called (using *ManagementEventWatcher.EventArrived*) or wait for the next event (by calling *ManagementEventWatcher.WaitForNextEvent*). If you call *ManagementEventWatcher.WaitForNextEvent*, it returns an instance of *ManagementBaseObject*, which you can use to retrieve the query-specific results.

The following code creates a WQL event query to detect a new process, waits for a new process to start, and then displays the information about the process:

```
'VB
' Create event query to be notified within 1 second of a change
' in a service
Dim query As New WqlEventQuery("__InstanceCreationEvent", _
    New TimeSpan(0, 0, 1), "TargetInstance isa ""Win32_Process""")

' Initialize an event watcher and subscribe to events that match this query
Dim watcher As New ManagementEventWatcher(query)

' Block until the next event occurs
Dim e As ManagementBaseObject = watcher.WaitForNextEvent()

' Display information from the event
Console.WriteLine("Process {0} has been created, path is: {1}", _
    DirectCast(e("TargetInstance"), ManagementBaseObject)("Name"), _
    DirectCast(e("TargetInstance"), ManagementBaseObject)("ExecutablePath"))

' Cancel the subscription
watcher.Stop()

//C#
// Create event query to be notified within 1 second of a change
// in a service
WqlEventQuery query = new WqlEventQuery("__InstanceCreationEvent",
       new TimeSpan(0, 0, 1),
       "TargetInstance isa \"Win32_Process\"");

// Initialize an event watcher and subscribe to events that match this query
ManagementEventWatcher watcher = new ManagementEventWatcher(query);

// Block until the next event occurs
ManagementBaseObject e = watcher.WaitForNextEvent();
```

```
// Display information from the event
Console.WriteLine("Process {0} has been created, path is: {1}",
   ((ManagementBaseObject)e["TargetInstance"])["Name"],
   ((ManagementBaseObject)e["TargetInstance"])["ExecutablePath"]);

// Cancel the subscription
watcher.Stop();
```

## Responding to WMI Events with an Event Handler

You can respond to the *ManagementEventWatcher.EventArrived* event to call a method each time a WMI event occurs. Your event handler must accept two parameters: an *object* parameter and an *EventArrivedEventArgs* parameter. *EventArrivedEventArgs.NewEvent* is a *ManagementBaseObject* that describes the event.

The following Console application demonstrates how to handle WMI events asynchronously. It performs the exact same task as the previous code sample:

```
'VB
Sub Main()
    Dim watcher As ManagementEventWatcher = Nothing

    Dim receiver As New EventReceiver()

    ' Create the watcher and register the callback.
    watcher = GetWatcher(New EventArrivedEventHandler( _
        AddressOf receiver.OnEventArrived))

    ' Watcher starts to listen to the Management Events.
    watcher.Start()

    ' Run until the user presses a key
    Console.ReadKey()
    watcher.Stop()
End Sub

' Create a ManagementEventWatcher object.
Public Function GetWatcher(ByRef handler As EventArrivedEventHandler) _
    As ManagementEventWatcher
    ' Create event query to be notified within 1 second of a change
    ' in a service
    Dim query As New WqlEventQuery("__InstanceCreationEvent", _
        New TimeSpan(0, 0, 1), "TargetInstance isa ""Win32_Process""")

    ' Initialize an event watcher and subscribe to events that match
    ' this query
    Dim watcher As New ManagementEventWatcher(query)

    ' Attach the EventArrived property to EventArrivedEventHandler method with the required
    handler to allow watcher object communicate to the application.
    AddHandler watcher.EventArrived, handler
    Return watcher
End Function

Class EventReceiver
    ' Handle the event and display the ManagementBaseObject properties.
    Public Sub OnEventArrived(ByVal sender As Object, _
        ByVal e As EventArrivedEventArgs)
        ' EventArrivedEventArgs is a management event.
        Dim evt As ManagementBaseObject = e.NewEvent

        ' Display information from the event
        Console.WriteLine("Process {0} has been created, path is: {1}", _
            DirectCast(evt("TargetInstance"), _
                ManagementBaseObject)("Name"),  _
            DirectCast(evt("TargetInstance"), _
                ManagementBaseObject)("ExecutablePath"))
    End Sub
End Class
```

**//C#**

```
static void Main(string[] args)
{
   ManagementEventWatcher watcher = null;

   EventReceiver receiver = new EventReceiver();

   // Create the watcher and register the callback
   watcher = GetWatcher(
       new EventArrivedEventHandler(receiver.OnEventArrived));

   // Watcher starts to listen to the Management Events.
   watcher.Start();

   // Run until the user presses a key
   Console.ReadKey();
    watcher.Stop();
}

// Create a ManagementEventWatcher object.
public static ManagementEventWatcher GetWatcher(
   EventArrivedEventHandler handler)
{
   // Create event query to be notified within 1 second of a
   // change in a service
   WqlEventQuery query = new WqlEventQuery("__InstanceCreationEvent",
       new TimeSpan(0, 0, 1),
       "TargetInstance isa \"Win32_Process\"");

   // Initialize an event watcher and subscribe to events that
   // match this query
   ManagementEventWatcher watcher = new ManagementEventWatcher(query);
```

```
        // Attach the EventArrived property to
        // EventArrivedEventHandler method with the
        // required handler to allow watcher object communicate to
        // the application.
        watcher.EventArrived += new EventArrivedEventHandler(handler);
        return watcher;
    }

// Handle the event and display the ManagementBaseObject
// properties.
class EventReceiver
{
    public void OnEventArrived(object sender,
        EventArrivedEventArgs e)
    {
        // EventArrivedEventArgs is a management event.
        ManagementBaseObject evt = e.NewEvent;

        // Display information from the event
        Console.WriteLine("Process {0} has been created, path is: {1}",
            ((ManagementBaseObject)
                evt["TargetInstance"])["Name"],
            ((ManagementBaseObject)
                evt["TargetInstance"])["ExecutablePath"]);
    }
}
```

## Lab: Create an Alarm Clock

In this lab, you create a WPF application that uses WMI events to trigger an alarm every minute.

### Exercise 1: Respond to a WMI Event

In this exercise, you create a WPF application that displays a dialog box every minute by responding to WMI events when the value of the computer's clock equals zero seconds.

1. Use Visual Studio to create a new WPF Application project named Alarm, in either Visual Basic.NET or C#.
2. In the XAML, add handlers for the *Loaded* and *Closing* events, as shown in bold here:

```
<Window x:Class="Alarm.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Loaded="Window_Loaded"
    Closing="Window_Closing"
    Title="Window1" Height="300" Width="300">
```

3. Add a reference to System.Management.dll to your project. Then add the *System .Management* namespace to the code file.
4. In the window's class, declare an instance of *ManagementEventWatcher* so that it can be accessible from all methods in the class. You need to use this instance to start and stop the *EventArrived* handler:

```
' VB
Class Window1
    Dim watcher As ManagementEventWatcher = Nothing
End Class

// C#
public partial class Window1 : Window
{
   ManagementEventWatcher watcher = null;
}
```

5. Add a class and a method to handle the WMI query event by displaying the current time in a dialog box. The following code sample demonstrates this:

```
' VB
Class EventReceiver
    Public Sub OnEventArrived(ByVal sender As Object, _
        ByVal e As EventArrivedEventArgs)
        Dim evt As ManagementBaseObject = e.NewEvent

        ' Display information from the event
        Dim time As String = [String].Format("{0}:{1:00}", _
            DirectCast(evt("TargetInstance"), _
                ManagementBaseObject)("Hour"), _
            DirectCast(evt("TargetInstance"), _
                ManagementBaseObject)("Minute"))

        MessageBox.Show(time, "Current time")
    End Sub
End Class

// C#
class EventReceiver
{
    public void OnEventArrived(object sender, EventArrivedEventArgs e)
    {
        ManagementBaseObject evt = e.NewEvent;

        // Display information from the event
        string time = String.Format("{0}:{1:00}",
            ((ManagementBaseObject)evt["TargetInstance"])["Hour"],
            ((ManagementBaseObject)evt["TargetInstance"])["Minute"]);

        MessageBox.Show(time, "Current time");
    }
}
```

6. Add a method to the *Window* class to create a WMI event query that is triggered when the number of seconds on the computer's clock is zero. This causes the event to be triggered every minute. Then register *OnEventArrived* as the event handler. The following code demonstrates this:

```
' VB
Public Shared Function GetWatcher(ByVal handler As _
    EventArrivedEventHandler) As ManagementEventWatcher
    ' Create event query to be notified within 1 second of a
    ' change in a service
    Dim query As New WqlEventQuery("__InstanceModificationEvent", _
        New TimeSpan(0, 0, 1), _
        "TargetInstance isa 'Win32_LocalTime' AND " + _
        "TargetInstance.Second = 0")

    ' Initialize an event watcher and subscribe to events that
    ' match this query
    Dim watcher As New ManagementEventWatcher(query)

    ' Attach the EventArrived property to EventArrivedEventHandler method
    ' with the required handler to allow watcher object communicate to the
    ' application.
    AddHandler watcher.EventArrived, handler
    Return watcher
End Function

// C#
public static ManagementEventWatcher GetWatcher(
    EventArrivedEventHandler handler)
{
    // Create event query to be notified within 1 second of a change in a
    // service
    WqlEventQuery query = new WqlEventQuery("__InstanceModificationEvent",
        new TimeSpan(0, 0, 1),
        "TargetInstance isa 'Win32_LocalTime' AND " +
        "TargetInstance.Second = 0");

    // Initialize an event watcher and subscribe to events that
    // match this query
    ManagementEventWatcher watcher = new ManagementEventWatcher(query);

    // Attach the EventArrived property to EventArrivedEventHandler method
    // with the required handler to allow watcher object communicate to the
    // application.
    watcher.EventArrived += new EventArrivedEventHandler(handler);
    return watcher;
}
```

7. Finally, handle the window's *Loaded* and *Closing* events to start and stop the event handler, as follows:

```
' VB
Private Sub Window_Loaded(ByVal sender As Object, _
    ByVal e As RoutedEventArgs)
    ' Event Receiver is a user-defined class.
    Dim receiver As New EventReceiver()

    ' Here, we create the watcher and register the callback with it
    ' in one shot.
    watcher = GetWatcher(New EventArrivedEventHandler( _
        AddressOf receiver.OnEventArrived))

    ' Watcher starts to listen to the Management Events.
    watcher.Start()
End Sub

Private Sub Window_Closing(ByVal sender As Object, _
    ByVal e As System.ComponentModel.CancelEventArgs)
    watcher.Stop()
End Sub

// C#
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Event Receiver is a user-defined class.
    EventReceiver receiver = new EventReceiver();

    // Here, we create the watcher and register the callback with it
    // in one shot.
    watcher = GetWatcher(
        new EventArrivedEventHandler(receiver.OnEventArrived));

    // Watcher starts to listen to the Management Events.
    watcher.Start();
}

private void Window_Closing(object sender,
    System.ComponentModel.CancelEventArgs e)
{
    watcher.Stop();
}
```

8. Build and run the application. When the number of seconds on your computer's clock equals zero, the *OnEventArrived* method displays a dialog box showing the current time.

## Lesson Summary

- You can examine processes by calling the *Process.GetProcesses* method. To start a process, call *Process.Start.*
- To read WMI data, first define the management scope by creating a new *ManagementScope* object and calling *ManagementScope.Connect*. Then create a query using an instance of *ObjectQuery*. With the scope and query defined, you can execute your query by creating a *ManagementObjectSearcher* object and then calling the *ManagementObjectSearcher.Get* method. You can also respond to WMI events by creating an instance of *WqlEventQuery*. Then, use the *WqlEventQuery* to create an instance of *ManagementEventWatcher*. You can then use *ManagementEventWatcher* to either create an event handler or wait for the next event.

## Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 3, "Managing Computers." The questions are also available on the companion CD if you prefer to review them in electronic form.

**NOTE Answers**

Answers to these questions and explanations of why each answer choice is right or wrong are located in the "Answers" section at the end of the book.

1. You need to retrieve a list of all running processes. Which method should you call?

   A. *Process.GetProcessesByName*

   B. *Process.GetCurrentProcess*

   C. *Process.GetProcesses*

   D. *Process.GetProcessById*

2. You need to query WMI for a list of logical disks attached to the current computer. Which code sample correctly runs the WMI query?

   A.

```
' VB
Dim searcher As New ObjectQuery("SELECT * FROM Win32_LogicalDisk")
Dim query As ManagementObject = searcher.Get()

// C#
ObjectQuery searcher = new ObjectQuery("SELECT * FROM Win32_LogicalDisk");
ManagementObject query = searcher.Get();
```

**B.**

```
' VB
Dim searcher As New ManagementObjectSearcher( _
    "SELECT * FROM Win32_LogicalDisk")
Dim queryCollection As ManagementObjectCollection = searcher.Get()

// C#
ManagementObjectSearcher searcher =
    new ManagementObjectSearcher("SELECT * FROM Win32_LogicalDisk");
ManagementObject query = searcher.Get();
```

**C.**

```
' VB
Dim searcher As New ObjectQuery("SELECT * FROM Win32_LogicalDisk")
Dim queryCollection As ManagementObjectCollection = searcher.Get()

// C#
ObjectQuery searcher = new ObjectQuery("SELECT * FROM Win32_LogicalDisk");
ManagementObjectCollection queryCollection = searcher.Get();
```

**D.**

```
' VB
Dim searcher As New ManagementObjectSearcher( _
    "SELECT * FROM Win32_LogicalDisk")
Dim queryCollection As ManagementObjectCollection = searcher.Get()

// C#
ManagementObjectSearcher searcher =
    new ManagementObjectSearcher("SELECT * FROM Win32_LogicalDisk");
ManagementObjectCollection queryCollection = searcher.Get();
```

3. You are creating an application that responds to WMI events to process new event log entries. Which of the following do you need to do? (Choose all that apply.)

   **A.** Call the *ManagementEventWatcher.Query* method.

   **B.** Create a *ManagementEventWatcher* object.

   **C.** Create an event handler that accepts *object* and *ManagementBaseObject* parameters.

   **D.** Register the *ManagementEventWatcher.EventArrived* handler.

## Chapter Review

To practice and reinforce the skills you learned in this chapter further, you can

- Review the chapter summary.
- Review the list of key terms introduced in this chapter.
- Complete the case scenarios. These scenarios set up real-world situations involving the topics of this chapter and ask you to create a solution.
- Complete the suggested practices.
- Take a practice test.

## Chapter Summary

- Before you can add events, you must register an event source by calling *EventLog.CreateEventSource*. You can then call *EventLog.WriteEntry* to add events. Read events by creating an instance of the *EventLog* class and accessing the *EventLog .Entries* collection. Use the *Debug* and *Trace* classes to log the internal workings of your application for troubleshooting purposes. *Debug* functions only in Debug releases. *Trace* can function with any release type. Users can configure listeners for *Debug* and *Trace* using the .config files.
- To monitor performance counters programmatically, create an instance of *PerformanceCounter*. To add custom performance counters, call the static *PerformanceCounterCategory.Create* method. To provide performance counter data, create a *PerformanceCounter* object and set the *ReadOnly* property to *false*.
- You can examine processes by calling the *Process.GetProcesses* method. To start a process, call *Process.Start*. To read WMI data, create a *ManagementScope* object and call *ManagementScope.Connect*. Then, create a query using an instance of *ObjectQuery*. You can also respond to WMI events by creating an instance of *WqlEventQuery*. Then use the *WqlEventQuery* to create an instance of *ManagementEventWatcher*. At this point, you can use *ManagementEventWatcher* to either create an event handler or wait for the next event.

## Key Terms

Do you know what these key terms mean? You can check your answers by looking up the terms in the glossary at the end of the book.

- Windows event log
- WMI Query Language (WQL)

## Case Scenarios

In the following case scenarios, you apply what you've learned about how to log application data and manage computer systems. You can find answers to these questions in the "Answers" section at the end of this book.

## Case Scenario 1: Improving the Manageability of an Application

You are an application developer for the Graphic Design Institute. For the last year, you and your team have been managing the first version of an internal application named Orders. You are now identifying requirements for the second version of the application. Your manager asks you to interview key people and then answer questions about your design choices.

### Interviews

The following is a list of company personnel interviewed and their statements.

- **IT Manager** "Orders v1 usually worked great. However, it was difficult to manage. Sometimes, users would complain about poor performance, and we had no way to isolate the source of the problem. Also, it would have been helpful to identify degrading performance proactively so we could take measures to prevent it from being worse. Also, we have a new event management system, and we need user logon and logoff events in the event log that we can collect for security purposes."
- **Development Manager** "Occasionally, IT discovers what they think is a bug in the application. Unfortunately, the only way to isolate the problem is to have them document how to re-create it and then have one of my developers attempt to re-create the problem with a debugger running. It would be much more useful if we could enable a troubleshooting mode in the application to have it create a log file while running on the end-user computer. Then, we could analyze the log file and attempt to isolate the problem."

## Questions

Answer the following questions for your manager:

1. How can you meet the requirements outlined by the IT manager?
2. How can you meet the requirements outlined by the development manager?

## Case Scenario 2: Collecting Information About Computers

You are an application developer working for Trey Research. Recently, an employee took confidential data out of the organization on a USB flash drive. Now, the IT department is requesting custom development to help them assess the storage currently attached to their computers and new storage devices that employees might attach.

The IT manager provides you with the following requests:

- **Storage inventory** Create a tool that IT can distribute to every computer. The tool should generate a list of all disks attached to the computer.
- **Storage change notification** Create an application that runs in the background when users log on. If a user connects a new disk, including a USB flash drive, it should display a warning message that the user should not remove confidential documents from the network. Then it should log the device connection.

### Questions

Answer the following questions for your manager:

1. How can you generate a list of all disks attached to the computer?
2. How can you detect when a USB flash drive is attached to the computer?

## Suggested Practices

To master the "Embedding configuration, diagnostic, management, and installation features into a .NET Framework application" exam objective, complete the following tasks.

### Manage an Event Log by Using the *System.Diagnostics* Namespace

For this task, you should complete at least Practice 1. If you want a better understanding of how events can be used in the real world and you have the resources, complete Practices 2 and 3 as well.

- **Practice 1** Go through your Application event log, or other custom event logs, and examine the events. Notice which events are the most useful for troubleshooting and which characteristics make them useful.
- **Practice 2** Configure event forwarding on computers running Windows Vista to forward events selectively from multiple computers to a single computer. Administrators often use event forwarding to assist in managing events.

- **Practice 3** Using a real-world application that you wrote, create a custom event log in the application's setup. Then add events to the event log when users log on or off or perform other tasks that might be relevant for security auditing.

## Manage System Processes and Monitor the Performance of a .NET Framework Application by Using the Diagnostics Functionality of the .NET Framework

For this task, you should complete at least Practices 1 and 2. If you want a better understanding of how events can be used in the real world and you have the resources, complete Practice 3 as well.

- **Practice 1** Create an application that adds a custom performance counter category with both single-instance and multi-instance counters.
- **Practice 2** Use the Performance snap-in to monitor the performance of a remote computer. Examine the Performance counters added by applications and think about how system administrators might use the counters in a real-world environment.
- **Practice 3** Using a real-world application that you wrote, add code to the setup procedure to establish a custom performance counter category. Then add code to the application to populate several counters revealing internal application metrics.

## Debug and Trace a .NET Framework Application by Using the *System.Diagnostics* Namespace

For this task, you should complete both practices.

- **Practice 1** Using a real-world application that you developed, add debugging and trace commands to allow you to follow the application's execution. Use debugging commands for information that would be useful only in a development environment. Use trace commands when the output might be useful for troubleshooting problems in a real-world environment.
- **Practice 2** Install the application you used in Practice 1. Then update the .config file to write trace output to a text file.

## Embed Management Information and Events into a .NET Framework Application

For this task, you should complete all three practices.

- **Practice 1** Create a program that displays new event log entries to the console.
- **Practice 2** Create a program that displays a dialog box when a user connects a USB flash drive.
- **Practice 3** Create a program that examines all network adapters connected to a computer and identifies the network adapter with the highest bandwidth.

## Take a Practice Test

The practice tests on this book's companion CD offer many options. For example, you can test yourself on just the content covered in this chapter, or you can test yourself on all the 70-536 certification exam content. You can set up the test so that it closely simulates the experience of taking a certification exam, or you can set it up in study mode so that you can look at the correct answers and explanations after you answer each question.

**MORE INFO Practice tests**

For details about all the practice test options available, see the section "How to Use the Practice Tests" in the Introduction of this book.

# Index