



Programming in C#



Exam Ref 70-483

Wouter de Kort

Exam Ref 70-483



Prepare for Microsoft Exam 70-483—and help demonstrate your real-world mastery of programming in C#. Designed for experienced software developers ready to advance their status, *Exam Ref* focuses on the critical-thinking and decision-making acumen needed for success at the Microsoft Specialist level.

Focus on the expertise measured by these objectives:

- Manage Program Flow
- Create and Use Types
- Debug Applications and Implement Security
- Implement Data Access

This Microsoft *Exam Ref*:

- Organizes its coverage by exam objectives.
- Features strategic, what-if scenarios to challenge you.
- Includes a 15% exam discount from Microsoft.
Offer expires 12/31/2017. Details inside.

microsoft.com/mspress

ISBN: 978-0-7356-7682-4

A standard linear barcode representing the ISBN number.



U.S.A. \$39.99

Canada \$41.99

[Recommended]

Certification/Microsoft Visual C#

Programming in C#

About the Exam

Exam 70-483 is one of three Microsoft exams focused on the skills and knowledge necessary to program the essential business/application logic for a variety of application types and hardware/software platforms using C#.

About Microsoft Certification

Passing this exam earns you a Microsoft Specialist certification. You also receive credit toward a **Microsoft Certified Solutions Developer (MCSD)** certification that proves your ability to build innovative solutions across multiple technologies, both on-premises and in the cloud.

Exams 70-483, 70-484, and 70-485 are required for MCSD: Windows Store Apps Using C# certification.

See full details at:
microsoft.com/learning/certification

About the Author

Wouter de Kort, MCSD, is an independent technical coach, trainer, and developer. He has worked with C# and .NET since their inception. His expertise also includes Visual Studio, Team Foundation Server, Entity Framework, Unit Testing, design patterns, ASP.NET, and JavaScript.



Microsoft

Exam Ref 70-483: Programming in C#

Wouter de Kort

Copyright © 2013 by Microsoft Press, Inc.

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

ISBN: 978-0-7356-7682-4

3 4 5 6 7 8 9 10 11 QG 8 7 6 5 4 3

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspininput@microsoft.com. Please tell us what you think of this book at <http://www.microsoft.com/learning/booksurvey>.

Microsoft and the trademarks listed at <http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions and Developmental Editor: Russell Jones

Production Editor: Melanie Yarbrough

Editorial Production: Box Twelve Communications

Technical Reviewer: Auri Rahimzadeh

Copyeditor: Ginny Munroe

Cover Design: Twist Creative • Seattle

Cover Composition: Ellie Volckhausen

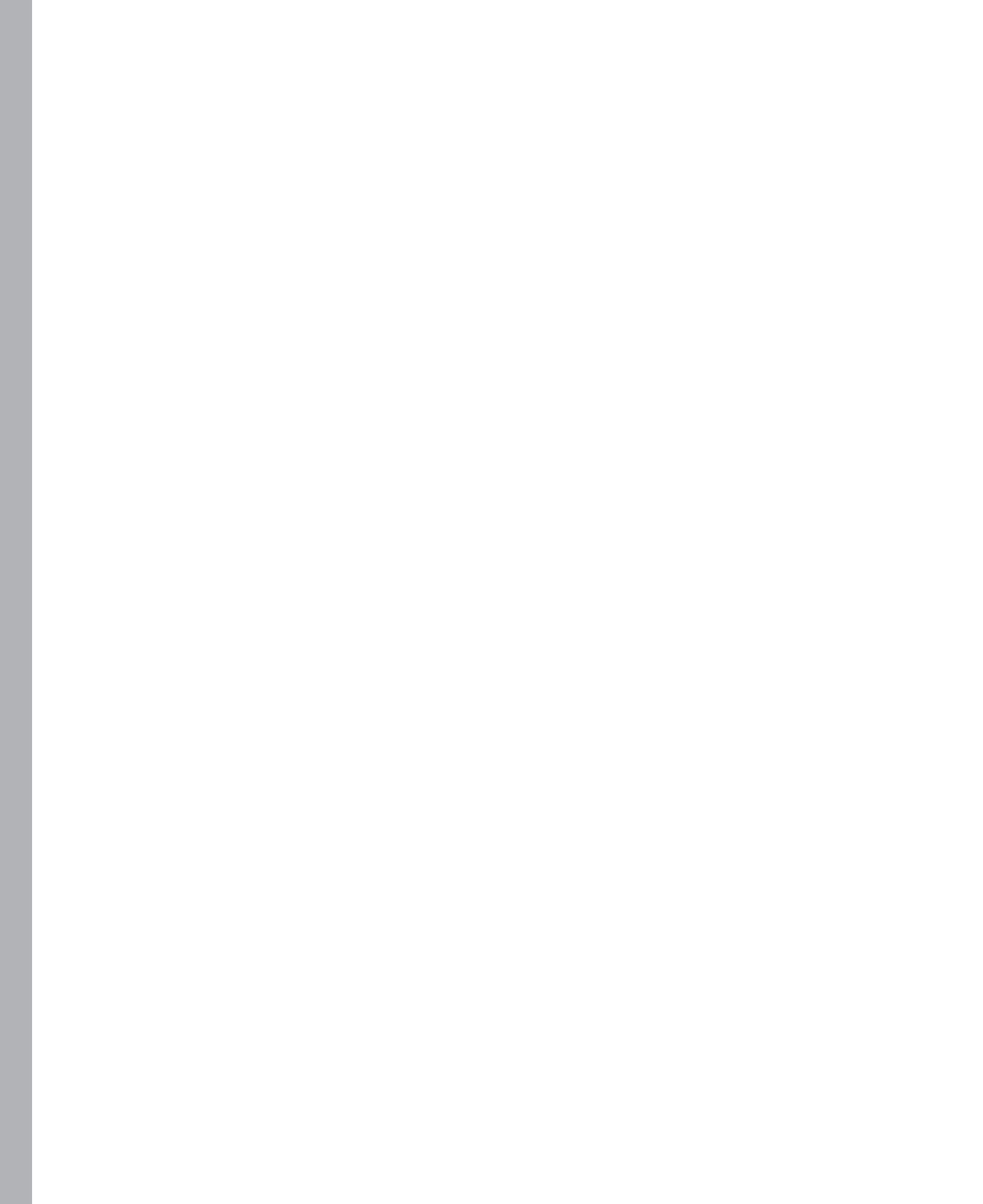
Illustrator: Rebecca Demarest

*Dedicated to my parents who encouraged me to start
programming when I was 7.*

—WOUTER DE KORT

Contents at a glance

<i>Introduction</i>	xv
<i>Preparing for the exam</i>	xxi
CHAPTER 1 Manage program flow	1
CHAPTER 2 Create and use types	89
CHAPTER 3 Debug applications and implement security	179
CHAPTER 4 Implement data access	253
<i>Index</i>	335



Contents

Introduction	xv
<i>Microsoft certifications</i>	xv
<i>Who should read this book</i>	xvi
<i>Organization of this book</i>	xvi
<i>System requirements</i>	xvii
<i>Conventions and features in this book</i>	xvii
<i>Preparing for the exam</i>	xxi
Chapter 1 Manage program flow	1
Objective 1.1: Implement multithreading and asynchronous processing.....	2
Understanding threads	2
Using <i>Tasks</i>	10
Using the <i>Parallel</i> class	16
Using <i>async</i> and <i>await</i>	17
Using Parallel Language Integrated Query (PLINQ)	21
Using concurrent collections	25
Objective summary	30
Objective review	30
Objective 1.2: Manage multithreading.....	31
Synchronizing resources	31
Canceling tasks	37
Objective summary	40
Objective review	40
Objective 1.3: Implement program flow.....	41
Working with Boolean expressions	41
Making decisions	44
Iterating across collections	50

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

Objective summary	55
Objective review	56
Objective 1.4: Create and implement events and callbacks	57
Understanding delegates	57
Using lambda expressions	59
Using events	61
Objective summary	67
Objective review	68
Objective 1.5: Implement exception handling.....	68
Handling exceptions	69
Throwing exceptions	75
Creating custom exceptions	79
Objective summary	81
Objective review	82
Chapter summary	82
Answers.....	83
Objective 1.1: Thought experiment	83
Objective 1.1: Review	83
Objective 1.2: Thought experiment	84
Objective 1.2: Review	84
Objective 1.3: Thought experiment	85
Objective 1.3: Review	86
Objective 1.4: Thought experiment	87
Objective 1.4: Review	87
Objective 1.5: Thought experiment	88
Objective 1.5: Review	88
Chapter 2 Create and use types	89
Objective 2.1: Create types	89
Choosing a type to create	90
Giving your types some body	93
Designing classes	99
Using generic types	101
Extending existing types	103

Objective summary	106
Objective review	106
Objective 2.2: Consume types	107
Boxing and unboxing	107
Converting between different types	108
Using dynamic types	112
Objective summary	114
Objective review	115
Objective 2.3: Enforce encapsulation	116
Using access modifiers	116
Using properties	120
Using explicit interface implementations	121
Objective summary	123
Objective review	124
Objective 2.4: Create and implement a class hierarchy	124
Designing and implementing interfaces	125
Creating and using base classes	128
Implementing standard .NET Framework interfaces	133
Objective summary	138
Objective review	138
Objective 2.5: Find, execute, and create types at runtime by using reflection	139
Creating and using attributes	139
Using reflection	143
Using CodeDom and lambda expressions to generate code	145
Objective summary	149
Objective review	149
Objective 2.6: Manage the object life cycle.....	150
Understanding garbage collection	150
Managing unmanaged resources	151
Objective summary	157
Objective review	158

Objective 2.7: Manipulate strings.....	158
Using strings in the .NET Framework	159
Manipulating strings	160
Searching for strings	162
Enumerating strings	163
Formatting strings	163
Objective summary	167
Objective review	167
Chapter summary	168
Answers.....	169
Objective 2.1: Thought experiment	169
Objective 2.1: Review	169
Objective 2.2: Thought experiment	170
Objective 2.2: Review	170
Objective 2.3: Thought experiment	171
Objective 2.3: Review	172
Objective 2.4: Thought experiment	173
Objective 2.4: Review	173
Objective 2.5: Thought experiment	174
Objective 2.5: Review	174
Objective 2.6: Thought experiment	175
Objective 2.6: Review	176
Objective 2.7: Thought experiment	177
Objective 2.7: Review	177
Chapter 3 Debug applications and implement security	179
Objective 3.1: Validate application input	179
Why validating application input is important	180
Managing data integrity	180
Using <i>Parse</i> , <i>TryParse</i> , and <i>Convert</i>	185
Using regular expressions	188
Validating JSON and XML	189
Objective summary	192
Objective review	192

Objective 3.2 Perform symmetric and asymmetric encryption	193
Using symmetric and asymmetric encryption	194
Working with encryption in the .NET Framework	195
Using hashing	199
Managing and creating certificates	202
Using code access permissions	204
Securing string data	206
Objective summary	208
Objective review	209
Objective 3.3 Manage assemblies	209
What is an assembly?	210
Signing assemblies using a strong name	211
Putting an assembly in the GAC	214
Versioning assemblies	214
Creating a WinMD assembly	217
Objective summary	219
Objective review	219
Objective 3.4 Debug an application	220
Build configurations	220
Creating and managing compiler directives	222
Managing program database files and symbols	226
Objective summary	230
Objective review	230
Objective 3.5 Implement diagnostics in an application.	231
Logging and tracing	231
Profiling your application	238
Creating and monitoring performance counters	241
Objective summary	245
Objective review	245
Chapter summary	246
Answers.	247
Objective 3.1: Thought experiment	247
Objective 3.1: Review	247
Objective 3.2: Thought experiment	248

Objective 3.2: Review	248
Objective 3.3: Thought experiment	249
Objective 3.3: Review	249
Objective 3.4: Thought experiment	250
Objective 3.4: Review	250
Objective 3.5: Thought experiment	251
Objective 3.5: Review	251
Chapter 4 Implement data access	253
Objective 4.1: Perform I/O operations	253
Working with files	254
Working with streams	260
The file system is not just for you	263
Communicating over the network	265
Implementing asynchronous I/O operations	266
Objective summary	269
Objective review	269
Objective 4.2: Consume data	270
Working with a database	271
Using web services	281
Consuming XML	284
Consuming JSON	289
Objective summary	290
Objective review	291
Objective 4.3: Query and manipulate data and objects by using LINQ .	291
Language features that make LINQ possible	292
Using LINQ queries	296
How does LINQ work?	300
Objective summary	305
Objective review	306
Objective 4.4: Serialize and deserialize data	307
Using serialization and deserialization	307
Using <i>XmlSerializer</i>	308
Using binary serialization	311
Using <i>DataContract</i>	314

Using JSON serializer	315
Objective summary	316
Objective review	317
Objective 4.5: Store data in and retrieve data from collections	317
Using arrays	318
Understanding generic versus nongeneric	319
Using <i>List</i>	319
Using <i>Dictionary</i>	321
Using sets	322
Using queues and stacks	323
Choosing a collection	324
Creating a custom collection	324
Objective summary	326
Objective review	326
Chapter summary	327
Answers	328
Objective 4.1: Thought experiment	328
Objective 4.1: Objective review	328
Objective 4.2: Thought experiment	329
Objective 4.2: Objective review	329
Objective 4.3: Thought experiment	330
Objective 4.3: Objective review	331
Objective 4.4: Thought experiment	332
Objective 4.4: Objective review	332
Objective 4.5: Thought experiment	333
Objective 4.5: Objective review	333
<i>Index</i>	335

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

Introduction

The Microsoft 70-483 exam focuses on a broad range of topics that you can use in your work as a C# developer. This book helps you understand both the basic and the more advanced areas of the C# language. It shows you how to use the C# language to create powerful software applications. This book also shows you how to use the new features that were added to the C# language, such as support for asynchronous code. This book is aimed at developers who have some experience with C# but want to deepen their knowledge and make sure they are ready for the exam. To use the examples in this book, you should be familiar with using Visual Studio to create a basic Console Application.

This book covers every exam objective, but it does not cover every exam question. Only the Microsoft exam team has access to the exam questions themselves and Microsoft regularly adds new questions to the exam, making it impossible to cover specific questions. You should consider this book a supplement to your relevant real-world experience and other study materials. If you encounter a topic in this book that you do not feel completely comfortable with, use the links to find more information and take the time to research and study the topic. Great information is available on MSDN, TechNet, in blogs, and in forums.

Microsoft certifications

Microsoft certifications distinguish you by proving your command of a broad set of skills and experience with 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 both on-premise and in the cloud. Certification brings a variety of benefits to the individual and to employers and organizations.

MORE INFO ALL MICROSOFT CERTIFICATIONS

For information about Microsoft certifications, including a full list of available certifications, go to <http://www.microsoft.com/learning/en/us/certification/cert-default.aspx>.

Who should read this book

This book is intended for developers who want to achieve certification for the C# programming language. This book prepares you to take and pass the exam 70-483: Programming in C#. Successfully passing the 70-483 exam also counts as credit toward the Microsoft Certified Solution Developer (MCSD): Windows Store Apps Using C#.

Assumptions

You should have at least one or more years of experience programming the essential business/application logic for a variety of application types and hardware/software platforms using C#. To run the examples from this book you should be able to create a console application in Visual Studio. As you progress with your learning through this book and other study resources, you will become proficient at developing complex applications. You will be able to use all features that C# offers. This book is focused on helping those of you whose goal is to become certified as a C# developer.

You can find information about the audience for Exam 70-483 in the exam preparation guide, available at <http://www.microsoft.com/learning/en/us/exam-70-483.aspx#fbid=x2KPCL1L6z8>.

Organization of this book

This book is divided into four chapters. Each chapter focuses on a different exam domain related to Exam 70-483: Programming in C#. Each chapter is further broken down into specific exam objectives that have been published by Microsoft; they can be found in the "Skills Being Measured" section of the Exam 70-483: Programming in C# website at <http://www.microsoft.com/learning/en/us/exam-70-483.aspx#fbid=x2KPCL1L6z8>.

The material covered by the exam domain and the objectives has been incorporated into the book so that you have concise, objective-by-objective content together with strategic real-world scenarios, thought experiments, and end-of-chapter review questions to provide readers with professional-level preparation for the exam.

System requirements

Where you are unfamiliar with a topic covered in this book, you should practice the concept on your study PC. You will need the following hardware and software to complete the practice exercises in this book:

- One study PC that can be installed with Visual Studio 2012 (see hardware specifications below) or a PC that allows the installation of Visual Studio 2012 within a virtualized environment. You can use Visual Studio 2012 Professional, Premium, or Ultimate if you own a license for one of these. If you don't have access to a licensed version, you can also use Visual Studio 2012 Express for Windows Desktop, which can be downloaded from <http://www.microsoft.com/visualstudio/eng/downloads>.
- Visual Studio 2012 supports the following operating systems: Windows 7 SP1 (x86 and x64), Windows 8 (x86 and x64), Windows Server 2008 R2 SP1 (x64), Windows Server 2012 (x64).
- Visual Studio 2012 requires the following minimum hardware requirements: 1.6 GHz or faster processor, 1 GB of RAM (1.5 GB if running on a virtual machine), 5 GB of available hard disk space, 100 MB of available hard disk space (language pack), 5400 RPM hard disk drive, DirectX 9-capable video card running at 1024 x 768 or higher display resolution.
- If you plan to install Visual Studio 2012 in a virtualized environment, you should consider using Hyper-V and ensure that the minimum hardware requirements are as follows: x64-based processor, which includes both hardware-assisted virtualization (AMD-V or Intel VT) and hardware data execution protection; 4 GB RAM (more is recommended); network card; video card; DVD-ROM drive; and at least 100 GB of available disk space available to allow for the storage of multiple virtual machines.

Conventions and features in this book

This book presents information using conventions designed to make the information readable and easy to follow:

- Each exercise consists of a series of tasks, presented as numbered steps listing each action you must take to complete the exercise.
- Boxed elements with labels such as "Note" provide additional information or alternative methods for completing a step successfully.
- Boxed elements with "Exam Tip" labels provide additional information that might offer helpful hints or additional information on what to expect on the exam.
- Text that you type (apart from code blocks) appear in bold.

- A plus sign (+) between two key names means that you must press those keys at the same time. For example, "Press Alt+Tab" means that you hold down the Alt key while you press the Tab key.
- A vertical bar between two or more menu items (for example, File | Close) means that you should select the first menu or menu item, then the next, and so on.

Acknowledgments

I'd like to thank the following people:

- To Jeff Riley for providing me the opportunity to write this book.
- To Ginny Munroe for helping me through the whole editing process. I learned a lot from your feedback and advice.
- To Auri Rahimzadeh for your technical reviewing skills.
- To my wife, Elise, for your support.

And to all the other people who played a role in getting this book ready. Thanks for your hard work!

Errata & book support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site:

<http://aka.ms/ER70-483/errata>

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at mspinput@microsoft.com.

Please note that product support for Microsoft software is not offered through the addresses above.

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

<http://www.microsoft.com/learning/booksurvey>

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: *<http://twitter.com/MicrosoftPress>*.

Preparing for the exam

Microsoft certification exams are a great way to build your resume and let the world know about your level of expertise. Certification exams validate your on-the-job experience and product knowledge. While there is no substitution for on-the-job experience, preparation through study and hands-on practice can help you prepare for the exam. We recommend that you round out your exam preparation plan by using a combination of available study materials and courses. For example, you might use this *Exam Ref* and another study guide for your "at home" preparation, and take a Microsoft Official Curriculum course for the classroom experience. Choose the combination that you think works best for you.

Note that this *Exam Ref* is based on publically available information about the exam and the author's experience. To safeguard the integrity of the exam, authors do not have access to the live exam.

CHAPTER 1

Manage program flow

If you could build only programs that execute all their logic from top to bottom, it would not be feasible to build complex applications. Fortunately, C# and the .NET Framework offer you a lot of options for creating complex programs that don't have a fixed program flow.

This chapter starts with looking at how to create *multithreaded* applications. Those applications can scale well and remain responsive to the user while doing their work. You will also look at the new language feature *async/await* that was added to C# 5.

You will learn about the basic C# language constructs to make decisions and execute a piece of code multiple times, depending on the circumstances. These constructs form the basic language blocks of each application, and you will use them often.

After that, you will learn how to create applications that are loosely coupled by using *delegates* and *events*. With events, you can build objects that can notify each other when something happens and that can respond to those notifications. Frameworks such as ASP.NET, Windows Presentation Foundation (WPF), and WinForms make heavy use of events; understanding events thoroughly will help you build great applications.

Unfortunately, your program flow can also be interrupted by errors. Such errors can happen in areas that are out of your control but that you need to respond to. Sometimes you want to raise such an error yourself. You will learn how exceptions can help you implement a robust error-handling strategy in your applications.

Objectives in this chapter:

- Objective 1.1: Implement multithreading and asynchronous processing
- Objective 1.2: Manage multithreading
- Objective 1.3: Implement program flow
- Objective 1.4: Create and implement events and callbacks
- Objective 1.5: Implement exception handling

IMPORTANT

Have you read page xxi?

It contains valuable information regarding the skills you need to pass the exam.

Objective 1.1: Implement multithreading and asynchronous processing

Applications are becoming more and more complex as user expectations rise. To fully take advantage of multicore systems and stay responsive, you need to create applications that use multiple threads, often called *parallelism*.

The .NET Framework and the C# language offer a lot of options that you can use to create multithreaded applications.

This objective covers how to:

- Understand threads.
- Use the Task Parallel Library.
- Use the *Parallel* class.
- Use the new *async* and *await* keywords.
- Use Parallel Language Integrated Query.
- Use concurrent collections.

Understanding threads

Imagine that your computer has only one *central processing unit* (CPU) that is capable of executing only one operation at a time. Now, imagine what would happen if the CPU has to work hard to execute a task that takes a long time.

While this operation runs, all other operations would be paused. This means that the whole machine would freeze and appear unresponsive to the user. Things get even worse when that long-running operation contains a bug so it never ends. Because the rest of the machine is unusable, the only thing you can do is restart the machine.

To remedy this problem, the concept of a *thread* is used. In current versions of Windows, each application runs in its own *process*. A process isolates an application from other applications by giving it its own *virtual memory* and by ensuring that different processes can't influence each other. Each process runs in its own *thread*. A *thread* is something like a virtualized CPU. If an application crashes or hits an infinite loop, only the application's process is affected.

Windows must manage all of the threads to ensure they can do their work. These management tasks do come with an overhead. Each thread is allowed by Windows to execute for a

certain time period. After this period ends, the thread is paused and Windows switches to another thread. This is called *context switching*.

In practice, this means that Windows has to do some work to make it happen. The current thread is using a certain area of memory; it uses CPU registers and other state data, and Windows has to make sure that the whole context of the thread is saved and restored on each switch.

But although there are certain performance hits, using threads does ensure that each process gets its time to execute without having to wait until all other operations finish. This improves the responsiveness of the system and gives the illusion that one CPU can execute multiple tasks at a time. This way you can create an application that uses *parallelism*, meaning that it can execute multiple threads on different CPUs in parallel.

Almost any device that you buy today has a CPU with multiple cores, which is similar to having multiple CPUs. Some servers not only have multicore CPUs but they also have more than one CPU. To make use of all these cores, you need multiple threads. Windows ensures that those threads are distributed over your available cores. This way you can perform multiple tasks at once and improve scalability.

Because of the associated overhead, you should carefully determine whether you need multithreading. But if you want to use threads for scalability or responsiveness, C# and .NET Framework offer you a lot of possibilities.

Using the *Thread* class

The *Thread* class can be found in the *System.Threading* namespace. This class enables you to create new threads, manage their priority, and get their status.

The *Thread* class isn't something that you should use in your applications, except when you have special needs. However, when using the *Thread* class you have control over all configuration options. You can, for example, specify the priority of your thread, tell Windows that your thread is long running, or configure other advanced options.

Listing 1-1 shows an example of using the *Thread* class to run a method on another thread. The *Console* class synchronizes the use of the output stream for you so you can write to it from multiple threads. *Synchronization* is the mechanism of ensuring that two threads don't execute a specific portion of your program at the same time. In the case of a console application, this means that no two threads can write data to the screen at the exact same time. If one thread is working with the output stream, other threads will have to wait before it's finished.

LISTING 1-1 Creating a thread with the *Thread* class

```
using System;
using System.Threading;

namespace Chapter1
{
    public static class Program
    {
        public static void ThreadMethod()
        {
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine("ThreadProc: {0}", i);
                Thread.Sleep(0);
            }
        }

        public static void Main()
        {
            Thread t = new Thread(new ThreadStart(ThreadMethod));
            t.Start();

            for (int i = 0; i < 4; i++)
            {
                Console.WriteLine("Main thread: Do some work.");
                Thread.Sleep(0);
            }

            t.Join();
        }
    }
}

// Displays
//Main thread: Do some work.
//ThreadProc: 0
//Main thread: Do some work.
//ThreadProc: 1
//Main thread: Do some work.
//ThreadProc: 2
//Main thread: Do some work.
//ThreadProc: 3
//ThreadProc: 4
//ThreadProc: 5
//ThreadProc: 6
//ThreadProc: 7
//ThreadProc: 8
//ThreadProc: 9
```

As you can see, both threads run and print their message to the console. The *Thread.Join* method is called on the main thread to let it wait until the other thread finishes.

Why the *Thread.Sleep(0)*? It is used to signal to Windows that this thread is finished. Instead of waiting for the whole time-slice of the thread to finish, it will immediately switch to another thread.

Both your process and your thread have a *priority*. Assigning a low priority is useful for applications such as a screen saver. Such an application shouldn't compete with other applications for CPU time. A higher-priority thread should be used only when it's absolutely necessary. A new thread is assigned a priority of Normal, which is okay for almost all scenarios.

Another thing that's important to know about threads is the difference between *foreground* and *background* threads. Foreground threads can be used to keep an application alive. Only when all foreground threads end does the common language runtime (CLR) shut down your application. Background threads are then terminated.

Listing 1-2 shows this difference in action.

LISTING 1-2 Using a background thread

```
using System;
using System.Threading;

namespace Chapter1
{
    public static class Program

    {
        public static void ThreadMethod()
        {
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine("ThreadProc: {0}", i);
                Thread.Sleep(1000);
            }
        }

        public static void Main()
        {
            Thread t = new Thread(new ThreadStart(ThreadMethod));
            t.IsBackground = true;
            t.Start();
        }
    }
}
```

If you run this application with the *IsBackground* property set to *true*, the application exits immediately. If you set it to *false* (creating a foreground thread), the application prints the *ThreadProc* message ten times.

The *Thread* constructor has another overload that takes an instance of a *ParameterizedThreadStart* delegate. This overload can be used if you want to pass some data through the start method of your thread to your worker method, as Listing 1-3 shows.

LISTING 1-3 Using the *ParameterizedThreadStart*

```
public static void ThreadMethod(object o)
{
    for (int i = 0; i < (int)o; i++)
    {
        Console.WriteLine("ThreadProc: {0}", i);

        Thread.Sleep(0);
    }
}

public static void Main()
{
    Thread t = new Thread(new ParameterizedThreadStart(ThreadMethod));
    t.Start(5);
    t.Join();
}
```

In this case, the value 5 is passed to the *ThreadMethod* as an object. You can cast it to the expected type to use it in your method.

To stop a thread, you can use the *Thread.Abort* method. However, because this method is executed by another thread, it can happen at any time. When it happens, a *ThreadAbortException* is thrown on the target thread. This can potentially leave a corrupt state and make your application unusable.

A better way to stop a thread is by using a shared variable that both your target and your calling thread can access. Listing 1-4 shows an example.

LISTING 1-4 Stopping a thread

```
using System;
using System.Threading;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()
        {

            bool stopped = false;
```

```

        Thread t = new Thread(new ThreadStart(() =>
    {
        while (!stopped)
        {
            Console.WriteLine("Running...");
            Thread.Sleep(1000);
        }
    }));
    t.Start();
    Console.WriteLine("Press any key to exit");
    Console.ReadKey();

    stopped = true;
    t.Join();
}
}
}

```

In this case, the thread is initialized with a lambda expression (which in turn is just a short-hand version of a delegate). The thread keeps running until *stopped* becomes *true*. After that, the *t.Join* method causes the console application to wait till the thread finishes execution.

A thread has its own call stack that stores all the methods that are executed. Local variables are stored on the call stack and are private to the thread.

A thread can also have its own data that's not a local variable. By marking a field with the *ThreadStatic* attribute, each thread gets its own copy of a field (see Listing 1-5).

LISTING 1-5 Using the *ThreadStaticAttribute*

```

using System;
using System.Threading;

namespace Chapter1
{
    public static class Program
    {
        [ThreadStatic]
        public static int _field;
        public static void Main()
        {

            new Thread(() =>
            {
                for(int x = 0; x < 10; x++)
                {
                    _field++;
                    Console.WriteLine("Thread A: {0}", _field);
                }
            }).Start();

            new Thread(() =>
            {

```

```

        for(int x = 0; x < 10; x++)
    {
        _field++;
        Console.WriteLine("Thread B: {0}", _field);
    }
}).Start();

Console.ReadKey();
}
}
}

```

With the *ThreadStaticAttribute* applied, the maximum value of *_field* becomes 10. If you remove it, you can see that both threads access the same value and it becomes 20.

If you want to use local data in a thread and initialize it for each thread, you can use the *ThreadLocal<T>* class. This class takes a delegate to a method that initializes the value. Listing 1-6 shows an example.

LISTING 1-6 Using *ThreadLocal<T>*

```

using System;
using System.Threading;

namespace Chapter1
{
    public static class Program
    {
        public static ThreadLocal<int> _field =
            new ThreadLocal<int>(() =>
        {
            return Thread.CurrentThread.ManagedThreadId;
        });

        public static void Main()
        {
            new Thread(() =>

            {

                for(int x = 0; x < _field.Value; x++)
                {
                    Console.WriteLine("Thread A: {0}", x);
                }

            }).Start();
            new Thread(() =>
            {
                for (int x = 0; x < _field.Value; x++)
                {
                    Console.WriteLine("Thread B: {0}", x);
                }

            });
        }
    }
}

```

```

        }).Start();

        Console.ReadKey();
    }
}

// Displays
// Thread B: 0
// Thread B: 1
// Thread B: 2
// Thread B: 3
// Thread A: 0
// Thread A: 1
// Thread A: 2

```

Here you see another feature of the .NET Framework. You can use the *Thread.CurrentThread* class to ask for information about the thread that's executing. This is called the thread's *execution context*. This property gives you access to properties like the thread's *current culture* (a *CultureInfo* associated with the current thread that is used to format dates, times, numbers, currency values, the sorting order of text, casing conventions, and string comparisons), *principal* (representing the current security context), *priority* (a value to indicate how the thread should be scheduled by the operating system), and other info.

When a thread is created, the runtime ensures that the initiating thread's execution context is flowed to the new thread. This way the new thread has the same privileges as the parent thread.

This copying of data does cost some resources, however. If you don't need this data, you can disable this behavior by using the *ExecutionContext.SuppressFlow* method.

Thread pools

When working directly with the *Thread* class, you create a new thread each time, and the thread dies when you're finished with it. The creation of a thread, however, is something that costs some time and resources.

A *thread pool* is created to reuse those threads, similar to the way a database connection pooling works. Instead of letting a thread die, you send it back to the pool where it can be reused whenever a request comes in.

When you work with a thread pool from .NET, you queue a work item that is then picked up by an available thread from the pool. Listing 1-7 shows how this is done.

LISTING 1-7 Queuing some work to the thread pool

```

using System;
using System.Threading;

namespace Chapter1
{
    public static class Program
    {

```

```

public static void Main()
{
    ThreadPool.QueueUserWorkItem((s) =>
    {
        Console.WriteLine("Working on a thread from threadpool");
    });

    Console.ReadLine();
}
}

```

Because the thread pool limits the available number of threads, you do get a lesser degree of parallelism than using the regular *Thread* class. But the thread pool also has many advantages.

Take, for example, a web server that serves incoming requests. All those requests come in at an unknown time and frequency. The thread pool ensures that each request gets added to the queue and that when a thread becomes available, it is processed. This ensures that your server doesn't crash under the amount of requests. If you spawn threads manually, you can easily bring down your server if you get a lot of requests. Each request has unique characteristics in the work they need to do. What the thread pool does is map this work onto the threads available in the system. Of course, you can still get so many requests that you run out of threads. Requests then start to queue up and this leads to your web server becoming unresponsive.

The thread pool automatically manages the amount of threads it needs to keep around. When it is first created, it starts out empty. As a request comes in, it creates additional threads to handle those requests. As long as it can finish an operation before a new one comes in, no new threads have to be created. If new threads are no longer in use after some time, the thread pool can kill those threads so they no longer use any resources.

MORE INFO THREAD POOL

For more information on how the thread pool works and how you can configure it, see <http://msdn.microsoft.com/en-us/library/system.threading.threadpool.aspx>.

One thing to be aware of is that because threads are being reused, they also reuse their local state. You may not rely on state that can potentially be shared between multiple operations.

Using Tasks

Queuing a work item to a thread pool can be useful, but it has its shortcomings. There is no built-in way to know when the operation has finished and what the return value is.

This is why the .NET Framework introduces the concept of a *Task*, which is an object that represents some work that should be done. The *Task* can tell you if the work is completed and if the operation returns a result, the *Task* gives you the result.

A *task scheduler* is responsible for starting the *Task* and managing it. By default, the *Task* scheduler uses threads from the thread pool to execute the *Task*.

Tasks can be used to make your application more responsive. If the thread that manages the user interface offloads work to another thread from the thread pool, it can keep processing user events and ensure that the application can still be used. But it doesn't help with scalability. If a thread receives a web request and it would start a new *Task*, it would just consume another thread from the thread pool while the original thread waits for results.

Executing a *Task* on another thread makes sense only if you want to keep the user interface thread free for other work or if you want to parallelize your work on to multiple processors.

Listing 1-8 shows how to start a new *Task* and wait until it's finished.

LISTING 1-8 Starting a new *Task*

```
using System;
using System.Threading.Tasks;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()
        {
            Task t = Task.Run(() =>
            {
                for (int x = 0; x < 100; x++)
                {
                    Console.Write('*');
                }
            });
            t.Wait();
        }
    }
}
```

This example creates a new *Task* and immediately starts it. Calling *Wait* is equivalent to calling *Join* on a thread. It waits till the *Task* is finished before exiting the application.

Next to *Task*, the .NET Framework also has the *Task<T>* class that you can use if a *Task* should return a value. Listing 1-9 shows how this works.

LISTING 1-9 Using a *Task* that returns a value.

```
using System;
using System.Threading.Tasks;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()
        {
            Task<int> t = Task.Run(() =>
            {
                return 42;
            });
            Console.WriteLine(t.Result); // Displays 42
        }
    }
}
```

Attempting to read the *Result* property on a *Task* will force the thread that's trying to read the result to wait until the *Task* is finished before continuing. As long as the *Task* has not finished, it is impossible to give the result. If the *Task* is not finished, this call will block the current thread.

Because of the object-oriented nature of the *Task* object, one thing you can do is add a *continuation task*. This means that you want another operation to execute as soon as the *Task* finishes.

Listing 1-10 shows an example of creating such a continuation.

LISTING 1-10 Adding a continuation

```
Task<int> t = Task.Run(() =>
{
    return 42;
}).ContinueWith((i) =>
{
    return i.Result * 2;
});

Console.WriteLine(t.Result); // Displays 84
```

The *ContinueWith* method has a couple of overloads that you can use to configure when the continuation will run. This way you can add different continuation methods that will run when an exception happens, the *Task* is canceled, or the *Task* completes successfully. Listing 1-11 shows how to do this.

LISTING 1-11 Scheduling different continuation tasks

```
Task<int> t = Task.Run(() =>
{
    return 42;
});

t.ContinueWith((i) =>
{
    Console.WriteLine("Canceled");
}, TaskContinuationOptions.OnlyOnCanceled);

t.ContinueWith((i) =>
{
    Console.WriteLine("Faulted");
}, TaskContinuationOptions.OnlyOnFaulted);

var completedTask = t.ContinueWith((i) =>
{
    Console.WriteLine("Completed");
}, TaskContinuationOptions.OnlyOnRanToCompletion);

completedTask.Wait();
```

Next to continuation *Tasks*, a *Task* can also have several *child Tasks*. The *parent Task* finishes when all the child tasks are ready. Listing 1-12 shows how this works.

LISTING 1-12 Attaching child tasks to a parent task

```
using System;
using System.Threading.Tasks;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()
        {
            Task<Int32[]> parent = Task.Run(() =>
            {
                var results = new Int32[3];
                new Task(() => results[0] = 0,
                    TaskCreationOptions.AttachedToParent).Start();
                new Task(() => results[1] = 1,
                    TaskCreationOptions.AttachedToParent).Start();
                new Task(() => results[2] = 2,
                    TaskCreationOptions.AttachedToParent).Start();

                return results;
            });

            var finalTask = parent.ContinueWith(
                parentTask => {
                    foreach(int i in parentTask.Result)
                        Console.WriteLine(i);
                });
        }
    }
}
```

```

                finalTask.Wait();
            }
        }
    }
}

```

The `finalTask` runs only after the parent `Task` is finished, and the parent `Task` finishes when all three children are finished. You can use this to create quite complex `Task` hierarchies that will go through all the steps you specified.

In the previous example, you had to create three `Tasks` all with the same options. To make the process easier, you can use a `TaskFactory`. A `TaskFactory` is created with a certain configuration and can then be used to create `Tasks` with that configuration. Listing 1-13 shows how you can simplify the previous example with a factory.

LISTING 1-13 Using a `TaskFactory`

```

using System;
using System.Threading.Tasks;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()
        {
            Task<Int32[]> parent = Task.Run(() =>
            {
                var results = new Int32[3];

                TaskFactory tf = new TaskFactory(TaskCreationOptions.AttachedToParent,
                    TaskContinuationOptions.ExecuteSynchronously);

                tf.StartNew(() => results[0] = 0);
                tf.StartNew(() => results[1] = 1);
                tf.StartNew(() => results[2] = 2);
                return results;
            });

            var finalTask = parent.ContinueWith(
                parentTask => {
                    foreach(int i in parentTask.Result)
                        Console.WriteLine(i);
                });
            finalTask.Wait();
        }
    }
}

```

Next to calling `Wait` on a single `Task`, you can also use the method `WaitAll` to wait for multiple `Tasks` to finish before continuing execution. Listing 1-14 shows how to use this.

LISTING 1-14 Using *Task.WaitAll*

```
using System.Threading;
using System.Threading.Tasks;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()
        {
            Task[] tasks = new Task[3];

            tasks[0] = Task.Run(() => {
                Thread.Sleep(1000);
                Console.WriteLine("1");
                return 1;
            });
            tasks[1] = Task.Run(() => {
                Thread.Sleep(1000);
                Console.WriteLine("2");
                return 2;
            });
            tasks[2] = Task.Run(() => {
                Thread.Sleep(1000);
                Console.WriteLine("3");
                return 3;
            });

            Task.WaitAll(tasks);
        }
    }
}
```

In this case, all three *Tasks* are executed simultaneously, and the whole run takes approximately 1000ms instead of 3000. Next to *WaitAll*, you also have a *WhenAll* method that you can use to schedule a continuation method after all *Tasks* have finished.

Instead of waiting until all tasks are finished, you can also wait until one of the tasks is finished. You use the *WaitAny* method for this. Listing 1-15 shows how this works.

LISTING 1-15 Using *Task.WaitAny*

```
using System;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()
        {
            Task<int>[] tasks = new Task<int>[3];
```

```

tasks[0] = Task.Run(() => { Thread.Sleep(2000); return 1; });
tasks[1] = Task.Run(() => { Thread.Sleep(1000); return 2; });
tasks[2] = Task.Run(() => { Thread.Sleep(3000); return 3; });

while (tasks.Length > 0)
{
    int i = Task.WaitAny(tasks);
    Task<int> completedTask = tasks[i];

    Console.WriteLine(completedTask.Result);

    var temp = tasks.ToList();
    temp.RemoveAt(i);
    tasks = temp.ToArray();

}
}
}
}

```

In this example, you process a completed *Task* as soon as it finishes. By keeping track of which *Tasks* are finished, you don't have to wait until all *Tasks* have completed.

Using the *Parallel* class

The *System.Threading.Tasks* namespace also contains another class that can be used for parallel processing. The *Parallel* class has a couple of static methods—*For*, *ForEach*, and *Invoke*—that you can use to parallelize work.

Parallelism involves taking a certain task and splitting it into a set of related tasks that can be executed concurrently. This also means that you shouldn't go through your code to replace all your loops with parallel loops. You should use the *Parallel* class only when your code doesn't have to be executed sequentially.

Increasing performance with parallel processing happens only when you have a lot of work to be done that can be executed in parallel. For smaller work sets or for work that has to synchronize access to resources, using the *Parallel* class can hurt performance.

The best way to know whether it will work in your situation is to measure the results.

Listing 1-16 shows an example of using *Parallel.For* and *Parallel.ForEach*.

LISTING 1-16 Using *Parallel.For* and *Parallel.ForEach*

```

Parallel.For(0, 10, i =>
{
    Thread.Sleep(1000);
});

var numbers = Enumerable.Range(0, 10);
Parallel.ForEach(numbers, i =>
{
    Thread.Sleep(1000);
});

```

You can cancel the loop by using the *ParallelLoopState* object. You have two options to do this: *Break* or *Stop*. *Break* ensures that all iterations that are currently running will be finished. *Stop* just terminates everything. Listing 1-17 shows an example.

LISTING 1-17 Using *Parallel.Break*

```
ParallelLoopResult result = Parallel.  
    For(0, 1000, (int i, ParallelLoopState loopState) =>  
{  
    if (i == 500)  
    {  
        Console.WriteLine("Breaking loop");  
        loopState.Break();  
    }  
    return;  
});
```

When breaking the parallel loop, the result variable has an *IsCompleted* value of *false* and a *LowestBreakIteration* of 500. When you use the *Stop* method, the *LowestBreakIteration* is *null*.

Using *async* and *await*

As you have seen, long-running CPU-bound tasks can be handed to another thread by using the *Task* object. But when doing work that's input/output (I/O)-bound, things go a little differently.

When your application is executing an I/O operation on the primary application thread, Windows notices that your thread is waiting for the I/O operation to complete. Maybe you are accessing some file on disk or over the network, and this could take some time.

Because of this, Windows pauses your thread so that it doesn't use any CPU resources. But while doing this, it still uses memory, and the thread can't be used to serve other requests, which in turn will lead to new threads being created if requests come in.

Asynchronous code solves this problem. Instead of blocking your thread until the I/O operation finishes, you get back a *Task* object that represents the result of the asynchronous operation. By setting a continuation on this *Task*, you can continue when the I/O is done. In the meantime, your thread is available for other work. When the I/O operation finishes, Windows notifies the runtime and the continuation *Task* is scheduled on the thread pool.

But writing asynchronous code is not easy. You have to make sure that all edge cases are handled and that nothing can go wrong. Because of this predicament, C# 5 has added two new keywords to simplify writing asynchronous code. Those keywords are *async* and *await*.

You use the *async* keyword to mark a method for asynchronous operations. This way, you signal to the compiler that something asynchronous is going to happen. The compiler responds to this by transforming your code into a *state machine*.

A method marked with *async* just starts running synchronously on the current thread. What it does is enable the method to be split into multiple pieces. The boundaries of these pieces are marked with the *await* keyword.

When you use the *await* keyword, the compiler generates code that will see whether your asynchronous operation is already finished. If it is, your method just continues running synchronously. If it's not yet completed, the state machine will hook up a continuation method that should run when the *Task* completes. Your method yields control to the calling thread, and this thread can be used to do other work.

Listing 1-18 shows a simple example of an asynchronous method.

LISTING 1-18 *async* and *await*

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

namespace Chapter1.Threads
{
    public static class Program
    {
        public static void Main()
        {
            string result = DownloadContent().Result;
            Console.WriteLine(result);
        }

        public static async Task<string> DownloadContent()
        {
            using(HttpClient client = new HttpClient())
            {

                string result = await client.GetStringAsync("http://www.microsoft.com");
                return result;
            }
        }
    }
}
```

Because the entry method of an application can't be marked as *async*, the example accesses the *Result* property in the *Main* method which blocks the code until the *async* method *DownloadContent* is finished. This class uses both the *async* and *await* keywords in the *DownloadContent* method.

The *GetStringAsync* uses asynchronous code internally and returns a *Task<string>* to the caller that will finish when the data is retrieved. In the meantime, your thread can do other work.

The nice thing about *async* and *await* is that they let the compiler do the thing it's best at: generate code in precise steps. Writing correct asynchronous code by hand is difficult, especially when trying to implement exception handling. Doing this correctly can become difficult quickly. Adding continuation tasks also breaks the logical flow of the code. Your code doesn't read top to bottom anymore. Instead, program flow jumps around, and it's harder to follow.

when debugging your code. The *await* keyword enables you to write code that looks synchronous but behaves in an asynchronous way. The Visual Studio debugger is even clever enough to help you in debugging asynchronous code as if it were synchronous.

So doing a CPU-bound task is different from an I/O-bound task. CPU-bound tasks always use some thread to execute their work. An asynchronous I/O-bound task doesn't use a thread until the I/O is finished.

If you are building a client application that needs to stay responsive while background operations are running, you can use the *await* keyword to offload a long-running operation to another thread. Although this does not improve performance, it does improve responsiveness. The *await* keyword also makes sure that the remainder of your method runs on the correct user interface thread so you can update the user interface.

Making a scalable application that uses fewer threads is another story. Making code scale better is about changing the actual implementation of the code. Listing 1-19 shows an example of this.

LISTING 1-19 Scalability versus responsiveness

```
public Task SleepAsyncA(int millisecondsTimeout)
{
    return Task.Run(() => Thread.Sleep(millisecondsTimeout));
}

public Task SleepAsyncB(int millisecondsTimeout)
{
    TaskCompletionSource<bool> tcs = null;
    var t = new Timer(delegate { tcs.TrySetResult(true); }, null, -1, -1);
    tcs = new TaskCompletionSource<bool>(t);
    t.Change(millisecondsTimeout, -1);
    return tcs.Task;
}
```

The *SleepAsyncA* method uses a thread from the thread pool while sleeping. The second method, however, which has a completely different implementation, does not occupy a thread while waiting for the timer to run. The second method gives you scalability.

When using the *async* and *await* keywords, you should keep this in mind. Just wrapping each and every operation in a task and awaiting them won't make your application perform any better. It could, however, improve responsiveness, which is very important in client applications.

The *FileStream* class, for example, exposes asynchronous methods such as *WriteAsync* and *ReadAsync*. They use an implementation that makes use of actual asynchronous I/O. This way, they don't use a thread while they are waiting on the hard drive of your system to read or write some data.

When an exception happens in an asynchronous method, you normally expect an *AggregateException*. However, the generated code helps you unwrap the *AggregateException* and throws the first of its inner exceptions. This makes the code more intuitive to use and easier to debug.

One other thing that's important when working with asynchronous code is the concept of a *SynchronizationContext*, which connects its application model to its threading model. For example, a WPF application uses a single user interface thread and potentially multiple background threads to improve responsiveness and distribute work across multiple CPUs. An ASP.NET application, however, uses threads from the thread pool that are initialized with the correct data, such as current user and culture to serve incoming requests.

The *SynchronizationContext* abstracts the way these different applications work and makes sure that you end up on the right thread when you need to update something on the UI or process a web request.

The *await* keyword makes sure that the current *SynchronizationContext* is saved and restored when the task finishes. When using *await* inside a WPF application, this means that after your *Task* finishes, your program continues running on the user interface thread. In an ASP.NET application, the remaining code runs on a thread that has the client's cultural, principal, and other information set.

If you want, you can disable the flow of the *SynchronizationContext*. Maybe your continuation code can run on any thread because it doesn't need to update the UI after it's finished. By disabling the *SynchronizationContext*, your code performs better. Listing 1-20 shows an example of a button event handler in a WPF application that downloads a website and then puts the result in a label.

LISTING 1-20 Using *ConfigureAwait*

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    HttpClient httpClient = new HttpClient();

    string content = await httpClient
        .GetStringAsync("http://www.microsoft.com")
        .ConfigureAwait(false);

    Output.Content = content;
}
```

This example throws an exception; the *Output.Content* line is not executed on the UI thread because of the *ConfigureAwait(false)*. If you do something else, such as writing the content to file, you don't need to set the *SynchronizationContext* to be set (see Listing 1-21).

LISTING 1-21 Continuing on a thread pool instead of the UI thread

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    HttpClient httpClient = new HttpClient();

    string content = await httpClient
        .GetStringAsync("http://www.microsoft.com")
        .ConfigureAwait(false);
```

```

        using (FileStream sourceStream = new FileStream("temp.html",
            FileMode.Create, FileAccess.Write, FileShare.None,
            4096, useAsync: true))
    {
        byte[] encodedText = Encoding.Unicode.GetBytes(content);
        await sourceStream.WriteAsync(encodedText, 0, encodedText.Length)
            .ConfigureAwait(false);
    };
}

```

Both *awaits* use the *ConfigureAwait(false)* method because if the first method is already finished before the *awaiter* checks, the code still runs on the UI thread.

When creating *async* methods, it's important to choose a return type of *Task* or *Task<T>*. Avoid the *void* return type. A *void* returning *async* method is effectively a fire-and-forget method. You can never inspect the *return* type, and you can't see whether any exceptions were thrown. You should use *async void* methods only when dealing with asynchronous events.

The use of the new *async/await* keywords makes it much easier to write asynchronous code. In today's world with multiple cores and requirements for responsiveness and scalability, it's important to look for opportunities to use these new keywords to improve your applications.



EXAM TIP

When using *async* and *await* keep in mind that you should never have a method marked *async* without any *await* statements. You should also avoid returning *void* from an *async* method except when it's an event handler.

Using Parallel Language Integrated Query (PLINQ)

Language-Integrated Query (LINQ) is a popular addition to the C# language. You can use it to perform queries over all kinds of data.

Parallel Language-Integrated Query (PLINQ) can be used on objects to potentially turn a sequential query into a parallel one.

Extension methods for using PLINQ are defined in the *System.Linq.ParallelEnumerable* class. Parallel versions of LINQ operators, such as *Where*, *Select*, *SelectMany*, *GroupBy*, *Join*, *OrderBy*, *Skip*, and *Take*, can be used.

Listing 1-22 shows how you can convert a query to a parallel query.

LISTING 1-22 Using *AsParallel*

```
var numbers = Enumerable.Range(0, 100000000);
var parallelResult = numbers.AsParallel()
    .Where(i => i % 2 == 0)
    .ToArray();
```

The runtime determines whether it makes sense to turn your query into a parallel one. When doing this, it generates *Task* objects and starts executing them. If you want to force PLINQ into a parallel query, you can use the *WithExecutionMode* method and specify that it should always execute the query in parallel.

You can also limit the amount of parallelism that is used with the *WithDegreeOfParallelism* method. You pass that method an integer that represents the number of processors that you want to use. Normally, PLINQ uses all processors (up to 64), but you can limit it with this method if you want.

One thing to keep in mind is that parallel processing does not guarantee any particular order. Listing 1-23 shows what can happen.

LISTING 1-23 Unordered parallel query

```
using System;
using System.Linq;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()
        {
            var numbers = Enumerable.Range(0, 10);
            var parallelResult = numbers.AsParallel()
                .Where(i => i % 2 == 0)
                .ToArray();

            foreach (int i in parallelResult)
                Console.WriteLine(i);
        }
    }
}

// Displays
// 2
// 0
// 4
// 6
// 8
```

As you can see, the returned results from this query are in no particular order. The results of this code vary depending on the amount of CPUs that are available. If you want to ensure that the results are ordered, you can add the *AsOrdered* operator. Your query is still processed in parallel, but the results are buffered and sorted. Listing 1-24 shows how this works.

LISTING 1-24 Ordered parallel query

```
using System;
using System.Linq;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()
        {
            var numbers = Enumerable.Range(0, 10);
            var parallelResult = numbers.AsParallel().AsOrdered()
                .Where(i => i % 2 == 0)
                .ToArray();

            foreach (int i in parallelResult)
                Console.WriteLine(i);
        }
    }

// Displays
// 0
// 2
// 4
// 6
// 8
```

If you have a complex query that can benefit from parallel processing but also has some parts that should be done sequentially, you can use the *AsSequential* to stop your query from being processed in parallel.

One scenario where this is required is to preserve the ordering of your query. Listing 1-25 shows how you can use the *AsSequential* operator to make sure that the *Take* method doesn't mess up your order.

LISTING 1-25 Making a parallel query sequential

```
var numbers = Enumerable.Range(0, 20);

var parallelResult = numbers.AsParallel().AsOrdered()
    .Where(i => i % 2 == 0).AsSequential();

foreach (int i in parallelResult.Take(5))
    Console.WriteLine(i);

// Displays
// 0
// 2
// 4
// 6
// 8
```

When using PLINQ, you can use the *ForAll* operator to iterate over a collection when the iteration can also be done in a parallel way. Listing 1-26 shows how to do this.

LISTING 1-26 Using *ForAll*

```
var numbers = Enumerable.Range(0, 20);

var parallelResult = numbers.AsParallel()
    .Where(i => i % 2 == 0);

parallelResult.ForAll(e => Console.WriteLine(e));
```

In contrast to *foreach*, *ForAll* does not need all results before it starts executing. In this example, *ForAll* does, however, remove any sort order that is specified.

Of course, it can happen that some of the operations in your parallel query throw an exception. The .NET Framework handles this by aggregating all exceptions into one *AggregateException*. This exception exposes a list of all exceptions that have happened during parallel execution. Listing 1-27 shows how you can handle this.

LISTING 1-27 Catching *AggregateException*

```
using System;
using System.Linq;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()
        {
            var numbers = Enumerable.Range(0, 20);

            try
            {
                var parallelResult = numbers.AsParallel()
                    .Where(i => IsEven(i));

                parallelResult.ForAll(e => Console.WriteLine(e));
            }
            catch (AggregateException e)
            {
                Console.WriteLine("There were {0} exceptions",
                    e.InnerExceptions.Count);
            }
        }

        public static bool IsEven(int i)
        {
            if (i % 10 == 0) throw new ArgumentException("i");

            return i % 2 == 0;
        }
    }
}
```

```

        }
    }

// Displays
// 4
// 6
// 8
// 2
// 12
// 14
// 16
// 18
// There were 2 exceptions

```

As you can see, two exceptions were thrown while processing the data. You can inspect those exceptions by looping through the *InnerExceptions* property.

Using concurrent collections

When working in a multithreaded environment, you need to make sure that you are not manipulating shared data at the same time without synchronizing access.

The .NET Framework offers some collection classes that are created specifically for use in concurrent environments, which is what you have when you're using multithreading. These collections are thread-safe, which means that they internally use synchronization to make sure that they can be accessed by multiple threads at the same time.

Those collections are the following:

- *BlockingCollection<T>*
- *ConcurrentBag<T>*
- *ConcurrentDictionary<TKey,T>*
- *ConcurrentQueue<T>*
- *ConcurrentStack<T>*

BlockingCollection<T>

This collection is thread-safe for adding and removing data. Removing an item from the collection can be blocked until data becomes available. Adding data is fast, but you can set a maximum upper limit. If that limit is reached, adding an item blocks the calling thread until there is room.

BlockingCollection is in reality a wrapper around other collection types. If you don't give it any specific instructions, it uses the *ConcurrentQueue* by default.

A regular collection blows up when being used in a multithreaded scenario because an item might be removed by one thread while the other thread is trying to read it.

Listing 1-28 shows an example of using a *BlockingCollection*. One *Task* listens for new items being added to the collection. It blocks if there are no items available. The other *Task* adds items to the collection.

LISTING 1-28 Using *BlockingCollection<T>*

```
using System;
using System.Collections.Concurrent;
using System.Threading.Tasks;

namespace Chapter1
{
    public static class Program
    {
        public static void Main()
        {
            BlockingCollection<string> col = new BlockingCollection<string>();
            Task read = Task.Run(() =>
            {
                while (true)
                {
                    Console.WriteLine(col.Take());
                }
            });

            Task write = Task.Run(() =>
            {
                while (true)
                {
                    string s = Console.ReadLine();
                    if (string.IsNullOrWhiteSpace(s)) break;
                    col.Add(s);
                }
            });

            write.Wait();
        }
    }
}
```

The program terminates when the user doesn't enter any data. Until that, every string entered is added by the write *Task* and removed by the read *Task*.

You can use the *CompleteAdding* method to signal to the *BlockingCollection* that no more items will be added. If other threads are waiting for new items, they won't be blocked anymore.

You can even remove the *while(true)* statements from Listing 1-28. By using the *GetConsumingEnumerable* method, you get an *IEnumerable* that blocks until it finds a new item. That way, you can use a *foreach* with your *BlockingCollection* to enumerate it (see Listing 1-29).

LISTING 1-29 Using *GetConsumingEnumerable* on a *BlockingCollection*

```
Task read = Task.Run(() =>
{
    foreach (string v in col.GetConsumingEnumerable())
        Console.WriteLine(v);
});
```

MORE INFO IENUMERABLE

For more information about using *IEnumerable*, see Chapter 2.

ConcurrentBag

A *ConcurrentBag* is just a bag of items. It enables duplicates and it has no particular order. Important methods are *Add*, *TryTake*, and *TryPeek*.

Listing 1-30 shows how to work with the *ConcurrentBag*.

LISTING 1-30 Using a ConcurrentBag

```
ConcurrentBag<int> bag = new ConcurrentBag<int>();

bag.Add(42);
bag.Add(21);

int result;
if (bag.TryTake(out result))
    Console.WriteLine(result);

if (bag.TryPeek(out result))
    Console.WriteLine("There is a next item: {0}", result);
```

One thing to keep in mind is that the *TryPeek* method is not very useful in a multithreaded environment. It could be that another thread removes the item before you can access it.

ConcurrentBag also implements *IEnumerable<T>*, so you can iterate over it. This operation is made thread-safe by making a snapshot of the collection when you start iterating it, so items added to the collection after you started iterating it won't be visible. Listing 1-31 shows this in practice.

LISTING 1-31 Enumerating a ConcurrentBag

```
ConcurrentBag<int> bag = new ConcurrentBag<int>();
Task.Run(() =>
{
    bag.Add(42);
    Thread.Sleep(1000);
    bag.Add(21);
});
Task.Run(() =>
{
    foreach (int i in bag)
        Console.WriteLine(i);
}).Wait();

// Displays
// 42
```

This code only displays 42 because the other value is added after iterating over the bag has started.

ConcurrentStack and ConcurrentQueue

A stack is a *last in, first out* (LIFO) collection. A queue is a *first in, first out* (FIFO) collection.

ConcurrentStack has two important methods: *Push* and *TryPop*. *Push* is used to add an item to the stack; *TryPop* tries to get an item off the stack. You can never be sure whether there are items on the stack because multiple threads might be accessing your collection at the same time.

You can also add and remove multiple items at once by using *PushRange* and *TryPopRange*. When you enumerate the collection, a snapshot is taken.

Listing 1-32 shows how these methods work.

LISTING 1-32 Using a ConcurrentStack

```
ConcurrentStack<int> stack = new ConcurrentStack<int>();  
  
stack.Push(42);  
  
int result;  
if (stack.TryPop(out result))  
    Console.WriteLine("Popped: {0}", result);  
  
stack.PushRange(new int[] { 1, 2, 3 });  
  
int[] values = new int[2];  
stack.TryPopRange(values);  
  
foreach (int i in values)  
    Console.WriteLine(i);  
  
// Popped: 42  
// 3  
// 2
```

ConcurrentQueue offers the methods *Enqueue* and *TryDequeue* to add and remove items from the collection. It also has a *TryPeek* method and it implements *IEnumerable* by making a snapshot of the data. Listing 1-33 shows how to use a *ConcurrentQueue*.

LISTING 1-33 Using a ConcurrentQueue.

```
ConcurrentQueue<int> queue = new ConcurrentQueue<int>();  
queue.Enqueue(42);  
  
int result;  
if (queue.TryDequeue(out result))  
    Console.WriteLine("Dequeued: {0}", result);  
  
// Dequeued: 42
```

ConcurrentDictionary

A *ConcurrentDictionary* stores key and value pairs in a thread-safe manner. You can use methods to add and remove items, and to update items in place if they exist.

Listing 1-34 shows the methods that you can use on a *ConcurrentDictionary*.

LISTING 1-34 Using a *ConcurrentDictionary*

```
var dict = new ConcurrentDictionary<string, int>();
if (dict.TryAdd("k1", 42))
{
    Console.WriteLine("Added");
}

if (dict.TryUpdate("k1", 21, 42))
{
    Console.WriteLine("42 updated to 21");
}

dict["k1"] = 42; // Overwrite unconditionally

int r1 = dict.AddOrUpdate("k1", 3, (s, i) => i * 2);
int r2 = dict.GetOrAdd("k2", () => 3);
```

When working with a *ConcurrentDictionary* you have methods that can atomically add, get, and update items. An *atomic operation* means that it will be started and finished as a single step without other threads interfering. *TryUpdate* checks to see whether the current value is equal to the existing value before updating it. *AddOrUpdate* makes sure an item is added if it's not there, and updated to a new value if it is. *GetOrAdd* gets the current value of an item if it's available; if not, it adds the new value by using a factory method.



Thought experiment

Implementing multithreading

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

You need to build a new application, and you look into multithreading capabilities. Your application consists of a client application that communicates with a web server.

- 1. Explain how multithreading can help with your client application.**
- 2. What is the difference between CPU and I/O bound operations?**
- 3. Does using multithreading with the TPL offer the same advantages for your server application?**

Objective summary

- A thread can be seen as a virtualized CPU.
- Using multiple threads can improve responsiveness and enables you to make use of multiple processors.
- The *Thread* class can be used if you want to create your own threads explicitly. Otherwise, you can use the *ThreadPool* to queue work and let the runtime handle things.
- A *Task* object encapsulates a job that needs to be executed. Tasks are the recommended way to create multithreaded code.
- The *Parallel* class can be used to run code in parallel.
- PLINQ is an extension to LINQ to run queries in parallel.
- The new *async* and *await* operators can be used to write asynchronous code more easily.
- Concurrent collections can be used to safely work with data in a multithreaded (concurrent access) environment.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. You have a lot of items that need to be processed. For each item, you need to perform a complex calculation. Which technique should you use?
 - A. You create a *Task* for each item and then wait until all tasks are finished.
 - B. You use *Parallel.For* to process all items concurrently.
 - C. You use *async/await* to process all items concurrently.
 - D. You add all items to a *BlockingCollection* and process them on a thread created by the *Thread* class.
2. You are creating a complex query that doesn't require any particular order and you want to run it in parallel. Which method should you use?
 - A. *AsParallel*
 - B. *AsSequential*
 - C. *AsOrdered*
 - D. *WithDegreeOfParallelism*

3. You are working on an ASP.NET application that retrieves some data from another web server and then writes the response to the database. Should you use *async/await*?
- A. No, both operations depend on external factors. You need to wait before they are finished.
 - B. No, in a server application you don't have to use *async/await*. It's only for responsiveness on the client.
 - C. Yes, this will free your thread to serve other requests while waiting for the I/O to complete.
 - D. Yes, this puts your thread to sleep while waiting for I/O so that it doesn't use any CPU.

Objective 1.2: Manage multithreading

Although multithreading can give you a lot of advantages, it's not easy to write a multi-threaded application. Problems can happen when different threads access some shared data. What should happen when both try to change something at the same time? To make this work successfully, *synchronizing* resources is important.

This objective covers how to:

- Synchronize resources.
- Cancel long-running tasks.

Synchronizing resources

As you have seen, with the TPL support in .NET, it's quite easy to create a multithreaded application. But when you build real-world applications with multithreading, you run into problems when you want to access the same data from multiple threads simultaneously. Listing 1-35 shows an example of what can go wrong.

LISTING 1-35 Accessing shared data in a multithreaded application

```
using System;
using System.Threading.Tasks;

namespace Chapter1
{
    public class Program
    {
        static void Main()
        {
            int n = 0;

            var up = Task.Run(() =>
            {
```

```

        for (int i = 0; i < 1000000; i++)
            n++;
    });

    for (int i = 0; i < 1000000; i++)
        n--;

    up.Wait();
    Console.WriteLine(n);
}
}
}

```

What would the output of Listing 1-35 be? The answer is, it depends. When you run this application, you get a different output each time. The seemingly simple operation of incrementing and decrementing the variable *n* results in both a lookup (check the value of *n*) and add or subtract 1 from *n*. But what if the first task reads the value and adds 1, and at the exact same time task 2 reads the value and subtracts 1? This is what happens in this example and that's why you never get the expected output of 0.

This is because the operation is not *atomic*. It consists of both a read and a write that happen at different moments. This is why access to the data you're working with needs to be *synchronized*, so you can reliably predict how your data is affected.

It's important to synchronize access to shared data. One feature the C# language offers is the *lock* operator, which is some syntactic sugar that the compiler translates in a call to *System.Threading.Monitor*. Listing 1-36 shows the use of the *lock* operator to fix the previous example.

LISTING 1-36 Using the *lock* keyword

```

using System;
using System.Threading.Tasks;

namespace Chapter1
{
    public class Program
    {
        static void Main()
        {
            int n = 0;

            object _lock = new object();

            var up = Task.Run(() =>
            {
                for (int i = 0; i < 1000000; i++)
                    lock (_lock)
                        n++;
            });

            for (int i = 0; i < 1000000; i++)

```

```

        lock (_lock)
        n--;

        up.Wait();
        Console.WriteLine(n);
    }
}
}

```

After this change, the program always outputs *0* because access to the variable *n* is now synchronized. There is no way that one thread could change the value while the other thread is working with it.

However, it also causes the threads to *block* while they are waiting for each other. This can give performance problems and it could even lead to a *deadlock*, where both threads wait on each other, causing neither to ever complete. Listing 1-37 shows an example of a deadlock.

LISTING 1-37 Creating a deadlock

```

using System;
using System.Threading;
using System.Threading.Tasks;

namespace Chapter1
{
    public class Program
    {
        static void Main()
        {
            object lockA = new object();
            object lockB = new object();

            var up = Task.Run(() =>
            {
                lock (lockA)
                {
                    Thread.Sleep(1000);
                    lock (lockB)
                    {
                        Console.WriteLine("Locked A and B");
                    }
                }
            });

            lock (lockB)
            {
                lock (lockA)
                {
                    Console.WriteLine("Locked B and A");
                }
            }
            up.Wait();
        }
    }
}

```

Because both locks are taken in reverse order, a deadlock occurs. The first *Task* locks *A* and waits for *B* to become free. The main thread, however, has *B* locked and is waiting for *A* to be released.

You need to be careful to avoid deadlocks in your code. You can avoid a deadlock by making sure that locks are requested in the same order. That way, the first thread can finish its work, after which the second thread can continue.

The lock code is translated by the compiler into something that looks like Listing 1-38.

LISTING 1-38 Generated code from a *lock* statement

```
object gate = new object();
bool __lockTaken = false;
try
{
    Monitor.Enter(gate, ref __lockTaken);
}
finally
{
    if (__lockTaken)
        Monitor.Exit(gate);
}
```

You shouldn't write this code by hand; let the compiler generate it for you. The compiler takes care of tricky edge cases that can happen.

It's important to use the *lock* statement with a reference object that is private to the class. A public object could be used by other threads to acquire a lock without your code knowing.

It should also be a reference type because a value type would get boxed each time you acquired a lock. In practice, this generates a completely new lock each time, losing the locking mechanism. Fortunately, the compiler helps by raising an error when you accidentally use a value type for the *lock* statement.

You should also avoid locking on the *this* variable because that variable could be used by other code to create a lock, causing deadlocks.

For the same reason, you should not lock on a string. Because of *string-interning* (the process in which the compiler creates one object for several strings that have the same content) you could suddenly be asking for a lock on an object that is used in multiple places.

Volatile class

The C# compiler is pretty good at optimizing code. The compiler can even remove complete statements if it discovers that certain code would never be executed.

The compiler sometimes changes the order of statements in your code. Normally, this wouldn't be a problem in a single-threaded environment. But take a look at Listing 1-39, in which a problem could happen in a multithreaded environment.

LISTING 1-39 A potential problem with multithreaded code

```
private static int _flag = 0;
private static int _value = 0;

public static void Thread1()
{
    _value = 5;
    _flag = 1;
}

public static void Thread2()
{
    if (_flag == 1)
        Console.WriteLine(_value);
}
```

Normally, if you would run *Thread1* and *Thread2*, you would expect no output or an output of 5. It could be, however, that the compiler switches the two lines in *Thread1*. If *Thread2* then executes, it could be that *_flag* has a value of 1 and *_value* has a value of 0.

You can use locking to fix this, but there is also another class in the .NET Framework that you can use: *System.Threading.Volatile*. This class has a special *Write* and *Read* method, and those methods disable the compiler optimizations so you can force the correct order in your code. Using these methods in the correct order can be quite complex, so .NET offers the *volatile* keyword that you can apply to a field. You would then change the declaration of your field to this:

```
private static volatile int _flag = 0;
```

It's good to be aware of the existence of the *volatile* keyword, but it's something you should use only if you really need it. Because it disables certain compiler optimizations, it will hurt performance. It's also not something that is supported by all .NET languages (Visual Basic doesn't support it), so it hinders language interoperability.

The *Interlocked* class

Referring to Listing 1-35, the essential problem was that the operations of adding and subtracting were not atomic. This because *n++* is translated into *n = n + 1*, both a read and a write.

Making operations atomic is the job of the *Interlocked* class that can be found in the *System.Threading* namespace. When using the *Interlocked.Increment* and *Interlocked.Decrement*, you create an atomic operation, as Listing 1-40 shows.

LISTING 1-40 Using the *Interlocked* class

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace Chapter1
{
```

```

public class Program
{
    static void Main()
    {
        int n = 0;

        var up = Task.Run(() =>
        {
            for (int i = 0; i < 1000000; i++)
                Interlocked.Increment(ref n);
        });

        for (int i = 0; i < 1000000; i++)
            Interlocked.Decrement(ref n);

        up.Wait();
        Console.WriteLine(n);
    }
}

```

Interlocked guarantees that the increment and decrement operations are executed atomically. No other thread will see any intermediate results. Of course, adding and subtracting is a simple operation. If you have more complex operations, you would still have to use a lock.

Interlocked also supports switching values by using the *Exchange* method. You use this method as follows:

```
if ( Interlocked.Exchange(ref isInUse, 1) == 0) { }
```

This code retrieves the current value and immediately sets it to the new value in the same operation. It returns the previous value before changing it.

You can also use the *CompareExchange* method. This method first checks to see whether the expected value is there; if it is, it replaces it with another value.

Listing 1-41 shows what can go wrong when comparing and exchanging a value in a non-atomic operation.

LISTING 1-41 Compare and exchange as a nonatomic operation

```

using System;
using System.Threading;
using System.Threading.Tasks;

public static class Program
{
    static int value = 1;

    public static void Main()
    {
        Task t1 = Task.Run(() =>
        {

```

```

        if (value == 1)
    {
        // Removing the following line will change the output
        Thread.Sleep(1000);
        value = 2;
    }
});

Task t2 = Task.Run(() =>
{
    value = 3;
});

Task.WaitAll(t1, t2);
Console.WriteLine(value); // Displays 2
}
}

```

Task t1 starts running and sees that *value* is equal to 1. At the same time, *t2* changes the value to 3 and then *t1* changes it back to 2. To avoid this, you can use the following *Interlocked* statement:

```
Interlocked.CompareExchange(ref value, newValue, compareTo);
```

This makes sure that comparing the value and exchanging it for a new one is an atomic operation. This way, no other thread can change the value between comparing and exchanging it.

Cancelling tasks

When working with multithreaded code such as the TPL, the *Parallel* class, or PLINQ, you often have long-running tasks. The .NET Framework offers a special class that can help you in canceling these tasks: *CancellationToken*.

You pass a *CancellationToken* to a *Task*, which then periodically monitors the token to see whether cancellation is requested.

Listing 1-42 shows how you can use a *CancellationToken* to end a task.

LISTING 1-42 Using a *CancellationToken*

```

CancellationTokenSource cancellationTokenSource =
    new CancellationTokenSource();
CancellationToken token = cancellationTokenSource.Token;

Task task = Task.Run(() =>
{
    while(!token.IsCancellationRequested)
    {

        Console.Write("*");
        Thread.Sleep(1000);
    }

}, token);

```

```

Console.WriteLine("Press enter to stop the task");
Console.ReadLine();
cancellationTokenSource.Cancel();

Console.WriteLine("Press enter to end the application");
Console.ReadLine();

```

The *CancellationToken* is used in the asynchronous *Task*. The *CancellationTokenSource* is used to signal that the *Task* should cancel itself.

In this case, the operation will just end when cancellation is requested. Outside users of the *Task* won't see anything different because the *Task* will just have a *RanToCompletion* state. If you want to signal to outside users that your task has been canceled, you can do this by throwing an *OperationCanceledException*. Listing 1-43 shows how to do this.

LISTING 1-43 Throwing *OperationCanceledException*

```

using System;
using System.Threading;
using System.Threading.Tasks;

namespace Chapter1.Threads
{
    public class Program
    {
        static void Main()
        {
            CancellationTokenSource cancellationTokenSource =
                new CancellationTokenSource();
            CancellationToken token = cancellationTokenSource.Token;

            Task task = Task.Run(() =>
            {
                while (!token.IsCancellationRequested)
                {

                    Console.Write("*");
                    Thread.Sleep(1000);
                }

                token.ThrowIfCancellationRequested();
            }, token);

            try
            {
                Console.WriteLine("Press enter to stop the task");
                Console.ReadLine();

                cancellationTokenSource.Cancel();
                task.Wait();
            }
        }
    }
}

```

```

        catch (AggregateException e)
        {
            Console.WriteLine(e.InnerException[0].Message);
        }
        Console.WriteLine("Press enter to end the application");
        Console.ReadLine();

    }

}

// Displays
// Press enter to stop the task
// **
// A task was canceled.
// Press enter to end the application

```

Instead of catching the exception, you can also add a continuation *Task* that executes only when the *Task* is canceled. In this *Task*, you have access to the exception that was thrown, and you can choose to handle it if that's appropriate. Listing 1-44 shows what such a continuation task would look like.

LISTING 1-44 Adding a continuation for canceled tasks

```

Task task = Task.Run(() =>
{
    while (!token.IsCancellationRequested)
    {
        Console.Write("*");
        Thread.Sleep(1000);
    }
throw new OperationCanceledException();

}, token).ContinueWith((t) =>
{
    t.Exception.Handle((e) => true);
    Console.WriteLine("You have canceled the task");
}, TaskContinuationOptions.OnlyOnCanceled);

```

If you want to cancel a *Task* after a certain amount of time, you can use an overload of *Task.WaitAny* that takes a timeout. Listing 1-45 shows an example.

LISTING 1-45 Setting a timeout on a task

```

Task longRunning = Task.Run(() =>
{
    Thread.Sleep(10000);
});

int index = Task.WaitAny(new[] { longRunning }, 1000);

if (index == -1)
    Console.WriteLine("Task timed out");

```

If the returned `index` is `-1`, the task timed out. It's important to check for any possible errors on the other tasks. If you don't catch them, they will go unhandled.



Thought experiment

Implementing multithreading

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

You are experiencing deadlocks in your code. It's true that you have a lot of locking statements and you are trying to improve your code to avoid the deadlocks.

1. How can you orchestrate your locking code to avoid deadlocks?
2. How can the `Interlocked` class help you?

Objective summary

- When accessing shared data in a multithreaded environment, you need to synchronize access to avoid errors or corrupted data.
- Use the `lock` statement on a private object to synchronize access to a piece of code.
- You can use the `Interlocked` class to execute simple atomic operations.
- You can cancel tasks by using the `CancellationTokenSource` class with a `CancellationToken`.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. You want to synchronize access by using a `lock` statement. On which member do you lock?
 - A. `this`
 - B. `string _lock = "mylock"`
 - C. `int _lock = 42;`
 - D. `object _lock = new object();`

2. You need to implement cancellation for a long running task. Which object do you pass to the task?
 - A. *CancellationTokenSource*
 - B. *CancellationToken*
 - C. Boolean *isCancelled* variable
 - D. *Volatile*
3. You are implementing a state machine in a multithreaded class. You need to check what the current state is and change it to the new one on each step. Which method do you use?
 - A. *Volatile.Write(ref currentState)*
 - B. *Interlocked.CompareExchange(ref currentState, ref newState, expectedState)*
 - C. *Interlocked.Exchange(ref currentState, newState)*
 - D. *Interlocked.Decrement(ref newState)*

Objective 1.3: Implement program flow

One important aspect of managing the flow of your program is making decisions in your application, including checking to see whether the user has entered the correct password, making sure that a certain value is within range, or one of the myriad other possibilities. C# offers a couple of statements that can be used when you need to make a decision.

Next to making decisions, another common task is working with collections. C# has language features that help you work with collections by allowing you to iterate over collections and access individual items.

This objective covers how to:

- Work with Boolean expressions.
- Make decisions in your application.
- Iterate across collections.
- Use explicit *jump* statements.

Working with Boolean expressions

When working with flow control statements, you will automatically work with *Boolean expressions*. A Boolean expression should always produce *true* or *false* as the end result, but in doing so they can be quite complex by using different *operators*.

One such an operator is the *equality operator* (`==`). You use this one to test that two values are equal to each other. Listing 1-46 shows some examples.

LISTING 1-46 Using the equality operator

```
int x = 42;
int y = 1;
int z = 42;

Console.WriteLine(x == y); // Displays false
Console.WriteLine(x == z); // Displays true
```

Table 1-1 shows the operators that you can use in C#.

TABLE 1-1 C# relational and equality operators

Operator	Description	Example
<	Less than	x < 42;
>	Greater than	x > 42;
<=	Less than or equal to	x <= 42;
>=	Greater than or equal to	x >= 42;
<code>==</code>	Equal to	x == 42;
<code>!=</code>	Not equal to	x != 42;

You can combine these operators by using the *OR* (`||`), *AND* (`&&`), and *Exclusive OR* (`^`) operators. These operators use both a left and a right operand, meaning the left and right part of the expression.

The *OR* operator returns *true* when one of both operands is *true*. If both are *false*, it returns *false*. If both are *true*, it will return *true*. Listing 1-47 shows an example.

LISTING 1-47 Boolean *OR* operator

```
bool x = true;
bool y = false;

bool result = x || y;
Console.WriteLine(result); // Displays True
```

If the runtime notices that the left part of your *OR* operation is *true*, it doesn't have to evaluate the right part of your expression. This is called short-circuiting. Listing 1-48 shows an example.

LISTING 1-48 Short-circuiting the *OR* operator

```
public void OrShortCircuit()
{
    bool x = true;
    bool result = x || GetY();
}

private bool GetY()
{
    Console.WriteLine("This method doesn't get called");
    return true;
}
```

In this case, the method *GetY* is never called and the line is not written to the console.

The *AND* operator can be used when both parts of an expression need to be *true*. If either one of the operands is *false*, the whole expression evaluates to *false*. Listing 1-49 uses the *AND* operator to check to see whether a value is within a certain range.

LISTING 1-49 Using the *AND* operator

```
int value = 42;
bool result = (0 < value) && (value < 100)
```

In this case, it's not required to add the extra parentheses around the left and right operand but it does add to the readability of your code. Just as with the *OR* operator, the runtime applies short-circuiting. Next to being a performance optimization, you can also use it to your advantage when working with *null* values. Listing 1-50 for example uses the *AND* operator to check if the *input* argument is not *null* and to execute a method on it. If short-circuiting wouldn't be used in this situation, the code would throw an exception each time the *input* parameter would be *null*.

LISTING 1-50 Short-circuiting the *AND* operator

```
public void Process(string input)
{
    bool result = (input != null) && (input.StartsWith("v"));
    // Do something with the result
}
```

The *Exclusive OR* operator (*XOR*) returns *true* only when exactly one of the operands is *true*. Table 1-2 gives the possibilities for the *XOR* operator.

TABLE 1-2 Possible values for the *XOR* operator

Left operand	Right operand	Result
True	True	False
True	False	True
False	True	True
False	False	False

Because the *XOR* operator has to check that exactly one of the operands is *true*, it doesn't apply short-circuiting. Listing 1-51 shows how to use the *XOR* operator.

LISTING 1-51 Using the *XOR* operator

```
bool a = true;
bool b = false;

Console.WriteLine(a ^ a); // False
Console.WriteLine(a ^ b); // True
Console.WriteLine(b ^ b); // False
```

Making decisions

C# offers several *flow control statements* that help you determine the path that your application follows. You can use the following statements:

- *if*
- *while*
- *do while*
- *for*
- *foreach*
- *switch*
- *break*
- *continue*
- *goto*
- Null-coalescing operator (??)
- Conditional operator (?:)

Using these constructs, you can create flexible applications that enable you to execute different behavior depending on the circumstances. It's important to know these statements and be able to choose between them.

The *if* statement

The most widely used flow control statement is the *if* statement. The *if* statement enables you to execute a piece of code depending on a specific condition. The general syntax for the *if statement* is this:

```
if (boolean-expression)
    statement to execute
```

The statement to execute is executed only if the *boolean* expression evaluates to *true*. Listing 1-52 shows an example of using *if*.

LISTING 1-52 Basic *if* statement

```
bool b = true;
if (b)
    Console.WriteLine("True");
```

In this case, the application outputs “*True*” because the condition for the *if statement* is *true*. If *b* would be *false*, the *Console.WriteLine* statement would not be executed.

Of course, passing a hard-coded value to the *if statement* is not very useful. Normally, you would use the *if statement* with a more dynamic value that can change during the execution of the application.

When working with program flow statements, it’s important to know the concept of a *code block*, which enables you to write multiple statements in a context in which only a single statement is allowed.

A block uses curly-braces to denote its start and end:

```
{  
    statements  
}
```

Listing 1-52 showed an *if statement* that executes a single line of code only if it’s *true*. You can, however, also use a *code block* after the *if statement*. All code in the block is executed based on the result of the *if statement*. You can see an example of this in Listing 1-53.

LISTING 1-53 An *if* statement with code block

```
bool b = true;
if (b)
{
    Console.WriteLine("Both these lines");
    Console.WriteLine("Will be executed");
}
```

Variables defined within a code block are accessible only within the code block and go out of scope at the end of the block. This means that you can declare variables inside a block, and use them within the block but not outside the block. Listing 1-54 shows the scoping differences. Variable *b* is declared outside the block and can be accessed both in the outer block and in the *if statement*. Variable *r*, however, can be accessed only in the *if statement*.

LISTING 1-54 Code blocks and scoping

```
bool b = true;
if (b)
{
    int r = 42;
    b = false;
}

// r is not accessible
// b is now false
```

You can also execute some code when the *if* statement evaluates to *false*. You can do this by using an *else* block. The general syntax looks like this:

```
if (boolean-expression)
    statement
else
    statement
```

Listing 1-55 shows an example of using an *else* statement. This outputs “*False*”.

LISTING 1-55 Using an *else* statement

```
bool b = false;

if (b)
{
    Console.WriteLine("True");
}
else
{
    Console.WriteLine("False");
}
```

You can use multiple *if/else* statements as shown in Listing 1-56.

LISTING 1-56 Using multiple *if/else* statements

```
bool b = false;
bool c = true;

if (b)
{
    Console.WriteLine("b is true");
}
else if (c)
{
    Console.WriteLine("c is true");
}
else
{
    Console.WriteLine("b and c are false");
}
```

You can also nest *if* and *else* statements. For readability, it’s nice to outline your code correctly. The following code is perfectly legal, but on first sight it’s hard to see what the code really does:

```
if (x) if (y) F(); else G();
```

When outlined correctly, the code is equal to the one in Listing 1-57, which is much easier to understand.

LISTING 1-57 A more readable nested *if* statement

```
if (x)
{
    if (y)
    {
        F();
    }
    else
    {
        G();
    }
}
```

The compiler optimizes your code and removes any unnecessary braces and statements. Under normal circumstances, you should worry more about readability than about the number of lines you produce. Team members especially appreciate it when you write code that's not only correct but also easier to maintain.

The null-coalescing operator

The `??` operator is called the *null-coalescing operator*. You can use it to provide a default value for nullable value types or for reference types.

The operator returns the left value if it's not *null*; otherwise, the right operand.

Listing 1-58 shows an example of using the operator.

LISTING 1-58 The null-coalescing operator

```
int? x = null;
int y = x ?? -1;
```

In this case, the value of *y* is *-1* because *x* is *null*.

You can also nest the null-coalescing operator, as Listing 1-59 shows.

LISTING 1-59 Nesting the null-coalescing operator

```
int? x = null;
int? z = null;
int y = x ??
           z ??
           -1;
```

Of course, you can achieve the same with an *if* statement but the null-coalescing operator can shorten your code and improve its readability.

The conditional operator

The *conditional operator* (`?:`) returns one of two values depending on a Boolean expression. If the expression is *true*, the first value is returned; otherwise, the second.

Listing 1-60 shows an example of how the operator can be used to simplify some code. In this case, the *if* statement can be replaced with the conditional operator.

LISTING 1-60 The conditional operator

```
private static int GetValue(bool p)
{
    if (p)
        return 1;
    else
        return 0;

    return p ? 1 : 0;
}
```

The *switch* statement

You can use the *switch* statement to simplify complex *if* statements. Take the example of Listing 1-61.

LISTING 1-61 A complex *if* statement

```
void Check(char input)
{
    if (input == 'a'
        || input == 'e'
        || input == 'i'
        || input == 'o'
        || input == 'u')
    {
        Console.WriteLine("Input is a vowel");
    }
    else
    {
        Console.WriteLine("Input is a consonant");
    }
}
```

The *switch* statement can be used to make this code more comprehensive. A *switch* statement checks the value of its argument and then looks for a matching label. Listing 1-62 shows the code from Listing 1-59 as a *switch* statement.

LISTING 1-62 A *switch* statement

```
void CheckWithSwitch(char input)
{
    switch (input)
    {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
    {
        Console.WriteLine("Input is a vowel");
```

```

        break;
    }
    case 'y':
    {
        Console.WriteLine("Input is sometimes a vowel.");
        break;
    }
    default:
    {
        Console.WriteLine("Input is a consonant");
        break;
    }
}
}

```

A *switch* can use one or multiple *switch-sections* that can contain one or more *switch-labels*. In Listing 1-62, all the vowels belong to the same *switch-section*. If you want, you can also add a *default label* that is used when none of the other labels matches.

The end point of a *switch* statement should not be reachable. You need to have a statement such as *break* or *return* that explicitly exits the *switch* statement, or you need to throw an exception. This avoids the fall-through behavior that C++ has. This makes it possible for *switch* sections to appear in any order without affecting behavior.

Instead of implicitly falling through to another label, you can use the *goto statement* (see Listing 1-63).

LISTING 1-63 *goto* in a *switch* statement

```

int i = 1;
switch (i)
{
    case 1:
    {
        Console.WriteLine("Case 1");
        goto case 2;
    }
    case 2:
    {
        Console.WriteLine("Case 2");
        break;
    }
}

// Displays
// Case 1
// Case 2

```

Iterating across collections

Another subject that has to do with the flow of your program is iterating across collections. Collections are widely used in C#, and the language offers constructs that you can use with them:

- *for*
- *foreach*
- *while*
- *do while*

The *for* loop

You can use a *for loop* when you need to iterate over a collection until a specific condition is reached (for example, you have reached the end of a collection). Listing 1-64 shows an example in which you loop through all items in an array.

LISTING 1-64 A basic *for* loop

```
int[] values = { 1, 2, 3, 4, 5, 6 };
for (int index = 0; index < values.Length; index++)
{
    Console.WriteLine(values[index]);
}

// Displays
// 123456
```

As you can see, the *for* loop consists of three different parts:

```
for(initial; condition; loop)
```

- The initial part is executed before the first iteration and declares and initializes the variables that are used in the loop.
- The condition is evaluated on each iteration. When the condition equals *false*, the loop is exited.
- The loop section is run during every iteration and is normally used to change the counter that's used to loop over the collection.

None of these parts is required. You can use `for(;;)` as a perfectly legal *for* loop that would never end. You can also use multiple statements in each part of your *for* loop (see Listing 1-65).

LISTING 1-65 A *for* loop with multiple loop variables

```
int[] values = { 1, 2, 3, 4, 5, 6 };
for (int x = 0, y = values.Length - 1;
    ((x < values.Length) && (y >= 0));
    x++, y--)
{
    Console.WriteLine(values[x]);
    Console.WriteLine(values[y]);
}

// Displays
// 162534435261
```

It's also not required to let the loop value increment or decrement with 1. For example, you can change Listing 1-64 to increment *index* with 2 to only display the odd numbers, as Listing 1-66 shows.

LISTING 1-66 A *for* loop with a custom increment

```
int[] values = { 1, 2, 3, 4, 5, 6 };
for (int index = 0; index < values.Length; index += 2)
{
    Console.WriteLine(values[index]);
}

// Displays
// 135
```

Normally, the *for* loop ends when the condition becomes *false*, but you can also decide to manually break out of the loop. You can do this by using the *break* or *return* statement when you want to completely exit the method. Listing 1-67 shows an example of the *break* statement.

LISTING 1-67 A *for* loop with a *break* statement

```
int[] values = { 1, 2, 3, 4, 5, 6 };
for (int index = 0; index < values.Length; index++)
{
    if (values[index] == 4) break;

    Console.WriteLine(values[index]);
}

// Displays
// 123
```

Next to breaking the loop completely, you can also instruct the *for* loop to continue to the next item by using the *continue* statement. Listing 1-68 shows an example in which the number 4 is skipped in the loop.

LISTING 1-68 A *for* loop with a *continue* statement

```
int[] values = { 1, 2, 3, 4, 5, 6 };

for (int index = 0; index < values.Length; index++)
{
    if (values[index] == 4) continue;

    Console.WriteLine(values[index]);
}

// Displays
// 12356
```

The *while* and *do-while* loop

Another looping construction is the *while loop*. A *for* loop is nothing more than a convenient way to write a *while* loop that does the checking and incrementing of the counter. Listing 1-69 shows an example. Notice the extra parenthesis to restrict the scope of the *loop* variable.

LISTING 1-69 Implementing a *for* loop with a *while* statement

```
int[] values = { 1, 2, 3, 4, 5, 6 };

{
    int index = 0;
    while (index < values.Length)
    {
        Console.WriteLine(values[index]);
        index++;
    }
}
```

As you can see, a *while* loop checks an expression and executes as long as this expression is *true*. You should use a *for* loop when you know the number of iterations in advance. A *while* loop can be used when you don't know the number of iterations.

If the condition of the *while* loop is *false*, it won't execute the code inside the loop. This is different when using a *do-while* loop. A *do-while* loop executes at least once, even if the expression is *false*. Listing 1-70 shows an example of using a *do-while* loop.

LISTING 1-70 *do-while* loop

```
do
{
    Console.WriteLine("Executed once!");
}
while (false);
```

Within a *while* or *do-while* loop, you can use the *continue* and *break* statements just as with a *for* loop.

The *foreach* loop

The *foreach loop* is used to iterate over a collection and automatically stores the current item in a *loop* variable. The *foreach* loop keeps track of where it is in the collection and protects you against iterating past the end of the collection.

Listing 1-71 shows an example of how to use the *foreach* loop.

LISTING 1-71 *foreach* loop

```
int[] values = { 1, 2, 3, 4, 5, 6 };

foreach (int i in values)
{
    Console.WriteLine(i);
}

// Displays 123456
```

As you can see, the *foreach* loop automatically stores the current item in a strongly typed variable. You can use the *continue* and *break* statements to influence the way the *foreach* loop works.

The *loop* variable cannot be modified. You can make modifications to the object that the variable points to, but you can't assign a new value to it. Listing 1-72 shows these differences.

LISTING 1-72 Changing items in a *foreach*

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

void CannotChangeForEachIterationVariable()
{
    var people = new List<Person>
    {
        new Person() { FirstName = "John", LastName = "Doe" },
        new Person() { FirstName = "Jane", LastName = "Doe" },
    };

    foreach (Person p in people)
    {
        p.LastName = "Changed"; // This is allowed
        // p = new Person(); // This gives a compile error
    }
}
```

You can understand this behavior when you know how *foreach* actually works. When the compiler encounters a *foreach* statement, it generates some code on your behalf; *foreach* is syntactic sugar that lets you write some code in a nice way. Listing 1-73 shows what's happening.

LISTING 1-73 The compiler-generated code for a *foreach* loop

```
List<Person>.Enumerator e = people.GetEnumerator();

try
{
    Person v;
    while (e.MoveNext())
    {
        v = e.Current;
    }
}
finally
{
    System.IDisposable d = e as System.IDisposable;
    if (d != null) d.Dispose();
}
```

If you change the value of *e.Current* to something else, the iterator pattern can't determine what to do when *e.MoveNext* is called. This is why it's not allowed to change the value of the iteration variable in a *foreach* statement.

MORE INFO ENUMERATORS

For more information on how the *GetEnumerator* method works and how you can implement your own enumerators see Chapter 2.

Jump statements

Another type of statement that can be used to influence program flow is a *jump statement*. You have already looked at two of those statements: *break* and *continue*. A *jump* statement unconditionally transfers control to another location in your code.

Another *jump* statement that can be used to change the flow of a program is *goto*. The *goto* statement moves control to a statement that is marked by a label. If the label can't be found or is not within the scope of the *goto* statement, a compiler error occurs.

Listing 1-74 shows an example of using *goto* and a *label*.

LISTING 1-74 *goto* statement with a label

```
int x = 3;
if (x == 3) goto customLabel;
x++;

customLabel:
Console.WriteLine(x);
// Displays 3
```

You cannot make a jump to a label that's not in scope. This means you cannot transfer control to another block of code that's outside of your current block. The compiler also makes sure that any *finally* blocks that intervene are executed.

The *jump* statements such as *break* and *continue* can have their uses in some situations. If possible, however, you should try to avoid them. By refactoring your code, you can remove them most of the time and this will improve the readability of your code.

The *goto* statement is even worse. It is considered a bad practice. Although C# restricts the way the *goto* operator behaves, as a guideline, you should try to avoid using *goto*. One area where *goto* is used is in generated code like the code the compiler generates when you use the new *async/await* feature in C# 5.

MORE INFO JUMP STATEMENTS

For more information about *jump* statements, see <http://msdn.microsoft.com/en-us/library/d96yfwee.aspx>.



Thought experiment

Choosing your program flow statements

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

You are updating an old C#2 console application to a WPF C#5 application. The application is used by hotels to keep track of reservations and guests coming and leaving. You are going through the old code base to determine whether there is code that can be easily reused. You notice a couple of things:

- The code uses the *goto* statement to manage flow.
 - There are a lot of long *if* statements that map user input.
 - The code uses the *for* loop extensively.
1. What is the disadvantage of using *goto*? How can you avoid using the *goto* statement?
 2. Which statement can you use to improve the long *if* statements?
 3. What are the differences between the *for* and *foreach* statement? When should you use which?

Objective summary

- Boolean expressions can use several operators: ==, !=, <, >, <=, >=, !. Those operators can be combined together by using *AND* (&&), *OR* (||) and *XOR* (^).
- You can use the *if-else* statement to execute code depending on a specific condition.
- The *switch* statement can be used when matching a value against a couple of options.

- The *for* loop can be used when iterating over a collection where you know the number of iterations in advance.
- A *while* loop can be used to execute some code while a condition is true; *do-while* should be used when the code should be executed at least once.
- *foreach* can be used to iterate over collections.
- *Jump* statements such as *break*, *goto*, and *continue* can be used to transfer control to another line of the program.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. You need to iterate over a collection in which you know the number of items. You need to remove certain items from the collection. Which statement do you use?
 - switch*
 - foreach*
 - for*
 - goto*
2. You have a lot of checks in your application for *null* values. If a value is not *null*, you want to call a method on it. You want to simplify your code. Which technique do you use?
 - for*
 - Conditional operator
 - Null-coalescing operator
 - The short-circuiting behavior of the and operator
3. You are processing some data from over the network. You use a *HasNext* and *Read* method to retrieve the data. You need to run some code on each item. What do you use?
 - for*
 - foreach*
 - while*
 - do-while*

Objective 1.4: Create and implement events and callbacks

An *event* can be used to provide notifications. You can subscribe to an event if you are interested in those notifications. You can also create your own events and raise them to provide notifications when something interesting happens. The .NET Framework offers built-in types that you can use to create events. By using delegates, lambda expressions, and anonymous methods, you can create and use events in a comfortable way.

This objective covers how to:

- Understand delegates.
- Use lambda expressions.
- Create and raise events.

Understanding delegates

In C#, *delegates* form the basic building blocks for events. A *delegate* is a type that defines a method signature. In C++, for example, you would do this with a function pointer. In C# you can instantiate a *delegate* and let it point to another method. You can invoke the method through the *delegate*.

Listing 1-75 shows an example of declaring a *delegate* and calling a method through it.

LISTING 1-75 Using a delegate

```
public delegate int Calculate(int x, int y);

public int Add(int x, int y) { return x + y; }
public int Multiply(int x, int y) { return x * y; }

public void UseDelegate()
{
    Calculate calc = Add;
    Console.WriteLine(calc(3, 4)); // Displays 7

    calc = Multiply;
    Console.WriteLine(calc(3, 4)); // Displays 12
}
```

As you can see, you use the special *delegate* keyword to tell the compiler that you are creating a *delegate* type. *Delegates* can be nested in other types and they can then be used as a nested type.

Instantiating *delegates* is easy since C# 2.0 added the automatic creation of a new *delegate* when a method group is assigned to a delegate type. An instantiated *delegate* is an object; you can pass it around and give it as an argument to other methods.

Another feature of *delegates* is that you can combine them together. This is called *multicasting*. You can use the + or += operator to add another method to the *invocation list* of an existing delegate instance. Listing 1-76 shows an example.

LISTING 1-76 A multicast delegate

```
public void MethodOne()
{
    Console.WriteLine("MethodOne");
}

public void MethodTwo()
{
    Console.WriteLine("MethodTwo");
}

public delegate void Del();

public void Multicast()
{
    Del d = MethodOne;
    d += MethodTwo;

    d();
}
// Displays
// MethodOne
// MethodTwo
```

You can also remove a method from an invocation list by using the decrement assignment operator (- or -=).

All this is possible because delegates inherit from the *System.MulticastDelegate* class that in turn inherits from *System.Delegate*. Because of this, you can use the members that are defined in those base classes on your delegates.

For example, to find out how many methods a multicast delegate is going to call, you can use the following code:

```
int invocationCount = del.GetInvocationList().GetLength(0);
```

When you assign a method to a *delegate*, the method signature does not have to match the *delegate* exactly. This is called *covariance* and *contravariance*. Covariance makes it possible that a method has a return type that is more derived than that defined in the delegate. Contravariance permits a method that has parameter types that are less derived than those in the delegate type.

Listing 1-77 shows an example of *covariance*.

LISTING 1-77 Covariance with delegates

```
public delegate TextWriter CovarianceDel();

public StreamWriter MethodStream() { return null; }
public StringWriter MethodString() { return null; }

CovarianceDel del;
del = MethodStream;
del = MethodString;
```

Because both *StreamWriter* and *StringWriter* inherit from *TextWriter*, you can use the *CovarianceDel* with both methods. An example of contravariance can be seen in Listing 1-78.

LISTING 1-78 Contravariance with delegates

```
void DoSomething(TextWriter tw) { }
public delegate void ContravarianceDel(StreamWriter tw);

ContravarianceDel del = DoSomething;
```

Because the method *DoSomething* can work with a *TextWriter*, it surely can also work with a *StreamWriter*. Because of contravariance, you can call the delegate and pass an instance of *StreamWriter* to the *DoSomething* method.

MORE INFO COVARIANCE AND CONTRAVARIANCE

For more information on covariance and contravariance and how they are implemented in C#, see the excellent series of blog posts that Eric Lippert wrote at <http://blogs.msdn.com/b/ericlippert/archive/tags/covariance+and+contravariance/>.

Using lambda expressions

Sometimes the whole signature of a method can be more code than the body of a method. There are also situations in which you need to create an entire method only to use it in a delegate.

For these cases, Microsoft added some new features to C#. In C# 2.0 *anonymous methods* were added. In C# 3.0, things became even better when *lambda expressions* were added. Lambda expressions are the preferred way to go when writing new code.

Listing 1-79 shows how you would write the example in Listing 1-73 with newer lambda syntax.

LISTING 1-79 Lambda expression to create a delegate

```
Calculate calc = (x, y) => x + y;
Console.WriteLine(calc(3, 4)); // Displays 7
calc = (x, y) => x * y;
Console.WriteLine(calc(3, 4)); // Displays 12
```

When reading this code, you can say *go* or *goes to* for the special lambda syntax. For example, the first lambda expression in Listing 1-79 is read as “*x and y goes to adding x and y.*”

The lambda function has no specific name as the methods in Listing 1-75 have. Because of this, lambda functions are called *anonymous functions*. You also don’t have to specify a *return type* explicitly. The compiler infers this automatically from your lambda. And in the case of Listing 1-79, the types of the parameters *x* and *y* are also not specified explicitly.

As you can see, the syntax for writing a lambda can be compact. If a lambda has only one parameter, you can even remove the parentheses around the parameter.

You can create lambdas that span multiple statements. You can do this by adding curly braces around the statements that form the lambda as Listing 1-80 shows.

LISTING 1-80 Creating a lambda expression with multiple statements

```
Calculate calc =
    (x, y) =>
{
    Console.WriteLine("Adding numbers");
    return x + y;
};

int result = calc(3, 4);
// Displays
// Adding numbers
```

Sometimes declaring a *delegate* for an *event* feels a bit cumbersome. Because of this, the .NET Framework has a couple of built-in *delegate* types that you can use when declaring *delegates*. For the *Calculate* examples, you have used the following *delegate*:

```
public delegate int Calculate(int x, int y);
```

You can replace this *delegate* with one of the built-in types namely *Func<int,int,int>*. The *Func<...>* types can be found in the *System* namespace and they represent *delegates* that return a type and take 0 to 16 parameters. All those types inherit from *System.MulticastDelegate* so you can add multiple methods to the invocation list.

If you want a *delegate* type that doesn’t return a value, you can use the *System.Action* types. They can also take 0 to 16 parameters, but they don’t return a value. Listing 1-81 shows an example of using the *Action* type.

LISTING 1-81 Using the *Action* delegate

```
Action<int, int> calc = (x, y) =>
{
    Console.WriteLine(x + y);
};

calc(3, 4); // Displays 7
```

Things start to become more complex when your *lambda* function starts referring to variables declared outside of the *lambda* expression (or to the *this* reference). Normally, when control leaves the scope of a variable, the variable is no longer valid. But what if a delegate refers to a *local* variable and is then returned to the calling method? Now, the delegate has a longer life than the variable. To fix this, the compiler generates code that makes the life of the captured variable at least as long as the longest-living delegate. This is called a *closure*.

Using events

A popular *design pattern* (a reusable solution for a recurring problem) in application development is that of *publish-subscribe*. You can subscribe to an *event* and then you are notified when the publisher of the *event* raises a new *event*. This is used to establish loose coupling between components in an application.

Delegates form the basis for the event system in C#. Listing 1-82 shows how a class can expose a public *delegate* and raise it.

LISTING 1-82 Using an *Action* to expose an event

```
public class Pub
{
    public Action OnChange { get; set; }

    public void Raise()
    {
        if (OnChange != null)
        {
            OnChange();
        }
    }
}

public void CreateAndRaise()
{
    Pub p = new Pub();
    p.OnChange += () => Console.WriteLine("Event raised to method 1");
    p.OnChange += () => Console.WriteLine("Event raised to method 2");
    p.Raise();
}
```

When calling *CreateAndRaise*, your code creates a new instance of *Pub*, subscribes to the event with two different methods and then raises the event by calling *p.Raise*. The *Pub* class is completely unaware of any subscribers. It just raises the *event*.

If there would be no subscribers to an *event*, the *OnChange* property would be *null*. This is why the *Raise* method checks to see whether *OnChange* is not *null*.

Although this system works, there are a couple of weaknesses. If you change the subscribe line for method 2 to the following, you would effectively remove the first subscriber by using *=* instead of *+=*:

```
p.OnChange = () => Console.WriteLine("Event raised to method 2");
```

In the example from Listing 1-82, the *Pub* class raises the *event*. However, nothing prevents outside users of the class from raising the *event*. By just calling *p.OnChange()* every user of the class can raise the *event* to all subscribers.

To overcome these weaknesses, the C# language uses the special *event* keyword. Listing 1-83 shows a modified example of the *Pub* class that uses the *event* syntax.

LISTING 1-83 Using the *event* keyword

```
public class Pub
{
    public event Action OnChange = delegate { };

    public void Raise()
    {
        OnChange();
    }
}
```

By using the *event* syntax, there are a couple of interesting changes. First, you are no longer using a public property but a public field. Normally, this would be a step back. However, with the *event* syntax, the compiler protects your field from unwanted access.

An *event* cannot be directly assigned to (with the `=` instead of `+=`) operator. So you don't have the risk of someone removing all previous subscriptions, as with the delegate syntax.

Another change is that no outside users can raise your *event*. It can be raised only by code that's part of the class that defined the *event*.

Listing 1-83 also uses some special syntax to initialize the *event* to an empty *delegate*. This way, you can remove the *null* check around raising the *event* because you can be certain that the *event* is never *null*. Outside users of your class can't set the *event* to *null*; only members of your class can. As long as none of your other class members sets the *event* to *null*, you can safely assume that it will always have a value.

There is, however, one change you still have to make to follow the coding conventions in the .NET Framework. Instead of using the *Action* type for your event, you should use the *EventHandler* or *EventHandler<T>*. *EventHandler* is declared as the following delegate:

```
public delegate void EventHandler(object sender, EventArgs e);
```

By default, it takes a *sender object* and some *event arguments*. The *sender* is by convention the object that raised the *event* (or *null* if it comes from a static method). By using *EventHandler<T>*, you can specify the type of *event arguments* you want to use. Listing 1-84 shows an example.

LISTING 1-84 Custom event arguments

```
public class MyArgs : EventArgs
{
    public MyArgs(int value)
    {
        Value = value;
    }

    public int Value { get; set; }
}

public class Pub
{
    public event EventHandler<MyArgs> OnChange = delegate { };

    public void Raise()
    {
        OnChange(this, new MyArgs(42));
    }
}

public void CreateAndRaise()
{
    Pub p = new Pub();

    p.OnChange += (sender, e)
        => Console.WriteLine("Event raised: {0}", e.Value);

    p.Raise();
}
```

The *Pub* class uses an *EventHandler<MyArgs>*, which specifies the type of the *event* arguments. When raising this *event*, you are required to pass an instance of *MyArgs*. Subscribers to the *event* can access the arguments and use it.

Although the *event* implementation uses a public field, you can still customize addition and removal of subscribers. This is called a *custom event accessor*. Listing 1-85 shows an example of creating a *custom event accessor* for an event.

LISTING 1-85 Custom event accessor

```
public class Pub
{
    private event EventHandler<MyArgs> onChange = delegate { };
    public event EventHandler<MyArgs> OnChange
    {
        add
        {
            lock (onChange)
            {
                onChange += value;
            }
        }
        remove
        {
            lock (onChange)
            {
                onChange -= value;
            }
        }
    }

    public void Raise()
    {
        onChange(this, new MyArgs(42));
    }
}
```

A *custom event accessor* looks a lot like a property with a *get* and *set accessor*. Instead of *get* and *set* you use *add* and *remove*. It's important to put a *lock* around adding and removing subscribers to make sure that the operation is thread safe.

MORE INFO USING LOCK

For more info on when and how to use locking, see the section titled "Objective 1.2: Manage multithreading," earlier in this chapter.

If you use the regular *event* syntax, the compiler generates the accessor for you. This makes it clear that *events* are not *delegates*; instead they are a convenient wrapper around *delegates*.

Delegates are executed in a sequential order. Generally, *delegates* are executed in the order in which they were added, although this is not something that is specified within the Common Language Infrastructure (CLI), so you shouldn't depend on it.

One thing that is a direct result of the sequential order is how to handle exceptions. Listing 1-86 shows an example in which one of the *event* subscribers throws an error.

LISTING 1-86 Exception when raising an event

```
public class Pub
{
    public event EventHandler OnChange = delegate { };
    public void Raise()
    {
        OnChange(this, EventArgs.Empty);
    }
}

public void CreateAndRaise()
{
    Pub p = new Pub();

    p.OnChange += (sender,e )
        => Console.WriteLine("Subscriber 1 called");

    p.OnChange += (sender, e)
        => { throw new Exception(); };

    p.OnChange += (sender,e )
        => Console.WriteLine("Subscriber 3 called");

    p.Raise();
}

// Displays
// Subscriber 1 called
```

As you can see, the first subscriber is executed successfully, the second one throws an exception, and the third one is never called.

If this is not the behavior you want, you need to manually raise the *events* and handle any exceptions that occur. You can do this by using the *GetInvocationList* method that is declared on the *System.Delegate* base class. Listing 1-87 shows an example of retrieving the subscribers and enumerating them manually.

LISTING 1-87 Manually raising events with exception handling

```
public class Pub
{
    public event EventHandler OnChange = delegate { };
    public void Raise()
    {
        var exceptions = new List<Exception>();

        foreach (Delegate handler in OnChange.GetInvocationList())
        {
            try
            {
                handler.DynamicInvoke(this, EventArgs.Empty);
            }
            catch (Exception ex)
            {
                exceptions.Add(ex);
            }
        }

        if (exceptions.Any())
        {
            throw new AggregateException(exceptions);
        }
    }
}

public void CreateAndRaise()
{
    Pub p = new Pub();

    p.OnChange += (sender, e)
        => Console.WriteLine("Subscriber 1 called");

    p.OnChange += (sender, e)
        => { throw new Exception(); };

    p.OnChange += (sender, e)
        => Console.WriteLine("Subscriber 3 called");

    try
    {
        p.Raise();
    }
    catch (AggregateException ex)
    {
        Console.WriteLine(ex.InnerExceptions.Count);
    }
}

// Displays
// Subscriber 1 called
// Subscriber 3 called
// 1
```

MORE INFO **EXCEPTION HANDLING**

For more information on how to work with exceptions, see the section titled "Objective 1.5: Implement exception handling" later in this chapter.



Thought experiment

Building a loosely coupled system

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

You are working on a desktop application that consists of multiple forms. Those

forms show different views of the same data and they should update in real time.

Your application is extensible, and third parties can add plug-ins that contain their own views of the data.

1. Should you use delegates or events in this system?

2. How can this help you?

Objective summary

- Delegates are a type that defines a method signature and can contain a reference to a method.
- Delegates can be instantiated, passed around, and invoked.
- Lambda expressions, also known as anonymous methods, use the => operator and form a compact way of creating inline methods.
- Events are a layer of syntactic sugar on top of delegates to easily implement the publish-subscribe pattern.
- Events can be raised only from the declaring class. Users of events can only remove and add methods to the invocation list.
- You can customize events by adding a custom event accessor and by directly using the underlying delegate type.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. You have a private method in your class and you want to make invocation of the method possible by certain callers. What do you do?
 - A. Make the method public.
 - B. Use an event so outside users can be notified when the method is executed.
 - C. Use a method that returns a delegate to authorized callers.
 - D. Declare the private method as a lambda.
2. You have declared an event on your class, and you want outside users of your class to raise this event. What do you do?
 - A. Make the event public.
 - B. Add a public method to your class that raises the event.
 - C. Use a public delegate instead of an event.
 - D. Use a custom event accessor to give access to outside users.
3. You are using a multicast delegate with multiple subscribers. You want to make sure that all subscribers are notified, even if an exception is thrown. What do you do?
 - A. Manually raise the events by using *GetInvocationList*.
 - B. Wrap the raising of the event in a *try/catch*.
 - C. Nothing. This is the default behavior.
 - D. Let subscribers return *true* or *false* instead of throwing an exception.

Objective 1.5: Implement exception handling

When you build your applications, sometimes errors occur. Maybe you want to write a file to disk, and the disk is full. You try to connect to a database, but the database server is unavailable or another unexpected condition exists. Instead of working with error codes, the .NET Framework uses exceptions to signal errors. You can also use these exceptions to signal errors that happen in your own applications and you can even create custom exception types to signal specific errors.

It's important to know how to work with exceptions so you can implement a well-designed strategy for dealing with or raising errors.

This objective covers how to:

- Handle exceptions.
- Throw exceptions.
- Create custom exceptions.

Handling exceptions

When an error occurs somewhere in an application, an *exception* is raised. *Exceptions* have a couple of advantages compared with error codes. An *exception* is an object in itself that contains data about the error that happened. It not only has a user-friendly message but it also contains the location in which the error happened and it can even store extra data, such as an address to a page that offers some help.

If an *exception* goes unhandled, it will cause the current process to terminate. Listing 1-88 shows an example of an application that throws an error and shuts down.

LISTING 1-88 Parsing an invalid number

```
namespace ExceptionHandling
{
    public static class Program
    {
        public static void Main()
        {
            string s = "NaN";
            int i = int.Parse(s);
        }
    }
}

// Displays
// Unhandled Exception: System.FormatException: Input string was not in a correct format.
//   at System.Number.StringToNumber(String str, NumberStyles options,
//       NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
//   at System.Number.ParseInt32(String s, NumberStyles style,
//       NumberFormatInfo info)
//   at System.Int32.Parse(String s)
//   at ExceptionHandling.Program.Main() in c:\Users\Wouter\Documents\
// Visual Studio 2012\Projects\ExamRefProgrammingInCSharp\Chapter1\Program.cs:line 9
```

The `int.Parse` method *throws* an *exception* of type `FormatException` when the string is not a valid number. *Throwing* an *exception* halts the execution of your application. Instead of continuing to the following line, the runtime starts searching for a location in which you handle the *exception*. If such a location cannot be found, the *exception* is unhandled and will terminate the application.

To handle an *exception*, you can use a *try/catch statement*. Listing 1-89 shows an example of catching the `FormatException`.

LISTING 1-89 Catching a *FormatException*

```
using System;
namespace ExceptionHandling
{
    public static class Program
    {
        public static void Main()
        {
            while (true)
            {
                string s = Console.ReadLine();

                if (string.IsNullOrWhiteSpace(s)) break;

                try
                {
                    int i = int.Parse(s);
                    break;
                }
                catch (FormatException)
                {
                    Console.WriteLine("{0} is not a valid number. Please try again", s);
                }
            }
        }
    }
}
```

You need to surround the code that can potentially throw an exception with a *try* statement. Following the *try* statement, you can add several different *catch blocks*. How much code you put inside each *try block* depends on the situation. If you have multiple statements that can throw the same exceptions that need to be handled differently, they should be in different *try blocks*.

A *catch block* can specify the type of the exception it wants to catch. Because all exceptions in the .NET Framework inherit from *System.Exception*, you can catch every possible *exception* by catching this base type. You can catch more specific *exception* types by adding extra *catch blocks*.

The *catch blocks* should be specified as most-specific to least-specific because this is the order in which the runtime will examine them. When an *exception* is thrown, the first matching *catch block* will be executed. If no matching *catch block* can be found, the exception will fall through. Listing 1-90 shows an example of catching two different *exception* types.

If the *string* *s* is *null*, an *ArgumentNullException* will be thrown. If the *string* is not a number, a *FormatException* will be thrown. By using different *catch blocks*, you can handle those exceptions each in their own way.

LISTING 1-90 Catching different *exception* types

```
try
{
    int i = int.Parse(s);
}
catch (ArgumentNullException)
{
    Console.WriteLine("You need to enter a value");
}
catch (FormatException)
{
    Console.WriteLine("{0} is not a valid number. Please try again", s);
}
```

In C# 1, you could also use a *catch block* without an *exception* type. This could be used to catch exceptions that were thrown from other languages like C++ that don't inherit from *System.Exception* (in C++ you can throw exceptions of any type). Nowadays, each exception that doesn't inherit from *System.Exception* is automatically wrapped in a *System.Runtime.CompilerServices.RuntimeWrappedException*. Since this exception inherits from *System.Exception*, there is no need for the empty catch block anymore.

It's important to make sure that your application is in the correct state when the *catch block* finishes. This could mean that you need to revert changes that your *try block* made before the exception was thrown.

Another important feature of exception handling is the ability to specify that certain code should always run in case of an exception. This can be done by using the *finally block* together with a *try* or *try/catch* statement. The *finally block* will execute whether an exception happens or not. Listing 1-91 shows an example of a *finally block*.

LISTING 1-91 Using a finally block

```
using System;

namespace ExceptionHandling
{
    public static class Program
    {
        public static void Main()
        {
            string s = Console.ReadLine();

            try
            {
                int i = int.Parse(s);
            }
            catch (ArgumentNullException)
            {
                Console.WriteLine("You need to enter a value");
            }
            catch (FormatException)

            {
                Console.WriteLine("{0} is not a valid number. Please try again", s);
            }
            finally
            {

                Console.WriteLine("Program complete.");
            }
        }
    }
}

// Displays
// a
// a is not a valid number. Please try again
// Program complete.
```

Of course, there are still situations in which a *finally block* won't run. For example, when the *try block* goes into an infinite loop, it will never exit the *try block* and never enter the *finally block*. And in situations such as a power outage, no other code will run. The whole operating system will just terminate.

There is one other situation that you can use to prevent a *finally block* from running. Of course, this isn't something you want to use on a regular basis, but you may have a situation in which just shutting down the application is safer than running *finally blocks*.

Preventing the *finally block* from running can be achieved by using *Environment.FailFast*. This method has two different overloads, one that only takes a *string* and another one that also takes an *exception*. When this method is called, the message (and optionally the exception) are written to the Windows application event log, and the application is terminated. Listing 1-92 shows how you can use this method. When you run this application without a debugger attached, a message is written to the event log.

LISTING 1-92 Using *Environment.FailFast*

```
using System;
namespace ExceptionHandling
{
    public static class Program
    {
        public static void Main()
        {
            string s = Console.ReadLine();

            try
            {
                int i = int.Parse(s);
                if (i == 42) Environment.FailFast("Special number entered");
            }
            finally
            {
                Console.WriteLine("Program complete.");
            }
        }
    }
}
```

The line *Program Complete* won't be executed if 42 is entered. Instead the application shuts down immediately.

When you catch an exception, you can use a couple of properties to inspect what's happened. Table 1-3 lists the properties of the base *System.Exception* class.

TABLE 1-3 *System.Exception* properties

Name	Description
<i>StackTrace</i>	A string that describes all the methods that are currently in execution. This gives you a way of tracking which method threw the exception and how that method was reached.
<i>InnerException</i>	When a new exception is thrown because another exception happened, the two are linked together with the <i>InnerException</i> property.
<i>Message</i>	A (hopefully) human friendly message that describes the exception.
<i>HelpLink</i>	A Uniform Resource Name (URN) or uniform resource locator (URL) that points to a help file.
<i>HResult</i>	A 32-bit value that describes the severity of an error, the area in which the exception happened and a unique number for the exception. This value is used only when crossing managed and native boundaries.
<i>Source</i>	The name of the application that caused the error. If the <i>Source</i> is not explicitly set, the name of the assembly is used.
<i>TargetSite</i>	Contains the name of the method that caused the exception. If this data is not available, the property will be null.
<i>Data</i>	A dictionary of key/value pairs that you can use to store extra data for your exception. This data can be read by other catch blocks and can be used to control the processing of the exception.

When using a catch block, you can use both an exception type and a named identifier. This way, you effectively create a variable that will hold the exception for you so you can inspect its properties. Listing 1-93 shows how to do this.

LISTING 1-93 Inspecting an exception

```
using System;

namespace ExceptionHandling
{
    public static class Program
    {
        public static void Main()
        {
            try
            {
                int i = ReadAndParse();
                Console.WriteLine("Parsed: {0}", i);
            }
            catch (FormatException e)
            {
                Console.WriteLine("Message: {0}", e.Message);
                Console.WriteLine("StackTrace: {0}", e.StackTrace);
                Console.WriteLine("HelpLink: {0}", e.HelpLink);
                Console.WriteLine("InnerException: {0}", e.InnerException);
                Console.WriteLine("TargetSite: {0}", e.TargetSite);
                Console.WriteLine("Source: {0}", e.Source);
            }
        }

        private static int ReadAndParse()
        {
            string s = Console.ReadLine();
            int i = int.Parse(s);
            return i;
        }
    }
}

//Displays
//Message: Input string was not in a correct format.
//StackTrace: at System.Number.StringToNumber(String str, NumberStyles options,
// NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
// at System.Number.ParseInt32(String s, NumberStyles style,
// NumberFormatInfo info)
// at System.Int32.Parse(String s)
// at ExceptionHandling.Program.ReadAndParse() in
// c:\Users\Wouter\Documents\Visual Studio 2012\Projects\
// ExamRefProgrammingInCSharp\Chapter1\Program.cs:line 27
// at ExceptionHandling.Program.Main() in c:\Users\Wouter\Documents\
// Visual Studio 2012\Projects\ExamRefProgrammingInCSharp\
// Chapter1\Program.cs:line 10
// HelpLink:
// InnerException:
```

```
// TargetSite: Void StringToNumber(System.String, System.Globalization.NumberStyles
// , NumberBuffer ByRef, System.Globalization.NumberFormatInfo, Boolean)
// Source: mscorlib
```

It's important to make sure that your *finally block* does not cause any exceptions. When this happens, control immediately leaves the *finally block* and moves to the next outer *try block*, if any. The original exception is lost and you can't access it anymore.

You should only catch an exception when you can resolve the issue or when you want to log the error. Because of this, it's important to avoid general catch blocks at the lower layers of your application. This way, you could accidentally swallow an important exception without even knowing that it happened. Logging should also be done somewhere higher up in your application. That way, you can avoid logging duplicate errors at multiple layers in your application.

Throwing exceptions

When you want to throw an error, you first need to create a new instance of an exception. You can then use the special *throw* keyword to throw the exception. After this, the runtime will start looking for *catch* and *finally blocks*.

Listing 1-94 shows how you can throw an exception.

LISTING 1-94 Throwing an *ArgumentNullException*

```
public static string OpenAndParse(string fileName)
{
    if (string.IsNullOrWhiteSpace(fileName))
        throw new ArgumentNullException("fileName", "Filename is required");

    return File.ReadAllText(fileName);
}
```

You should not try to reuse exception objects. Each time you throw an exception, you should create a new one, especially when working in a multithreaded environment, the stack trace of your exception can be changed by another thread.

When catching an exception, you can choose to *rethrow* the exception. You have three ways of doing this:

- Use the *throw* keyword without an identifier.
- Use the *throw* keyword with the original exception.
- Use the *throw* keyword with a new exception.

The first option rethrows the exception without modifying the call stack. This option should be used when you don't want any modifications to the exception. Listing 1-95 shows an example of using this mechanism.

LISTING 1-95 Rethrowing an exception

```
try
{
    SomeOperation();
}
catch (Exception logEx)
{
    Log(logEx);
    throw; // rethrow the original exception
}
```

When you choose the second option, you reset the call stack to the current location in code. So you can't see where the exception originally came from, and it is harder to debug the error.

Using the third option can be useful when you want to raise another exception to the caller of your code.

Say, for example, that you are working on an order application. When a user places an order, you immediately put this order in a message queue so another application can process it.

When an internal error happens in the message queue, an exception of type *MessageQueueException* is raised. To users of your ordering application, this exception won't make any sense. They don't know the internal workings of your module and they don't understand where the message queue error is coming from.

Instead, you can throw another exception, something like a custom *OrderProcessingException*, and set the *InnerException* to the original exception. In your *OrderProcessingException* you can put extra information for the user of your code to place the error in context and help them solve it. Listing 1-96 shows an example. The original exception is preserved, including the stack trace, and a new exception with extra information is added.

LISTING 1-96 Throwing a new exception that points to the original one

```
try
{
    ProcessOrder();
}
catch (MessageQueueException ex)
{
    throw new OrderProcessingException("Error while processing order", ex);
}
```

**EXAM TIP**

Make sure that you don't swallow any exception details when rethrowing an exception. Throw a new exception that points to the original one when you want to add extra information; otherwise, use the *throw* keyword without an identifier to preserve the original exception details.

In C# 5, a new option is added for rethrowing an exception. You can use the *ExceptionDispatchInfo.Throw* method, which can be found in the *System.Runtime.ExceptionServices* namespace. This method can be used to throw an exception and preserve the original stack trace. You can use this method even outside of a catch block, as shown in Listing 1-97.

LISTING 1-97 Using *ExceptionDispatchInfo.Throw*

```
ExceptionDispatchInfo possibleException = null;

try
{
    string s = Console.ReadLine();
    int.Parse(s);
}
catch (FormatException ex)
{
    possibleException = ExceptionDispatchInfo.Capture(ex);
}

if (possibleException != null)
{
    possibleException.Throw();
}

// Displays
// Unhandled Exception: System.FormatException:
// Input string was not in a correct format.
// at System.Number.StringToNumber(String str, NumberStyles options,
//     NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
// at System.Number.ParseInt32(String s, NumberStyles style,
//     NumberFormatInfo info)
// at System.Int32.Parse(String s)
// at ExceptionHandling.Program.Main() in c:\Users\Wouter\Documents\
//     Visual Studio 2012\Projects\ExamRefProgrammingInCSharp\Chapter1\
//     Program.cs:line 17
//--- End of stack trace from previous location where exception was thrown ---
// at System.Runtime.ExceptionServices.ExceptionDispatchInfo.Throw()
// at ExceptionHandling.Program.Main() in c:\Users\Wouter\Documents\
//     Visual Studio 2012\Projects\ExamRefProgrammingInCSharp\Chapter1\
//     Program.cs:line 6
```

When looking at the stack trace, you see this line, which shows where the original exception stack trace ends and the *ExceptionDispatchInfo.Throw* is used:

```
--- End of stack trace from previous location where exception was thrown ---
```

This feature can be used when you want to catch an exception in one thread and throw it on another thread. By using the *ExceptionDispatchInfo* class, you can move the exception data between threads and throw it. The .NET Framework uses this when dealing with the *async/await* feature added in C# 5. An exception that's thrown on an *async* thread will be captured and rethrown on the executing thread.

MORE INFO THREADS AND ASYNC/AWAIT

For more information on working with threads and the new *async/await* feature, see the section titled "Objective 1.1: Implement multithreading and asynchronous processing" earlier in this chapter.

You shouldn't throw exceptions when dealing with expected situations. You know that when users start entering information into your application, they will make mistakes. Maybe they enter a number in the wrong format or forget to enter a required field. Raising an exception for these kinds of expected situations is not recommended.

Exception handling changes the normal expected flow of your program. This makes it harder to read and maintain code that uses exceptions, especially when they are used in normal situations.

Using exceptions also incurs a slight performance hit. Because the runtime has to search all outer catch blocks until it finds a matching block, and when it doesn't, has to look if a debugger is attached, it takes slightly more time to handle. When a real unexpected situation occurs that will terminate the application, this won't be a problem. But for regular program flow, it should be avoided. Instead you should have proper validation and not rely solely on exceptions.

When you need to throw an exception of your own, it's important to know which exceptions are already defined in the .NET Framework. Because developers will be familiar with these exceptions, they should be used whenever possible.

Some exceptions are thrown only by the runtime. You shouldn't use those exceptions from your own code. Table 1-4 lists those exceptions.

TABLE 1-4 Runtime exceptions in the .NET Framework

Name	Description
<i>ArithmeticException</i>	A base class for other exceptions that occur during arithmetic operations.
<i>ArrayTypeMismatchException</i>	Thrown when you want to store an incompatible element inside an array.
<i>DivideByZeroException</i>	Thrown when you try to divide a value by zero.
<i>IndexOutOfRangeException</i>	Thrown when you try to access an array with an index that's less than zero or greater than the size of the array.
<i>InvalidCastException</i>	Thrown when you try to cast an element to an incompatible type.
<i>NullReferenceException</i>	Thrown when you try to reference an element that's null.
<i>OutOfMemoryException</i>	Thrown when creating a new object fails because the CLR doesn't have enough memory available.
<i>OverflowException</i>	Thrown when an arithmetic operation overflows in a checked context.

Name	Description
<i>StackOverflowException</i>	Thrown when the execution stack is full. This can happen in a recursive operation that doesn't exit.
<i>TypeInitializationException</i>	Thrown when a static constructor throws an exception that's goes unhandled.

You shouldn't throw these exceptions in your own applications. Table 1-5 shows popular exceptions in the .NET Framework that you can use.

TABLE 1-5 Popular exceptions in the .NET Framework

Name	Description
<i>Exception</i>	The base class for all exceptions. Try avoiding throwing and catching this exception because it's too generic.
<i>ArgumentException</i>	Throw this exception when an argument to your method is invalid.
<i>ArgumentNullException</i>	A specialized form of ArgumentException that you can throw when one of your arguments is null and this isn't allowed.
<i>ArgumentOutOfRangeException</i>	A specialized form of ArgumentException that you can throw when an argument is outside the allowable range of values.
<i>FormatException</i>	Throw this exception when an argument does not have a valid format.
<i>InvalidOperationException</i>	Throw this exception when a method is called that's invalid for the object's current state.
<i> NotImplementedException</i>	This exception is often used in generated code where a method has not been implemented yet.
<i>NotSupportedException</i>	Throw this exception when a method is invoked that you don't support.
<i>ObjectDisposedException</i>	Throw when a user of your class tries to access methods when Dispose has already been called.

You should avoid directly using the *Exception* base class both when catching and throwing exceptions. Instead you should try to use the most specific exception available.

Creating custom exceptions

Once throwing an exception becomes necessary, it's best to use the exceptions defined in the .NET Framework. But there are situations in which you want to use a *custom exception*. This is especially useful when developers working with your code are aware of those exceptions and can handle them in a more specific way than the framework exceptions.

A custom exception should inherit from *System.Exception*. You need to provide at least a parameterless constructor. It's also a best practice to add a few other constructors: one that

takes a *string*, one that takes both a *string* and an *exception*, and one for *serialization*. Listing 1-98 shows an example of a custom exception.

LISTING 1-98 Creating a custom exception

```
[Serializable]
public class OrderProcessingException : Exception, ISerializable
{
    public OrderProcessingException(int orderId)
    {
        OrderId = orderId;
        this.HelpLink = "http://www.mydomain.com/infoaboutexception";
    }
    public OrderProcessingException(int orderId, string message)
        : base(message)
    {
        OrderId = orderId;
        this.HelpLink = "http://www.mydomain.com/infoaboutexception";
    }

    public OrderProcessingException(int orderId, string message,
                                    Exception innerException)
        : base(message, innerException)
    {
        OrderId = orderId;
        this.HelpLink = "http://www.mydomain.com/infoaboutexception";
    }

    protected OrderProcessingException(SerializationInfo info, StreamingContext context)
    {
        OrderId = (int)info.GetValue("OrderId", typeof(int));
    }

    public int OrderId { get; private set; }

    public void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        info.AddValue("OrderId", OrderId, typeof(int));
    }
}
```

By convention, you should use the *Exception* suffix in naming all your custom exceptions. It's also important to add the *Serializable* attribute, which makes sure that your exception can be serialized and works correctly across application domains (for example, when a web service returns an exception).

When creating your custom exception, you can decide which extra data you want to store. Exposing this data through properties can help users of your exception inspect what has gone wrong.

You should never inherit from *System.ApplicationException*. The original idea was that all C# runtime exceptions should inherit from *System.Exception* and all custom exceptions from *System.ApplicationException*. However, because the .NET Framework doesn't follow this pattern, the class became useless and lost its meaning.



Thought experiment

Planning your error strategy

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

You are designing a new application and you want to implement a proper error-handling strategy. You are discussing the topic with some colleagues, and one of them says that you should use regular error codes to signal errors because that's faster and your company has used it in the past.

You are also having a discussion about when to create a custom exception and when to use the built-in .NET Framework exceptions.

1. Explain to your colleague the advantages of Exceptions compared to error codes.
2. When should you create a custom exception?

Objective summary

- In the .NET Framework, you should use exceptions to report errors instead of error codes.
- Exceptions are objects that contain data about the reason for the exception.
- You can use a try block with one or more catch blocks to handle different types of exceptions.
- You can use a finally block to specify code that should always run after, whether or not an exception occurred.
- You can use the *throw* keyword to raise an exception.
- You can define your own custom exceptions when you are sure that users of your code will handle it in a different way. Otherwise, you should use the standard .NET Framework exceptions.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. You are checking the arguments of your method for illegal *null* values. If you encounter a *null* value, which exception do you throw?

- A. *ArgumentException*.
 - B. *InvalidOperationException*.
 - C. *NullReferenceException*.
 - D. *ArgumentNullException*.
- 2. Your code catches an *IOException* when a file cannot be accessed. You want to give more information to the caller of your code. What do you do?
 - A. Change the message of the exception and rethrow the exception.
 - B. Throw a new exception with extra information that has the *IOException* as *InnerException*.
 - C. Throw a new exception with more detailed info.
 - D. Use throw to rethrow the exception and save the call stack.
- 3. You are creating a custom exception called *LogonFailedException*. Which constructors should you at least add? (Choose all that apply.)
 - A. *LogonFailed()*
 - B. *LogonFailed(string message)*
 - C. *LogonFailed(string message, Exception innerException)*
 - D. *LogonFailed(Exception innerException)*

Chapter summary

- Multithreading can help you create programs that are responsive and scalable. You can use the TPL, the Parallel class, and PLINQ for this. The new *async/await* keywords help you write asynchronous code.
- In a multithreaded environment, it's important to manage synchronization of shared data. You can do this by using the *lock* statement.
- C# offers statements for making decisions—*if*, switch, conditional operator (?) and null-coalescing operator (??)—iterating (*for*, *foreach*, *while*, *do-while*), and *jump* statements (*break*, *continue*, *goto*, *return* and *throw*).
- Delegates are objects that point to a method and can be used to invoke the method. Lambda expressions are a shorthand syntax for creating anonymous methods inline.
- Events are a layer on top of delegates that help you with creating a publish-subscribe architecture.
- Exceptions are the preferred way to work with errors in the .NET Framework. You can throw exceptions, catch them, and run code in a finally block.

Answers

This section contains the solutions to the thought experiments and answers to the lesson review questions in this chapter.

Objective 1.1: Thought experiment

1. Multithreading can improve the responsiveness in a client application. The UI thread can process requests from the user while background threads execute other operations.
2. A CPU-bound operation needs a thread to execute. In a client application, it can make sense to execute a CPU-bound operation on another thread to improve responsiveness. In a server application, you don't want an extra thread for a CPU-bound operation. Asynchronous I/O operations don't require a thread while executing. Using asynchronous I/O frees the current thread to do other work and improves scalability.
3. Using multithreading in a server environment can help you distribute operations over multiple CPUs. This way, you can improve performance. Using the TPL to create another thread to execute a CPU-bound operation while the originating thread has to wait for it won't help you with increasing performance.

Objective 1.1: Review

1. **Correct answer:** B
 - A. **Incorrect:** Manually creating and managing tasks is not necessary. The `Parallel` class takes care of this and uses the optimal configuration options.
 - B. **Correct:** `Parallel.For` is ideal for executing parallel operations on a large set of items that have to do a lot of work.
 - C. **Incorrect:** `async/await` does not process items concurrently. Instead it waits until the current task has finished and then continues executing the code.
 - D. **Incorrect:** The `BlockingCollection` can be used to share data between multiple threads. Using one producer and one consumer thread, however, won't improve scalability. The `Parallel` class is designed for this scenario and should be used.
2. **Correct answer:** A
 - A. **Correct:** `AsParallel` makes a sequential query parallel if the runtime thinks this will improve performance.
 - B. **Incorrect:** `AsSequential` is used to make a parallel query sequential again.
 - C. **Incorrect:** `AsOrdered` is used to make sure that the results of a parallel query are returned in order.
 - D. **Incorrect:** `WithDegreeOfParallelism` is used to specify how many threads the parallel query should use.

3. Correct answer: C

- A. **Incorrect:** Because you have to wait for external factors (the database and web response), you should use `async/await` to free your thread. That way your thread can do some other work while waiting for the external responses to come back.
- B. **Incorrect:** `Async/await` can be used to improve responsiveness on the client but it can also be used in server scenarios. Especially when waiting for an I/O-bound operation, you can use asynchronous code to free the thread from waiting.
- C. **Correct:** The operating system waits for the I/O request to complete and then activates a thread that can process the response. In the meantime, the thread can do other work.
- D. **Incorrect:** `Async/await` does not put your thread to sleep in an I/O-bound situation. Instead, your thread can process other work while the operating system monitors the status of the request. When the request finishes, a thread is used to process the response. With a CPU-bound operation, your thread waits for the operation to finish on another thread.

Objective 1.2: Thought experiment

- 1. It's important to make sure that all locking follows the same order when locking multiple objects. As soon as you start locking dependent objects in different orders, you start getting deadlocks.
- 2. The `Interlocked` class can help you to execute small, atomic operations without the need for locking. When you use locking a lot for these kind of operations, you can replace them with the `Interlocked` statement.

Objective 1.2: Review

1. Correct answer: D

- A. **Incorrect:** You should never lock on this. Another part of your code may already be using your object to execute a lock.
- B. **Incorrect:** You shouldn't use a string for locking. With string-interning, one object can be used for multiple strings, so you would be locking on an object that is also in use in other locations.
- C. **Incorrect:** Locking on a value type will generate a compile error. The value type will be boxed each time you lock on it, resulting in a unique lock each time.
- D. **Correct:** A private lock of type `object` is the best choice.

2. Correct answer: B

- A. **Incorrect:** The *CancellationTokenSource* is used to generate a *CancellationToken*. The token should be passed to the task, and the *CancellationTokenSource* can then be used to request cancellation on the token.
- B. **Correct:** A *CancellationToken* generated by a *CancellationTokenSource* should be passed to the task.
- C. **Incorrect:** A Boolean variable can be used to cancel a task, but it's not the preferred way. A *CancellationToken* offers more flexibility and should be used.
- D. **Incorrect:** The *volatile* keyword should be used to signal to the compiler that the order of reads and writes on a field is important and that the compiler shouldn't change it.

3. Correct answer: B

- A. **Incorrect:** *Volatile.Write* is used to signal to the compiler that writing the value to a field should happen at that exact location.
- B. **Correct:** *CompareExchange* will see whether the current state is correct and it will then change it to the new state in one atomic operation.
- C. **Incorrect:** *Exchange* only changes the value; it doesn't check to see whether the current state is correct.
- D. **Incorrect:** *Decrement* is used to subtract one off the value in an atomic operation.

Objective 1.3: Thought experiment

1. Using the *goto* statement makes your code much harder to read because the application flow jumps around. *goto* is mostly used in looping statements. You can then replace *goto* with *while* or *do-while*.
2. The *switch* statement can be used to improve long *if* statements.
3. The *for* statement can be used to iterate over a collection by using an *index*. You can modify the collection while iterating. You need to use the *index* to retrieve each item. *foreach* is syntactic sugar over the iterator pattern. You don't use an *index*; instead the compiler gives you a variable that points to each iteration item.

Objective 1.3: Review

1. **Correct answer: C**
 - A. **Incorrect:** *switch* is used as a decision statement. You map a value to certain labels to execute specific code; it doesn't iterate over collections.
 - B. **Incorrect:** Although the *foreach* statement can be used to iterate over a collection; it doesn't allow changes to the collection while iterating.
 - C. **Correct:** With *for*, you can iterate over the collection while modifying it. It's your own job to make sure that the *index* stays correct.
 - D. **Incorrect:** *goto* is a *jump* statement that should be avoided.
2. **Correct answer: D**
 - A. **Incorrect:** *for* is an iteration statement that can't be used to check for *null* values.
 - B. **Incorrect:** The conditional operator can be used to shorten *if* statements. It's not useful to conditionally call a method.
 - C. **Incorrect:** The null-coalescing operator does check for *null* values but it's used to provide a default value. It's not useful when calling a method if the value is not *null*.
 - D. **Correct:** Short-circuiting enables you to see whether a value is *null* and call a member on it in one *and* statement. If the left value is *null*, the right operand won't be executed.
3. **Correct answer: C**
 - A. **Incorrect:** A *for* statement is most useful when iterating over a collection in which you know the number of items beforehand.
 - B. **Incorrect:** *foreach* can be used only on types that implement *IEnumerable*. It can't be easily used with your two custom methods.
 - C. **Correct:** You can use *while (o.HasNext) { var i = o.Read(); }* to process the items. When *o.HasNext* returns *false*, you automatically end the loop.
 - D. **Incorrect:** *Do-while* will run the code at least once. If there are no items on the network, the code doesn't have to run.

Objective 1.4: Thought experiment

1. Events are a nice layer on top of delegates that make them easier and safer to use. In this case, you should use events to make sure that other users won't be able to clear all subscriptions. It also makes sure that they can't raise the event on their own. They can only listen to changes.

- The advantage of using an event system in an application like this is that you can achieve loose coupling. Your forms don't have to know anything about each other. The class that monitors data changes and raises the event doesn't have to know how many forms are listening and how they look. Third-party plug-ins can easily subscribe to the events at runtime to be able to respond to changes without tightly coupling to the existing system.

Objective 1.4: Review

1. Correct answer: C

- A. Incorrect:** Making the method public gives access to all users of your class.
- B. Incorrect:** This doesn't give users of your class the ability to execute the method.
- C. Correct:** The method can see whether the caller is authorized and then return a delegate to the private method that can be invoked at will.
- D. Incorrect:** Changing the method to a lambda doesn't change the fact that outside users can't access the method.

2. Correct answer: B

- A. Incorrect:** The compiler restricts the use of events outside of the class where it's defined. They can only add and remove subscribers. Only the class itself can raise the event.
- B. Correct:** The public method can be called by outside users of your class. Internally it can raise the event.
- C. Incorrect:** Using a delegate does allow it to be invoked from outside the class. However, you lose the protection that an event gives you. A public delegate can be completely modified by outside users without any restrictions.
- D. Incorrect:** Canonical name (CNAME) records map an alias or nickname to the real or canonical name that might lie outside the current zone.

3. Correct answer: A

- A. Correct:** You can handle each individual error and make sure that all subscribers are called.
- B. Incorrect:** Wrapping the raising of the event in one *try/catch* will still cause the invocation to stop at the first exception. Later subscribers won't be notified.
- C. Incorrect:** By default, the invocation of subscribers stops when the first unhandled exception happens in one of the subscribers.
- D. Incorrect:** Exceptions are the preferred way of dealing with errors. Returning a value from each event still requires you to invoke them manually one by one to check the return value.

Objective 1.5: Thought experiment

- Exceptions are objects, so they can store extra information that can't be done with only an error code. The .NET Framework also offers special support for dealing with exceptions. For example, you can use catch blocks to handle certain types of exceptions and you can use a finally block to make sure that certain code will always run.
- You should create a custom exception only if you expect developers to handle it or perform custom logging. If a developer won't be able to fix the specific error, it won't make any sense to create a more specific exception. Custom logging can happen when you throw more detailed exceptions, so developers can differentiate between the errors that happen.

Objective 1.5: Review

- Correct answer:** D
 - Incorrect:** Although the exception has to do with an argument to your method, you should throw the more specialized *ArgumentNullException*.
 - Incorrect:** *InvalidOperationException* should be used when your class is not in the correct state to handle a request.
 - Incorrect:** *NullReferenceException* is thrown by the runtime when you try to reference a *null* value.
 - Correct:** *ArgumentNullException* is the most specialized exception that you can use to tell which argument was *null* and what you expect.
- Correct answer:** B
 - Incorrect:** The *Message* property of an exception is read-only. You can't change it after the exception is created.
 - Correct:** The new exception can contain extra info. Setting the *InnerException* makes sure that the original exception stays available.
 - Incorrect:** Throwing a brand-new exception loses the original exception and the information that it had.
 - Incorrect:** Using *throw* without an identifier will rethrow the original exception while maintaining the stack trace, but it won't add any extra information.
- Correct answers:** A, B, C
 - Correct:** You should always add a default empty constructor.
 - Correct:** A second constructor should take a descriptive message of why the error occurred.
 - Correct:** An *InnerException* can be set to correlate two exceptions and show what the original *error* was.
 - Incorrect:** You don't have to define a constructor that only takes an *InnerException* without a message.

Index

A

abstract base classes, 130–131
accessing data. *See* data access
accessing shared data, multithreaded applications, 31–32
access modifiers, 116–120

- internal, 118–120
- private, 117–118
- protected, 118–119
- public, 116

Action type, 147
adding

- behaviors to body of types, 93–95
- continuations to tasks, 12–13
- data to body of types, 96–98

Add method, ConcurrentBag, 27
AddOrUpdate method, ConcurrentDictionary, 29
address, WCF service, 282
Advanced Encryption Standard (AES) algorithm, 195
AES (Advanced Encryption Standard) algorithm, 195
AesManaged class, 195
AggregateExceptions, 19, 24–25
AJAX (Asynchronous JavaScript and XML), 290
algorithms, encryption

- .NET Framework, 195–199
- symmetric and asymmetric encryption, 194–196

AllowMultiple parameter, 142
AND operator, 42
anonymous functions, 60
anonymous methods, 147
anonymous types, LINQ, 295–296
APIs (application programming interfaces), Office automation, 112–113
app.config files, connection strings, 273
application configuration files, 215
application programming interfaces (APIs), Office automation, 112–113

applications

- debugging, 220–230
- assembly management, 209–218
- compiler directives, 222–226
- configurations, 220–222
- diagnostics, 231–244
- encryption, 193–208
- PDB files and symbols, 226–229
- validating application input, 179–191

multithreaded, 2–31

- asynchronous processing, 17–21
- concurrent collections, 25–29
- Parallel class, 16–17
- PLINQ, 21–25
- tasks, 10–16
- threads, 2–10

applying attributes, 139–140
ArgumentException, 190
ArgumentNullException, 71, 186
arguments

- named, 95–96
- optional, 95–96
- overloading, 95–96

Array class, 318
arrays, collections, 318–320
ASCII, 261
as operator, 111–112
AsParallel method, 21–22
assemblies

- internal access modifiers, 118–120
- management, 209–218
 - creating a WinMD assembly, 217–219
 - defining assemblies, 210–211
 - putting an assembly in the GAC, 214–215
 - signing with a strong name, 211–214
 - versioning assemblies, 214–216

AssemblyFileVersionAttribute, 215

AssemblyInfo.cs (class library), 140
AssemblyInfo.cs files, 215
AssemblyVersionAttribute, 215
AsSequential method, 23
asymmetric encryption, 194–196
asynchronous I/O operations, 266–269
Asynchronous JavaScript and XML (AJAX), 290
asynchronous processing, multithreaded applications, 17–21
async keyword, 266–268
async keyword, asynchronous processing, 17–21
Atomicity property (transactions), 278
atomic operations, 29
attributes
 adding metadata to an application, 139–142
 AllowMultiple parameter, 142
 AssemblyFileVersionAttribute, 215
 AssemblyVersionAttribute, 215
 ConditionalAttribute, 225
 DebuggerDisplayAttribute, 226–227
 Flags, enums and, 91
 InternalsVisibleToAttribute, 119
 OnDeserializedAttribute, 312
 OnDeserializingAttribute, 312
 OnSerializedAttribute, 312
 OnSerializingAttribute, 312
 OperationContract, 281
 OptionalFieldAttribute, 312
 SecurityPermission, 314
 SerializableAttribute, 139
 ServiceContract, 281
 ThreadStatic, 7–8
 Trait, 141
 XmlSerializer, 309–310
authentication of users. *See* encryption
auto-implemented properties, 121
automation APIs, 112–113
AverageTimer32 (performance counter), 243
await keyword, 266–268
await keyword, asynchronous processing, 17–21

B

background threads, 5
base classes, 128–133
 abstract, 130–131
 behavior, 129–130
 sealed, 130–131

behaviors
 adding to body of types, 93–95
 base classes, 129–130
BigEndianUnicode, 261
binary serialization, 311–314
BindingFlags enumeration, 144
bindings, WCF service, 282
BlockingCollection collection, 25–26
blueprints, creating types, 98–99
body (creating types), 93–99
 adding behaviors, 93–95
 adding data, 96–98
 blueprints, 98–99
 named and optional arguments, 95–96
Boolean expressions, program flow, 41–44
boxing value types, 107–108
branch instructions, 221
breakpoints, setting in Visual Studio, 221–222
BufferedStream, 263
Build Number (assembly), 215

C

CA (Certificate Authority), 202
calling methods, 94
call stacks, 7, 205
Call Stack window, 228
canceling tasks, 37–40
CancellationToken method, 37–40
CancellationTokenSource method, 38
CAS (code access security), 204
catch blocks, 70
central processing units (CPUs), and threads, 2–3
Certificate Authority (CA), 202
certificates, digital, 202–204
chaining constructors, 99
child classes, 129
child elements, XML documents, 285
child Tasks, 13–14
ciphertext, 194
classes
 AesManaged, 195
 Array, 318
 Console synchronization, 3
 creating base classes, 128–133
 CultureInfo, 186
 DbConnection base, 271

DbConnectionStringBuilder, 272
DbContext, 121
Debug, 232
 design principles, 99–100
Dictionary, 201, 321
Directory, 255
DirectorySecurity, 255–256
DriveInfo, 254
DSACryptoServiceProvider, 197
 dynamic types
 DynamicObject, 113
 ExpandoObject, 113–114
EventLog, 235–237
 exposing a public event, 61–62
File, 258
FileStream asynchronous methods, 19
Hashtable, 201
 Interlocked synchronizing resources for multithread
 applications, 35–36
IXPathNavigable, 288
List, 320
Marshal, 207
OracleConnectionStringBuilder, 272
Path, 259
PerformanceCounter, 242–245
Queue, 323
RSACryptoServiceProvider, 197
SqlConnectionStringBuilder, 272
Stopwatch, 238–241
Stream, 260–261
 strings, 160–161
SymmetricAlgorithm, 197
System.Exception properties, 73–74
System.Linq.ParallelEnumerable, 21
System.MulticastDelegate, 58
Task<T>, 12
Thread, 3–9
Thread.CurrentThread, 9
ThreadLocal<T>, 8–9
TraceSource, 232
TransactionScope, 278–279
 Volatile, synchronizing resources for multithreaded
 applications, 34–35
WebRequest, 265
WebResponse, 265
XElement, 303
XmlDocument, 285, 287–289
XmlReader, 286–287
XmlWriter, 285–286
XPathNavigator, 285
 class hierarchies, 124–137
 base classes, 128–133
 designing and implementing interfaces, 125–127
 .NET Framework standard interfaces, 133–137
 closures, 61
 CLR (common language runtime) type storage, 92
 code access permissions, 204–206
 code access security (CAS), 204
 codebase element, locating assemblies, 216
 code block, if statements, 45
CodeCompileUnit, 145
CodeDOM, 145–146
 Code First approach (Entity Framework), 280
 collections
 BlockingCollection, 25–26
 ConcurrentBag, 27
 ConcurrentDictionary, 29
 ConcurrentQueue, 28
 ConcurrentStack, 28
 data access, 317–325
 arrays, 318–320
 choosing a collection type, 324
 creating custom collections, 324–326
 Dictionary, 321–322
 generic versus nongeneric versions, 319
 List, 319–320
 queues and stacks, 323–324
 sets, 322–323
 FIFO (first in, first out), 28
 iterating across, 50–55
 for each loops, 53–54
 for loop, 50–51
 jump statements, 54
 while and do-while loops, 52–53
 LIFO (last in, first out), 28
 Combine method, 259
 COM (Component Object Model) integration using
 PIA, 112
 COM Interop classes, **IUnknown** interface, 137
 common language runtime (CLR) type storage, 92
CompareExchange method, 36
CompareTo method, 133
 compiler directives, 222–226
CompleteAdding method, 26
 Component Object Model (COM) integration using
 PIA, 112

.ConcurrentBag collection

ConcurrentBag collection, 27
concurrent collections, multithread applications, 25–29
ConcurrentDictionary collection, 29
ConcurrentQueue collection, 28
ConcurrentStack collection, 28
ConditionalAttribute, 140, 225
conditional operators, 47
ConfigurationManager.ConnectionStrings
 property, 273
configurations, debugging applications, 220–222
ConfigureAwait method, 20–21
configuring TraceListener, 234–235
confirming type conversions, 111–112
connected data (ADO.NET), 271
connecting to databases, 271–274
connection pooling, 273–274
connection strings, 271–273
Consistency property (transactions), 278
Console class, synchronization, 3
Console.WriteLine method, 96
constraints, generic type parameter, 102
constructors, 98–100
consuming data
 databases, 271–281
 connecting to, 271–274
 ORM (Object Relational Mapper), 279–280
 parameters, 277–278
 providers, 271
 reading data, 274–276
 transactions, 278–279
 updating data, 276–277
 JSON, 289–290
 web services, 281–283
 XML, 284–289
 .NET Framework, 285
 XMLDocument class, 287–289
 XMLReader class, 286–287
 XMLWriter, 287
consuming types, 107–114
 boxing and unboxing value types, 107–108
 conversions, 108–111
 dynamic types, 112–114
Contains method, 200
content, XML documents, 285
context switching, 3
continuation Tasks, 12–13
ContinueWith method, 12
contracts, WCF service, 282
contravariance, 58
conversions, types, 108–111, 185–187
copying files, 259–260
covariance, 58
CPU sampling option, profiling applications, 240
CPUs (central processing units), and threads, 2–3
CreateAndRaise property, 61
Create method, 265
CreateText method, 261
creating
 anonymous methods, 294
 anonymous types, 295–296
 attributes, 141–142
 base classes, 128–133
 compiler directives, 222–226
 custom collections, 324–326
 custom exceptions, 79–80
 custom struct, 92
 deadlocks, 33–34
 directories, 255
 extension methods, 103–104
 FileStream, 260
 finalizers, 152–153
 generic types, 147
 methods, 94
 properties, 120–121
 threads, 3–4
 types, 89–105
 body, 93–99
 categories, 90
 designing classes, 99–100
 enums, 90–91
 extension methods, 103–105
 generic types, 101–103
 value and reference types, 91–93
 WinMD assembly, 217–219
Critical option (TraceEventType enum), 233
cryptography, 194
C# types
 class hierarchies, 124–137
 consuming, 107–114
 creating, 89–105
 enforcing encapsulation, 116–123
 managing object life cycle, 150–157
 manipulating strings, 158–166
 reflection, 139–148
CultureInfo class, 186
CultureInfo, formatting values, 165

custom collections, creating, 324–326
 custom event accessors, 63
 custom event arguments, 62
 custom exceptions, 79–80
 custom struct, creating, 92
 cyclic references, 307

D

data
 adding to body of types, 96–98
 exporting to Excel, dynamic keyword, 112–113

data access
 collections, 317–325
 arrays, 318–320
 choosing a collection type, 324
 creating custom collections, 324–326
 Dictionary, 321–322
 generic versus nongeneric versions, 319
 List, 319–320
 queues and stacks, 323–324
 sets, 322–323

consuming data, 270–291
 databases, 271–281
 JSON, 289
 web services, 281–284
 XML, 284–289

I/O operations, 253–269
 asynchronous I/O operations, 266–269
 file infrastructure, 254–260
 file system, 263–264
 network communication, 265–266
 streams, 260–263

LINQ, 291–304
 functionality, 300–304
 language features, 292–295
 queries, 296–299

serialization/deserialization, 307–316
 binary serialization, 311–314
 DataContract Serializer, 314–315
 functionality, 307
 JSON serializer, 315–316
 XmlSerializer, 308–311

Database First approach (Entity Framework), 280

databases, 271–281
 connecting to, 271–274
 ORM (Object Relational Mapper), 279–280
 parameters, 277–278

providers, 271
 reading data, 274–276
 transactions, 278–279
 updating data, 276–277

data consumption. *See consuming data*

DataContract Serializer, 314–315

Data Encryption Standard (DES) algorithm, 194

data integrity, validating application input, 180–185

data transfer object (DTO), 307

DbConnection base class, 271

DbConnectionStringBuilder classes, 272

DbContext class, 121

deadlocks, 33–34

Debug class, 232

DebuggerDisplayAttribute, 226

debugging applications, 220–230
 assembly management, 209–218
 creating a WinMD assembly, 217–219
 defining assemblies, 210–211
 putting an assembly in the GAC, 214–215
 signing with a strong name, 211–214
 versioning assemblies, 214–216
 compiler directives, 222–226
 configurations, 220–222
 diagnostics, 231–244
 logging and tracing, 231–237
 performance counters, 241–244
 profiling applications, 238–241

encryption, 193–208
 code access permissions, 204–206
 digital certificates, 202–204
 hashing, 199–202
 .NET Framework, 195–199
 securing string data, 206–208
 symmetric and asymmetric encryption, 194–196

PDB files and symbols, 226–229

validating application input, 179–191
 data integrity, 180–185
 JSON and XML data, 189–191
 methods for converting between types, 185–187
 reasons for validation, 180
 regular expressions, 188–189

Debugging Tools, PDBCopy, 229

debug mode default build configuration, 220

decision-making statements, program flow, 44–49
 conditional operators, 47
 if statement, 44–47
 null-coalescing operators, 47
 switch statements, 48–49

declarative CAS, 205
declaring an array, 318–319
decoding standards, 261–262
decorator pattern, streams, 262–263
Decrement operators, 35–36
decryption, 194
DefaultTraceListener, 234
deferred execution, 300
#define directive, 223
delegates, 57–59
 covariance and contravariance, 58
 multicasting, 58–59
 ParameterizedThreadStart, 6
deleting
 directories, 255
 files, 258
deployment, assemblies, 214–215
Dequeue method, 323
derived classes, 129
Descendants method, 302
DES (Data Encryption Standard) algorithm, 194
deserialization (data), 307–316
design
 classes, 99–100
 interfaces, 125–127
 SOLID principles, 100
diagnostics, 231–244
 logging and tracing, 231–237
 performance counters, 241–244
 profiling applications, 238–241
Dictionary class, 201, 321
Dictionary collection, data access, 321–322
digital certificates, 202–204
directories (files), 255–258
Directory class, 255
DirectoryInfo object, 255
DirectoryNotFoundException, 255
DirectorySecurity class, 255–256
directory tree, building, 256
disconnected data (ADO.NET), 271
displaying drive information, 254
Dispose method, 136, 153, 272
Document Type Definition (DTD) files, 285
domain integrity (data integrity), 181
do-while loops, iterating across collections, 52–53
DriveInfo class, 254
drives (files), 254
DSACryptoServiceProvider class, 197
DTD (Document Type Definition) files, 285

DTO (data transfer object), 307
Durability property (transactions), 278
dynamic keyword, types, 112–114
DynamicObject class, 113

E

else blocks, 46–47
encapsulation, 116–123
 access modifiers, 116–120
 internal, 118–120
 private, 117–118
 protected, 118–119
 public, 116
 explicit interface implementations, 121–122
 properties, 120–121
encoding standards, 261–262
encryption, 193–208
 code access permissions, 204–206
 digital certificates, 202–204
 hashing, 199–202
 .NET Framework, 195–199
 securing string data, 206–208
 symmetric and asymmetric encryption, 194–196
EndsWith method, 162
enforcing encapsulation, 116–123
 access modifiers, 116–120
 internal, 118–120
 private, 117–118
 protected, 118–119
 public, 116
 explicit interface implementations, 121–122
 properties, 120–121
Enqueue method, 28, 323
Entity Framework
 approaches, 280
 DbContext class, 121
 NuGet package, 182
entity integrity (data integrity), 181
EntityValidationErrors property, 184
enumerating a ConcurrentBag collection, 27
enumeration strings, 163
enums
 creating, 90–91
 TraceEventType, 233–234
 TransactionScopeOption, 279
Environment.FailFast method, 72

equality operators, 42
#error directive, 224
error handling
 diagnostics
 performance counters, 241–244
 profiling applications, 238–241
 logging and tracing, 231–237
 validating application input, 179–191
 data integrity, 180–185
 JSON and XML data, 189–191
 methods for converting between types, 185–187
 reasons for validation, 180
 regular expressions, 188–189
Error option (`TraceEventType` enum), 233
errors, exception handling, 68–80
 custom exceptions, 79–80
 throwing exceptions, 75–80
`EventHandler`, 62
`EventHandler<T>`, 62
event ID number, trace methods, 233
event keyword, 62–63
`EventLog` class, 235–237
events, 57–66
 custom accessors, 63
 custom arguments, 62
 delegates, 57–59
 lambda expressions, 59–60
 manually raising, 65–66
 using events, 61–66
Event Viewer, 236–237
Excel, exporting data to using dynamic keyword, 112–113
`ExceptionDispatchInfo.Throw` method, 77–78
exception handling, 68–80
 custom exceptions, 79–80
 throwing exceptions, 75–80
exceptions
 `AggregateExceptions`, 19, 24–25
 `ArgumentException`, 190
 `ArgumentNullException`, 71, 186
 `DirectoryNotFoundException`, 255
 `FileNotFoundException`, 216
 `FormatException`, 69, 186
 inspecting, 74–75
 `InvalidCastException`, 108
 `MessageQueueException`, 76
 `ObjectDisposedException`, 153
 `OperationCanceledException`, 38–39
 `OverflowException`, 187
 popular, 79–80
 runtime, 78–79
 `ThreadAbortException`, 6
 throwing, 75–80
 `UnauthorizedAccessException`, 255
Exclusive OR operator, 42–44
`ExecuteNonQueryAsync` method, 276
`ExecuteNonQuery` method, 276
`ExecutionContext.SuppressFlow` method, 9
execution context (threads), 9
`Exists` method, 255
`ExpandoObject` class, 113–114
explicit cast operators, 302
explicit conversions, types, 109
explicit interface implementations, encapsulation, 121–122
explicit typing, 292
exporting
 data to Excel, dynamic keyword, 112–113
 public keys, 197
exposing public events, 61–62
expression trees, reflection, 147
Extensible Markup Language (XML)
 consuming data, 284–289
 .NET Framework, 285
 `XmlDocument` class, 287–289
 `XMLReader` class, 286–287
 `XMLWriter`, 287
 data validation, 189–191
 LINQ to, 301
extension methods. LINQ, 295
extension methods, types, 103–105

F

fields, 96–97, 120–121
FIFO (first in, first out) collections, 28, 323
`File` class, 258
`FileInfo` object, 258
`FileIOPermission`, 205
`FileNotFoundException`, 216
files
 `app.config`, connection strings, 273
 application configuration, 215
 `AssemblyInfo.cs`, 215
 DTD (Document Type Definition), 285

infrastructure, 254–260
directories, 255–258
drives, 254
manipulating files, 258–259
paths, 259–260
machine configuration, 215
private symbol, 229
public symbol, 229
publisher policy, 215
WinMD (Windows Metadata), 217

FileStream class, asynchronous methods, 19

FileStream, creating, 260

file system, I/O operations, 263–264

finalization, unmanaged resources, 151–156

finalizers
 creating, 152–153
 implementation, 154–155

finally blocks, 71–72

first in, first out (FIFO) collections, 28, 323

Flags attribute, enums and, 91

flow control statements, 44–49
 conditional operators, 47
 if statements, 44–47
 null-coalescing operators, 47
 switch statements, 48–49

ForAll operators, 24–25

for each loops, iterating across collections, 53–54

ForEach method, parallelism, 16–17

foreground threads, 5

foreign keys, 181

for loops, iterating across collections, 50–51

FormatException, 69, 186

formatting strings, 163–166

format, versioning assemblies, 214

For method, parallelism, 16–17

Func<...> delegates, 60

functional construction, 303

functionality
 deserialization (data), 307
 LINQ, 300–304
 creating XML, 303
 LINQ to XML, 301
 querying XML, 301–302
 updating XML, 303–304
 reflection, 143–145
 serialization (data), 307

functions, 93

Func<..> type, 147

G

GAC (global assembly cache), deploying assemblies, 214–215

garbage collection, 150–151

Generation 0, garbage collection, 151–152

generic collections, 319

generic type parameter, creating an interface, 126

generic types
 creating, 147
 support for Nullables, 101–103

get and set accessor, 121

GetConsumingEnumerable method, 26

GetData function, **WeakReference**, 156

GetDirectoryName method, 259

GetEnumerator method, 135, 163

GetExtensions method, 259

GetFileName method, 259

GetHashCode method, 200

GetInvocationList method, 65

GetObjectData method, 314

GetOrAdd method, 29

GetPathRoot method, 259

GetStringAsync method, 18, 267

global assembly cache (GAC), deploying assemblies, 214–215

globally unique identifier (GUID), 227

goto statements, 49

grouping LINQ operation, 299

GUID (globally unique identifier), 227

GZipStream, 262

H

handling errors
 diagnostics
 performance counters, 241–244
 profiling applications, 238–241
 logging and tracing, 231–237
 validating application input, 179–191
 data integrity, 180–185
 JSON and XML data, 189–191
 methods for converting between types, 185–187
 reasons for validation, 180
 regular expressions, 188–189

handling exceptions, 68–80
 custom exceptions, 79–80
 throwing exceptions, 75–80
 hashing, 199–202
 Hashtable class, 201
 heaps (type storage), 92, 150–151
 helper classes, type conversion, 110

I

IComparable interface, 133–135
 IDisposable interface, 136–137, 153
 IEnumerable interface, 134–136
 IEnumerator interface, 134–136
 #if directive, 222
 IFormatProvider interface, 166–167
 IFormattable interface, 111, 166–167
 if statements, 44–47
 immutability, strings, 159
 imperative CAS, 205
 implementation
 data access
 collections, 317–325
 consuming data, 270–291
 I/O operations, 253–269
 LINQ, 291–304
 exception handling, 68–80
 custom exceptions, 79–80
 throwing exceptions, 75–80
 interfaces, 125–127
 program flow, 41–56
 Boolean expressions, 41–44
 decision-making statements, 44–49
 iterating across collections, 50–55
 security
 assembly management, 209–218
 debugging applications, 220–230
 diagnostics, 231–244
 encryption, 193–208
 validating application input, 179–191
 sets, 199–200
 implicit conversions, types, 108
 implicitly typed variables, LINQ, 292–293
 Increment operator, 35–36
 indexers, 97
 IndexOf method, 162

infrastructure, files, 254–260
 directories, 255–258
 drives, 254
 manipulating files, 258–259
 paths, 259–260
 inheritance
 class hierarchies, 124–137
 base classes, 128–133
 designing and implementing interfaces, 125–127
 .NET Framework standard interfaces, 133–137
 Liskov substitution principle, 131
 protected access modifier, 118
 initialization vector (IV), 195
 InnerExceptions property, 25
 innocent users, 180
 input (application), validation, 179–191
 data integrity, 180–185
 JSON and XML data, 189–191
 methods for converting between types, 185–187
 reasons for validation, 180
 regular expressions, 188–189
 input/output operations. *See* I/O operations
 inspecting exceptions, 74–75
 instantiating concrete types, interface implementation, 126
 instantiating delegates, 58
 Instrumentation method, profiling applications, 240
 interfaces
 designing and implementing, 125–127
 encapsulation, 121–122
 IDisposable, 153
 IFormatProvider, 166–167
 IFormattable, 111, 166–167
 IPlugin, 143
 ISerializable, 313
 .NET Framework, 133–137
 Interlocked class, synchronizing resources for multi-threaded applications, 35–36
 internal access modifier, 118–120
 InternalsVisibleToAttribute attribute, 119
 int.Parse method, 69
 InvalidCastException, 108
 Invoke method, parallelism, 16–17
 I/O operations, 253–269
 asynchronous I/O operations, 266–269
 asynchronous processing, 17–21
 file infrastructure, 254–260
 directories, 255–258

IPlugin interface

drives, 254
manipulating files, 258–259
paths, 259–260
file system, 263–264
network communication, 265–266
streams, 260–263
IPlugin interface, 143
IsDefined method, 141
ISerializable interface, 313
Isolation property (transactions), 278
is operator, 111–112
iterating across collections, 50–55
 for each loops, 53–54
 for loop, 50–51
 jump statements, 54
 while and do-while loops, 52–53
iterator pattern, 300
IUnknown interface, 137
IV (initialization vector), 195
IXPathNavigable interface, 288

J

jagged arrays, 318
JavaScript Object Notation (JSON)
 consuming data, 289–290
 data validation, 189–191
JavaScriptSerializer, 190
join operators, 299
JSON (JavaScript Object Notation)
 consuming data, 289–290
 data validation, 189–191
JSON serializer, 315–316
jump statements, iterating across collections, 54

K

key containers, storing encryption keys, 198
keys, algorithms, 194–195
keywords
 async, 266–268
 await, 266–268
 dynamic, 112–114
 sealed, 104–105
 struct, 92–93
var, 292–293
yield, 136

L

lambda expressions, 59–60
LINQ, 294–295
reflection, 147–148
language features, LINQ, 292–295
 anonymous types, 295–296
 extension methods, 295
 implicitly typed variables, 292–293
 lambda expressions, 294–295
 object initialization syntax, 293–294
Language-Integrated Query. *See LINQ*
LastIndexOf method, 162
last-in, first-out (LIFO) collections, 28, 323
LIFO (last-in, first-out) collections, 28, 323
#line directive, 224
LINQ (Language-Integrated Query), 21, 104, 291–304
 functionality, 300–304
 creating XML, 303
 LINQ to XML, 301
 querying XML, 301–302
 updating XML, 303–304
 language features, 292–295
 anonymous types, 295–296
 extension methods, 295
 implicitly typed variables, 292–293
 lambda expressions, 294–295
 object initialization syntax, 293–294
 queries, 296–299
LINQPad, 300
Liskov substitution principle, 131
List class, 320
List collection, data access, 319–320
Listeners property, TraceSource classes, 233
List<T> collection type, 97
lock operators, 32–33
lock statement, synchronization, 32–34
logging strategy, 231–237
long-running tasks, CancellationToken, 37–39
loops, iterating across collections
 do-while, 52–53
 for each loops, 53–54
 for loops, 50–51
 while, 52–53

M

- machine configuration files, 215
- Major Version (assembly), 215
- Makecert.exe tool, 202
- malicious users, 180
- Managed Extensibility Framework (MEF), 143
- management
 - assemblies, 209–218
 - creating a WinMD assembly, 217–219
 - defining assemblies, 210–211
 - putting an assembly in the GAC, 214–215
 - signing with a strong name, 211–214
 - versioning assemblies, 214–216
 - compiler directives, 222–226
 - object life cycle, 150–157
 - garbage collection, 150–151
 - unmanaged resources, 151–156
 - PDB files and symbols, 226–229
 - program flow
 - events and callbacks, 57–66
 - exception handling, 68–80
 - managing multithreading, 31–41
 - multithread applications, 2–31
 - manipulating strings, 158–166
 - classes, 160–161
 - enumeration, 163
 - formatting, 163–166
 - .NET Framework, 159
 - searching fro strings, 162–164
 - manually raising events, 65–66
 - Marshal class, 207
 - MEF (Managed Extensibility Framework), 143
 - memory management, garbage collector. *See* garbage collector
 - MessageQueueException, 76
 - metadata, reflection, 139–148
 - attributes, 139–142
 - CodeDOM, 145–146
 - expression trees, 147
 - functionality, 143–145
 - lambda expressions, 147–148
 - methods
 - adding behaviors to classes, 93–95
 - AddOrUpdate, 29
 - anonymous, 147
 - AsParallel, 21–22
 - AsSequential, 23
 - associated with SecureString, 207
 - Attribute, 302
 - calling, 94
 - CancellationToken, 37–40
 - CancellationTokenSource, 38
 - Combine, 259
 - CompareExchange, 36
 - CompareTo, 133
 - CompleteAdding, 26
 - ConcurrentDictionary collections, 29–30
 - ConfigureAwait, 20–21
 - Contains, 200
 - ContinueWith, 12
 - Create, 265
 - CreateAndRaise, 61
 - CreateText, 261
 - creating, 94
 - Dequeue, 323
 - Descendants, 302
 - EndsWith, 162
 - Enqueue, 28, 323
 - Environment.FailFast, 72
 - ExceptionDispatchInfo.Throw, 77
 - ExecuteNonQuery, 276
 - ExecuteNonQueryAsync, 276
 - ExecutionContext.SuppressFlow, 9
 - Exists, 255
 - ForEach, parallelism, 16–17
 - For, parallelism, 16–17
 - functions versus, 93
 - GetConsumingEnumerable, 26
 - GetDirectoryName, 259
 - GetEnumerator, 135, 163
 - GetExtensions, 259
 - GetFileName, 259
 - GetHashCode, 200
 - GetInvocationList, 65
 - GetObjectData, 314
 - GetOrAdd, 29
 - GetPathRoot, 259
 - GetStringAsync, 18, 267
 - IndexOf, 162
 - int.Parse, 69
 - Invoke, parallelism, 16–17
 - IsDefined, 141
 - LastIndexOf, 162
 - MoveNext, 135, 301
 - MoveTo, 257

overriding, 104–105, 129
Parallel.Break, 17
Parallel.Stop, 17
parameters, 95
Parse, 110
Peek, 323
Pop, 323
Push, 28, 323
PushRange, 28
Raise, 61
Read, 35
ReadAsync, 19
returning data, 96
SignHash, 204
StartsWith, 162
static, 98
string.Concat, 107
Thread.Abort, 6
Thread.Join, 5
t.Join, 7
ToList, 301
ToString, 164
ToXmlString, 197
TryDequeue, 28
TryParse, 110
TryPeek, 27
TryPop, 28
TryPopRange, 28
TryTake, 27
TryUpdate, 29
type conversions, 185–187
VerifyHash, 204
WaitAll, 14–15
WaitAny, 15–16
WhenAll, 15
WithDegreeOfParallelism, 22
WithExecutionMode, 22
Write, 35
WriteAsync, 19, 267
XmlWriter.Create, 161
Minor Version (assembly), 215
Model First approach (Entity Framework), 280
Modules window, 227
MoveNext method, 135, 301
MoveTo method, 257
moving files, 258
multicasting delegates, 58–59
multidimensional arrays, 318
multiple inheritance, 127
multithreaded applications, 2–31. *See also* parallelism
asynchronous processing, 17–21
concurrent collections, 25–29
Parallel class, 16–17
PLINQ, 21–25
synchronizing resources
 Interlocked class, 35–36
 Volatile class, 34–35
tasks, 10–16
threads, 2–10
 Thread class, 3–9
 thread pools, 9–10

N

named arguments, 95–96
namespaces
 System.Threading
 Parallel class, 16–17
 Thread class, 3–9
ADO.NET, data access code, 271
.NET Framework
 binary serialization, 311–314
 built-in types, 185–186
 consuming XML, 285
 creating web services, 281–283
 data access code, 271
 database providers, 271
 DataContract, 314–315
 dynamic types, classes, 113–114
 encryption, 195–199
 Nullables, 101–102
 standard interfaces, 133–137
 strings, 159
 TraceListeners, 234
 transactions, 278–279
 XmlSerializer, 308–311
.NET memory allocation method, profiling applications, 240
network communication, I/O operations, 265–266
Newtonsoft.Json, 290
nonatomic operations, 36–37
nongeneric collections, 319
NonSerialized attribute, 312
no-operation (NOP) instructions, 221
NOP (no-operation) instructions, 221

NuGet, 182
 Nullables, 101–102
 null-coalescing operators, 47
 NumberOfItems32 (performance counter), 243
 NumberOfItems64 (performance counter), 243

O

object constructors, 98–100
 ObjectDisposedException, 153
 object initialization syntax, LINQ, 293–294
 object life cycle, management, 150–157
 garbage collection, 150–151
 unmanaged resources, 151–156
 object-relational impedance mismatch, 279–280
 Object Relational Mapper (ORM), 279–280
 objects
 cyclic references, 307
 DirectoryInfo, 255
 encapsulation. *See* encapsulation
 FileInfo, 258
 ParallelLoopState, 17
 SqlCommand, 274
 XmlNode, 287
 Office automation APIs, 112–113
 OnChange property, 61
 OnDeserializedAttribute, 312
 OnDeserializingAttribute, 312
 OnSerializedAttribute, 312
 OnSerializingAttribute, 312
 OperationCanceledException, 38–39
 OperationContract attribute, 281
 operators
 AND, 42
 as, 111–112
 Decrement, 35–36
 equality, 42
 Exclusive OR, 42–44
 explicit cast, 302
 Increment, 35–36
 is, 111–112
 OR, 42
 OrderBy, 302
 standard LINQ query operators, 297–299
 Where, 302
 optional arguments, 95–96
 OptionalFieldAttribute, 312

OracleConnectionStringBuilder class, 272
 OrderBy operator, 302
 ordered parallel queries, 22–23
 OrderProcessingException, 76
 ORM (Object Relational Mapper), 279–280
 OR operators, 42
 OverflowException, 187
 overloading visibility, 95–96
 overriding methods, 104–105, 129

P

paging LINQ operation, 299
 parallel asynchronous operations, 268
 Parallel.Break method, 17
 Parallel class, multithread applications, 16–17
 parallelism. *See also* multithreaded applications
 ForEach method, 16–17
 For method, 16–17
 Invoke method, 16–17
 Parallel Language-Integrated Query (PLINQ), 21–25
 ParallelLoopState object, 17
 parallel queries
 converting queries to, 21–22
 making sequential, 23
 ordered, 23–24
 unordered, 22
 Parallel.Stop method, 17
 parameterized SQL statements, 277–278
 ParameterizedThreadStart delegate, 6
 parameters
 databases, 277–278
 methods, 95
 parent classes, 129
 parent Tasks, attaching child Tasks to, 13–14
 Parse method, 110
 partial signing, assemblies, 213
 Path class, 259
 paths, files, 259–260
 PDBCopy tool, 229
 PDB (program database) files, 226–229
 Peek method, 323
 performance
 counters, 241–244
 symmetric and asymmetric encryption, 194
 PerformanceCounter class, 242–245
 Performance Monitor, 241

Performance Wizard, 239
permissions, code access, 204–206
PIA (Primary Interop Assembly), integrating with COM, 112
PKI (Public Key Infrastructure), 202
PLINQ (Parallel Language-Integrated Query), 21–25
pointer types, 90
pooling, connection, 273–274
Pop method, 323
popular exceptions, 79–80
#pragma warning directive, 224
preprocessor directives, 222
Primary Interop Assembly (PIA), integrating with COM, 112
primary keys, 181
private access modifier, 117–118
private algorithms, 194
private keys, strong-named assemblies, 211
private symbol files, 229
probing section, locating assemblies, 216–217
procedural code, updating XML, 304–305
profiling applications, 238–241
profiling tools, 239
program flow
 events and callbacks, 57–66
 delegates, 57–59
 lambda expressions, 59–60
 using events, 61–66
exception handling, 68–80
 custom exceptions, 79–80
 throwing exceptions, 75–80
implementation, 41–56
 Boolean expressions, 41–44
 decision-making statements, 44–49
 iterating across collections, 50–55
multithreaded applications, 2–41
 asynchronous processing, 17–21
 canceling tasks, 37–40
 concurrent collections, 25–29
 Parallel class, 16–17
 PLINQ, 21–25
 synchronizing resources, 31–37
 tasks, 10–16
 threads, 2–10
projection LINQ operation, 299
prolog, 284
properties
 auto-implemented, 121
 CreateandRaise, 61

creating, 120–121
enforcing encapsulation, 120–121
InnerExceptions, 25
OnChange, 61
System.Exception class, 73–74
protected access modifier, 118–119
protecting applications. *See* security providers, databases, 271
public access modifier, 116
public algorithms, 194
public events. exposing, 61–62
Public Key Infrastructure (PKI), 202
public keys
 exporting, 197
 strong-named assemblies, 211
public symbol files, 229
publisher policy files, 215
publish-subscribe design pattern, 61–64
Push method, 28, 323
PushRange method, 28

Q

queries
 converting to parallel queries, 21–22
 databases, 274–276
 LINQ, 296–299
 LINQ (Language-Integrated Query), 21
 PLINQ (Parallel Language-Integrated Query), 21–25
query syntax, 296
Queue class, 323
queues
 data access collections, 323–324
 FIFO (first in, first out) collections, 28
 queuing work to the thread pool, 9–10

R

Raise method, 61
raising events
 exceptions, 64
 manually, 65–66
RanToCompletion state, 38
RateOfCountsPerSecond32 (performance counter), 243
RateOfCountsPerSecond64 (performance counter), 243
ReadAsync method, 19

reading
 attributes, 140–141
 data, databases, 274–276
 from a stream, 260
Read method, 35
reference types, creating, 91–93
referential integrity (data integrity), 181
reflection, 139–148
 attributes, 139–142
 CodeDOM, 145–146
 expression trees, 147
 functionality, 143–145
 lambda expressions, 147–148
ReflectionPermission, 205
regex. See **regular expressions**
regular assemblies, 211, 211–214
regular expressions, validating application input, 188–189
release build, 221
release mode default build configuration, 220
Repository base class, 128
resource contention data method, profiling applications, 240
resource synchronization, multithreading, 31–37
 Interlocked class, 35–36
 Volatile class, 34–35
Resume option (**TraceEventType** enum), 233
rethrowing exceptions, 75
returning a value (tasks), 12
returning data, methods, 96
Revision (assembly), 215
rights, code access permissions, 204
RSACryptoServiceProvider class, 197
runtime exceptions, 78–79
runtime, reflection, 139–148
 attributes, 139–142
 CodeDOM, 145–146
 expression trees, 147
 functionality, 143–145
 lambda expressions, 147–148

S

scalable applications, 19
sealed base classes, 130–131
sealed keyword, 104–105
searching strings, 162–164

search pattern, wildcards, 257
SecureString, 206–207
security
 assembly management, 209–218
 creating a WinMD assembly, 217–219
 defining assemblies, 210–211
 putting an assembly in the GAC, 214–215
 signing with a strong name, 211–214
 versioning assemblies, 214–216
debugging applications, 220–230
 compiler directives, 222–226
 configurations, 220–222
 PDB files and symbols, 226–229
diagnostics, 231–244
 logging and tracing, 231–237
 performance counters, 241–244
 profiling applications, 238–241
encryption, 193–208
 code access permissions, 204–206
 digital certificates, 202–204
 hashing, 199–202
 .NET Framework, 195–199
 securing string data, 206–208
 symmetric and asymmetric encryption, 194–196
 validating application input, 179–191
 data integrity, 180–185
 JSON and XML data, 189–191
 methods for converting between types, 185–187
 reasons for validation, 180
 regular expressions, 188–189
SecurityPermission, 205
SecurityPermission attribute, 314
seeking, streams, 260
sequential parallel queries, 23
SerializableAttribute attribute, 139
serialization (data), 307–316
 binary serialization, 311–314
 DataContract Serializer, 314–315
 functionality, 307
 JSON serializer, 315–316
 XmlSerializer, 308–311
ServiceContract attribute, 281
set-based collections, 324
set implementation, 199–200
sets, data access collections, 322–323
SHA256Managed algorithm, 201
shared data, accessing in multithreaded applications, 31–32

short-circuiting OR operators

short-circuiting OR operators, 42–43
side-by-side hosting, 215
SignHash method, 204
signing data, digital certificates, 203–204
Signing page (Visual Studio), 211
single-dimensional arrays, 318
Skip operator, 299
SOLID principles, 100
SqlCommand, 274
SqlCommand object, 274
SqlConnection, 271
SqlConnectionStringBuilder class, 272
SqlDataReader, 275
SQL injection, 277
SQL script, creating tables, 274
stacks, 92
 data access collections, 323–324
 garbage collection, 150–151
 LIFO (last in, first out) collections, 28
standard interfaces, .NET Framework, 133–137
standard LINQ query operators, 297–299
starting tasks, 11
Start option (TraceEventType enum), 233
StartsWith method, 162
static methods, 98
Stop option (TraceEventType enum), 233
stopping a thread, 6–7
Stopwatch class, 238–241
storage
 asymmetric keys, 198–199
 types, 92
 value types, 92, 150–151
Stream class, 260–261
StreamReader, 262
streams, 260–263
 base Stream class, 260–261
 decorator pattern, 262–263
 encoding and decoding, 261–262
StringBuilder class, 160
string.Concat method, 107
string data, securing, 206–208
StringReader class, 161
strings, 158–166
 application input, 185
 classes, 160–161
 enumeration, 163
 formatting, 163–166
.NET Framework, 159
searching, 162–164

StringWriter class, 161
strong-named assemblies, 211–214
Strong Name tool, 212
struct keyword, 92
Suspend option (TraceEventType enum), 233
switch statements, 48–49
Symbol Server, PDB files, 228
SymmetricAlgorithm class, 197
symmetric encryption, 194–196
synchronization, Console class, 3
SynchronizationContext, asynchronous code, 20
synchronizing resources, multithreading, 31–37
 Interlocked class, 35–36
 lock statement, 32–34
 Volatile class, 34–35
synchronous code, 266
System.BitConverter helper class, 110
System.Data strong-named assembly, 212
System.Exception class, properties, 73–74
System.Linq.ParallelEnumerable class, 21
System.MulticastDelegate class, 58
System.Threading namespace
 Parallel class, 16–17
 Thread class, 3–9

T

Take operator, 299
TaskFactory, 14
Tasks
 canceling, 37–40
 setting a timeout, 39
task scheduler, 11
Tasks, multithread applications, 10–16
 attaching child tasks to parent tasks, 13–14
 continuation, 12–13
 TaskFactory, 14
 WaitAll method, 14–15
 WaitAny method, 15–16
 WhenAll method, 15
Task<T> class, 12
Task.WaitAll method, 14–15
Team Foundation Server (TFS), 229
TextWriterTraceListener, 234
TFS (Team Foundation Server), 229
ThreadAbortException, 6
Thread.Abort method, 6

Thread class, 3–9
Thread.CurrentThread class, 9
Thread.Join method, 5
ThreadLocal<T> class, 8–9
 thread pools, 9–10
 threads, 2–10

- Thread class, 3–9
- thread pools, 9–10

ThreadStatic attribute, 7–8
 throwing exceptions, 75–80
 timeout, Tasks, 39
t.Join method, 7
ToList method, 301
 tools

- Makecert.exe**, 202
- PDBCopy**, 229
- profiling, 239
- Strong Name, 212
- XML Schema Definition, 190

ToString method, 164
ToXmlString method, 197
TraceEventType enum, 233–234
TraceListeners, 233–234
TraceSource class, 232
 tracing strategy, 231–237
Trait attribute, 141
TransactionScope class, 278–279
TransactionScopeOption enum, 279
 transactions, databases, 278–279
 transactions, data integrity, 185
 Transfer option (**TraceEventType** enum), 233
 transformation, 303
 triggers, 182
 try/catch statements, handling exceptions, 69–70
TryDequeue method, 28
TryParse method, 110
TryPeek method, 27
TryPop method, 28
TryPopRange method, 28
TryTake method, 27
TryUpdate method, 29
 type parameters, constraints, 102
 types

- class hierarchies, 124–137
- base classes, 128–133
- designing and implementing interfaces, 125–127
- .NET Framework standard interfaces, 133–137

 consuming, 107–114
 boxing and unboxing value types, 107–108
 conversions, 108–111
 dynamic types, 112–114
 conversions, 185–187
 creating in C#, 89–105

- body, 93–99
- categories, 90
- designing classes, 99–100
- enums, 90–91
- extension methods, 103–105
- generic types, 101–103
- value and reference types, 91–93

 enforcing encapsulation, 116–123

- access modifiers, 116–120
- explicit interface implementations, 121–122
- properties, 120–121

 generic, creating, 147
 managing object life cycle, 150–157

- garbage collection, 150–151
- unmanaged resources, 151–156

 manipulating strings, 158–166

- classes, 160–161
- enumeration, 163
- formatting, 163–166
- .NET Framework, 159
- searching fro strings, 162–164

 reflection, 139–148

- attributes, 139–142
- CodeDOM, 145–146
- expression trees, 147
- functionality, 143–145
- lambda expressions, 147–148

 storage, 92

U

UnauthorizedAccessException, 255
 unboxing value types, 107–108
#undef directive, 224
Unicode, 261
Unicode Consortium, 261
 uniform resource locator (URL), 91
UnitTestAttribute, 142
 unmanaged resources, finalization, 151–156
 unordered parallel queries, 22
 updating data, databases, 276–277

updating XML, LINQ, 303–304
URL (uniform resource locator), 91
User Account Control. *See* UAC
user-defined conversions, types, 109
user-defined integrity (data integrity), 181
users
 authentication. *See* encryption
 validating application input, 180
UTF7 encoding standard, 261
UTF8 encoding standard, 261
UTF32 encoding standard, 261

V

validation
 application input, 179–191
 data integrity, 180–185
 JSON and XML data, 189–191
 methods for converting between types, 185–187
 reasons for validation, 180
 regular expressions, 188–189
 type conversions, 111–112
ValidationEventHandler, 192
value types
 boxing and unboxing, 107–108
 creating, 91–93
 enums, 90–91
 storage, 92, 150–151
var keyword, 292–293
Verbose option (TraceEventType enum), 233
VerifyHash method, 204
verifying data, digital certificates, 203–204
versioning assemblies, 214–216
Visual Studio
 Performance Wizard, 239
 setting breakpoints, 221–222
 Signing page, 211
Visual Studio debugger, asynchronous code, 19
Volatile class, synchronizing resources for multithreaded applications, 34–35

W

WaitAny method, 15–16
#warning directive, 224
Warning option (TraceEventType enum), 233
WCF (Windows Communication Foundation), creating services, 281–283
WeakReference, 155
WebRequest class, 265
WebResponse class, 265
web services, data storage, 281–283
WhenAll method, 15
Where operator, 302
while loops, iterating across collections, 52–53
wildcards, search patterns, 257
windows
 Call Stack, 228
 Modules, 227
Windows Communication Foundation (WCF) services, 281–283
Windows Metadata (WinMD) files, 217
Windows Runtime component, 218–219
WinMD assembly, creating, 217–219
WinMD (Windows Metadata) files, 217
WinRT runtime, 217
WithDegreeOfParallelism method, 22
WithExecutionMode method, 22
wizards, Performance, 239
wrapper classes, IUnknown interface, 137
WriteAsync method, 19, 267
Write method, 35
writing from a stream, 260

X

X.509 certificates, 202
 XElement class, 303
 XElement constructor, 303
 XmlArray attribute, 309
 XmlArrayItem attribute, 309
XmlAttribute, 309
 XmlDocument class, 285, 287–289
XmlElement attribute, 309

XML (Extensible Markup Language)
 consuming data, 284–289
 .NET Framework, 285
 `XMLDocument` class, 287–289
 `XMLReader` class, 286–287
 `XMLWriter`, 287
 data validation, 189–191
 LINQ to, 301
`XmIgnore` attribute, 309
`XmlNode` objects, 287
`XmlReader` class, 285, 286–287
XML Schema Definition Tool, 190
`XmlSerializer`, 308–311
`XmlWriter` class, 285, 287
`XmlWriter.Create` method, 161
XOR (Exclusive OR) operators, 43–44
XPath, 288
`XPathNavigator` class, 285

Y

`yield` keyword, 136
`yield` statements, 300

About the author



WOUTER DE KORT is an independent technical coach, trainer, and developer at Seize IT. He is MCSD certified. As a software architect, he has directed the development of complex web applications. He has also worked as a technical evangelist, helping organizations stay on the cutting edge of web development. Wouter has worked with C# and .NET since their inception; his expertise also includes Visual Studio, Team Foundation Server, Entity Framework, Unit Testing, design patterns, ASP.NET, and JavaScript.



Now that
you've
read the
book...

Tell us what you think!

Was it useful?

Did it teach you what you wanted to learn?

Was there room for improvement?

Let us know at <http://aka.ms/tellpress>

Your feedback goes directly to the staff at Microsoft Press,
and we read every one of your responses. Thanks in advance!



Microsoft