Praise for
Sams Teach Yourself Visual C# 2010 in 24 Hours

“The Teach Yourself in 24 Hours series of books from Sams has been a staple of anyone wanting to quickly come up-to-speed on a new technology. This book is not just a simple refresh of last year’s book, Scott has written it from the ground up for the Visual Studio 2010 and .NET 4.0 release. From the C# type system, to events and data, from ASP.NET Web to WPF Windows applications, Sams Teach Yourself Visual C# 2010 in 24 Hours will provide any developer new to the C# language a great foundation to build upon.”
—Shawn Weisfeld, Microsoft Visual C# MVP

“The key to learning software development is to have a great foundation. Sams Teach Yourself Visual C# 2010 in 24 Hours is a must-read for anyone who wants to learn C# from the beginning, or just brush up on its features. Scott Dorman brings a very knowledgeable, yet casual approach to his book that anyone with the desire to learn to program in .NET can be inspired by. I found a few gems that will enhance my future programming projects.”
—Chris “Woody” Woodruff, Co-Host of Deep Fried Bytes Podcast

“This book is an excellent resource for anyone who is learning C# for the first time, migrating from Visual Basic, or catching up on the latest features of C#. It is full of information and should be on the desks of any developer who is becoming familiar with C# 2010.”
—Jeff Julian, Managing Partner, AJI Software, Founder of GeeksWithBlogs.NET
“Scott Dorman has written an excellent reference book that not only covers the basic fundamentals of .NET 4.0 C# development, but also includes instruction and guidance on the finer points of advanced C# and development with Visual Studio 2010.

The book is written in a clear and concise manner, with liberal usage of ‘Did You Know,’ ‘By the Way,’ and ‘Watch Out!’ sidebars that help provide the reader with informative ‘sign posts’ along their journey for re-enforcing key concepts, best practices, and anti-patterns. These invaluable sign posts really help to ‘bring-it-home’ to the reader with Scott’s real-world commentary about why certain topics are critical in the overall understanding and use of the C# language and associated constructs.

Whether you are a novice, intermediate, or professional developer, this book will certainly become a very handy, well-thumbed, desk reference for today’s highly productive .NET 4.0 C# developer.”

—Jeff Barnes, Architect Microsoft Developer & Platform Evangelism, Microsoft Corporation

“This book covers all the bases, from the C# language, through the frameworks you’ll use it with and the tools you need to be productive. The best way to learn is to do, and there is no shortage of doing here.”

—Chris Burrows, C# Compiler Team, Microsoft Corporation

“Sams Teach Yourself Visual C# 2010 in 24 Hours gives you the jump start you need to be productive quickly. I found the book extremely clear to follow and laid out logically hour by hour to flow you through related topics. From novices to C# veterans, this book gives you all you need to understand all that is new in the 2010 release.”

—Richard Jones, Microsoft MVP
Contents at a Glance

Introduction ................................................................................................................. 1

Part I: C# Fundamentals

HOUR 1  The .NET Framework and C# ................................................................. 7
2  Understanding C# Types .................................................................................. 35
3  Understanding Classes and Objects the C# Way .............................................. 63
4  Inheritance, Interfaces, and Abstract Classes ................................................. 93
5  Creating Enumerated Types and Structures ................................................... 113
6  Events and Event Handling ........................................................................... 131

Part II: Programming in C#

HOUR 7  Controlling Program Flow ...................................................................... 147
8  Using Strings and Regular Expressions ........................................................... 167
9  Working with Arrays and Collections .............................................................. 195
10 Handling Errors Using Exceptions .................................................................... 227
11 Understanding Generics ................................................................................... 245
12 Understanding Query Expressions ................................................................... 267

Part III: Working with Data

HOUR 13 Using Files and Streams ......................................................................... 289
14 Working with XML ............................................................................................ 311
15 Working with Databases .................................................................................... 329

Part IV: Building an Application Using Visual Studio

HOUR 16 Debugging in Visual Studio ..................................................................... 347
17 Building a Windows Application ....................................................................... 363
18 Using Data Binding and Validation .................................................................... 385
19 Building a Web Application ............................................................................... 407
Part V: Diving Deeper

HOUR 20 Programming with Attributes ................................................... 427
21 Dynamic Types and Language Interoperability .................................. 439
22 Memory Organization and Garbage Collection .................................. 451
23 Understanding Threads, Concurrency, and Parallelism .................... 461
24 Next Steps: Silverlight, PowerShell, and Entity Framework ............. 479
Index .................................................................................................. 485
# Table of Contents

Introduction ................................. 1

- Audience and Organization .................................................. 2
- Conventions Used in This Book ................................................. 3
- Closing Thoughts ......................................................................... 3

## Part I C# Fundamentals

**HOUR 1 The .NET Framework and C#** ........................................ 7

- The .NET Framework ................................................................. 7
- The C# Language ........................................................................ 17
- Visual Studio 2010 ...................................................................... 24
- Writing Your First Program ...................................................... 27
- Q&A ................................................................................................. 31
- Workshop ..................................................................................... 32

**HOUR 2 Understanding C# Types** ........................................... 35

- Types .............................................................................................. 36
- Predefined Types .......................................................................... 37
- Operators ........................................................................................ 47
- Default Values .............................................................................. 53
- Nullable and Nullable Types ...................................................... 53
- Casting and Conversion ............................................................... 55
- Q&A ................................................................................................. 59
- Workshop ..................................................................................... 60

**HOUR 3 Understanding Classes and Objects the C# Way** ............. 63

- Object-Orientated Programming ................................................. 64
- Component-Oriented Programming ............................................. 65
- Classes in C# ............................................................................... 65
- Scope and Declaration Space ...................................................... 66
HOUR 4 Inheritance, Interfaces, and Abstract Classes

Inheritance and Polymorphism
Abstract Classes and Members
Interfaces
Q&A
Workshop

HOUR 5 Creating Enumerated Types and Structures

Enumerated Types
Structures
Q&A
Workshop

HOUR 6 Events and Event Handling

Understanding Events
Subscribing and Unsubscribing
Publishing an Event
Raising an Event
Q&A
Workshop

Part II Programming in C#

HOUR 7 Controlling Program Flow

Selection Statements
Iteration Statements
Jump Statements
# Contents

Q&A .............................................................................................. 162
Workshop ...................................................................................... 163

**HOUR 8** Using Strings and Regular Expressions 167

- **Strings** ................................................................. 168
- **Mutable Strings Using StringBuilder** .......... 177
- **Type Formatting** ................................................. 178
- **Regular Expressions** ............................................. 187
- Q&A ........................................................................... 190
- Workshop .................................................................... 191

**HOUR 9** Working with Arrays and Collections 195

- **Arrays** ................................................................. 196
- **Indexers** ............................................................ 200
- **Generic Collections** ............................................ 203
- **Collection Initializers** ........................................... 217
- **Collection Interfaces** ............................................ 219
- **Enumerable Objects and Iterators** ..................... 220
- Q&A ........................................................................... 223
- Workshop .................................................................... 224

**HOUR 10** Handling Errors Using Exceptions 227

- **Understanding Exceptions** .................................. 228
- **Throwing Exceptions** .......................................... 231
- **Handling Exceptions** ............................................ 232
- **Rethrowing Caught Exceptions** ......................... 239
- **Overflow and Integer Arithmetic** ....................... 241
- Q&A ........................................................................... 243
- Workshop .................................................................... 243

**HOUR 11** Understanding Generics 245

- **Why You Should Use Generics** ......................... 246
- **Using Generic Methods** ..................................... 253
- **Creating Generic Classes** ................................. 254
Part IV Building an Application Using Visual Studio

HOUR 16 Debugging in Visual Studio 347
Commenting Your Code ................................................................. 348
Compiler and Runtime Errors ......................................................... 349
Debugging in Visual Studio .............................................................. 350
Visualizing Data ............................................................................ 359
Q&A .............................................................................................. 361
Workshop ...................................................................................... 361

HOUR 17 Building a Windows Application 363
Understanding WPF ......................................................................... 364
Creating a WPF Application ............................................................. 370
Styling the Layout .......................................................................... 379
Q&A .............................................................................................. 382
Workshop ...................................................................................... 382

HOUR 18 Using Data Binding and Validation 385
Understanding Data Binding .............................................................. 386
Converting Data .............................................................................. 390
Binding to Collections ..................................................................... 395
Working with Data Templates .......................................................... 399
Validating Data ............................................................................... 400
Q&A .............................................................................................. 404
Workshop ...................................................................................... 405

HOUR 19 Building a Web Application 407
Understanding Web Application Architecture ................................. 408
Working with ASP.NET ..................................................................... 408
Creating a Web Application ............................................................. 411
Understanding Data Validation ......................................................... 420
Q&A .............................................................................................. 423
Workshop ...................................................................................... 424
Part V Diving Deeper

HOUR 20  Programming with Attributes

Understanding Attributes ................................................................. 428
Working with the Common Attributes ............................................. 430
Using Custom Attributes ................................................................. 433
Accessing Attributes at Runtime ...................................................... 434
Q&A ........................................................................................... 436
Workshop ..................................................................................... 436

HOUR 21  Dynamic Types and Language Interoperability

Using Dynamic Types ....................................................................... 439
Understanding the DLR .................................................................... 444
Interoperating with COM ................................................................. 447
Reflection Interoperability ................................................................ 448
Q&A ........................................................................................... 449
Workshop ..................................................................................... 450

HOUR 22  Memory Organization and Garbage Collection

Memory Organization .......................................................................... 452
Garbage Collection .......................................................................... 452
Understanding IDisposable .............................................................. 453
Using the Dispose Pattern ................................................................. 455
Declaring and Using Finalizers ......................................................... 456
Q&A ........................................................................................... 458
Workshop ..................................................................................... 459

HOUR 23  Understanding Threads, Concurrency, and Parallelism

Understanding Threads and Threading ............................................. 462
Concurrency and Synchronization ..................................................... 463
Understanding the Task Parallel Library .......................................... 467
Working with Parallel LINQ (PLINQ) ............................................... 472
Potential Pitfalls ............................................................................... 473
Q&A ........................................................................................... 475
Workshop ..................................................................................... 476
HOUR 24 Next Steps: Silverlight, PowerShell, and Entity Framework 479

Understanding the Entity Framework 479
Introducing PowerShell 482
Silverlight 483
Index 485
This page intentionally left blank
Foreword

Over a decade ago, a small team of designers met in a small conference room on the second floor of Building 41 at Microsoft to create a brand-new language, C#. The guiding principles of the language emphasized simplicity, familiarity, safety, and practicality. Of course, all those principles needed to balance against one another; none are absolutes. The designers wanted the language to be simple to understand but not simplistic, familiar to C++ and Java programmers but not a slavish copy of either, safe by default but not too restrictive, and practical but never abandoning a disciplined, consistent, and theoretically valid design.

After many, many months of thought, design, development, testing, and documentation, C# 1.0 was delivered to the public. It was a pretty straightforward object-oriented language. Many aspects of its design were carefully chosen to ensure that objects could be organized into independently versionable components, but the fundamental concepts of the language came from ideas developed in object-oriented and procedural languages going back to the 1970s or earlier.

The design team continued to meet three times a week in that same second-floor conference room to build upon the solid base established by C# 1.0. By working with colleagues in Microsoft Research Cambridge and the CLR team across the street, the type system was extended to support parametric polymorphism on generic types and methods. They also added “iterator blocks” (sometimes known as “generators” in other languages) to make it easier to build iterable collections and anonymous methods. Generics and generators had been pioneered by earlier languages such as CLU and Ada in the 1970s and 1980s; the idea of embedding anonymous methods in an existing method goes all the way back to the foundations of modern computer science in the 1950s.

C# 2.0 was a huge step up from its predecessor, but still the design team was not content. They continued to meet in that same second-floor conference room three times a week. This time, they were thinking about fundamentals. Traditional “procedural” programming languages do a good job of basic arithmetic, but the problems faced by modern developers go beyond adding a column of numbers to find the average. They realized that programmers manipulate data by combining relatively simple operations in complex ways. Operations typically include sorting, filtering, grouping, joining, and projecting collections of data. The concept of a syntactic pattern for “query comprehensions” that concisely describes these operations was originally developed in functional languages such as Haskell but also works well in a more imperative language like C#. And thus LINQ—Language Integrated Query—was born.
After ten years of meeting for six hours a week in the same conference room, the need to teleconference with offsite team members motivated a change of venue to the fifth floor. The design team looked back on the last ten years to see what real-world problems were not solved well by the language, where there were “rough edges,” and so on. The increasing need to interoperate with both modern dynamic languages and legacy object models motivated the design of new language features like the “dynamic” type in C# 4.0.

I figured it might be a good idea to do a quick look at the evolution of the C# language here, in the Foreword, because this is certainly not the approach taken in this book. And that is a good thing! Authors of books for novices often choose to order the material in the order they learned it, which, as often as not, is the order in which the features were added to the language. What I particularly like about this book is that Scott chooses a sensible order to develop each concept, moving from the most basic arithmetic computations up to quite complex interrelated parts. Furthermore, his examples are actually realistic and motivating while still being clear enough and simple enough to be described in just a few paragraphs.

I’ve concentrated here on the evolution of the language, but of course the evolution of one language is far from the whole story. The language is just the tool you use to access the power of the runtime and the framework libraries; they are large and complex topics in themselves. Another thing I like about this book is that it does not concentrate narrowly on the language, but rather builds upon the language concepts taught early on to explain how to make use of the power afforded by the most frequently used base class library types.

As my brief sketch of the history of the language shows, there’s a lot to learn here, even looking at just the language itself. I’ve been a user of C# for ten years, and one of its designers for five, and I’m still finding out new facts about the language and learning new programming techniques every day. Hopefully your first 24 hours of C# programming described in this book will lead to your own decade of practical programming and continual learning. As for the design team, we’re still meeting six hours a week, trying to figure out what comes next. I’m looking forward to finding out.

Eric Lippert
Seattle, Washington
March 2010
Dedication

This book is first and foremost dedicated to Nathan, who I hope follows in my footsteps and someday writes books of his own.

Thank you for giving me a unique perspective and showing me the world through the eyes of a child.

About the Author

Scott Dorman has been designated by Microsoft as a C# Most Valued Professional in recognition for his many contributions to the C# community. Scott has been involved with computers in one way or another for as long as he can remember. He has been working with computers professionally since 1993 and with .NET and C# since 2001. Currently, Scott’s primary focus is developing commercial software applications using Microsoft .NET technologies. Scott runs a software architecture-focused user group, speaks extensively (including at Microsoft TechEd and community-sponsored code camps), and contributes regularly to online communities such as The Code Project and StackOverflow. Scott also maintains a .NET Framework and C#-focused technology blog at http://geekswithblogs.com/sdorman.
Acknowledgments

When I decided to undertake this project, I wasn’t prepared for just how difficult it is to actually write a book. As I look back on the amount of time and effort it took, I realize that, although I was the one responsible for writing the content, I couldn’t have done it without the help and support of others. First, I need to thank Brook for giving me the idea of writing this book for Sams Publishing in the first place and taking the chance on a new author. The rest of the editors at Sams, without whom the book would never have been published, were also great to work with. I also want to thank Keith Elder, Shawn Weisfeld, Brad Abrams, and Krzysztof Cwalina for their early input on the table of contents and helping me focus the content and overall direction of the book. My technical editors, Claudio and Eric, also deserve a huge amount of thanks; they have both provided an incredible amount of comments and insight. Of course, without the entire C#, .NET Framework, and Visual Studio product teams, I wouldn’t have anything to write about in the first place.

I wrote this book for the development community, which has given so much to me. Without its encouragement and support, I wouldn’t have been in a position to write this book at all. This includes everyone associated with the Microsoft MVP program and the Microsoft field evangelists, particularly Joe “devfish” Healy, Jeff Barnes, and Russ “ToolShed” Fustino.

Finally, of course, I have to thank my family for being so patient and understanding of the many long nights and weekends it took to finish this book. Although Nathan is too young right now to understand why I spent so much time on the computer rather than playing with him, I hope he will appreciate it as he gets older. The biggest thing it did was introduce him to computers at a very early age, as at 21 months old, he received his first laptop (an old IBM ThinkPad 770 that was collecting dust). To my stepson, Patrick, thank you for understanding all the canceled amusement park plans. Last, but certainly not least, thank you Erin for your support and patience. I know you are happy that everything is done and I can start having more family time.
We Want to Hear from You

As the reader of this book, you are our most important critic and commentator.

We value your opinion and want to know what we’re doing right, what we could do better, what areas you’d like to see us publish in, and any other words of wisdom you’re willing to pass our way.

You can email or write me directly to let me know what you did or didn’t like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book’s title and author, as well as your name and contact information. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: neil.rowe@pearson.com

Mail: Neil Rowe
Executive Editor
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.
This page intentionally left blank
Introduction

In late December 1998, Microsoft began working on a new development platform that would result in an entirely new way to create and run next-generation applications and web services. This new platform was called the .NET Framework and was publicly announced in June 2000.

The .NET Framework unified the existing Windows interfaces and services under a single application programming interface (API) and added many of the emerging industry standards, such as Simple Object Access Protocol (SOAP), and many existing Microsoft technologies, such as the Microsoft Component Object Model (COM and COM+) and Active Server Pages (ASP). In addition to providing a consistent development experience, the .NET Framework enabled developers to focus on the application logic rather than more common programming tasks with the inclusion of one of the largest available class libraries.

Finally, by running applications in a managed runtime environment that automatically handled memory allocation and provided a “sandboxed” (or restricted access) environment, many common programming errors and tasks were reduced and, in some cases, eliminated.

Now, nearly 10 years later, the .NET Framework continues to evolve by supporting new technologies and industry standards, adding support for dynamic languages and providing even more classes that are built-in. At Microsoft’s Professional Developer Conference (PDC) in 2008, one of the themes was “make the simple things easy and the difficult things possible.” The .NET Framework achieved that with its first release, and each release after that continues to realize that goal.

The C# (pronounced “See Sharp”) programming language was developed with the .NET Framework by Anders Hejlsberg, Scott Wiltamuth, and Peter Golde and was first available in July 2000. Having been written specifically for the .NET Framework, it is considered by many to be the canonical language of the .NET Framework. As a language, C# drew inspiration for its syntax and primary features from Delphi 5, C++, and Java 2. C# is a general-purpose, object-oriented, type-safe programming language used for writing applications of any type. Just as the .NET Framework has continued to evolve, C# has evolved to keep pace with the changes in the .NET Framework and to introduce new language features that continue to make the simple things easy and the difficult things possible.
Although there are more than 50 different programming languages supported by the .NET Framework, C# continues to be one of the most popular and modern general-purpose languages.

**Audience and Organization**

This book is targeted toward the non-.NET programmer who is venturing into .NET for the first time or an existing .NET programmer trying to learn C#. If you are first learning how to program, this book can help you on your way, but it isn't intended to be a beginning programming book. The book is designed with the purpose of getting you familiar with how things are done in C# and becoming productive as quickly as possible. I take a different approach in this book by using a more holistic view of the language. I chose this approach to give you the most complete understanding of the C# language by focusing on how the current language features enable you to solve problems.

This book is divided into five parts, each one focusing on a different aspect of the language. These parts progress from the simple fundamentals to more advanced topics, so I recommend reading them in order:

- **Part I, “C# Fundamentals,”** teaches you about the .NET Framework, the object-oriented programming features of C#, the fundamentals of C# type system, and events.

- **Part II, “Programming in C#,”** teaches you the fundamentals of programming. You learn how to perform loops and work with strings, regular expressions, and collections. Then we move to more advanced topics, such as exception management and generics. Finally, we finish with anonymous functions (lambdas), query expressions (LINQ), and how to interact with dynamic languages.

- **Part III, “Working with Data,”** shows how to interact with the file system and streams, create and query XML documents, and work with databases.

- **Part IV, “Building an Application Using Visual Studio,”** starts with an introduction to Visual Studio 2010 and debugging applications. We then build a Windows client application using data binding and validation. Next, you learn how to build an application for the web.

- **Part V, “Diving Deeper,”** introduces the advanced concepts of attribute programming, dynamic types, and language interoperability. You learn the fundamentals of how the .NET Framework organizes memory, how the garbage collector works, and how the .NET Framework provides mechanisms for deterministic finalization. Next, you learn how to use multiple threads and parallel processing. Finally, you look at some of the newer technologies from Microsoft.
built on the .NET Framework, such as Silverlight, PowerShell, and the Entity Framework.

By the Way boxes provide useful sidebar information that you can read immediately or circle back to without losing the flow of the topic at hand.

Did You Know? boxes highlight information that can make your programming more effective.

Watch Out! boxes focus your attention on problems or side effects that can occur under certain situations.

Throughout the book, I use examples that show real-world problems and how to solve them using C# and the .NET Framework. In Part IV, we actually build some complete applications from scratch that draw on the skills you learned in the previous three parts.

Conventions Used in This Book

This book uses several design elements and conventions to help you prioritize and reference the information it contains.

New terms appear in bold for emphasis.

In addition, this book uses various typefaces to help you distinguish code from regular English. Code is presented in a monospace font. Placeholders—words or characters that represent the real words or characters you would type in code—appear in italic monospace. When you are asked to type or enter text, that text appears in bold.

Some code statements presented in this book are too long to appear on a single line. In these cases, a line continuation character is used to indicate that the following line is a continuation of the current statement.

Closing Thoughts

The Microsoft .NET Framework and C# continue to be one of the most powerful yet elegant languages I’ve worked with and provide many exciting opportunities for developing the next “killer application.” You won’t be an expert in C# when you finish this book, but I hope you feel comfortable about creating applications in .NET and C#.
This page intentionally left blank
This page intentionally left blank
A class is the fundamental programming concept in C#, defining both representation and behavior in a single unit. Classes provide the language support required for object-oriented and component-oriented programming and are the primary mechanism you use to create user-defined types. Traditionally, object-oriented programming languages have used the term “type” to refer to behavior, whereas value-oriented programming languages have used it to refer to data representation. In C#, it is used to mean both data representation and behavior. This is the basis of the common type system and means two types are assignment-compatible if, and only if, they have compatible representations and behaviors.

In this hour, you learn the basics of both object-oriented and component-oriented programming. When you understand these concepts, you move on to creating a class in C# and examining how it fulfills the goals of object-oriented and component-oriented programming. You learn about the different accessibility models, how to create and use properties and methods, and about optional and named parameters.
Object-Oriented Programming

Before we start talking about classes in detail, you need to understand the benefits of object-oriented programming and understand how it relates to C#. Object-oriented programming helps you think about the problem you want to solve and gives you a way to represent, or model, that problem in your code. If you do a good job modeling the problem, you end up with code that’s easy to maintain, easy to understand, and easy to extend.

Maintainable Code

There is, of course, more to creating code that’s easy to maintain, understand, and extend than just getting the model correct. The implementation also has to be correct, readable, and correctly organized.

As previously mentioned, classes are the fundamental programming concept in C#, defining both representation and behavior in a single unit. Put another way, a class is a data structure that combines data storage with methods for manipulating that data. Classes are simply another data type that becomes available to you in much the same way any of the predefined types are available to you. Classes provide the primary mechanism you use to create user-defined types.

The four primary concepts of object-oriented programming are encapsulation, abstraction, inheritance, and polymorphism. In this hour, you learn about encapsulation and abstraction. In the next hour, you learn about inheritance and polymorphism.

Encapsulation and Abstraction

Encapsulation enables a class to hide the internal implementation details and to protect itself from unwanted changes that would result in an invalid or inconsistent internal state. For that reason, encapsulation is also sometimes referred to as data hiding.

As an example of encapsulation at work, think about your car. You start your car in the morning by inserting a key and turning it (or simply pushing a button, in some cases). The details of what happens when you turn the key (or push the button) that actually causes the engine to start running are hidden from you. You don’t need to know about them to start the car. It also means you can’t influence or change the internal state of the engine except by turning the ignition key.

By hiding the internal details and data, you create a public interface or abstraction representing the external details of a class. This abstraction describes what actions the
class can perform and what information the class makes publicly available. As long as the public interface does not change, the internal details can change in any way required without having an adverse affect on other classes or code that depends on it.

By keeping the public interface of a class small and by providing a high degree of fidelity between your class and the real-world object it represents, you help ensure that your class will be familiar to other programmers who need to use it.

Let's look at our car example again. By encapsulating the details of what happens when you start your car and providing an action, \texttt{StartCar}, and information, such as \texttt{IsCarStarted}, we have defined a public interface, thereby creating an abstraction (or at least a partial abstraction, because cars do much more than just start) of a car.

**Component-Oriented Programming**

Component-oriented programming is a technique of developing software applications by combining pre-existing and new components, much the same way automobiles are built from other components. Software components are self-contained, self-describing packages of functionality containing definitions of types that expose both behavior and data.

C# supports component-oriented programming through the concepts of properties, methods, events, and attributes (or metadata), allowing self-contained and self-describing components of functionality called assemblies.

**Classes in C#**

Now that you have a basic understanding of object-oriented and component-oriented programming, it is time to see how C# enables these concepts to become reality by using classes. You have actually already used classes in the examples and exercises from the previous two hours.

Classes in C# are reference types that implicitly derive from \texttt{object}. To define a class, you use the \texttt{class} keyword. Look at the application you built at the end of Hour 1, “The .NET Framework and C#.” Everything you did was inside a class named \texttt{Program}.

The \texttt{body} of the class, defined by the opening and closing braces, is where you define the data and behavior for the class.
Scope and Declaration Space

We briefly mentioned scope and declaration space in Hour 1, saying that scope defines where you can use a name, whereas declaration space focuses on where that name is unique. Scope and declaration space are closely related, but there are a few subtle differences.

A more formal definition is that **scope** is an enclosing context or region that defines where a name can be used without qualification.

In C#, both scope and declaration space is defined by a statement block enclosed by braces. That means namespaces, classes, methods, and properties all define both a scope and a declaration space. As a result, scopes can be nested and overlap each other.

If scope defines the visibility of a name and scopes are allowed to overlap, any name defined in an outer scope is visible to an inner scope, but not the other way around.

In the code shown in Listing 3.1, the field **age** is in scope throughout the entire body of **Contact**, including within the body of **F** and **G**. In **F**, the use of **age** refers to the field named **age**.

**LISTING 3.1  **Scope and Declaration Space**

```csharp
class Contact
{
    public int age;

    public void F()
    {
        age = 18;
    }

    public void G()
    {
        int age;
        age = 21;
    }
}
```

However, in **G**, the scopes overlap because there is also a local variable named **age** that is in scope throughout the body of **G**. Within the scope of **G**, when you refer to **age**, you are actually referring to the locally scoped entity named **age** and not the one in the outer scope. When this happens, the name declared in the outer scope is hidden by the inner scope.

Figure 3.1 shows the same code with the scope boundaries indicated by the dotted and dashed rectangles.
Declaration space, on the other hand, is an enclosing context or region in which no two entities are allowed to have the same name. In the Contact class, for example, you are not allowed to have anything else named age in the body of the class, excluding the bodies of F and G. Likewise, inside the body of G, when you redeclare age, you aren’t allowed to have anything else named age inside the declaration space of G.

You learn about method overloading a bit later this hour, but methods are treated a little differently when it comes to declaration spaces. If you consider the set of all overloaded methods with the same name as a single entity, the rule of having a unique name inside a declaration space is still satisfied.

Try It Yourself

Working with Scope
To explore the differences between scope and declaration space, follow these steps. Keep Visual Studio open at the end of this exercise because you will use this application later.

1. Create a new Console application.

2. Add a new class file named Contact.cs that looks like Listing 3.1.

4. In the `Main` method of the `Program.cs` file, enter the following code to create a new instance of the `Contact` class and print the current value of `age`:

```csharp
Contact c = new Contact();
Console.WriteLine(c.age);
c.F();
Console.WriteLine(c.age);
c.G();
Console.WriteLine(c.age);
```

5. Run the application using Ctrl+F5 and observe that the output matches what is shown in Figure 3.2.

---

**Accessibility**

Accessibility enables you to control the visibility, or accessibility, of an entity outside of its containing scope. C# provides this through access modifiers, which specify constraints on how members can be accessed outside the boundary of the class and, in some cases, even constrain inheritance. A particular class member is accessible when access to that member has been allowed; conversely, the member is inaccessible when access has been disallowed.

These access modifiers follow a simple set of contextual rules that determine when certain types of accessibility are permitted:

- Namespaces are not allowed to have any access modifiers and are always public.
- Classes default to internal accessibility but are allowed to have either public or internal declared accessibility. A nested class, which is a class defined inside of another class, defaults to private accessibility but can have any of the five kinds of declared accessibility.
- Class members default to private accessibility but can have any of the five kinds of declared accessibility.

These rules also define the default accessibility, which occurs when a member does not include any access modifiers.
Explicitly Declaring Accessibility

Although C# provides reasonable default access modifiers, you should always explicitly declare the accessibility of your class members. This prevents unintended ambiguity, indicates that the choice was a conscious decision, and is self-documenting.

The access modifiers supported by C# are shown in Table 3.1.

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Access is not limited.</td>
</tr>
<tr>
<td>protected</td>
<td>Access is limited to the containing class or types derived from the containing class.</td>
</tr>
<tr>
<td>internal</td>
<td>Access is limited to the containing assembly.</td>
</tr>
<tr>
<td>protected internal</td>
<td>Access is limited to the containing assembly or types derived from the containing class.</td>
</tr>
<tr>
<td>private</td>
<td>Access is limited to the containing class only.</td>
</tr>
</tbody>
</table>

Protected Internal

Be careful when using protected internal accessibility because it is effectively protected or internal. C# does not provide a concept of protected and internal.

Fields and Constants

Fields are variables that represent data associated with a class. In other words, a field is simply a variable defined in the outermost scope of a class. If you recall from Hour 1, a field can be either an instance field or a static field, and for both types of field, you can specify any of the five access modifiers. Typically, fields are private, which is the default.

If a field, no matter whether it is an instance or static field, is not given an initial value when it is declared, it is assigned the default value appropriate for its type.

Similar to fields, constants can be declared with the same access modifiers. Because a constant must have a value that can be computed at compile time, it must be assigned a value as part of its declaration. One benefit of requiring a value that can be computed at compile time is that a constant can depend on other constants.
A constant is usually a value type or a string literal because the only way to create a non-null value of a reference type other than string is to use the new operator, which is not permitted.

**Watch Out!**

**Constants Should Be Constant**

When creating constants, you should be sure that the value is something that is logically constant forever. Good constants are things that never change, such as the value of Pi, the year Elvis was born, or the number of items in a mol.

If you need to create a field that has constant-like behavior but uses a type not allowed in a constant declaration, you can use a static read-only field instead by specifying both the static and readonly modifiers. A read-only field can be initialized only as part of its declaration or in a constructor.

**Try It Yourself**

**Working with Fields**

By following these steps, you explore how to create a class containing data and how to provide access to that data. If you closed Visual Studio, repeat the previous exercise first. Keep Visual Studio open at the end of this exercise because you will use this application later.

1. Create a new Console application.

2. Add a new class file named `Contact.cs`. Inside the body of the class, declare three private fields named `firstName`, `lastName`, and `dateOfBirth` of type `string`, `string`, and `DateTime`, respectively.

3. Add the following method to the class. You learn more about methods later in this hour and more about the `StringBuilder` class in Hour 8, “Using Strings and Regular Expressions”:

   ```csharp
   public override string ToString()
   {
       StringBuilder stringBuilder = new StringBuilder();
       stringBuilder.AppendFormat("Name: {0} {1}\n\n", this.firstName,
                                 this.lastName);
       stringBuilder.AppendFormat("Date of Birth: {0}\n", this.dateOfBirth);
       return stringBuilder.ToString();
   }
   ```

4. In the `Main` method of the `Program.cs` file, enter the following:

   ```csharp
   Contact c = new Contact();
   Console.WriteLine(c.ToString());
   ```
5. Run the application using Ctrl+F5 and observe that the output matches what is shown in Figure 3.3.

![Figure 3.3: Working with fields](image)

**Properties**

If fields represent state and data but are typically private, there must be a mechanism that enables the class to provide that information publicly. Knowing the different accessibility options allowed it would be tempting to simply declare the class fields to have public accessibility.

This would allow us to satisfy the rules of abstraction, but this would then violate the rules of encapsulation because the fields could be directly manipulated. How, then, is it possible to satisfy both the rules of encapsulation and abstraction? What is needed is something accessed using the same syntax as a field but that can define different accessibility than the field itself. Properties enable us to do exactly that. A property provides a simple way to access a field, called the **backing field**, which can be publicly available while still allowing the internal details of that field to be hidden. Just as fields can be static, properties can also be static and are not associated with an instance of the class.

Although fields declare variables, which require storage in memory, properties do not. Instead, properties are declared with accessors that enable you to control whether a value can be read or written and what should occur when doing so. The **get** accessor enables the property value to be read, whereas the **set** accessor enables the value to be written.

Listing 3.2 shows the simplest way to declare a property. When using this syntax, known as **automatic properties**, you omit the backing field declaration and must always include both the **get** and **set** accessor without a declared implementation, which the compiler provides.

**Listing 3.2  Declaring an Automatic Property**

```csharp
class Contact
{
    public string FirstName
    {
        get;
        set;
    }
}
```
In fact, the compiler transforms the code shown in Listing 3.2 into code that looks roughly like that shown in Listing 3.3.

**Listing 3.3  Declaring a Property**

```csharp
class Contact
{
    private string firstName;

    public string FirstName
    {
        get
        {
            return this.firstName;
        }
        set
        {
            this.firstName = value;
        }
    }
}
```

**Automatic Properties**

Automatic properties are convenient, especially when you implement a large number of properties. This convenience does come at a slight cost, however. Because you don’t provide a body for the accessors, you can’t specify any logic that executes as part of that accessor, and both accessors must be declared using the automatic property syntax. As a result, if at some point later you realize that you need to provide logic for either of the accessors, you need to add a backing field and the appropriate logic to both accessors.

Fortunately, this change doesn’t affect the public interface of your class, so it is safe to make, although it might be a bit tedious.

The **get** accessor uses a `return` statement, which simply instructs the accessor to return the value indicated. In the **set** accessor of the code in Listing 3.3, the class field `firstName` is set equal to `value`, but where does `value` come from? From Table 1.6 in Chapter 1, you know that `value` is a contextual keyword. When used in a property **set** accessor, the **value** keyword always means “the value that was provided by the caller” and is always typed to be the same as the property type.

By default, the property accessors inherit the accessibility declared on the property definition itself. You can, however, declare a more restrictive accessibility for either the **get** or the **set** accessor.
You can also create calculated properties that are read-only and do not have a back-
ing field. These calculated properties are excellent ways to provide data derived from
other information.

Listing 3.4 shows a calculated FullName property that combines the firstName and
lastName fields.

**LISTING 3.4 Declaring a Calculated Property**

```csharp
class Contact
{
    private string firstName;
    private string lastName;

    public string FullName
    {
        get
        {
            return this.firstName + " " + this.lastName;
        }
    }
}
```

**Read-Only and Write-Only Properties**

For explicitly declared properties, you are allowed to omit either accessor. By
including only the get accessor, you create a read-only property. To create the
equivalent of a read-only property using automatic properties, you would declare
the set accessor to be private.

By including only the set accessor, or declaring the get accessor to be private,
you create a write-only property. In practice, you should avoid write-only properties.

Because properties are accessed as if they were fields, the operations performed in the
accessors should be as simple as possible. If you need to perform more complex oper-
ations or perform an operation that could be time-consuming or expensive (resource
consuming), it might be better to use a method rather than a property.

**Try It Yourself**

**Working with Properties**

To modify the Contact class to allow access to the private data using properties,
and to use automatic and calculated properties, follow these steps. If you closed
Visual Studio, repeat the previous exercise first. Be sure to keep Visual Studio
open at the end of this exercise because you will use this application later.
1. Open the Contact.cs file.

2. Add a new public property named DateOfBirth that enables reading and writing to the dateOfBirth field.

3. Remove the firstName and lastName fields and create a FirstName and LastName property as automatic properties.

4. Add a calculated property named FullName, which combines the values of the FirstName and LastName properties. This should be similar to the calculated property shown in Listing 3.4.

5. Modify the ToString method to make use of the new FullName property instead of performing the string concatenation directly.

6. In the Main method of the Program.cs file, enter the following code after the Console.WriteLine statement:
   ```csharp
   c.FirstName = "Jim";
   c.LastName = "Morrison";
   c.DateOfBirth = new DateTime(1943, 12, 8);
   Console.WriteLine(c.ToString());
   ```

7. Run the application using Ctrl+F5, and observe that the output matches what is shown in Figure 3.4.

Methods

Methods define and implement a behavior or action that can be performed. The WriteLine action of the Console class you have been using in the examples and exercises so far is an example of a method.

Listing 3.5 shows how to add a method to the Contact class that verifies an email address. In this case, the VerifyEmailAddress method specifies void as the return type, meaning that it does not return a value.
LISTING 3.5  Declaring a Method

class Contact
{
    public void VerifyEmailAddress(string emailAddress)
    {
    }
}

Listing 3.6 shows the same method declared to have a bool as the return type.

LISTING 3.6  Declaring a Method That Returns a Value

class Contact
{
    public bool VerifyEmailAddress(string emailAddress)
    {
        return true;
    }
}

A method declaration can specify any of the five access modifiers. In addition to the access modifiers, a method can also include the static modifier. Just as static properties and fields are not associated with an instance of the class, neither are static methods. The WriteLine method is actually a static method on the Console class.

Methods can accept zero or more parameters, or input, declared by the formal parameter list, which consists of one or more comma-separated parameters. Each parameter must include both its type and an identifier. If a method accepts no parameters, an empty parameter list must be specified.

Parameters are divided into three categories:

- **Value parameters**—The most common. When a method is called, a local variable is implicitly created for each value parameter and assigned the value of the corresponding argument in the argument list.

---

**Parameter Arrays**

Parameter arrays, declared with the params keyword, can be thought of as a special case of value parameters and declare a single parameter that can contain zero or more arguments of the given type in the argument list.

A method’s formal parameter list can include only a single parameter array; in which case it must be the last parameter in the list. A parameter array can also be the only parameter.
**Reference parameters**—Do not create a new storage location but represent the same storage location as the corresponding argument in the argument list. Reference parameters are declared using the `ref` keyword, which must be present both in the parameter list and the argument list.

**Output parameters**—Similar to reference parameters but require the `out` keyword to be present in both the parameter and invocation lists. Unlike reference parameters, they must be given a definite value before the method returns.

For a method to actually perform its desired action on the object, it must be invoked, or called. If the method requires input parameters, those values must be provided in an argument list, and if the method provides an output value, that value can also be stored in a variable.

The argument list is normally a one-to-one relationship with the parameter list, meaning that for each parameter, you must provide a value of the appropriate type in the same order when you call the method.

### Methods as Input

Methods that return a value and properties can also be used as input to other methods, as long as the return value type is compatible with the parameter type. This capability greatly increases the usefulness of both methods and properties, allowing you to chain method or property calls to form behaviors that are more complex.

Looking at the `VerifyEmailAddress` method that has a `void` return type from the earlier examples, you would call the method like this:

```csharp
Contact c = new Contact();
c.VerifyEmailAddress("joe@example.com");
```

However, for the `VerifyEmailAddress` method defined to return a `bool`, you would call the method like this:

```csharp
Contact c = new Contact();
bool result = c.VerifyEmailAddress("joe@example.com");
```

Just as you do with the parameter list, if a method invocation requires no arguments, you must still specify an empty list.

### Method Overloading

Ordinarily, two entities cannot have the same name within a declaration space, except for overloaded methods. When two or more methods have the same name in a declaration space but have different method signatures, they are **overloaded**.
The **method signature** is made up of the method name and the number, types, and modifiers of the formal parameters and must be different from all other method signatures declared in the same class; the method name must be different from all other non-methods declared in the class.

**Method Signatures**

The return type is not part of the method signature, so methods cannot differ only in return type.

Although the formal parameter list is part of the method signature, methods cannot differ based on a parameter being a ref or out parameter. For the purposes of the method signature, the ref or out attribute of the parameter is not considered.

Overloaded methods can vary only by signature. More appropriately, they can vary only by the number and types of parameters. Consider the `Console.WriteLine` method you have already used; there are 19 different overloads from which you can choose.

Overloading methods is common in the .NET Framework and enables you to give the users of your class a single method with which they interact and provide different input. Based on that input, the compiler figures out which method should actually be used.

**Overloads with Different Return Types**

Because method signatures do not include the return type, it is possible for overloaded methods to have different return types. Even though this might be legal C# code, you should avoid it to minimize the possibility for confusion.

Method overloading is useful when you want to provide several different possibilities for initiating an action, but method overloading can become unwieldy when there are many options. An example of method overloading is shown in Listing 3.7.

**LISTING 3.7  Method Overloading**

```csharp
public void Search(float latitude, float longitude)
{
    Search(latitude, longitude, 10, "en-US");
}

public void Search(float latitude, float longitude, int distance)
{
    Search(latitude, longitude, distance, "en-US");
}

public void Search(float latitude, float longitude, int distance, string culture)
{
}
```
Working with Methods

Continuing to expand the Contact class, add the VerifyEmailAddress and Search methods by following these steps. If you closed Visual Studio, repeat the previous exercises first. Be sure to keep Visual Studio open at the end of this exercise because you will use this application later.

1. Open the Contact.cs file.

2. Add the VerifyEmailAddress method shown in Listing 3.6 so that it returns true if the email address entered is “joe@example.com”.

3. Add the overloaded methods shown in Listing 3.7.

4. In the last overloaded Search method, enter a Console.WriteLine call that prints the values of the parameters.

5. In the Main method of the Program.cs file, enter the following code after the last Console.WriteLine statement:

   ```csharp
   c.Search(37.479444f, -122.450278f);
   c.Search(37.479444f, -122.450278f, 50);
   c.Search(37.479444f, -122.450278f, 50, "en");
   Console.WriteLine(c.VerifyEmailAddress("joe@example.com"));
   Console.WriteLine(c.VerifyEmailAddress("jim@example.com"));
   ```

6. Run the application using Ctrl+F5 and observe that the output matches what is shown in Figure 3.5.

Optional Parameters and Named Arguments

Optional parameters enable you to omit that argument in the invocation list when calling a method. Only value parameters can be optional, and all optional parameters must appear after required parameters, but before a parameter array.

To declare a parameter as optional, you simply provide a default value for it. The modified Search method using optional parameters is shown here:
The latitude and longitude parameters are required, whereas distance and culture are both optional. The default values used are the same values provided by the first overloaded Search method.

Looking at the Search method overloads from the previous section, it should become clear that the more parameters you have the more overloads you need to provide. In this case, there are only a few overloads, but that is still more than providing a single method with optional parameters. Although overloads are the only option in some cases, particularly those that don’t imply a reasonable default for a parameter, often you can achieve the same result using optional parameters.

Optional and Required Parameters

A parameter with a default argument is an optional parameter, whereas a parameter without a default argument is a required parameter.

Optional parameters are also particularly useful when integrating with unmanaged programming interfaces, such as the Office automation APIs, which were written specifically with optional parameters in mind. In these cases, the original API call might require a large number of arguments (sometimes as many as 30), most of which have reasonable default values.

A method that contains optional parameters can be invoked without explicitly passing arguments for those parameters, allowing the default arguments to be used instead. If, however, the method is invoked and provides an argument for an optional parameter, that argument is used instead of the default.

Listing 3.8 shows an example of calling the Search method, allowing the default values to be used.

**LISTING 3.8 Using Optional Parameters**

```
public void Search(float latitude, float longitude, int distance = 10,
                    string culture = "en-US");
```

```
Search(27.966667f, 82.533333f, 3);
Search(27.966667f, 82.533333f, 3, "en-GB");
Search(27.966667f, 82.533333f);
```

The drawback to optional parameters is that you cannot omit arguments between the commas, meaning you could not call the Search method like this:

```
Search(27.966667f, 82.533333f, , "en-GB");
```

To resolve this situation, C# enables any argument to be passed by name, whereby you are explicitly indicating the relationship between the argument and its...
corresponding parameter. Using named arguments, the different method calls in Listing 3.8 and the illegal call just shown could be written as shown in Listing 3.9.

**Listing 3.9 Using Named Arguments**

```csharp
Search(latitude: 27.966667f, longitude: 82.533333f, distance: 3);
Search(latitude: 27.966667f, longitude: 82.533333f, distance: 3, culture: "en-GB");
Search(latitude: 27.966667f, longitude: 82.533333f);
Search(27.966667f, 82.533333f, culture: "en-GB");
Search(latitude: 27.966667f, longitude: 82.533333f, culture: "en-GB");
```

All these calls are equivalent. The first three calls are the same as the calls in Listing 3.8 except that each parameter is explicitly named. The last two calls show how we can omit an argument in the middle of the parameter list and are also the same, although one uses a mixture of named and positional arguments.

**By the Way**

**Named and Positional Arguments**

Arguments that are not passed by name are called positional arguments. Positional arguments are the most common.

Named arguments are most often used with optional parameters, but they can be used without them as well. Unlike optional parameters, named arguments can be used with value, reference, and output parameters. You can also use named arguments with parameter arrays, but you must explicitly declare a new array to contain the values, as shown here:

```csharp
Console.WriteLine(String.Concat(values: new string[] { "a", "b", "c" }));
```

As you can see from the `Search` method, by enabling you to explicitly indicate the name of an argument, C# provides an additional (and powerful) way to help write fully describing and self-documenting code.

**Did you Know?**

**Changing the Order of Arguments**

Arguments are always evaluated in the order they are specified. Although not generally needed, named arguments enable you to change the order an argument appears in the invocation list:

```
Search(longitude: 82.533333f, latitude: 27.966667f);
Search(latitude: 27.966667f, longitude: 82.533333f);
```
Try It Yourself

Working with Optional Parameters and Named Arguments

To modify the Search methods previously defined to use optional parameters rather than overloads, follow these steps. If you closed Visual Studio, repeat the previous exercises first. Be sure to keep Visual Studio open at the end of this exercise because you will use this application later.

1. Open the Contact.cs file.

2. Remove the first two Search methods, leaving only the method containing all four parameters, and modify that method so that distance and culture are optional, using 10 and "en-US" as the default values.

3. Run the application using Ctrl+F5 and observe that the output matches what is shown in Figure 3.6.

4. In the Main method of the Program.cs file, change the calls to the Search method to use different combinations of named parameters and observe the output after each change.

Instantiating a Class

Unlike the predefined value types in which you could simply declare a variable and assign it a value, to use a class in your own programs, you must create an instance of that class.

Remember, even though you create new objects directly using the new keyword, the virtual execution system is responsible for actually allocating the memory required, and the garbage collector is responsible for deallocating that memory.

Instantiating a class is accomplished using the new keyword, like this:

```csharp
Contact c = new Contact();
```
A newly created object must be given an initial state, which means any fields declared must be given an initial value either by explicitly providing one or accepting the default values (see Table 2.13 in Chapter 2).

Sometimes this level of initialization is sufficient, but often it won’t be. To provide additional actions that occur during initialization, C# provides an instance constructor (sometimes just called a constructor), which is a special method executed automatically when you create the instance.

A constructor has the same name of the class but it cannot return a value, which is different from a method that returns void. If the constructor has no parameters, it is the default constructor.

**By the Way**

**Default Constructors**

Every class must have a constructor, but you don’t always have to write one. If you don’t include any constructors, the C# compiler creates a default constructor for you. This constructor won’t actually do anything, but it will be there.

Because the compiler only generates the default constructor if you don’t provide any additional constructors, it is easy to break the public interface of your class by adding an additional constructor that has parameters and forgetting to also explicitly add the default constructor. As a result, it is a good idea to always provide a default constructor rather than letting the compiler generate it for you.

The default constructor (or any constructor) can have any of the accessibility modifiers, so it is entirely possible to create a private default constructor. This is useful if you want to allow your class to be created but want to ensure that certain information is always provided when the object is instantiated.

Listing 3.10 shows the default constructor for the Contact class.

**LISTING 3.10  Declaring a Default Constructor**

```csharp
public class Contact
{
    public Contact()
    {
    }
}
```

Just as it is possible to overload regular methods, it is also possible to overload constructors. The signature for a constructor is the same as it is for a regular method, so the set of overloaded constructors must also vary by signature.

Some reasons for providing specialized constructors follow:

- There is no reasonable initial state without parameters.
- Providing an initial state is convenient and reasonable for the type.
Constructing the object can be expensive, so you want to ensure that the object has the correct initial state when it is created.

A non-public constructor restricts who can create objects using it.

Looking at the Contact class you have been using, it would certainly be useful if you provided values for the firstName, lastName, and dateOfBirth fields when creating a new instance. To do that, you would declare an overloaded constructor like the one shown in Listing 3.11.

**Listing 3.11  Declaring a Constructor Overload**

```csharp
public class Contact
{
    public Contact(string firstName, string lastName, DateTime dateOfBirth)
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.dateOfBirth = dateOfBirth;
    }
}
```

In the constructor overload from Listing 3.11, you assigned the value of the parameter to its corresponding private field.

Typically, although not always, when a class contains multiple constructors, those constructors are chained together. To chain constructors together, you use a special syntax that uses the `this` keyword.

### The this Keyword

The `this` keyword refers to the current instance of the class. It is similar to the `Me` keyword in Visual Basic, a self identifier in F#, the `__self__` attribute in Python, and `self` in Ruby.

The common uses of this follow:

- To qualify members hidden by similar names
- To pass an object as a parameter to other methods
- To specify which constructor should be called from another constructor overload
- To indicate the extended type in an extension method

Because static members exist at the class level and are not associated with an instance, you can’t use the `this` keyword.

In Listing 3.11, the `this` keyword is used to distinguish between the class field and the parameter because both have the same name.
Listing 3.12 shows the Contact class with both constructors from Listing 3.10 and Listing 3.11 using constructor chaining.

**LISTING 3.12  Constructor Chaining**

```csharp
public class Contact
{
    public Contact()
    {
    }

    public Contact(string firstName, string lastName, DateTime dateOfBirth)
        : this()
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.dateOfBirth = dateOfBirth;
    }
}
```

One benefit of constructor chaining is that you can chain in any constructor provided by the class, not just the default constructor. When you use constructor chaining, it is important to understand the order in which the constructors execute. The constructor chain is followed until it reaches the last chained constructor, and then constructors will be executed in order going back out of the chain. Listing 3.13 shows a class, `C`, with three constructors, each chained through to the default constructor.

**LISTING 3.13  Chained Constructor Order of Execution**

```csharp
public class C
{
    string c1;
    string c2;
    int c3;

    public C()
    {
        Console.WriteLine("Default constructor");
    }

    public C(int i, string p1) : this(p1)
    {
        Console.WriteLine(i);
    }

    public C(string p1) : this()
    {
        Console.WriteLine(p1);
    }
}
```

Figure 3.7 shows the sequence in which each constructor would execute when instantiated using the second constructor (the one that takes an `int` and a `string` as input).
C c = new C(3, "C2");

**Static Construction**

Instance constructors, like you have just seen, implement the actions required to initialize instances of the class. In some cases, a class might require specific initialization actions to occur at most once and before any instance members are accessed.

To accomplish this, C# provides a **static constructor**, which has the same form as the default constructor with the addition of the `static` modifier instead of one of the access modifiers. Because static constructors initialize the class, you cannot directly call a static constructor.

A static constructor executes at most once and will be executed the first time an instance is created or the first time any of the static class members are referenced.

**Nested Classes**

A **nested class** is one that is fully enclosed, or nested, inside another class declaration. Nested classes are a convenient way to allow an outer class to create and use objects without making them accessible outside of that class. Although nested classes can be convenient, they are also easy to overuse, which can make your class more difficult to work with.

Nested classes implicitly have at least the same access level as the containing class. For example, if the nested class is `public` but the containing class is `internal`, the nested class is implicitly `internal` as well, and only members of that assembly can access the nested class. However, if the containing class is `public`, the nested class follows the same accessibility rules as a non-nested class.
You should consider implementing a class as a nested class if it has no stand-alone significance and can be logically contained by another class or members of the class need to access private data of the containing class. Nested classes should generally not be public because they are for the internal use of the containing class.

**Partial Classes**

Partial classes enable you to split the declaration of a class into multiple parts, typically across multiple files. Partial classes are implemented in exactly the same way as normal classes but contain the keyword `partial` just before the `class` keyword. When working with partial classes, all the parts must be available during compilation and have the same accessibility to form the complete class.

Code-generation tools, such as the visual designers in Visual Studio, which generate a class for you representing the visual control being designed, use partial classes extensively. The machine-generated code is added to one part of the partial class, allowing you to modify the other part of the partial class without concern that your changes will be lost when the machine-generated portion is regenerated.

Partial classes can also be used in other scenarios that don’t involve machine-generated code. Large class declarations can benefit from using partial classes; however, this can sometimes mean that your class is trying to do too much and would be better split into multiple classes.

**Did you know?**

Even though C# does not require a single class per file, like Java, it is often helpful to follow that structure. When using nested classes, this isn’t possible unless the containing class is a partial class.

**Static Classes**

So far, you have seen the `static` modifier applied to constructors, fields, methods, and properties. You can also apply the `static` modifier to a class, which defines a static class. A static class can have only a static constructor, and as a result, it is not possible to create an instance of a static class. For that reason, static classes most commonly contain utility or helper methods that do not require a class instance to work.
Implicit Static Members
Static classes can contain only static members, but those members are not automatically static. You must explicitly include the static modifier; however, you can declare any static member as public, private, or internal.

Extension Methods
Extension methods are regular static methods, but the first parameter includes the this modifier and represents the type instance being extended, typically called the type extension parameter. Extension methods must be declared in a non-nested, non-generic static class.

When the namespace containing an extension class is in scope through a using directive, the extension methods appear as if they were native instance methods on the extended type. This allows them to be called in a natural and intuitive manner.

Because an extension method is nothing more than a specially marked static method, it does not have any special access to the type being extended and can work only with the public interface of the extended type. It also enables you to call the extension method in the more traditional way by referring to its fully qualified name.

Access to Internals
An extension method defined in the same assembly as the type being extended also has access to internal members of that type.

Although an extension method matching the signature of an actual method on the type can be defined, it will not be visible. The compiler ensures that during method resolution, any actual class methods take precedence over extension methods. This ensures that an extension method cannot change the behavior of a standard class method, which would cause unpredictable, or at least unexpected, behavior.

Try It Yourself

Working with Extension Methods
By following these steps, you add an extension method on the Contact class and modify the Main method of Program.cs to use this new extension method. If you closed Visual Studio, repeat the previous exercises first.

1. Create a new file named Extensions.cs.

2. Make the Extensions class static and create a new extension named GetFullName that extends Contact and uses the same logic as you used for the FullName property.
3. Remove the FullName property in the Contact class and modify the ToString method to use this new extension method.

4. Run the application using Ctrl+F5 and observe that the output matches what is shown in Figure 3.8.

Object Initializers

You have seen how to create constructors for your class that provide a convenient way to set the initial state. However, as with method overloading, the more fields you require to be set, the more overloaded constructors you might need to provide. Although constructors support optional parameters, sometimes you want to set properties when you create the object instance.

Classes provide an object initialization syntax that enables you to assign values to any publicly accessible fields or properties as part of the constructor call. This allows a great deal of flexibility and can significantly reduce the number of overloaded constructors you need to provide.

Listing 3.14 shows code similar to what you wrote in the “Working with Properties” section, followed by code using an object initializer. The code generated by the compiler in both cases is almost the same.

Listing 3.14  Object Initializers

```csharp
Contact c1 = new Contact();
c1.FirstName = "Jim";
c1.LastName = "Morrison";
c1.DateOfBirth = new DateTime(1943, 12, 8);
Console.WriteLine(c1.ToString());

Contact c2 = new Contact
{
    FirstName = "Jim",
    LastName = "Morrison",
    DateOfBirth = new DateTime(1943, 12, 8)
};

Console.WriteLine(c2.ToString());
```
As long as there are no dependencies between fields or properties, object initializers are an easy and concise way to instantiate and initialize an object at the same time.

**Summary**

At this point, you should have a good understanding of how classes in C# provide a language implementation for object-oriented programming. You learned how scope affects the visibility of members in a class and how you can change accessibility using the different access modifiers. From there, you built a class and instantiated an instance of that class. You then learned about methods and properties, including method overloading, optional, and named parameters. Finally, we talked about nested and partial classes.

Departing from the simple examples you worked with in the previous hours, the samples and exercises in this hour focused on building more real-world classes.

**Q&A**

Q. **What are the four primary principles of object-oriented programming?**

A. The four primary principles of object-oriented programming are encapsulation, abstraction, inheritance, and polymorphism.

Q. **Why are encapsulation and abstraction important?**

A. By using encapsulation and abstraction, you can change internal implementation details without affecting already-written code that uses that class.

Q. **What is method overloading?**

A. Method overloading is creating more than one method of the same name in a given type. Overloaded methods must have different signatures.

Q. **How do properties enable a class to meet the goals of encapsulation?**

A. A property provides a simple way to access a field that can be publicly available while still allowing the internal details of that field to be hidden.

Q. **What are partial classes?**

A. A partial class contains the keyword `partial` on all class declarations and is typically split across multiple source code files.
Q. *What is the benefit of using extension methods?*

A. Using extension methods enables additional functionality to be added to an existing type without requiring the use of inheritance. This additional functionality can then be used in a natural and intuitive way.

## Workshop

### Quiz

1. What are the five access modifiers available in C#?
2. What is the default accessibility for a class?
3. What is a constructor?
4. Can the default constructor of a class have parameters?
5. Using the code shown in Listing 3.13, what is the output of the following statement?
   
   ```csharp
   C c = new C(3, "C2");
   ```
6. When can a read-only field be assigned?
7. What is method overloading?
8. Are there limitations when using automatic properties?
9. What is a nested class?
10. Can extension methods access private members of the type being extended?
11. What happens when the new operator is executed?

### Answers

1. The five access modifiers available in C# are public, protected, internal, protected internal, and private.
2. Classes default to internal accessibility but are allowed to have either public or internal declared accessibility. Nested classes default to private accessibility but are allowed to have any accessibility.
3. A constructor is a special method that is executed automatically when you create an object to provide additional initialization actions.
4. No, the default constructor of a class must always have no parameters.

5. The output of the statement is
   
   ```csharp
default Constructor
C2
3
   ```

6. A read-only field can be initialized only as part of its declaration or in a constructor.

7. Method overloading is creating more than one method of the same name that differs only by the number and type of parameters.

8. Automatic properties do not provide a way to access the implicit backing field, do not enable you to specify additional statements that execute as part of the get or set accessor, and do not enable a mixture of regular and automatic syntax.

9. A nested class is one that is fully enclosed inside another class declaration.

10. Because extension methods are simply static methods, they do not have any special access to the type they extend. However, an extension method defined in the same assembly as the type being extended also has access to internal members of that type.

11. The two primary actions that occur when the new operator is executed are 1) Memory is allocated from the heap and 2) the constructor for the class is executed to initialize the allocated memory.

**Exercise**

1. Add a class to the PhotoViewer project that represents a photo. This class should be named `Photo` and be in the `PhotoViewer` namespace. The class should have the following private fields and a read-only property to retrieve the value of those fields:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>Exists</td>
</tr>
<tr>
<td>BitmapFrame</td>
<td>image</td>
</tr>
<tr>
<td>Uri</td>
<td>source</td>
</tr>
</tbody>
</table>
Add the following constructor:

```csharp
public Photo(Uri path)
{
    if (path.IsFile)
    {
        this.source = path;
    }
}
```
Symbols

+ (addition operator), 47
= (assignment operator), 47
/* */ (comments), 348-349
// (comments), 348
&& (conditional And operator), 50
|| (conditional OR operator), 50
- - (decrement operator), 48-49
/ (division operator), 48
== (equals operator), 50
\ (escape sequence), 169
\" (escape sequence), 169
\\ (escape sequence), 169
^ (exclusive OR operator), 50
> (greater than operator), 50
(>=) (greater than or equals operator), 50
++ (increment operator), 48
incrementing a value listing, 48
sample application, 49
=> (lambda operator), 280
< (less than operator), 50
(<=) (less than or equals operator), 50
& (logical AND operator), 50
| (logical OR operator), 47
% (modulus operator), 48
* (multiplication operator), 48
! (not operator), 51
!= (not equals operator), 50
?? (null-coalescing operator), 54
. regular expression metacharacter, 188
? regular expression metacharacter, 188
$ regular expression metacharacter, 188
* regular expression metacharacter, 188
( ) regular expression metacharacter, 188
[ ] regular expression metacharacter, 188
[^ ] regular expression metacharacter, 188
^ regular expression metacharacter, 188
| regular expression metacharacter, 188
+ regular expression metacharacter, 188
; (semicolons), 19
- (subtraction operator), 48
/// (XML comments), 348
A escape sequence, 169
Abs method (BigInteger type), 46
absolute paths, 290
AbsoluteUri method (Uri class), 44
abstract classes, 104-105
    interface combinations, 107
    listing, 104-105
    sample application, 105
abstractions, 65
access modifiers
    rules, 68
    supported, 69
accessibility
    access modifiers
        rules, 68
        supported, 69
default, 68
defined, 68
explicit declaration, 69
accessing
    attributes at runtime, 434
    multiple attributes, 435
    single attributes, 434-435
    web applications, 408
AccessViolationException, 229
Add method
    collection initializers, 218
    Dictionary<TKey, TValue> class, 209
    HashSet<T> class, 212
    SortedSet<T> class, 212
    SortedDictionary<TKey, TValue> class, 210
    SortedList<TKey, TValue>/class, 210
    TimeSpan type, 42
    XML elements, 324-325
AddAfterSelf method, 324
AddBeforeSelf method, 324
AddDays method, 42
AddFirst method, 324
AddHours method, 42
addition operator (+), 48
additive operators, 47
AddMinutes method, 42
AddMonths method, 42
AddressChangedEventArgs class, 140-141
AddYears method, 42
ADO.NET, 330
    connection pooling, 332
    data providers, 330
    data source connections, 331
    DataSet class, 330, 331
    queries, 332
    read-only database access, 331
    required references, 331
    sample application, 333
Aggregate() operator, 473
AND
    conditional operator (&&), 50
    logical operator (&), 50
animation (WPF), 369
anonymous methods (event handlers), 136
anonymous types (LINQ queries), 270
Append method, 177
AppendAllLines method, 298
AppendAllText method, 298
AppendFormat method, 177
AppendText method
    File class, 298
    FileInfo class, 294
Application class, 366
application model (WPF), 366-367
    browser-hosted applications, 367
    standalone applications, 367
applications
    ADO.NET, 333
        arithmetic operators, 49
        array indexing, 197-198
        base class members, 103
        binary files, 302-303
        break statements, 160-161
classes
    abstract classes, 105
        creating, 71
        inheritance, 99
collection pooling, 332
collections, deriving, 208
constructor chaining, 100
do statements, 155
\a escape sequence, 169
DataTips, 354-355
Exception Assistant, 351-352
dynamic types, 446-447
data validation, 403
dating, 333
expressions side effects, 353-354
data source connections, 331
directory class, 300
directory information, 295-296
do statements, 155
dictionaries, 211-212
directory class, 300
directory information, 295-296
error messages, 441
error messages, 441
user-provided divisors, 350
variable windows, 352-354
dictionary, 211-212
directory class, 300
directory information, 295-296
do statements, 155
dynamic types, 446-447
error messages, 441
output, 441
runtime exceptions, 441
enumerations, 116-117
events, publishing/raising, 140-141
exception handling, 235-239
console window, 235
Exception assistant, 237
Exception details dialog, 237
JIT Debugger dialog, 237
JIT Debugging support, 236
multiple catch handlers in wrong order, 239
multiple exceptions, 238
runtime exception, 238
stack trace information, 236
Visual Studio with MSDN editions, 236
Windows Vista or later dialog, 235
expression lambdas, 282
File class, 300
FileInfo class, 295-296
flag enumerations, 118-119
foreach statements, 159
generic variant interfaces, 259-260
array contents, printing, 259
compiler errors, 259
results, 260
Hello world
class declaration, 29
default code, 28-29
directives, 28-29
namespace declaration, 29
running, 30
text, displaying, 29
if statements, 150
integer minimum values, finding
with generics, 249-250
without generics, 247-248
integer stacks, implementing, 216
interfaces, 108
LINQ queries
filtering data, 272
grouping data, 274-275
ordering data, 274-275
selecting data, 270-271
LINQ to SQL, 337-340
Add Connection dialog, 338
Add Item dialog, 337
expanded data connection, 339
O/R Designer, 337
O/R Designer displaying table, 339
results, 340
Server Explorer, 338
List<T> class, 205-206
logical/relational operators, 51
managed, 8-9
methods, 78
named arguments, 81
nullable types, 55
operator overloading, 123-124
optional parameters, 81
polymorphism, 99
predefined types, 40-41
properties, 74
queues, 217
scope and declaration space comparison, 68
sets, 214-215
standalone, 367
for statements, 367
strings
comparisons, 174
modifying, 175
substrings, creating, 172
switches, 152-153
text files, reading and writing, 305
tuples, 262
value type conversions, 58
web. See web applications
web-based photo viewer application
ASPX editor, 413
CSS, 416-418
data binding, 418-420
default page, 415
HTML, 416
layout, creating, 414-416, 414-415
layout styling, 416-418
New Project dialog, 412
Properties window, 413
selected element outline, 413
Visual Studio, viewing, 412
while statements, 154
WPF, creating, 370
Collection Editor, 373-374
completed layout, 378-379
controls, adding, 375
directories, choosing, 377
document outline, 372
event handlers, 375-376
grid row/column definitions, 374
grid rows/columns, creating, 373
New Project dialog, 370

How can we make this index more useful? Email us at indexes@samspublishing.com
Properties window, 372
routed events, 376
selected element outline, 372
ShowDialog extension method, 375
sizing grid rows/columns, 374
structure, editing, 373
tooltips, displaying, 377
Visual Studio, viewing, 371
XML bindings, debugging, 377-378
XML designer, 371
XML/Design tabs, 371-372
XML written, 364
layout styling. See WPF, layout styling
XML
documents, creating, 316, 319
modifications, 325
selecting, 321-322
architecture
DLR, 444
web applications, 408
ArgumentException, 229, 230
ArgumentNullException, 230
ArgumentOutOfRangeException, 230
arguments
checking, 231
named, 80
listing, 80
sample application, 81
optional parameters, 79
order, 80
positional, 80
validating, 230
array methods
DateTime type, 42
standard mathematical operations, 48
arithmetic operators, 47-48
increment/decrement, 48
incrementing a value listing, 48
overflow, 241
checked/unchecked blocks, 242
checked/unchecked expressions, 242
sample application, 49
standard mathematical operations, 48
Array class, 200
arrays, 196
Array class, 200
C# versus C, 196
five integer values, creating, 196
generic combinations, 257-258
indexing, 197-198
initializers, 198-200
listing, 198-199
multidimensional arrays, 199-200
single-dimensional arrays, 199
integers. See integers
jagged rectangular, 197
lists, compared, 203
multidimensional, 197
parameter, 75
size, 196
syntax, 196
AsOrdered() operator, 473
AsParallel() operator, 472
ASP.NET, 408-411
ASPX files, 409
CSS, 416-418
applying, 417-418
applying to elements, 417
child page links, 417
classes, 417
inline styles, compared, 418
master page links, 417
data binding, 418-420
embedded code blocks, 419
expressions, 418, 419
formatting strings, 419
text/navigation URL for controls, 418-419
data validation
client-side validation, 420-421
server-side validation, 420
validation controls, 420-422
Validator example, 422
event handlers
Global.asax, 411
Page, 410
Hello world application, 409-410
code listing, 409
code-behind file, 409
output, 410
Page directive, 409
master pages, 414-415
MVC Framework, 408
web-based photo viewer application, 411
ASPSX editor, 413
default page, 415
HTML, 416
layout, creating, 414-416
layout styling, 416-418
New Project dialog, 412
Properties window, 413
selected element outline, 413
Visual Studio, viewing, 412
ASPX files, 409
assembly identity attributes, 432
assembly manifest, 432
assembly manifest attributes, 433
AsSequential() operator, 472
assignment operators, 47-48
=, 47
compound, 48, 49
incrementing a value listing, 48
sample application, 49
atomization (XML namespaces), 319
Attribute method, 321
attributes, 428
applying, 428
applying multiple, 428
class, 417
Conditional, 431-432
CssClass, 417
custom, 433
applying, 434
creating, 433-434
retrieving at runtime, 434-435
#endif preprocessor symbol, 431-432
Flags, 118
global, 432-433
assembly identity, 432
assembly manifest, 432, 433
common, 433
informational, 433
#if preprocessor symbols, 431-432
Name, 376
names, 428
Obsolete, 430-431
parameters, 428
listing, 429
named, 429
positional, 429
runtime access, 434
multiple attributes, 435
single attributes, 434-435
targets
identifying, 430
listing of, 429
x:Class, 365
XML
adding, 324-325
changing values, 323
removing, 324
replacing data, 324
selecting, 321
values, changing, 323
Attributes method
FileInfo class, 294
XML attributes, selecting, 321
automatic memory management.
See garbage collection
automatic properties
declaring, 71
disadvantage, 72
automatic reset events, 466
AutoResetEvent class, 466
BigInteger type, 45-46
listing, 46
methods, 46
binary files, reading and writing, 301-303
listing, 302
ReadAllBytes method, 306
sample application, 302-303
WriteAllBytes method, 306
BinarySearch method, 200
BlockingCollection<T> class, 469
bool type, 37, 38
Boolean values, 38
boxing operations, 56
break statements, 152, 160-161
listing, 160
sample application, 160-161
breaking on exceptions, 351
breaking the stack, 240
breakpoints, 355-357
disabling, 356
enabling, 356
reenabling, 357
setting, 356
Visual Studio MSDN edition
features, 356-357
browser-hosted applications, 367
buffered streams, 303-304
BufferedStream class, 303
button Click event, 133-134
byte type, 37

How can we make this index more useful? Email us at indexes@samspublishing.com
C++ templates

calculated properties
creating, 73
declaring, 73
Call Stack window, 358-359
calling methods, 76
camel casing, 23
CancelEventArgs class, 138
CanRead method, 301
CanWrite method, 301
Capacity method
lists, 203
SortedDictionary<TKey, TValue> class, 210
SortedList<TKey, TValue> class, 210
Capture class, 189
Cascading Style Sheet. See CSS
case-sensitivity
identifiers, 23
strings, 173
catch handlers, 232-233
declaring, 233
variables, 233
chaining
constructors, 83, 84, 100
LINQ queries, 283
ChangeExtension method, 290
char type, 37
characters
encoding, 316
escape sequences, 168-169
CIL (common intermediate language), 9
class attribute, 417
class keyword, 65
class library (.NET Framework). See Framework class library
classes
abstract, 104-105
interface combinations, 107
listing, 104-105
sample application, 105
accessibility
access modifiers, 68, 69
default, 68
defined, 68
explicit declaration, 69
AddressChangedEventArgs, 140-141
Application, 366
Array, 200
AutoResetEvent, 466
BufferedStream, 303
CancelEventArgs, 138
Collection<T>
methods/properties, 206
sample application, 206-207
virtual methods, 207
CollectionViewSource, 396
concurrent collection, 469-470
Console, 29
constants, 70
Contact
creating, 71
default constructors, 82
dispose pattern, 455
extension methods, 88
LINQ query data, flattening, 277-278
LINQ query data, joining, 275-276
properties application, 74
creating, 71
CSS, 417
DataContext, 341-342
DataSet, 330
database tables, 331
populating, 334
DateTime, 88
declaration space
defined, 67
listing, 66
scope comparison
application, 68
statement blocks, 66
declaring, 29
defining, 64, 65
derived, 99-100
Dictionary<TKey, TValue>, 209
Directory
DirectoryInfo class,
compared, 298
listing, 297-298
methods, 297
sample application, 300
DirectoryInfo, 292-293
Directory class, compared, 298
listing, 293
methods, 293
sample application, 295-296
downcasting, 97
DynamicObject, 444
capsulation, 64-65
EventArgs, 137
Exception, 228-229
ExpandObject, 444, 445
ExternalException, 229
fields
constant-like behavior,
creating, 70
default values, 69
defined, 69
sample application, 71
File, 297
listing, 299-300
methods, 298-299
sample application, 300
FileInfo, 292
methods, 294
sample application, 295-296
generic
creating, 254-255
inheritance, 256-257
interface implementation, 257
type parameter
constraints, 252
variant interfaces, 258
HashSet<T>, 212-213
inheritance, 92-103
class hierarchies,
designing, 96
derived classes, 94-95
designing, 103
implementation, 96
interfaces, 106
member hiding, 101
member overriding, 101-103
multiple, 94
sample application, 99
sealing members, 103
trust but verify philosophy, 98
instantiating, 81-82
Interlocked, 466
JournalEntry
LINQ query data,
flattening, 277-278
LINQ query data, joining, 275-276
LinkedList<T>, 204
List<T>, 205-206
ManualResetEvent, 466
ManualResetEventSlim, 466
methods
calling, 76
declaring, 74-75
defined, 74
extension, 87-88
as input, 76
overloading, 76-77
sample application, 78
signatures, 77
static, 75
parameters. See
parameters
Monitor, 465
nested, 85-86
object initializers, 88-89
Parallel, 468
ParallelEnumerable, 472
partial, 86
Path, 290-291
Program, 29
properties
automatic, 71, 72
calculated, 73
declaring, 71-72
defined, 71
operations, 73
read-only, 73
sample application, 74
set accessor, 72
write-only, 73
Queue<T>, 216-217
ReaderWriterLockSlim, 467
ReadOnlyCollection<T>, 208-209
scope
declaration space
comparison application, 68
defined, 66
listing, 66
nesting and hiding, 66-67
statement blocks, 66
switches, 152
variables, 20
visibility, 66
SortedDictionary<TKey, TValue>, 210-211
SortedList<TKey, TValue>, 210-211
SortedSet<T>, 212-213
SpinLock, 466
Stack<T>
methods, 215
sample application, 216
static, 86-87
Stream, 300-301
StreamReader, 304-305
StreamWriter, 304-305
String, 171, 175
StringBuilder, 177
Append/AppendFormat
methods, 177
capacity, 178
listing, 177
structures, compared,
119-120, 125-126
SystemException, 229
Task, 471
tuples, 261
upcasting, 97
Uri, 44-45
methods, 44
properties, 45
UriBuilder, 45
WPInteropExtensions, 375
XAttribute, 314
XDocument class, 313
XElement, 314
navigation properties, 320
SetAttributeDefaultValue method, 315
SetElementValue method, 315
values, retrieving, 316
XName, 317
XNamespace, 317-319
XNode, 319-320
classes
   group, 273-274
   join, 276
   orderby, 273
Clear method
   Array class, 200
   Dictionary<TKey, TValue> class, 209
   HashSet<T> class, 212
   Queue<T> class, 216
   SortedDictionary<TKey, TValue> class, 210
   SortedList<TKey, TValue> class, 210
   SortedSet<T> classes, 212
   Stack<T> class, 215
ClearItems method, 207
CLI (common language infrastructure), 14
Click events, 133-134
client-side validation, 420-421
Close method, 301
CLR (common language runtime), 8-9
code
   comments, 348
   benefits, 349
delimited, 348-349
   syntax, 348
   writing, 349
   XML, 348
debugging
   breaking on exceptions, 351
   compiler errors, 349
DataTips, 354-355
   Exception Assistant, 351-352
   expression side effects, 353-354
   runtime errors, 349
   user-provided divisors, 350
   variable windows, 352-354
   XAML bindings, 377-378
   listings. See listings
maintainable, 64
   unit tests, 349-350
code-behind, 365-366
coding errors, 231
Collection Editor dialog, 373-374
collection views, 395-396
current item pointers, 398-399
collection views, 395
   default, 396
   filtering data, 398
   grouping, 397
   sorting, 396-397
Collection<T> class
   methods/properties, 206
   sample application, 206-207
   virtual methods, 207
collections, 203
current item pointers, 398-399
collection views, 395-396
   filtering data, 398
   grouping data, 397
I NotifyCollectionChanged interface, 396
I NotifyPropertyChanged interface, 396
photo viewer application example, 395
sorting data, 396-397
Collection<T> class, 206-207
   methods/properties, 206
   sample application, 206-207
   virtual methods, 207
concurrent collection classes, 469-470
deriving, 208
dictionaries, 209-212
   Dictionary<TKey, TValue> class, 209
   sample application, 211-212
   sorting elements, 210-211
generic, 203
initializers, 217-218
   Add method, 218
   complex, 218-219
   listing, 218
   syntax, 218
   interfaces, 218-220
   specific collection behaviors, 219
   supporting implementations, 219-220
lists
   arrays, compared, 203
capacity, 203
   LinkedList<T> class, 204
   
   get-process, 482
   select, 482
cmdlets, 482
cmdlets, 482
cmdlets, 482
null
console applications

Visual Studio with MSDN editions, 236
Windows Vista or later dialog, 235
FileInfo class, 295-296
generic variant interfaces, 259-260
array contents, printing, 259
compiler errors, 259
results, 260
Hello world
class declaration, 29
default code, 28-29
directives, 28-29
namespace declaration, 29
running, 30
text, displaying, 29
integer minimum values, finding
with generics, 249-250
without generics, 247-248
integer stacks, implementing, 216
LINQ to SQL, 337-340
Add Connection dialog, 338
Add Item dialog, 337
dispensible data
connection, 339
O/R Designer, 337
O/R Designer displaying table, 339
results, 340
Server Explorer, 338
List<T> class, 205-206
logical/relational operators, 51
methods, 78
nullable types, 55
operator overloading, 123-124
polymorphism, 99
predefined types, 40-41
properties, 74
queues, 217
scope and declaration space comparison, 68
sets, 214-215
substrings, 172
tuples, 262
value type conversions, 58
Console class, 29
cursor window, 29
constants
declaring, 21, 22
defined, 21, 70
magic numbers, 21-22
values, 70
variables, compared, 21
constraints
generic methods, 253
generic type parameters, 250-252
class listing, 252
listing of, 251
method listing, 252
multiple, 251
value equality testing, 251-252
constructors
chaining, 83-84, 100
default, 82
derived classes, 99-100
overloading, 82-83
specialized, 82-83
static, 85
structures, 124-125
Contact class
creating, 71
default constructors, 82
dispose pattern, 455
extension methods, 88
LINQ query data, flattening, 277-278
joining, 275-276
properties application, 74
Contains method
HashSet<T> class, 212
Queue<T> class, 216
ReadOnlyCollection<T> class, 208-209
SortedSet<T> class, 212
Stack<T> class, 215
string comparisons, 173
ContainsKey method
Dictionary<TKey, TValue> class, 209
SortedDictionary<TKey, TValue> class, 210
SortedList<TKey, TValue> class, 210
contextual keywords, 24
continue statements, 161-162
listing, 161
sample application, 161-162
contravariance
generic interfaces, 258
interfaces, extending, 260-261
control flow statements
iteration statements, 153
do, 154-155
for, 155-158
foreach, 158-159
while, 153-154
jump statements, 159
break, 160-161
continue, 161-162
return, 162
selection statements, 148
if, 148-150
switches, 150-153
controls
adding to WPF applications, 375
validation, 420-422
ASPX page, 422
combining, 421
error messages, summarizing, 421
listing of, 420-421
properties, 421
ControlToValidate property, 421
conversion operators, 122
conversions
boxing/unboxing operations, 56
dynamic, 442
value types to reference
types, 54
boxing/unboxing operations, 56
explicit conversions, 57
implicit conversions, 56
sample application, 58
Convert method, 391, 393
ConvertBack method, 393
ConverterParameter property, 393
Copy method, 298
CopyTo method
FileInfo class, 294
Stream class, 301
corrupted state exceptions, 239
Count method
Dictionary<TKey, TValue> class, 209
HashSet<T> class, 212
Queue<T> class, 216
ReadOnlyValueCollection<T> class, 208-209
SortedDictionary<TKey, TValue> class, 210
SortedList<TKey, TValue> class, 210
SortedSet<T> class, 212
Stack<T> class, 215
covariance, 258, 260, 261
Create method
DirectoryInfo class, 292
File class, 299
FileInfo class, 294
CreateDirectory method, 297
CreateNavigator method, 321
CreateSubdirectory method, 292
CreateText method
File class, 299
FileInfo class, 294
CSS (Cascading Style Sheet),
ASP.NET applications, 416-418
applying, 417-418
applying to elements, 417
child page links, 417
classes, 417
inline styles, compared, 418
master page links, 417
CssClass attribute, 417
CTS (common type system), 9-10
CLS (common language specification), 11
type safety, 10
culture parameter, 393
current item pointers, 398-399
custom attributes, 433
applying, 434
creating, 433-434
retrieving at runtime, 434-435
custom cmdlets, creating, 483
custom dynamic types
creating, 445-446
functionality, 446
custom format strings, 183-186
DateTime instance listing, 183-186
specifiers, 183-186
custom indexers, creating, 201-203
custom validation rules, 401
CustomValidator control, 421
data binding (ASP.NET), 418-420
embedded code blocks, 419
expressions, 418-419
formatting strings, 419
text/navigation URL for controls, 418-419
data binding (WPF), 369, 386
binding to collections, 395
collection views, 395-396
current item pointers, 398-399
filtering data, 398
grouping data, 397
INotifyCollectionChanged interface, 396
INotifyPropertyChanged interface, 396
photo viewer application example, 395
sorting data, 396-397
components, 386

How can we make this index more useful? Email us at indexes@samspublishing.com
data binding (WPF)

creating, 387
data converters, 390
adding to XAML file, 391
creating, 390-391
culture-aware, 393
multivalue, 392-393
multivalue with Converter
parameter, 394
photo viewer application, 394
event handlers, 389
OneTime, 387
OneWay, 386
OneWayToSource, 387
photo viewer application, 389-390
source object, 388-389
source property, updating, 387
target objects, 386
target properties, 386
TwoWay, 387
validating data, 400-403
AddressBook application, 403
custom validation rules, 401
validation checks, adding, 401
visual feedback, 402
XAML application code, 389
data converters (WPF data
binding), 390
adding to XAML file, 391
creating, 390-391
culture-aware, 393
multivalue, 392-393
multivalue with Converter
parameter, 394
photo viewer application, 394
data hiding, 64-65
data parallelism, 468
ForEach method example, 468
guidelines, 473-474
loop execution, controlling, 468
Parallel class, 468
thread-safe collections, 469-470
Data property, 229
data providers
ADO.NET, 330
EntityClient, 481
data templates, 399-400
associating with controls, 400
defining, 399-400
data validation, 400-403
AddressBook application, 403
custom validation rules, 401
validation checks, adding, 401
visual feedback, 402
web applications, 420
client-side, 420-421
server-side, 420
validation controls, 420-422
Validator example, 422
databases
ADO.NET, 330
connection pooling, 332
data providers, 330
data source connections, 331
DataSet class, 330-331
queries, 332
required references, 331
sample application, 333
LINQ to ADO.NET, 333
LINQ to DataSet, 333-335
LINQ to SQL. See LINQ to
SQL
prerequisites, 329-330
read-only access, 331
records
adding, 340
deleting, 341
updating, 340-341
DataContext class, 341-342
DataSet class, 330
database tables, 331
populating, 334
DataTips, 354-355
floating, 354
pinning, 354
sharing, 355
date and time values
DateTime type, 39-42
arithmetic methods, 42
properties, 41-42
TimeSpan type, 42-43
Date property, 41
dateOfBirth field, 71
DateOfBirth property, 74
DateTime class, 88
DateTime type, 39-42
arithmetic methods, 42
properties, 41-42
Day property, 41
DayOfWeek property, 41
Days enumeration
as flag enumeration, 118-119
implementing, 116-117
Days method, 42
deadlocks, 463
debugging code
errors
compiler, 349
runtime, 349
user-provided divisors, 350
Visual Studio debugger, 350
breaking on exceptions, 351
DirectoryName method, 294
Display property, 421
disposable types, 454
Dispose method
disposable types, 454
dispose pattern, 456
dispose pattern
benefits, 456
Contact class example, 455
Dispose method, 456
implementing, 455
division operator (/), 48
DivRem method, 46
DLR (dynamic language runtime),
8, 16, 444
architecture, 444
custom dynamic types, 446
creating, 445-447
functionality, 446
dynamic operations, defining, 445
ExpandObject class, 445
IDynamicMetaObjectProvider, 444
interoperability support, 444
language binders, 444
do statements, 154-155
listing, 155
sample application, 155
syntax, 154
DockPanel, creating, 375
document support (WPF), 369
DOM (Document Object Model),
XML, 312-313
creating XML, 314-315
XPath queries, 321
double type, 37, 38
downcasting, 97
dynamic keyword, 440
dynamic language runtime.
See DLR
dynamic languages, 9
dynamic types, 39, 439
COM interoperability, 447
conversions, 442
custom
application, 446-447
creating, 445-446
functionality, 446
dynamic keyword, 439
methods, invoking, 449
overload resolution, 442-443
runtime, 441-442
sample application, 440-441
variables, 440
DynamicObject class, 444

E
Ecma International, 14
ElapsedEventHandler delegate,
132-133
embedded ASP.NET code
blocks, 419
Empty method, 44
empty strings, 170-171
testing, 170
whitespace characters,
170-171
capsulation, 64-65
#endif preprocessor symbol,
431-432
EndsWith method, 173
Enqueue method, 216
Enter method, 465
EnterReaderLock method, 467
EnterWriterLock method, 467
Entity Data Model Designer,
480-481
Entity Data Model Wizard, 481
Entity Framework, 479-480
conceptual model queries, 480
data models, creating, 481
Entity Data Model Designer,
480-481
Entity Data Model Wizard,
481
EntityClient data provider, 481
querying, 481-482
EntityClient data provider, 481
enum keyword, 114
EnumerateDirectories method
Directory class, 297
DirectoryInfo class, 292
EnumerateFiles method
Directory class, 297
DirectoryInfo class, 292
EnumerateFileSystemEntries
method, 297
EnumerateFileSystemInfos
method, 293
tenumeralions, 114, 220
Days
as flag enumeration,
118-119
implementing, 116-117
defining, 114
flag, 117-119
Flags attribute, 118
listing, 117-118
sample application,
118-119
values, combining, 118
listing, 114-115
multiple named values, 115
operations, 116
sample application, 116-117
underlying types, 116
values
commas, 114
numeric, 115-116
explicit interface implementation,
109
explicit keyword, 122
Exposure Time converter, 391
expressions
ASP.NET data binding, 418-419
defined, 19
lambdas, 281-282
examples, 281
method calls, 282
sample application, 282
regular, 187
classes, 189
compatibility, 187
metacharacters, 188
string validation, 189-190
substring matches, 190
side effects, 353-354
for statements, 156
extending
generic variant interfaces,
260-261
interfaces, 106
Extensible Application Markup
Language. See XAML
Extensible Markup Language. See
XML
extension methods, 87-88, 294
ExternalException class, 229
extracting substrings, 171

F
\f escape sequence, 169
fall through (classes), 152
fields, 20
constant-like behavior,
creating, 70
dateOfBirth field, 71
default values, 53, 69
defined, 69
firstName, 70
lastName, 70
listing, 20-21
public, 126
read-only, 21
sample application, 71
File class, 297
listing, 299-300
methods, 298-299
sample application, 300
FileInfo class, 292
listing, 295
methods, 294
sample application, 295-296
files, 290
ASPX, 409
binary, reading and writing,
301-303, 306
listing, 302
ReadAllBytes method, 306
sample application, 302-303
WriteAllBytes method, 306
File class, 297
listing, 299-300
methods, 298-299
sample application, 300
FileInfo class, 292
listing, 295
methods, 294
sample application, 295-296
paths, 290-291
absolute, 290
Path class, 290-291
relative, 290
text, reading and writing,
304-307
filtering
collection views, 398
LINQ queries, 271-272
finalizers, 456
implementing, 456-457
rules, 457
writing, 458
finally handlers, 232
Find method, 200
FindAll method, 200
finding special directories, 291
FirstAttribute property, 320
firstName field, 70
FirstName property, 74
FirstNode property, 319-320
flag enumerations, 117-119
Flags attribute, 118
listing, 117-118
sample application, 118-119
values, combining, 118
Flags attribute, 118
FlagsAttribute attribute, 428
flattening LINQ query data,
277-278
Contact class and
JournalEntry class listing,
277-278
enumerable collections, 278
listing, 278
float type, 37
Flush method, 301
for statements, 155-158
expressions, 156
infinite loops, 156
initializer, 156
sample application, 157-158
syntax, 156
while statements, compared,
156-157
ForAll() operator, 473
ForEach method
Array class, 200
data parallelism, 468
foreach statements, 158-159
iteration variables, 158
listing, 158
sample application, 159
syntax, 158

formatting
composite formatting, 186-187
types, 178

Fragment property, 45
Framework class library, 8, 13
available types, 13
Base Class Libraries, 14
functional areas, 14
namespaces, 14-16
common, 15-16
type names, 15

FromDays method, 42
FromHours method, 42
FromMilliseconds method, 43
FromMinutes method, 43
FromSeconds method, 43

FullName method
declaring, 73
DirectoryInfo class, 293
FileInfo class, 294

g

garbage collection, 452
dispose pattern, 455
benefits, 456
Contact class example, 455
Dispose method, 456
implementing, 455
finalizers, 456
implementing, 456-457
rules, 457

generics
array combinations, 257-258
C++ templates, compared, 249
classes
creating, 254-255
inheritance, 256-257
interface implementation, 257
collections, 203
integer minimum values,
finding
with generics application, 249-250
with generics code listing, 248-249
objects, 246-247
without generics
application, 247-248
without generics code
listing, 246
interfaces, 255-256, 258-261
Java generics, compared, 249
methods, 253
calling, 253-254
constraints, 253
nongeneric classes, 253
printing array items
example, 257-258
type inference, 254
type parameter hiding, 253
type parameters, 254
structures, 255
type parameters, 250
constraints, 250-252
default values, 252
multiple, 250
type safety, 247
value equality, 251-252
get accessor (indexers), 201
GetAttributes method, 299
GetCurrentDirectory method, 297
GetCustomAttribute method,
434-435
GetDirectoryName method, 290
GetDirectoryRoot method, 297
GetEnumerator method, 220
GetExtension method, 290
GetFileName method, 290
GetFileNameWithoutExtension
method, 290
GetFolderPath method, 291-292
GetLogicalDrives method, 297
GetParent method, 297
GetPathRoot method, 290
gt-process cmdlet, 482
GetRandomFileName method,
290
GetTempFileName method, 291
GetTempPath method, 291
global attributes, 432-433
assembly identity, 432
assembly manifest, 432-433
common, 433
informational, 433
Global.asax event handlers, 411
globally unique identifiers
(GUIDs), 43-44
goto statements, 159
graphics (WPF), 369
greater than operator (>) , 50
greater than or equals operator
(>=), 50

How can we make this index more useful? Email us at indexes@samspublishing.com
GreatestCommonDivisor method

GreatestCommonDivisor method, 46
Group class, 189
group clause (LINQ queries), 273-274
group joins (LINQ queries), 276-277
grouping
  collection views, 397
  LINQ query data, 273
  listings, 273-274
  sample application, 274-275
GUIDs (globally unique identifiers), 43-44

H

handlers
  events
    anonymous methods, 136
    attaching to events, 132-133
    Global.asax, 411
    Page, 410
    raising events, 139
    routed events, 376
    WPF applications, adding, 375-376
    WPF data binding, 389
exceptions, 232
  catch handler variable, 233
  catch handlers, 232-233
  catch handlers, declaring, 233
  cleanup activities, 234
corrupted state
  exceptions, 239
critical system exceptions, avoiding, 239
finally handler, 232
multiple exceptions, catching, 233-234
nonspecific exceptions, avoiding, 239
protected regions, 232-233
sample application, 235-239
swallowing exceptions, 234
tasks, 471-472
HashSet<T> class, 212-213
heap memory, 452
Hello world application
  ASP.NET, 409-410
  code listing, 409
  code-behind file, 409
  output, 410
  Page directive, 409
  class declaration, 29
  default code, 28-29
directives, 28-29
  namespace declaration, 29
running, 30
text, displaying, 29
HelpLink property, 229
hiding
  base class members, 101
  scopes, 66-67
Host property, 45
Hour property, 41
Hours method, 43
HTTP (Hypertext Transfer Protocol), 408
Internet Information Services (IIS)

Internet Information Services (IIS), 410
interoperability
  COM, 447
    with dynamic types
      example, 448
  Primary Interop
    Assemblies, 447
    without dynamic types
      example, 447-448
  reflection, 448-449
    invoking methods
      dynamically, 449
    invoking methods in C#, 448
  invoking methods
    reflectively, 448
  IronPython dynamic objects, 449
IntersectWith method, 212
InvalidOperationException, 229, 230
Invoke method, 470
invoking methods
  C#, 448
    dynamically, 449
    reflectively, 448
IronPython dynamic objects, 449
ISet<T> interface, 219
IsEven method, 46
IsFile method, 44
IsNamespaceDeclaration property, 317
IsNullOrEmpty method, 170
IsNullOrWhiteSpace method, 170-171
IsOne method, 46
IsProperSubsetOf method, 213
IsProperSupersetOf method, 213
IsReadOnly method, 294
IsSubsetOf method, 213
IsSupersetOf method, 213
IsZero method, 46
iteration statements, 153
  do, 154-155
    listing, 155
    sample application, 155
    syntax, 154
  for, 155-158
    expressions, 156
    infinite loops, 156
    initializer, 156
    sample application, 157-158
    syntax, 156
    while statements, 156-157
    compared, 156-157
    foreach, 158-159
    iteration variables, 158
    listing, 158
    sample application, 159
    syntax, 158
  while, 153-154
    listing, 154
    sample application, 154
    for statements, compared, 156-157
    syntax, 153
iteration variables, 158
iterators, 220
  complex, 222
  foreach statement listings, 220
  iterators, 215
  listing, 221-222
  multiple iterations over same
    source, 221
  ordered sequence of values, 221
IValueConverter interface, 391
J
  jagged rectangular arrays, 197
  Java generics, 249
  JIT (Just-in-Time) compiler, 12
  JIT Debugger dialog, 237
  join clause (LINQ query data), 276
  Join method, 176-177
  joining
    LINQ query data, 275-277
      Contact class and
        JournalEntry class
        listing, 275-276
        equals operator, 277
        group joins, 276-277
        join clause, 276
        ordering, 276-277
        strings, 176-177
    JournalEntry class (LINQ query
data)
      flattening, 277-278
      joining, 275-276
  jump statements, 159
    break, 160-161
      listing, 160
      sample application, 160
    continue, 161-162
      listing, 161
      sample application, 161-162
    return, 162
  Just-in-Time compilations, 12
K
  Keys method, 209
  keywords
    class, 65
    common, 23-24
How can we make this index more useful? Email us at indexes@samspublishing.com
LINQ queries

SQL syntax, compared, 269
standard query operator
methods, 279-280
syntax, 268
XElement class, 320-321

LINQ to ADO.NET, 333
LINQ to DataSet, 333-335
queries, 334
required references, 335
LINQ to SQL, 335-342
adding database records, 340
DataContext class, 341-342
deleting database records, 341
object model, creating, 335-336
projection, 340
queries, 336
required references, 336
sample application, 337-340
updating database records, 340-341

LINQ to DataSet, 333-335
queries, 334
required references, 335

LINQ to SQL, 335-342
database records
adding, 340
deleting, 341
updating, 340-341
DataContext class, 341-342
object model, creating, 335-336
projection, 340
queries, 336
required references, 336
sample application, 337-340
Add Connection dialog, 338
Add Item dialog, 337
expanded data
connection, 339
O/R Designer, 337, 339
results, 340
Server Explorer, 338

LINQ to XML, 313
character encoding, 316
creating XML, 314
modification application, 325
namespaces
atomization, 319
creating XML, 317-318
declaring, 317
expanded names, 319
prefixes, 318
sample application, 319
selecting XML, 321-322
SetAttributeValue method, 315
SetElementValue method, 315
source XML, transforming, 323
XAttribute class, 314
XDocument class, 313
XElement class, 314, 316
XML documents, creating, 316
XName class, 317
XNamespace class, 317-319
XPath queries, 321

Lippert, Eric, xv-xvi
List<T> class application, 205-206

listings
abstract class, 104-105
Add method, 325
ADO.NET query, 332
array initializers, 198-199
ASP.NET
binding expressions, adding, 419
child page stylesheet links, 417
CSS styles, applying, 417-418
Hello world application, 409
master page stylesheet links, 417
Page_Load method, 418
validation controls, 422
attributes
Conditional, 431-432
FlagsAttribute attribute, 428
multiple attributes, 428
Obsolete, 430-431
parameters, 429
runtime access, 435
target identification, 430

BigInteger type, 46
binary files, reading and writing, 302, 306
binding to collections, 395
break statements, 160
buffered streams, 303-304
calculated properties, 73
classes versus structures, 125-126
code-behind class, 365-366
collection initializers, 218
collection views
current item pointers, 399
filtering, 398
grouping, 397
sorting, 397
COM interoperability
with dynamic types, 448
without dynamic types, 447-448
collection initializers, 218-219
complex iterators, 222
constructor chaining, 84
continue statements, 161
creating XML, 314-315
custom attributes
applying, 434
creating, 433-434
custom cmdlets, creating, 483
custom dynamic types
creating, 445-446
functionality, 446
data bindings, creating, 387
data parallelism, ForEach method, 468
data templates
associating with controls, 400
defining, 400
database records
adding, 340
deleting, 341
updating, 340-341
DataContext class, 342
default constructors,
declaring, 82
derived class constructors, 99-100
Directory class, 297-298
DirectoryInfo class, 293
dispose pattern, 455
do statements, 155
DockPanel, creating, 375
dynamic types
implicit conversions, 442
overload conversions, 443
empty strings
testing, 170
whitespace characters, 170-171
Entity Framework, querying, 481-482
enumerations, 114-116
event publishing
custom EventArgs derived class, 137
field-like syntax, 137
property-like syntax, 138
events
raising, 139
subscribing, 132-133
exception handling
catch handlers, declaring, 233
multiple exceptions, catching, 233-234
ExpandObject class, 445
Exposure Time converter, 391
fields, 20-21
File class, 299-300
FileInfo class, 295
finalizers, implementing, 456-457
flag enumerations, 117-118
foreach statements, 158
foreach statement iterator, 220
generic class inheritance, 256
open constructed classes, 257
open constructed classes with constraints, 257
generic methods
calling, 253-254
type inference, 254
type parameter hiding, 253
generic type parameter constraints
classes, 252
methods, 252
generic variant interfaces
covariant/contravariant, extending, 260, 261
extending, 260
grid row/column definitions, 374
IDisposable interface, 453
if statements
cascading, 149
nesting, 148
implementation inheritance, 96
incrementing a value, 48
integer arithmetic operations
checked/unchecked blocks, 242
checked/unchecked expressions, 242
integer minimum values, finding
with generics, 248-249
objects, 246-247
without generics code listing, 246
invoking methods
C#, 448
dynamically, 449
reflectively, 448
iterators, 221-222
label control styles, 380-381
lambdas
expression, 281
statement, 282-283
LINQ queries
against XElement class, 320-321
Contact object collection, 268
data selection, 269-270
enumerable collections, 278
filtering data, 272
flattening data, 277-278
group joins, 277
joining data, 275-276
standard query operator methods, 280
LINQ to DataSet query, 334
LINQ to SQL query, 336
lock statement, 464-465
methods declaring, 74-75
overloading, 77
multiple interface inheritance, 108
multivalue converters
adding to XAML, 393
Converter parameter, 394
size example, 392
named arguments, 80
null-coalescing operator, 54
object initializers, 88-89
overloading constructors, 83
photo viewer application layout, 378-379
PLINQ query, 473
printing array items with generic method, 257-258
read-only fields, 21
Remove method, 324
ReplaceWith method, 324
reshaped XML, returning, 322-323
resource dictionaries, 380
rethrowing exceptions, 240
scope and declaration space, 66
SetValue method, 323
source XML, transforming, 323
strings
  composite formatting, 187
  concatenation, 176
custom format strings, 183-186
joining and splitting, 176-177
litertals, 169
standard format, 181
StringBuilder class, 177
ToString method,
  overriding, 182
validation, 190
substrings, creating, 171-172
switches, 151-152
tasks
creating, 470-471
exception handling, 471-472
waiting to complete, 471
text files, reading and writing, 304-305, 307
ToString method, overriding, 178-179
trust but verify philosophy, 98
UriBuilder class, 45
using statement, 454
validating data
custom validation rules, 401
validation checks, adding, 401
visual feedback, 402
web-based photo viewer application, 416
while statements, 154
WPF
  application event handlers, 375-376
data binding, 389
wrapping exceptions, 241
XAML, 365
XML, creating
  DOM, 312
  LINQ to XML, 314
namespace prefixes, 318
namespaces, 317-318
XML tree node navigation, 320
arrays, compared, 203
capacity, 203
LinkedList<T> class, 204
List<T> class
  application, 205-206
  methods/properties, 204
strings
  character escape sequences, 168-169
  listing, 169
  verbatim, 168-169
values, 21-22
local variables, 20
LocalPath method, 44
Locals window, 352-353
lock statement, 464
Enter/TryEnter methods, 465
listing, 464-465
lock expressions to avoid, 464
Monitor class, 465
locks, 463-466
exclusive, 464
lock statement, 464
Enter/TryEnter methods, 465
expressions to avoid, 465
listing, 464-465
Monitor class, 465
SpinLock class, 466
Wait method, 465
logical operators, 50-51
   AND (&), 47
   listing of, 51
   OR (|), 47
   rules, 51
   sample application, 51
   XOR, 47
long type, 37
loops
   infinite, 156
   statements. See iteration statements
M
magic numbers, 21-22
Main function, 29
maintainable code, 64
MakeRelativeUri method, 44
managed applications, 8-9
managed code, 8, 17
managed threads, 462
managing memory. See memory management
manipulating strings, 171
manual reset events, 466
ManualResetEvent class, 466
ManualResetEventSlim class, 466
markup, 364-365
master pages (ASP.NET), 414-415
Match class, 189
MatchCollection class, 189
Max method
   BigInteger type, 46
   HashSet<T> class, 212
   SortedSet<T> class, 212
memory management
   automatic, 12-13
   dispose pattern, 455
   benefits, 456
   Contact class example, 455
   Dispose method, 456
   implementing, 455
   finalizers, 456
   implementing, 456-457
   rules, 457
   writing, 458
   garbage collection, 453
   heap, 452
   IDisposable interface, 453
   stack, 452
   using statement, 454
   compiler generated code, 454-455
   syntax, 454
Message property, 228
method group inference, 133
methods
   Add
      collection initializers, 218
      XML elements, 324-325
   AddAfterSelf, 324
   AddBeforeSelf, 324
   AddFirst, 324
   anonymous, 136
   Append, 177
   AppendFormat, 177
   arithmetic, 42
   Array class, 200
   Attribute, 321
   Attributes, 321
   BigInteger type, 46
   calling, 76
   Collection<T> class, 206-207
   Compare, 173
   CompareOrdinal, 173
   CompareTo, 173
   Contains, 173
   Convert, 391, 393
   ConvertBack, 393
   declaring, 74-75
   defined, 74
   Dictionary<TKey, TValue> class, 209
   Directory class, 297
   DirectoryInfo class, 293
   Dispose
disposable types, 454
dispose pattern, 456
   EndsWith, 173
   Enter, 465
   EnterReaderLock, 467
   EnterWriterLock, 467
   event raiser, 140
   extension, 87-88
   File class, 298-299
   FileInfo class, 294
   finalizers, 456
   implementing, 456-457
   rules, 457
   writing, 458
   ForEach
      Array class, 200
data parallelism, 468
generic, 253
calling, 253-254
   constraints, 253
   nongeneric classes, 253
   printing array items
      example, 257-258
type inference, 254
type parameters, 254
type parameters, constraints, 252
type parameters, hiding, 253
   GetCustomAttribute, 434-435
   GetEnumerator, 220
methods

GetFolderPath, 291-292
Guid type, 40-44
HashSet<T> class, 213
IndexOf, 172
as input, 76
InsertOnSubmit, 340
Invoke, 470
invoking
C#, 448
dynamically, 449
reflectively, 448
IsNullOrEmpty, 170
IsNullOrWhiteSpace, 170-171
Join, 176-177
LastIndexOf, 172
List<T> class, 204
onPropertyChanged, 388
overloading, 76-77
example listing, 77
return types, 77
signatures, 77
PadLeft, 175
PadRight, 175
Page_Load, 418
parameters, 75-76
output, 76
reference, 76
value, 75
optional. See optional parameters
Path class, 290-291
Queue<T> class, 216-217
ReadOnlyCollection<T> class, 208-209
Remove
string characters, 175
XML elements/attributes, 324
RemoveAttributes, 324
RemoveNodes, 324
Replace, 175
ReplaceAll, 324
ReplaceAttributes, 324
ReplaceNodes, 324
ReplaceWith, 316-324
sample application, 78
Search, 78-79
SetAttributeValue, 315
SetElementValue, 315
SetValue, 323
ShowDialog, 375
signatures, 77
SortedList<TKey, TValue> class, 210-211
Standard query operator, 279-280
StartNew, 471
StartsWith, 173
static, 75
Stream class, 301
string comparisons, 173-174
structures, 120
TimeSpan type, 43
ToLowerInvariant, 174
ToUpperInvariant, 174
Trim, 175
TrimEnd, 175
TrimStart, 175
VerifyEmailAddress
calling, 76
declaring, 74-75
Wait, 465
Task class, 471
thread signals, 466
WaitAll, 471
WaitAny, 471
XPath namespace, 321. See also properties
Milliseconds method, 43
Min method
BigInteger type, 46
HashSet<T> class, 212
SortedSet<T> class, 212
MinusOne method, 46
Minute property, 41
Minutes method, 43
mismatched else problem, 149
modifying
indexers, 201
strings, 174-175
XML, 323
adding elements, 324-325
changing data, 323
removing
elements/attributes, 324
replacing data, 324
sample application, 325
modulus operator (%), 48
Monitor class, 465
Month property, 41
Move method
Directory class, 297
File class, 299
MoveTo method
DirectoryInfo class, 293
FileInfo class, 294
multidimensional arrays, 197
multiple exceptions, catching, 233-234
multiple inheritance, 94, 108
multiple threads, 462
multiplication operator (*), 48
multiplicative operators, 47
multithreading events, 140
multivalue converters, 392-393
adding to XAML, 393
Converter parameter, 394
size example, 392
mutable strings
appending data, 177
characters, replacing, 178
data, adding/deleting, 178
StringBuilder class, 177
capacity, 178
listing, 177
mutex (thread synchronization), 467

N

\n escape sequence, 169
{n} regular expression
metacharacter, 188
{n, } regular expression
metacharacter, 188
Name attribute, 376
Name method
DirectoryInfo class, 293
FileInfo class, 294
?<name> regular expression
metacharacter, 188
?name' regular expression
metacharacter, 188
named arguments, 80
listing, 80
sample application, 81
named parameters, 429
names
attributes, 428
expanded, 319
identifiers, 23
XML, 317
namespaces
declaring, 29
Framework class library, 14-16
common, 15-16
type names, 15
System, 15
System.Collections.Generic, 15
System.Data, 15
System.Diagnostics, 15
System.Globalization, 15
System.Linq, 16
System.Net, 16
System.ServiceModel, 16
System.Text, 16
System.Web, 16
System.Windows, 16
System.Windows.Controls, 16
System.Windows.Forms, 16
System.Xml, 16
XML, 317
atomization, 319
creating XML, 317-318
declaring, 317
expanded names, 319
prefixes, 318
sample application, 319
XPath, 321
navigating XML tree nodes
LINQ queries against
XElement class, 320-321
properties
XElement, 320
XNode, 319-320
nesting
classes, 85-86
if statements, 148
scopes, 66-67
.NET Framework
class library, 8, 13
available types, 13
Base Class Libraries, 14
functional areas, 14
namespaces, 14-16
CLR (common language runtime)
common intermediate language, 9
CTS (common type system), 9-10
memory management, 12-13
virtual execution system, 9-12
CLS (common language specification), 11, 38-39
components, 8
CTS (common type system), 9-10
CLS (common language specification), 11
type safety, 10
dynamic language runtime.
See DLR
evolution, 344
functions, 484
JIT (Just-in-Time) compiler, 12
LINQ (Language Integrated Query), 268
managed applications, 8-9
managed code/unmanaged code, 8
parallel computing platform, 8, 16-17
Silverlight version, 203
New Project dialog box, 27-28
NewGuid method, 44
next statements, 359
NextAttribute property, 320
NextNode property, 319-320
nodes (XML tree, navigating)
LINQ queries against
XElement class, 320-321

How can we make this index more useful? Email us at indexes@samspublishing.com
nodes (XML tree, navigating)

properties
  XElement, 320
  XNode, 319-320
non-generic collections, 203
not equals operator (!=), 50
not operator (!), 51
Now property, 41
{n,m} regular expression
  metacharacter, 188
null types, 53
nullable types
  defined, 54
  null-coalescing operators, 54
  sample application, 55
  syntax, 54
  values, 54
null-coalescing operator (??), 54
null-coalescing operators, 47
NullReferenceException, 229
'?number' regular expression
  metacharacter, 188
?<number> regular expression
  metacharacter, 188

\o escape sequence, 169
object initializers, 88-89
object type, 38-39
object-oriented programming, 18
  benefits, 64
  encapsulation, 64-65
  polymorphism, 95
object lifetime, 453
Obsolete attribute, 430-431
OnApplicationStartup event
  handler, 389
One method, 46
OneTime data binding, 387
OneWay data binding, 386
OneWayToSource data binding, 387
onPropertyChanged method, 388
Open method, 294
OpenRead method
  File class, 299, 301
  FileInfo class, 294
OpenText method
  File class, 299
  FileInfo class, 294
OpenWrite method
  File class, 299, 301
  FileInfo class, 294
operators
  additive, 47
  arithmetic, 47-48
  increment/decrement, 48
  incrementing a value
    listing, 48
  sample application, 49
  standard mathematical
    operations, 48
  assignment, 47-48
    =, 47
    compound, 48
    incrementing a value
      listing, 48
    sample application, 49
  conditional, 47, 52
  right-associative, 52
  short-circuit evaluation, 52
  syntax, 52
  types, 52
  conditional
    AND (&&,), 50
    OR (||), 50
  conversion, 122
  defined, 47
  enumerations, 116
  equality, 47
  exclusive OR operator (^), 50
  increment/decrement, 48
  incrementing a value listing, 48
  integer arithmetic operations,
    overflow, 241-242
  lambda, 47, 280
  logical, 50-51
    AND (&), 50
    listing of, 51
    OR (|), 47
    rules, 51
    sample application, 51
    XOR, 47
  multiplicative, 47
  not (!), 51
  null-coalescing, 47
  overloading, 120-122
  language support, 121
    listing of, 121
    sample application, 123-124
    symmetrical groups, 121-122
  PLINQ, 473
  precedence, 47
  primary, 47
  relational, 47, 49, 51
  shift, 47
  type testing, 47
  unary, 47
optional parameters, 78-79, 81
  arguments, 79
  declaring, 78
  disadvantage, 79
  listing, 79
  required, compared, 79
  unmanaged programming
    interfaces, 79
OR
  conditional operator (||), 50
  exclusive operator (^), 50
  logical operator (|), 47
OrderablePartitioner<TSource> class, 470
orderby clause, 273
ordering
  arguments, 80
  LINQ joins, 276-277
  LINQ query data, 273
  listings, 273
  sample application, 274-275
OutOfMemoryException, 230
output parameters, 76
overflowing integer arithmetic operations, 241
  checked/unchecked blocks, 242
  checked/unchecked expressions, 242
Overlaps method, 213
overloading
  constructors, 82-83
  dynamic types, 442-443
  methods, 76-77
    example listing, 77
    return types, 77
    signatures, 77
  operators, 120-122
    language support, 121
    listing of, 121
    sample application, 123-124
    symmetrical groups, 121-122
overriding
  base class members, 101-103
  ToString method, 178-179, 182
P
  padding strings, 175
  PadLeft method, 175
  PadRight method, 175
  Page event handlers, 410
  Page_Load method, 418
  Parallel class, 468
  parallel computing platform (.NET Framework), 8, 16-17
  Parallel LINQ. See PLINQ
  ParallelEnumerable class, 472
parallelism. See data parallelism
data, 468
  ForEach method example, 468
  guidelines, 473-474
  loop execution, controlling, 468
  Parallel class, 468
  thread-safe collections, 469-470
  tasks, 469-472
  exception handling, 471-472
  guidelines, 473-474
  Invoke method, 470
  task creation, 470-471
  waiting on tasks, 471
parameters, 75-76
  arguments
    order, 80
    positional, 80
    named. See named arguments
  arrays, 75
  attributes, 428
  listing, 429
  named, 429
  positional, 429
  culture, 393
generic type, 250
  constraints, 250-252
  default values, 252
  multiple, 250
  optional, 78-79, 81
  arguments, 79
  declaring, 78
  disadvantage, 79
  listing, 79
  required, compared, 79
  unmanaged programming interfaces, 79
output, 76
  reference, 76
  required, 79
  value, 75
Parent method
  DirectoryInfo class, 293
  XNode class, 319-320
Parse method, 44
  partial classes, 86
Partitioner class, 470
Partitioner<TSource> class, 470
Pascal casing, 23
Password property, 45
Path class, 290-291
Path property, 45
paths, 290-291
  absolute, 290
  Path class, 290-291
  relative, 290
 Peek method
  Queue<T> class, 217
  Stack<T> class, 215
photo viewer application
  data binding, adding, 389-390
  data converters, 394
  document outline, 372
  label control styles, 380-381
  layout
    Collection Editor, 373-374
    completed, 378-379
controls, adding, 375
directories, choosing, 377
event handlers, 375-376
grid row/column
definitions, 374
grid rows/columns,
creating, 373
ShowDialog extension
method, 375
sizing grid rows/columns,
374
structure, editing, 373
tooltips, displaying, 377
New Project dialog, 370
Properties window, 372
resource dictionaries, 380
selected element outline, 372
Visual Studio, viewing, 371
web-based, 411
ASPx editor, 413
CSS, 416-418
data binding, 418-420
default page, 415
HTML, 416
layout, creating, 414-416
layout styling, 416-418
New Project dialog, 412
Properties window, 413
selected element outline,
413
Visual Studio, viewing,
412
XAML designer, 371-372
PhotoSizeConverter, 393
PIAs (Primary Interop
Assemblies), 447
PLINQ (Parallel LINQ), 472
defined, 472
operators, 473
queries, creating, 473
pointer types, 36
d polymorphism, 95, 99
Pop method, 215
Port property, 45
positional arguments, 80
positional parameters, 429
post-events, 138
PowerShell, 482
cmdlets, 482
custom, creating, 483
get-process, 482
select, 482
running processes script, 482
precedence (operators), 47
predefined delegates, 283
predefined types, 37-38
bool, 37
byte, 37
char, 37
CLS-compliance, 38-39
decimal, 37
double, 37
float, 37
implicit conversions, 56
int, 37
long, 37
object, 38-39
sample applications, 40-41
sbyte, 37
short, 37
string, 38
uint, 37
ulong, 38
preemptive multitasking, 462
pre-events, 138
PreviousAttribute property, 320
PreviousNode property, 319-320
Primary Interop Assemblies
(PIAs), 447
primary operators, 47
private accessibility, 69
Program class, 29
projects, 26
creating, 27-28
New Project commands,
27
New Project dialog box,
27-28
saving, 30
viewing, 26
properties
automatic
declaring, 71
disadvantage, 72
calculated
creating, 73
declaring, 73
Capacity, 203
Collection<T> class, 206
ColumnDefinitions, 373
ConverterParameter, 393
DateOfBirth, 74
DateTime type, 41-42
declaring, 71-72
defined, 71
Exception class, 228-229
FirstAttribute, 320
FirstName, 74
FirstNode, 319-320
FullName,
indexers, compared, 201
IsNamespaceDeclaration,
317
LastAttribute, 320
LastName, 74
LastNode, 319-320
Length, 171, 178
List<T> class, 204
NextAttribute, 320
NextNode, 319-320
operations, 73
Parent, 319-320
PreviousAttribute, 320
queries

ADO.NET, 332
Entity Framework, 481-482
Entity Framework conceptual model, 480
LINQ to DataSet, 334
LINQ to SQL, 336
PLINQ, 473
XPath, 321
queries (LINQ)

chaining, 283
Contact object collection listing, 268
data selection, 269-271
anonymous types, 270
concatenating data, 269-270
sample application, 270-271
defered execution, 283
filtering data, 271-272
flattening data, 277-278
Contact class and JournalEntry class listing, 277-278
enumerable collections, 278
listing, 278
grouping data, 273-274
listings, 273-274
sample, 274-275
joining data, 275-277
Contact class and JournalEntry class listing, 275-276
equals operator, 277
group joins, 276-277
join clause, 276
order, 276-277
lambda expressions, 280
delegate replacement, 280
expression, 281-282
lambda operator, 280
statement, 282-283
variables, capturing, 281
lazy evaluation, 283
ordering data, 273
listings, 273
sample application, 274-275
predefined delegates, 283
query comprehension syntax, 269
SQL syntax, compared, 269
standard query operator methods, 279-280
ten syntax, 268
XElement class, 320-321
XML

LINQ queries, 320-321
reshaped XML, returning, 322-323
XPath queries, 321
Queue<T> class, 216-217
queues, 215
Queue<T> class, 216-217
sample application, 217
R

\r escape sequence, 169
raising events, 139-141
event handlers, 139
method names, 140
property-like syntax, 140
sample application, 140-141
RangeValidator control, 420
Read method

Read method
  binary files, 301
  Stream class, 301
  StreamReader class, 304-305

ReadAllBytes method
  binary files, 306
  File class, 299

ReadAllLines method
  File class, 299
text files, 306

ReadAllText method
  File class, 299
text files, 306

StreamReader class, 304-305

ReadAllBytes method, 306
File class, 299

ReadAllLines method
  File class, 299
text files, 306

ReaderWriterLockSlim class, 467

ReadLines method
  File class, 299
text files, 307

ReaderWriterLockSlim class, 467
reading
  binary files, 301-303
  listing, 302
  ReadAllBytes method, 306
  sample application, 302-303
  buffered streams, 303-304
  read-only database access, 331
  text files, 306-307

ReadLines method
  File class, 299
text files, 307

read-only fields, 21
read-only properties, 73

ReadOnlyCollection<T> class, 208-209

Recent Projects list (Visual Studio Start page), 26
records (databases)
  adding, 340
deleting, 341
  updating, 340-341
reference parameters, 76
reference types, 19, 36
categories, 36

value type conversions, 54
boxing/unboxing operations, 56
explicit conversions, 57
implicit conversions, 56
sample application, 58
value types, compared, 19
reflection interoperability, 448-449

invoking methods
  C#, 448
dynamically, 449
  reflectively, 448

IronPython dynamic objects, 449

Refresh method
  DirectoryInfo class, 293
  FileInfo class, 294

Regex class, 189
regular expressions, 187
classes, 189
compatibility, 187
metacharacters, 188
string validation, 189-190
substring matches, 190

RegularExpressionValidator control, 420
relational operators, 47, 49, 51

Relations property, 331
relative paths, 290

Remainder method, 46
Remove method
  Dictionary<TKey, TValue> class, 209
  HashSet<T>/SortedSet<T> classes, 213
  SortedDictionary<TKey, TValue> class, 210
  SortedList<TKey, TValue> class, 210

RemoveAttributes method, 324
RemoveItem method, 207
RemoveNodes method, 324
RemoveWhere method, 213

Replace method
  File class, 299
  FileInfo class, 294
  string characters, 175
RemoveAttributes method, 324
RemoveItem method, 207
RemoveNodes method, 324
RemoveWhere method, 213

Replace method
  File class, 299
  FileInfo class, 294
string characters, 175

ReplaceAll method, 324
ReplaceAttributes method, 324
ReplaceNodes method, 324
ReplaceWith method, 324
required parameters, 79
RequiredFieldValidator control, 420
reshaped XML, returning, 322-323

Resize method, 200
resource cleanup
dispose pattern
  benefits, 456
  Contact class example, 455
  Dispose method, 456
  implementing, 455
finalizers, 456
  implementing, 456-457
  rules, 457
  writing, 458
resource dictionaries, 380
rethrowing exceptions, 239-240
  breaking the stack, 240
  listing, 240
return statements, 162
return types, 77
Reverse method, 213

Root method, 293
routed events, 376
rules
  access modifiers, 68
custom validation rules, 401
finalizers, 457
identifiers, 21
logical operators, 51
set accessors
  indexers, 201
  properties, 72
setAttributes method, 299
setAttributeValue method, 315
SetElementValue method, 315
SetEquals method, 213
setItem method, 207
select cmdlet, 482
selecting
  LINQ query data, 269-271
  anonymous types, 270
  concatenating data, 269-270
  sample application, 270-271
  XML
    attributes, 321
    LINQ to XML, 321-322
selection statements, 148
  if, 148-150
  cascading, 149
  mismatched else problem, 149
  nesting, 148
  sample application, 150
  syntax, 148
  switches, 150-153
  expression values, 151
  fall through, 152
  listing, 151
  sample application, 152-153
  scope, 152
  sections, 151
  syntax, 150-151
semaphores, 467
semicolons (;), 19
server-side validation, 420

S
\s regular expression metacharacter, 188
\S regular expression metacharacter, 188
safety
  threads, 463
types, 247
saving
  projects, 30
  solutions, 30
sbyte type, 37
Scheme property, 45
scope
  declaration space comparison
    application, 68
  defined, 66
set accessors
  indexers, 201
  properties, 72
SetAttributes method, 299
setAttributeValue method, 315
SetElementValue method, 315
SetEquals method, 213
setItem method, 207
sets, 212
class methods, 213
classes, 212
sample application, 214-215
SetValue method, 323
sharing DataTips, 355
shift operators, 47
short type, 37
short-circuit evaluation, 52
ShowDialog method, 375
Sign method, 46
signals, 466
signatures
  indexers, 201
  methods, 77
Silverlight, 203, 483-484
size
  arrays, 196
  WPF application grid
    rows/columns, 374
Solution Explorer, 26
solutions, 26
  saving, 30
  viewing, 26
SortDescription structure, 397
SortedDictionary<TKey, TValue> class, 210-211
SortedList<TKey, TValue> class, 210-211
SortedSet<T> class, 212-213
sorting collection views, 396-397
special directories, 291-292
  finding, 291
    SpecialFolder values, 291-292
specialized constructors, 82-83
SpinLock class, 466
splitting strings, 176-177
Stack<T> class
  methods, 215
  sample application, 216
stack memory, 452
StackOverflowException, 230
stacks, 215-216
  breaking, 240
  integer, implementing, 216
  Stack<T> class, 215
StackTrace property, 228
standalone applications, 367
standard exceptions, 229-230
standard format strings, 179-182
  Days enumeration value
    listing, 181
  specifiers
    defining, 181-182
    listing of, 179-181
  ToString method, overriding, 182
standard query operator
  methods, 279-280
star sizing, 374
starting
  Solution Explorer, 26
  Visual Studio, 25-26
    command section, 26
    Recent Projects list, 26
    tabbed content area, 26
StartNew method, 471
StartsWith method, 173
starvation, 463
statements
  blocks, 19, 66
  control flow. See control flow statements
definition, 22
defined, 19
goto, 159
iteration, 153
do, 154-155
for, 155-158
foreach, 158-159
while, 153-154
jump, 159
  break, 160-161
  continue, 161-162
return, 162
lambdas, 282-283
lock, 464
  Enter/TryEnter methods, 465
  listing, 464-465
  lock expressions to avoid, 464
  Monitor class, 465
next, 359
selection, 148
  if, 148-150
  switches, 150-153
styles, 19
using, 454
  compiler generated code, 454-455
  syntax, 454
  whitespace, 19
static classes, 86-87
static constructors, 85
static methods, 75
static variables, 20
Stream class, 300-301
StreamReader class, 304-305
streams
  buffered, 303-304
disposing, 295
  Stream class, 300-301
StreamWriter class, 304-305
String class, 171-175
StringBuilder class, 177
Append/AppendFormat
  methods, 177
capacity, 178
listing, 177
strings, 38, 168
case, 173-174
characters, deleting, 175
comparisons, 173-174
  Compare method, 173
  CompareOrdinal method, 173
  CompareTo method, 173
  Contains method, 173
  EndsWith method, 173
  rules, 173
  sample application, 174
  StartsWith method, 173
composite formatting,
  186-187
  listing, 187
  syntax, 186-187
concatenation, 176
custom format, 183-186
  DateTime instance listing,
    183-186
  specifiers, 183-186
empty, 170-171
testing, 170
  whitespace characters,
    170-171
interning, 168
joining, 176-177
literals
  character escape
    sequences, 168-169
displaying, 169
verbatim, 168-169
modifying, 174-175
mutable
  appending data, 177
  characters, replacing, 178
data, adding/deleting, 178
StringBuilder class, 177-178
StringBuilder class listing, 177
number of characters, 171
padding, 175
regular expressions, 187
classes, 189
compatibility, 187
metacharacters, 188
string validation, 189-190
substring matches, 190
splitting, 176-177
standard format, 179-182
Days enumeration value listing, 181
specifiers, 179-182
ToString method, overriding, 182
String class, 171
ToString method, 170
type formatting, 178
validation, 189-190
zero-based counting, 172
structures
classes, compared, 119-120, 125-126
common, 120
constructors, 124-125
conversion operators, 122
custom default constructors, 125
declaring, 120
defined, 119
generic, 255
initializing, 125
methods, 120
operator overloading, 120-122
language support, 121
listing of, 121
sample application, 123-124
symmetrical groups, 121-122
properties, 126
public fields, 126
SortDescription, 397
subscribing, events, 132
anonymous methods, 136
button Click event example, 133-134
event handlers, attaching, 132-133
method group inference, 133
user interface control published, 135-136
substrings, 171-172
creating, 171-172
extracting, 171
IndexOf/LastIndexOf methods, 172
regular expression matches, 190
Subtract method, 43
subtraction operator (-), 48
swallowing exceptions, 234
switches, 150-153
expression values, 151
core through, 152
listing, 151
sample application, 152-153
scope, 152
sections, 151
syntax, 150-151
symmetric operator overload groups, 121-122
SymmetricExceptWith method, 213
synchronizing threads, 463
interlocked operations, 466
locks, 463-466
mutex, 467
reader-writer locks, 467
semaphores, 467
signals, 466
System namespace, 15
System.Collections.Generic namespace, 15
System.Data namespace, 15
System.Diagnostics namespace, 15
System.Environment.FailFast namespace, 231
System.Exception class, 229
System.Globalization namespace, 15
System.IO namespace, 15
System.Linq namespace, 16
System.Net namespace, 16
System.Security namespace, 16
System.ServiceModel namespace, 16
System.Text namespace, 16
System.Web namespace, 16
System.Windows namespace, 16
System.Windows.Controls namespace, 16
System.Xml namespace, 16
	 escape sequence, 169
tabbed content area (Visual Studio Start page), 26
Tables property, 331
targets (attributes)
identifying, 430
listing of, 429

How can we make this index more useful? Email us at indexes@samspublishing.com
Task class, 471

Task Parallel Library. See TPL

tasks
creating
explicitly, 470
StartNew method, 471
parallelism, 469-472
exception handling,
471-472
guidelines, 473-474
Invoke method, 470
task creation, 470-471
waiting on tasks, 471
waiting to complete, 471

templates, 399-400
associating with controls, 400
defining, 399-400

ternary operators. See conditional operators
testing
empty strings, 170
unit tests, 349-350
value equality with generics,
251-252
text
console window, displaying,
29
WPF, 369
text files, reading and writing,
306-307
ReadAllLines method, 306
ReadAllText method, 306
ReadLines method, 307
ReadLine method, 304-305
sample application, 305
WriteAllLines method, 306-307
WriteAllText method, 306-307
WriteLine method, 304-305

Text property, 421
this keyword, 83

threads, 462
concurrency problems, 463
data parallelism, 468
ForEach method example,
468
loop execution, controlling,
468
Parallel class, 468
thread-safe collections,
469-470
disadvantages, 463
managed, 462
multiple, 462
preemptive multitasking, 462
safety, 463
synchronizing, 463
interlocked operations,
466
locks, 463-466
mutex, 467
reader-writer locks, 467
semaphores, 467
signals, 466
task parallelism, 469-472
exception handling,
471-472
Invoke method, 470
task creation, 470-471
waiting on tasks, 471

TPL. See TPL
tostring method, 170

throw keyword, 231, 239
throwing exceptions, 231
no handlers, 231
rethrowing, 239-240
breaking the stack, 240
listing, 240
timing, 231
wrapped, 240-241

 Today property, 41
ToLowerInvariant method, 174
tostring method, 170

Trim method, 175
TrimEnd method, 175
TrimExcess method
Hashset<T> class, 213
Queue<T> class, 217
SortedDictionary<TKey,
TValue> class, 211
SortedList<TKey, TValue> class, 211
SortedSet<T> class, 213
Stack<T> class, 215
TrimStart method, 175
trust but verify philosophy, 98
try keyword, 232
TryCreate method, 44
TryEnter method, 465
TryGetValue method, 211
TryParse method, 44
tuples, 261
classes, 261
sample application, 262
TwoWay data binding, 387
type parameters (generics), 250
constraints, 250-252
default values, 252
multiple, 250
types
anonymous, 270
BigInteger, 45-46
listing, 46
methods, 46
categories, 36-37
comparison, 19
DateTime, 39-42
arithmetic methods, 42
properties, 41-42
default values, 53
declared, 19
disposable, 454
dynamic, 39, 440
COM interoperability, 448
conversions, 442
custom, creating, 445-447
dynamic keyword, 439-440
methods, invoking, 449
overload resolution, 442-443
runtime, 441-442
sample application, 440-441
variables, 440
enumerations, 114
Days, 116-117
defining, 114
flag enumerations, 117-119
listing, 114-115
multiple named values, 115
numeric values, 115-116
operations, 116
sample application, 116-117
underlying types, 116
values, 114
formatting, 178
Guid, 43-44
inference, 254
null, 53
nullable
declared, 54
null-coalescing operator, 54
sample application, 55
syntax, 54
values, 54
pointers, 36
predefined, 36-38
bool, 37-38
byte, 37
cchar, 37
CLP-compliance, 38-39
decimal, 37-38
double, 37-38
float, 37
implicit conversions, 56
int, 37
long, 37
object, 38-39
sample applications, 40-41
sbyte, 37
short, 37
string, 38
uint, 37
ulong, 38
ushort, 38
reference, 19, 36
categories, 36
value type conversions, 54
safety, 247
testing operators, 47
TimeSpan, 42-43
unified type system, 37
URIs
Uri class, 44-45
UriBuilder class, 45
value, 19, 36, 58
var, 39
variant, 39
void, 39

t
uint type, 37
ulong type, 38
unary operators, 47
unboxing operations, 56
underlying types, 116
unexpected error conditions, 232
unified type system, 37
uniform resource identifiers. See URIs
UnionWith method, 213
unit tests, 349-350
unmanaged code, 8
unsubscribing, events, 136

How can we make this index more useful? Email us at indexes@samspublishing.com
upcasting, 97
UpdateSourceTrigger property
  ValidationRules collection, 402
  WPF data binding, 387
updating database records, 340-341
Uri class, 44-45
  methods, 44
  properties, 45
Uri property, 45
UriBuilder class
  listing, 45
  properties, 45
URIs (uniform resource identifiers), 44-45
UserName property, 45
using statement, 454
  compiler generated code, 454-455
  syntax, 454
UtcNow property, 42
\uxxxxx escape sequence, 169
\uxxxxxxxxx escape sequence, 169

V
\v escape sequence, 169
ValidatesOnDataErrors property, 401
ValidatesOnExceptions property, 401
validation, 400-403
  AddressBook application, 403
  arguments, 230
  controls, 420-422
  ASPX page, 422
  combining, 421
  error messages, 421
  listing of, 420-421
  properties, 421
  custom validation rules, 401
  strings, 189-190
  validation checks, adding, 401
  visual feedback, 402
  web applications, 420
  client-side, 420-421
  server-side, 420
  validation controls, 420-422
  Validator example, 422
ValidationRules property, 401
ValidationSummary control, 421
value parameters, 75
value types, 19, 36
  bool, 37-38
  byte, 37
  char, 37
  converting to reference types, 54
  boxing/unboxing operations, 56
  explicit conversions, 57
  implicit conversions, 56
  sample application, 58
decimal, 37-38
double, 37-38
float, 37
int, 37
long, 37
null, 53
nullable
defined, 54
  null-coalescing operator, 54
  sample application, 55
  syntax, 54
  values, 54
reference types, compared, 19
  sbyte, 37
  short, 37
  switch expressions, 151
  uint, 37
  ulong, 38
values
  constants, 70
date and time values. See date and time values
default, 53
enumerations
  commas, 114
  multiple named values, 115
  numeric, 115-116
flag enumerations, 118
generic types, 252
GUIDs, 40-44
incrementing/decrementing, 48
table minimum, finding
  with generics code listing, 248-249
  objects, 246-247
  with generics application, 249-250
  without generics application, 247-248
  without generics code listing, 246
integers, 45-46
literals, 21-22
nullable types, 54
URIs
  Uri class, 44-45
  UriBuilder class, 45
XML elements/attributes, 323
Values method, 209
var type, 39
variable windows (Visual Studio debugger), 352-354
  Locals window, 352-353
  Watch window, 352-353
variables
capturing, 281
catch handlers, 233
constants, compared, 21
declaring, 20, 22
default values, 53
defined, 20
dynamic types, 440
fields. See fields
instance, 20
iteration, 158
lifetime, 20
local, 20
scope, 20
spoa, 20
static, 20
variance (generic interfaces), 258-261
class implementation, 258
contravariance, 258
covariance, 258
extending, 260-261
listing of, 258-259
sample application, 259-260
variant type, 39
verbatim string literals, 168-169
VerifyEmailAddress method
calling, 76
declaring, 74-75
VES (virtual execution system), 9-12
virtual methods, 207
Visual Studio
  benefits, 24
debugger, 350
breaki on exceptions, 351
breakpoints, 355-357
Call Stack window, 358-359
compiler errors, 349
DataTips, 354-355
Exception Assistant, 351-352
expression side effects, 353-354
Immediate window, 355
MSDN edition features, 352
next statements, 359
runtime errors, 349
user-provided divisors, 350
stepping through code, 357-358
variable windows, 352-354
visualizers, 350-360
editions, 25
features, 24
overview, 24
projects, 26, 30
Solution Explorer, 26
solutions, 26, 30
Start page, 25-26
command section, 26
Recent Projects list, 26
tabbed content area, 26
visualizers, 350-360
void type, 39
\W regular expression
metacharacter, 188
Wait method, 465
  Task class, 471
  thread signals, 466
WaitAll method, 471
WaitAny method, 471
waiting on tasks, 471
Watch window, 352-353
web applications
  accessing, 408
  architecture, 408
  ASP.NET, 408-411
  ASPX files, 409
  client-side validation, 420-421
  CSS, 416-418
  data binding, 418-420
  embedded code blocks, 419
  Global.asax event handlers, 411
  Hello world application, 409
  master pages, 414-415
  MVC Framework, 408
  Page event handlers, 410
  server-side validation, 420
  validation controls, 420-422
  web-based photo viewer application. See web-based photo viewer
data validation, 420
  client-side, 420-421
  server-side, 420
  validation controls, 420-422
  Validator example, 422
HTTP 408
IIS, 410

How can we make this index more useful? Email us at indexes@samspublishing.com
web applications

- performance, 408
- Visual C# Express edition, 407
- web-based photo viewer application, 411
  - ASPX editor, 413
  - CSS, 416-418
  - data binding, 418-420
  - default page, 415
  - HTML, 416
  - layout, creating, 414-416
  - layout styling, 416-418
  - New Project dialog, 412
  - Properties window, 413
  - selected element outline, 413
  - Visual Studio, viewing, 412
- while statements, 153-154
  - listing, 154
  - sample application, 154
  - for statements, compared, 156-157
  - syntax, 153
- whitespace
  - empty strings, 170-171
  - statements, 19
- windows
  - Call Stack, 358-359
  - console, 29
  - Immediate, 355
  - variable, 352-354
    - Locals window, 352-353
    - Watch window, 352-353
- WPF (Windows Presentation Foundation), 364
  - animation, 369
  - application model, 366-367
    - browser-hosted
      - applications, 367
      - standalone applications, 367
  - applications, creating, 370
    - Collection Editor, 373-374
    - completed layout, 378-379
    - controls, adding, 375
    - directories, choosing, 377
    - document outline, 372
    - event handlers, 375-376
    - grid row/column definitions, 374
    - grid rows/columns, creating, 373
    - New Project dialog, 370
    - Properties window, 372
    - routed events, 376
    - selected element outline, 372
    - ShowDialog extension method, 375
    - sizing grid rows/columns, 374
    - structure, editing, 373
    - tooltips, displaying, 377
    - Visual Studio, viewing, 371
    - XAML bindings, debugging, 377-378
    - XAML designer, 371-372
    - binding to collections, 395
    - collection views, 395-396
    - current item pointers, 398-399
    - filtering data, 398
    - grouping data, 397
    - INotifyCollectionChanged interface, 396
    - INotifyPropertyChanged interface, 396
    - photo viewer application example, 395
    - sorting data, 396-397
  - code-behind, 365-366
  - data binding, 369, 386
    - components, 386
    - creating, 387-389
    - event handler, 389
    - OneTime, 387
    - OneWay, 386
    - OneWayToSource, 387
    - photo viewer application, 389-390
    - source object, 388-389
    - source property, updating, 387
    - target objects, 386
    - target properties, 386
    - TwoWay, 387
    - XAML application code, 389
  - data converters, 390
    - adding to XAML file, 391
    - creating, 390-391
    - culture-aware, 393
    - multivalue, 392-393
    - multivalue with Converter parameter, 394
    - photo viewer application, 394
  - data templates, 399-400
    - associating with controls, 400
    - defining, 399-400
  - document support, 369
  - elements, nesting, 368
  - graphics, 369
  - layouts, 367-368
    - default layout controls, 367-368
    - DockPanel example, 368
  - label control styles, 380-381
  - resource dictionaries, 380
style/data template
resources, 380
markup, 364-365
text, 369
validating data, 400-403
AddressBook application, 403
custom validation rules, 401
validation checks, adding, 401
visual feedback, 402
WPFInteropExtensions class, 375
wrapping exceptions, 240-241
Write method
binary files, 301
Stream class, 301
WriteAllBytes method
binary files, 306
File class, 299
WriteAllLines method
File class, 299
text files, 306-307
WriteAllText method
306-307
WriteLine method, 304-305
unit tests, 349-350

X

XAML (Extensible Application Markup Language), 364-365
bindings, debugging, 377-378
CollectionViewSource class, 396
data converters, adding, 391
designer, 371
Name attribute, 376
photo viewer application layout, 378-379
XAttribute class, 314
XBAPs (XAML browser applications), 367
x:Class attribute, 365
XDocument class, 313
XElement class, 314
navigation properties, 320
SetElementValue/SetAttributeValue methods, 315
values, retrieving, 316
XML (Extensible Markup Language), 309
attributes, selecting, 321
comments, 348
DOM, 312-313
creating XML, 314-315
XPath queries, 321
elements/attributes adding, 324-325
removing, 324
replacing data, 324
values, changing, 323
LINQ to XML, 313
tree nodes, navigating
LINQ queries against XElement class, 320-321
character encoding, 316
creating XML, 314
modification application, 325
selecting XML, 321-322
SetElementValue/SetAttributeValue methods, 315
source XML, transforming, 323
XAttribute class, 314
XDocument class, 313
XElement class, 314
XElement values, retrieving, 316
XML documents, creating, 316
XName class, 317
XNamespace class, 317-319
XPath queries, 321
namespaces. See LINQ to XML, namespaces
modifying, 323
names, 317
namespaces, 317
atomization, 319
creating XML, 317-318
declaring, 317
expanded names, 319
prefixes, 318
sample application, 319
reshaped, returning, 322-323
selecting with LINQ to XML, 321-322
tree nodes, navigating
LINQ queries against XElement class, 320-321

How can we make this index more useful? Email us at indexes@samspublishing.com
XML (Extensible Markup Language)

- XElement properties, 320
- XNode class properties, 319-320
- XName class, 317
- XNamespace class, 317-319
- XNode class, 319-320
- XOR logical operator, 47
- XPath namespace, 321
- XPath queries, 321
- XPathEvaluate method, 321
- XPathSelectElements method, 321

Y

Year property, 42

Z

Zero method, 46