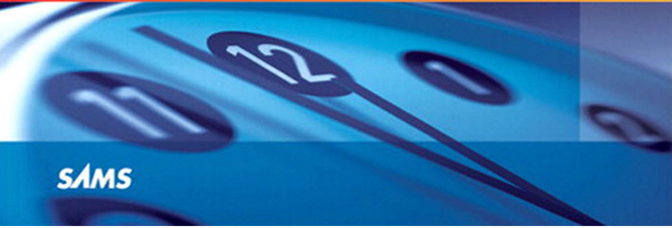
Scott Dorman

Foreword by Eric Lippert
Senior Developer, Microsoft Visual C# Team

STARTER KIT
DVD includes
Visual C# 2010
Express Edition

Sams Teach Yourself

Visual C#® 2010 in 24 Hours



SAMS

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

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*

Scott Dorman

Sams Teach Yourself Visual C#® 2010 Complete Starter Kit in 24 Hours

SAMS 800 East 96th Street, Indianapolis, Indiana, 46240 USA

Sams Teach Yourself Visual C#® 2010 in 24 Hours: Complete Starter Kit

ISBN-13: 978-0-672-33101-5
ISBN-10: 0-672-33101-2

Library of Congress Cataloging-in-Publication Data

Dorman, Scott, 1973-
Sams teach yourself Visual C# 2010 : in 24 hours / Scott Dorman.
p. cm.
Includes index.
ISBN 978-0-672-33101-5
1. C# (Computer program language) 2. Microsoft Visual C#. I. Millspaugh, A. C. (Anita C.) II. Title.
QA76.73.C154D57 2010
005.13'3—dc22
2010018992

Printed in the United States on America

First Printing June 2010

Trademarks

Warning and Disclaimer

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales

1-800-382-3419

corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales

international@pearsoned.com

Sams Publishing
800 East 96th Street
Indianapolis, Indiana, 46240 USA

Editor-in-Chief
Karen Gettman

Executive Editor
Neil Rowe

Acquisitions Editor
Brook Farling

Development Editor
Mark Renfrow

Managing Editor
Kristy Hart

Senior Project Editor
Lori Lyons

Copy Editor
Apostrophe Editing Services

Indexer
Publishing Works, Inc.

Proofreader
Water Crest Publishing, Inc.

Technical Editors
Claudio Lasalla
Eric Lippert

Publishing Coordinator
Cindy Teeters

Book Designer
Gary Adair

Composition
Jake McFarland
Nonie Ratcliff

Contents at a Glance

Part V: Diving Deeper

Table of Contents

Part IV Building an Application Using Visual Studio

Part V Diving Deeper

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 in to 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.

By the Way

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

Did you Know?

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

Watch Out!

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

HOUR 3

Understanding Classes and Objects the C# Way

What You'll Learn in This Hour:

- Object- and Component-Oriented Programming
- Classes in C#
- Scope and Accessibility
- Methods and Properties
- Nested and Partial Classes
- Static Classes and Data
- Object Initializers

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.

By the Way

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, `StartCar`, and information, such as `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 `object`. To define a class, you use the `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 `Program`.

The **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

```
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.

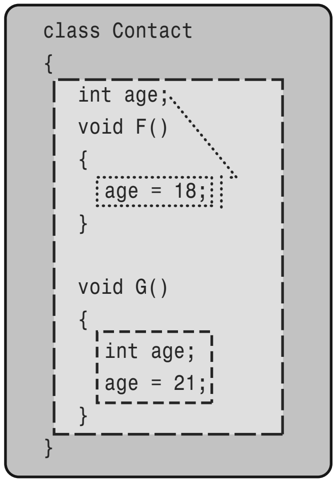


FIGURE 3.1
Nested scopes and hiding

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.
3. In `G`, add a `Console.WriteLine` statement at the end of the method that prints the value of `age`.

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`:

```
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.

FIGURE 3.2
Working with scope

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.

By the Way

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 3.1 Access Modifiers

Modifier	Description
`public`	Access is not limited.
`protected`	Access is limited to the containing class or types derived from the containing class.
`internal`	Access is limited to the containing assembly.
`protected internal`	Access is limited to the containing assembly or types derived from the containing class.
`private`	Access is limited to the containing class only.

Watch Out!

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":

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

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

```
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.

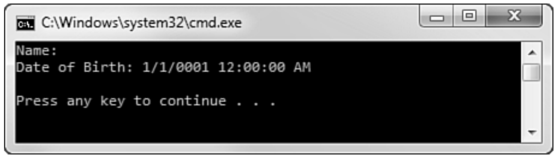


FIGURE 3.3
Working with fields

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

```
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

```
class Contact
{
   private string firstName;

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

Watch Out!

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 backing 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

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

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

By the Way

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 operations 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

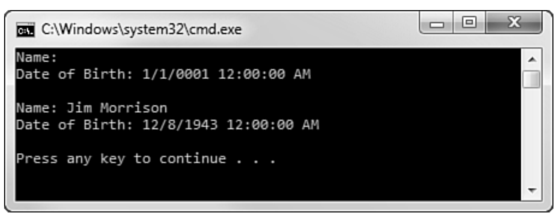
To modify the `Contact` class to allow access to the private data using properties, and to use automatic and calculated properties, follow these steps. If you closed Visual Studio, repeat the previous exercise first. Be sure to keep Visual Studio open at the end of this exercise because you will use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

```
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.

FIGURE 3.4
Working with properties



Methods

If fields and properties define and implement data, methods, which are also called **functions**, 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.

By the Way

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.

By the Way

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:

```
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:

```
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.

Watch Out!

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.

Watch Out!

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

```
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)
{
}
```

Try It Yourself

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:

```
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.

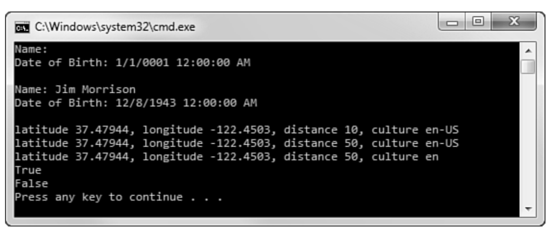


FIGURE 3.5
Working with methods

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:

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

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.

By the Way

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

```
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

```
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:

```
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.

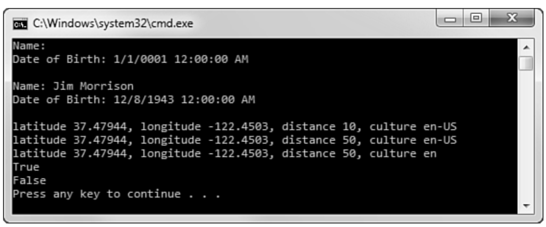


FIGURE 3.6 Working with optional parameters and named arguments

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:

```
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

```
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

```
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.

By the Way

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

```
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

```
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).

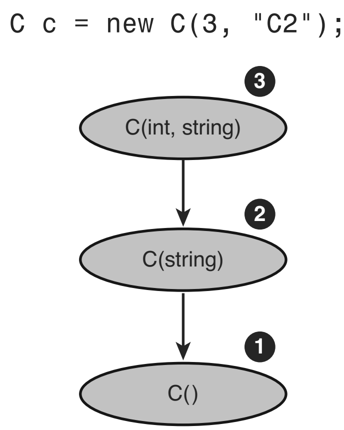


FIGURE 3.7 Constructor chaining sequence

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?

Nested Classes with Partial Classes

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.

Watch Out!

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.

GO TO ▶ Hour 11, "Understanding Generics," for more information on generic classes.

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.

By the Way

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.

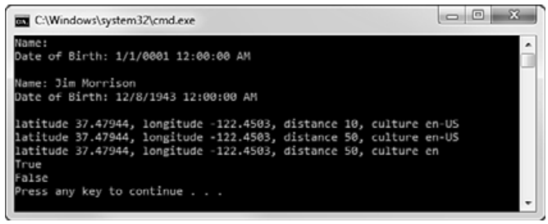


FIGURE 3.8
Results of working with extension methods

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

```
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?

   ```
   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

```
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:

Data Type	Field Name
`bool`	`Exists`
`BitmapFrame`	`image`
`Uri`	`source`

Add the following constructor:

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

Index

Symbols

A

B

C

D

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

E

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

F

H

I

J

K

N

O

Q

R

T

U

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

V

W

X

Y

Z