"The **Essential C#** series is a classic, and teaming up with famous C# blogger Eric Lippert on the new edition is another masterstroke!...[The authors] share a great gift of providing clarity and elucidation, and by combining their 'inside' and 'outside' perspectives on C#, this book reaches a new level of completeness. Welcome to one of the greatest collaborations you could dream of in the world of C# books!"

—From the Foreword by **Mads Torgersen,** C# Program Manager, Microsoft

Essential C# 5.0

Windows Development Series

Mark **Michaelis**
with Eric **Lippert**

IntelliTect

Essential C# 5.0

Microsoft Windows Development Series

Addison-Wesley

Visit informit.com/mswinseries for a complete list of available publications.

The Windows Development Series grew out of the award-winning Microsoft .NET Development Series established in 2002 to provide professional developers with the most comprehensive and practical coverage of the latest Windows developer technologies. The original series has been expanded to include not just .NET, but all major Windows platform technologies and tools. It is supported and developed by the leaders and experts of Microsoft development technologies, including Microsoft architects, MVPs and RDs, and leading industry luminaries. Titles and resources in this series provide a core resource of information and understanding every developer needs to write effective applications for Windows and related Microsoft developer technologies.

"This is a great resource for developers targeting Microsoft platforms. It covers all bases, from expert perspective to reference and how-to. Books in this series are essential reading for those who want to judiciously expand their knowledge and expertise."

– JOHN MONTGOMERY, Principal Director of Program Management, Microsoft

"This series is always where I go first for the best way to get up to speed on new technologies. With its expanded charter to go beyond .NET into the entire Windows platform, this series just keeps getting better and more relevant to the modern Windows developer."

– CHRIS SELLS, Vice President, Developer Tools Division, Telerik

Make sure to connect with us!
informit.com/socialconnect

Addison Wesley | informIT the trusted technology learning source | Safari Books Online

Essential C# 5.0

Mark Michaelis
with Eric Lippert

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Michaelis, Mark.
Essential C# 5.0 / Mark Michaelis with Eric Lippert.
pages cm
Includes index.
ISBN 0-321-87758-6 (pbk. : alk. paper)
1. C# (Computer program language) 2. Microsoft .NET Framework. I. Lippert, Eric. II. Title.
QA76.73.C154M5238 2013
006.7'882—dc23 2012036148

ISBN-13: 978-0-321-87758-1
ISBN-10: 0-321-87758-6
Text printed in the United States on recycled paper at Edwards Brothers Malloy in Anne Arbor, Michigan.
First printing, November 2012

To my family: Elisabeth, Benjamin, Hanna, and Abigail,

You have sacrificed a husband and daddy for countless hours of writing, frequently at times when he was needed most.

Thanks!

This page intentionally left blank

Contents at a Glance

Contents

Figures

Tables

Foreword

WELCOME TO ONE of the greatest collaborations you could dream of in the world of C# books—and probably far beyond! Mark Michaelis' Essential C# series is already a classic, and teaming up with famous C# blogger Eric Lippert on the new edition is another masterstroke!

You may think of Eric as writing blogs and Mark as writing books, but that is not how I first got to know them.

In 2005 when LINQ (Language Integrated Query) was disclosed, I had only just joined Microsoft, and I got to tag along to the PDC conference for the big reveal. Despite my almost total lack of contribution to the technology, I thoroughly enjoyed the hype. The talks were overflowing, the printed leaflets were flying like hotcakes: It was a big day for C# and .NET, and I was having a great time.

It was pretty quiet in the hands-on labs area, though, where people could try out the technology preview themselves with nice scripted walkthroughs. That's where I ran into Mark. Needless to say, he wasn't following the script. He was doing his own experiments, combing through the docs, talking to other folks, busily pulling together his own picture.

As a newcomer to the C# community, I think I may have met a lot of people for the first time at that conference—people that I have since formed great relationships with. But to be honest, I don't remember it. The only one I remember is Mark. Here is why: When I asked him if he was liking the new stuff, he didn't just join the rave. He was totally level-headed: "*I don't know yet. I haven't made up my mind about it*." He wanted to absorb and

understand the full package, and until then he wasn't going to let anyone tell him what to think.

So instead of the quick sugar rush of affirmation I might have expected, I got to have a frank and wholesome conversation, the first of many over the years, about details, consequences, and concerns with this new technology. And so it remains: Mark is an incredibly valuable community member for us language designers to have, because he is super smart, insists on understanding everything to the core, and has phenomenal insight into how things affect real developers. But perhaps most of all because he is forthright and never afraid to speak his mind. If something passes the Mark Test then we know we can start feeling pretty good about it!

These are the same qualities that make Mark such a great writer. He goes right to the essence and communicates with great integrity, no sugarcoating, and a keen eye for practical value and real-world problems.

Eric is, of course, my colleague of seven years on the C# team. He's been here much longer than I have, and the first I recall of him, he was explaining to the team how to untangle a bowl of spaghetti. More precisely, our C# compiler code base at the time was in need of some serious architectural TLC, and was exceedingly hard to add new features to—something we desperately needed to be able to do with LINQ. Eric had been investigating what kind of architecture we ought to have (Phases! We didn't even really have those!), and more importantly, how to get from here to there, step by step. The remarkable thing was that as complex as this was, and as new as I was to the team and the code base, I immediately understood what he was saying!

You may recognize from his blogs the super-clear and well-structured untangling of the problem, the convincing clarity of enumerated solutions, and the occasional unmitigated hilarity. Well, you don't know the half of it! Every time Eric is grappling with a complex issue and is sharing his thoughts about it with the team, his emails about it are just as meticulous and every bit as hilarious. You fundamentally can't ignore an issue raised by Eric because you can't wait to read his prose about it. They're even purple, too! So I essentially get to enjoy a continuous supply of what amounts to unpublished installments of his blog, as well as, of course, his pleasant and insightful presence as a member of the C# compiler team and language design team.

In summary, I am truly grateful to get to work with these two amazing people on a regular basis: Eric to help keep my thinking straight and Mark

to help keep me honest. They share a great gift of providing clarity and elucidation, and by combining their "inside" and "outside" perspective on C#, this book reaches a new level of completeness. No one will help you get C# 5.0 like these two gentlemen do.

Enjoy!

—Mads Torgersen,
C# Program Manager,
Microsoft

This page intentionally left blank

Preface

THROUGHOUT THE HISTORY of software engineering, the methodology used to write computer programs has undergone several paradigm shifts, each building on the foundation of the former by increasing code organization and decreasing complexity. This book takes you through these same paradigm shifts.

The beginning chapters take you through **sequential programming structure** in which statements are executed in the order in which they are written. The problem with this model is that complexity increases exponentially as the requirements increase. To reduce this complexity, code blocks are moved into methods, creating a **structured programming model.** This allows you to call the same code block from multiple locations within a program, without duplicating code. Even with this construct, however, programs quickly become unwieldy and require further abstraction. Object-oriented programming, discussed in Chapter 5, was the response. In subsequent chapters, you will learn about additional methodologies, such as interface-based programming, LINQ (and the transformation it makes to the collection API), and eventually rudimentary forms of declarative programming (in Chapter 17) via attributes.

This book has three main functions.

- It provides comprehensive coverage of the C# language, going beyond a tutorial and offering a foundation upon which you can begin effective software development projects.
- For readers already familiar with C#, this book provides insight into some of the more complex programming paradigms and provides

in-depth coverage of the features introduced in the latest version of the language, C# 5.0 and .NET Framework 4.5.

- It serves as a timeless reference, even after you gain proficiency with the language.

The key to successfully learning C# is to start coding as soon as possible. Don't wait until you are an "expert" in theory; start writing software immediately. As a believer in iterative development, I hope this book enables even a novice programmer to begin writing basic C# code by the end of Chapter 2.

A number of topics are not covered in this book. You won't find coverage of topics such as ASP.NET, ADO.NET, smart client development, distributed programming, and so on. Although these topics are relevant to the .NET Framework, to do them justice requires books of their own. Fortunately, Addison-Wesley's Microsoft Windows Development Series provides a wealth of writing on these topics. *Essential C# 5.0* focuses on C# and the types within the Base Class Library. Reading this book will prepare you to focus on and develop expertise in any of the areas covered by the rest of the series.

Target Audience for This Book

My challenge with this book was to keep advanced developers awake while not abandoning beginners by using words such as *assembly, link, chain, thread,* and *fusion,* as though the topic was more appropriate for blacksmiths than for programmers. This book's primary audience is experienced developers looking to add another language to their quiver. However, I have carefully assembled this book to provide significant value to developers at all levels.

- *Beginners:* If you are new to programming, this book serves as a resource to help transition you from an entry-level programmer to a C# developer, comfortable with any C# programming task that's thrown your way. This book not only teaches you syntax, but also trains you in good programming practices that will serve you throughout your programming career.

- *Structured programmers:* Just as it's best to learn a foreign language through immersion, learning a computer language is most effective when you begin using it before you know all the intricacies. In this vein, this book begins with a tutorial that will be comfortable for those familiar with structured programming, and by the end of Chapter 4, developers in this category should feel at home writing basic control flow programs. However, the key to excellence for C# developers is not memorizing syntax. To transition from simple programs to enterprise development, the C# developer must think natively in terms of objects and their relationships. To this end, Chapter 5's Beginner Topics introduce classes and object-oriented development. The role of historically structured programming languages such as C, COBOL, and FORTRAN is still significant but shrinking, so it behooves software engineers to become familiar with object-oriented development. C# is an ideal language for making this transition because it was designed with object-oriented development as one of its core tenets.
- *Object-based and object-oriented developers:* C++ and Java programmers, and many experienced Visual Basic programmers, fall into this category. Many of you are already completely comfortable with semicolons and curly braces. A brief glance at the code in Chapter 1 reveals that, at its core, C# is similar to the C and C++ style languages that you already know.
- *C# professionals:* For those already versed in C#, this book provides a convenient reference for less frequently encountered syntax. Furthermore, it provides answers to language details and subtleties that are seldom addressed. Most importantly, it presents the guidelines and patterns for programming robust and maintainable code. This book also aids in the task of teaching C# to others. With the emergence of C# 3.0, 4.0, and 5.0, some of the most prominent enhancements are
  - Implicitly typed variables (see Chapter 2)
  - Extension methods (see Chapter 5)
  - Partial methods (see Chapter 5)
  - Anonymous types (see Chapter 11)
  - Generics (see Chapter 11)
  - Lambda statements and expressions (see Chapter 12)
  - Expression trees (see Chapter 12)

- Standard query operators (see Chapter 14)
- Query expressions (see Chapter 15)
- Dynamic programming (Chapter 17)
- Multithreaded programming with the Task Programming Library and `async` (Chapter 18)
- Parallel query processing with PLINQ (Chapter 18)
- Concurrent collections (Chapter 19)

These topics are covered in detail for those not already familiar with them. Also pertinent to advanced C# development is the subject of pointers, in Chapter 21. Even experienced C# developers often do not understand this topic well.

Features of This Book

Essential C# 5.0 is a language book that adheres to the core C# Language 5.0 Specification. To help you understand the various C# constructs, it provides numerous examples demonstrating each feature. Accompanying each concept are guidelines and best practices, ensuring that code compiles, avoids likely pitfalls, and achieves maximum maintainability.

To improve readability, code is specially formatted and chapters are outlined using mind maps.

C# Coding Guidelines

One of the more significant enhancements added to *Essential C# 5.0,* and not explicitly called out in previous editions, was the addition of C# coding guidelines, as shown in the following example taken from Chapter 16:

Guidelines

DO ensure that equal objects have equal hash codes.

DO ensure that the hash code of an object never changes while it is in a hash table.

DO ensure that the hashing algorithm quickly produces a well-distributed hash.

DO ensure that the hashing algorithm is robust in any possible object state.

These guidelines are the key to differentiating a programmer who knows the syntax from an expert who is able to discern the most effective code to write based on the circumstances. Such an expert not only gets the code to compile, but does so while following best practices that minimize bugs and enable maintenance well into the future. The coding guidelines highlight some of the key principles that readers will want to be sure to incorporate into their development.

Code Samples

The code snippets in most of this text can run on any implementation of the Common Language Infrastructure (CLI), including the Mono, Rotor, and Microsoft .NET platforms. Platform- or vendor-specific libraries are seldom used, except when communicating important concepts relevant only to those platforms (appropriately handling the single-threaded user interface of Windows, for example). Any code that specifically requires C# 3.0, 4.0, or 5.0 compliance is called out in the C# version indexes at the end of the book.

Here is a sample code listing.

LISTING 1.9: Declaring and Assigning a Variable

```
class MiracleMax
{
  static void Main()
  {
        data type
        string max;
               variable
      max = "Have fun storming the castle!";

      System.Console.WriteLine(max);
  }
}
```

The formatting is as follows.

- Comments are shown in italics.

```
/* Display a greeting to the console
   using composite formatting. */
```

- Keywords are shown in bold.

```
    static void Main()
```

- Highlighted code calls out specific code snippets that may have changed from an earlier listing, or demonstrates the concept described in the text.

```
    System.Console.Write /* No new line */ (
```

  Highlighting can appear on an entire line or on just a few characters within a line.

```
System.Console.WriteLine(
     "Your full name is {0} {1}.",
```

- Incomplete listings contain an ellipsis to denote irrelevant code that has been omitted.

```
// ...
```

- Console output is the output from a particular listing that appears following the listing.

OUTPUT 1.4

```
>HeyYou.exe
Hey you!
Enter your first name: Inigo
Enter your last name: Montoya
```

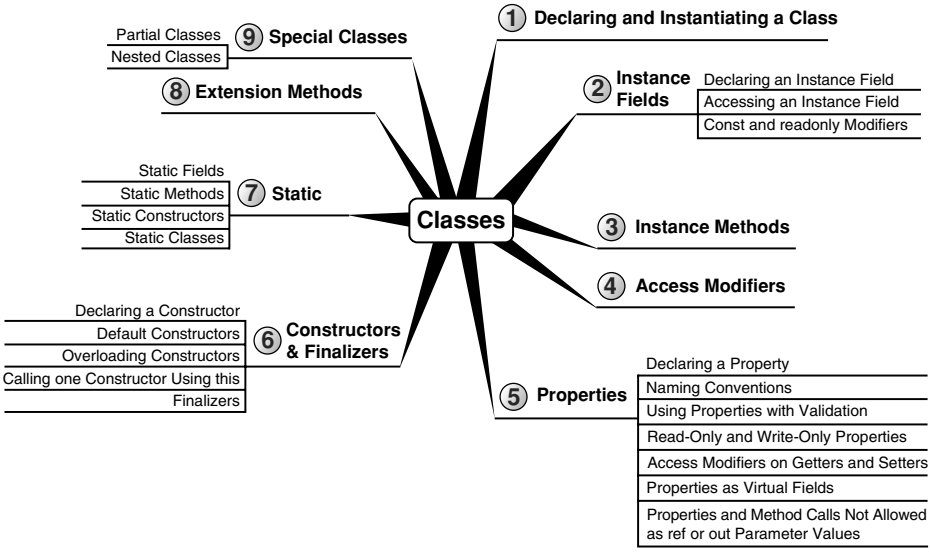
User input for the program appears in boldface.

Although it might have been convenient to provide full code samples that you could copy into your own programs, doing so would detract you from learning a particular topic. Therefore, you need to modify the code samples before you can incorporate them into your programs. The core omission is error checking, such as exception handling. Also, code samples do not explicitly include `using System` statements. You need to assume the statement throughout all samples.

You can find sample code at intellitect.com/essentialcsharp and at informit.com/mswinseries.

Mind Maps

Each chapter's introduction includes a **mind map,** which serves as an outline that provides at-a-glance reference to each chapter's content. Here is an example (taken from Chapter 5).



The theme of each chapter appears in the mind map's center. High-level topics spread out from the core. Mind maps allow you to absorb the flow from high-level to more detailed concepts easily, with less chance of encountering very specific knowledge that you might not be looking for.

Helpful Notes

Depending on your level of experience, special code blocks will help you navigate through the text.

- Beginner Topics provide definitions or explanations targeted specifically toward entry-level programmers.
- Advanced Topics enable experienced developers to focus on the material that is most relevant to them.
- Callout notes highlight key principles in callout boxes so that readers easily recognize their significance.
- Language Contrast sidebars identify key differences between C# and its predecessors to aid those familiar with other languages.

How This Book Is Organized

At a high level, software engineering is about managing complexity, and it is toward this end that I have organized *Essential C# 5.0*. Chapters 1–4 introduce structured programming, which enable you to start writing simple functioning code immediately. Chapters 5–9 present the object-oriented constructs of C#. Novice readers should focus on fully understanding this section before they proceed to the more advanced topics found in the remainder of this book. Chapters 11–13 introduce additional complexity-reducing constructs, handling common patterns needed by virtually all modern programs. This leads to dynamic programming with reflection and attributes, which is used extensively for threading and interoperability in the chapters that follow.

The book ends with a chapter on the Common Language Infrastructure, which describes C# within the context of the development platform in which it operates. This chapter appears at the end because it is not C# specific and it departs from the syntax and programming style in the rest of the book. However, this chapter is suitable for reading at any time, perhaps most appropriately immediately following Chapter 1.

Here is a description of each chapter (in this list, chapter numbers shown in **bold** indicate the presence of C# 3.0–5.0 material).

- *Chapter 1—Introducing C#:* After presenting the C# HelloWorld program, this chapter proceeds to dissect it. This should familiarize readers with the look and feel of a C# program and provide details on how to compile and debug their own programs. It also touches on the context of a C# program's execution and its intermediate language.
- ***Chapter 2****—Data Types:* Functioning programs manipulate data, and this chapter introduces the primitive data types of C#. This includes coverage of two type categories, value types and reference types, along with conversion between types and support for arrays.
- *Chapter 3—Operators and Control Flow:* To take advantage of the iterative capabilities in a computer, you need to know how to include loops and conditional logic within your program. This chapter also covers the C# operators, data conversion, and preprocessor directives.
- ***Chapter 4****—Methods and Parameters:* This chapter investigates the details of methods and their parameters. It includes passing by value,

passing by reference, and returning data via an out parameter. In C# 4.0 default parameter support was added and this chapter explains how to use them.

- ***Chapter 5**—Classes:* Given the basic building blocks of a class, this chapter combines these constructs together to form fully functional types. Classes form the core of object-oriented technology by defining the template for an object.
- *Chapter 6—Inheritance:* Although inheritance is a programming fundamental to many developers, C# provides some unique constructs, such as the new modifier. This chapter discusses the details of the inheritance syntax, including overriding.
- ***Chapter 7**—Interfaces:* This chapter demonstrates how interfaces are used to define the "versionable" interaction contract between classes. C# includes both explicit and implicit interface member implementation, enabling an additional encapsulation level not supported by most other languages.
- *Chapter 8—Value Types:* Although not as prevalent as defining reference types, it is sometimes necessary to define value types that behave in a fashion similar to the primitive types built into C#. This chapter describes how to define structures, while exposing the idiosyncrasies they may introduce.
- *Chapter 9—Well-Formed Types:* This chapter discusses more advanced type definition. It explains how to implement operators, such as + and casts, and describes how to encapsulate multiple classes into a single library. In addition, the chapter demonstrates defining namespaces and XML comments, and discusses how to design classes for garbage collection.
- *Chapter 10—Exception Handling:* This chapter expands on the exception-handling introduction from Chapter 4 and describes how exceptions follow a hierarchy that enables creating custom exceptions. It also includes some best practices on exception handling.
- ***Chapter 11**—Generics:* Generics is perhaps the core feature missing from C# 1.0. This chapter fully covers this 2.0 feature. In addition, C# 4.0 added support for covariance and contravariance—something covered in the context of generics in this chapter.

- ***Chapter 12****—Delegates and Lambda Expressions:* Delegates begin clearly distinguishing C# from its predecessors by defining patterns for handling events within code. This virtually eliminates the need for writing routines that poll. Lambda expressions are the key concept that make C# 3.0's LINQ possible. This chapter explains how lambda expressions build on the delegate construct by providing a more elegant and succinct syntax. This chapter forms the foundation for the new collection API discussed next.
- *Chapter 13—Events:* Encapsulated delegates, known as events, are a core construct of the Common Language Runtime. Anonymous methods, another C# 2.0 feature, are also presented here.
- ***Chapter 14****—Collection Interfaces with Standard Query Operators:* The simple and yet elegantly powerful changes introduced in C# 3.0 begin to shine in this chapter as we take a look at the extension methods of the new Enumerable class. This class makes available an entirely new collection API known as the standard query operators and discussed in detail here.
- ***Chapter 15****—LINQ with Query Expressions:* Using standard query operators alone results in some long statements that are hard to decipher. However, query expressions provide an alternative syntax that matches closely with SQL, as described in this chapter.
- ***Chapter 16****—Building Custom Collections:* In building custom APIs that work against business objects, it is sometimes necessary to create custom collections. This chapter details how to do this, and in the process introduces contextual keywords that make custom collection building easier.
- ***Chapter 17****—Reflection, Attributes, and Dynamic Programming:* Object-oriented programming formed the basis for a paradigm shift in program structure in the late 1980s. In a similar way, attributes facilitate declarative programming and embedded metadata, ushering in a new paradigm. This chapter looks at attributes and discusses how to retrieve them via reflection. It also covers file input and output via the serialization framework within the Base Class Library. In C# 4.0 a new keyword, dynamic, was added to the language. This removed all type checking until runtime, a significant expansion of what can be done with C#.

- ***Chapter 18****—Multithreading:* Most modern programs require the use of threads to execute long-running tasks while ensuring active response to simultaneous events. As programs become more sophisticated, they must take additional precautions to protect data in these advanced environments. Programming multithreaded applications is complex. This chapter discusses how to work with threads and provides best practices to avoid the problems that plague multithreaded applications.
- ***Chapter 19****—Thread Synchronization:* Building on the preceding chapter, this one demonstrates some of the built-in threading pattern support that can simplify the explicit control of multithreaded code.
- ***Chapter 20****—Platform Interoperability and Unsafe Code:* Given that C# is a relatively young language, far more code is written in other languages than in C#. To take advantage of this preexisting code, C# supports interoperability—the calling of unmanaged code—through P/Invoke. In addition, C# provides for the use of pointers and direct memory manipulation. Although code with pointers requires special privileges to run, it provides the power to interoperate fully with traditional C-based application programming interfaces.
- *Chapter 21—The Common Language Infrastructure:* Fundamentally, C# is the syntax that was designed as the most effective programming language on top of the underlying Common Language Infrastructure. This chapter delves into how C# programs relate to the underlying runtime and its specifications.
- *Appendix A—Downloading and Installing the C# Compiler and CLI Platform:* This appendix provides instructions for setting up a C# compiler and the platform on which to run the code, Microsoft .NET or Mono.
- *Appendix B—Tic-Tac-Toe Source Code Listing:* This appendix provides a full listing of the source code displayed in parts within Chapter 3 and Chapter 4.
- *Appendix C—Interfacing with Mulithreading Patterns prior to the TPL and C# 5.0:* This appendix provides details on multithreading patterns for development prior to C# 5.0 and/or the Task Parallel Library.
- *Appendix D—Timers prior to the Async/Await Pattern of C# 5.0:* This appendix describes three different types of timers for use when .NET 4.5/C# 5.0 is not available.

- *C# 3.0, 4.0, 5.0 Index:* These indexes provide a quick reference for the features added in C# 3.0–5.0. They are specifically designed to help programmers quickly update their language skills to a more recent version.

I hope you find this book to be a great resource in establishing your C# expertise and that you continue to reference it for those areas that you use less frequently well after you are proficient in C#.

—Mark Michaelis
IntelliTect.com/mark
Twitter: @Intellitect, @MarkMichaelis

Acknowledgments

NO BOOK CAN be published by the author alone, and I am extremely grateful for the multitude of people who helped me with this one. The order in which I thank people is not significant, except for those that come first. By far, my family has made the biggest sacrifice to allow me to complete this. Benjamin, Hanna, and Abigail often had a Daddy distracted by this book, but Elisabeth suffered even more so. She was often left to take care of things, holding the family's world together on her own. I would like to say it got easier with each edition but, alas, no; as the kids got older, life became more hectic, and without me Elisabeth was stretched to the breaking point virtually all the time. A huge sorry and ginormous "Thank You!"

Many technical editors reviewed each chapter in minute detail to ensure technical accuracy. I was often amazed by the subtle errors these folks still managed to catch: Paul Bramsman, Kody Brown, Ian Davis, Doug Dechow, Gerard Frantz, Thomas Heavey, Anson Horton, Brian Jones, Shane Kercheval, Angelika Langer, Eric Lippert, John Michaelis, Jason Morse, Nicholas Paldino, Jon Skeet, Michael Stokesbary, Robert Stokesbary, John Timney, and Stephen Toub. Thanks also to Mandy Frei who diligently kept notes of changes needed for reprints.

Eric is no less than amazing. His grasp of the C# vocabulary is truly astounding and I am very appreciative of his edits, especially when he pushed for perfection in terminology. His improvements to the C# 3.0 chapters were incredibly significant, and in the second edition my only regret was that I didn't have him review all the chapters. However, that regret is no longer. Eric painstakingly reviewed every *Essential C# 5.0* chapter with amazing

detail and precision. I am extremely grateful for his contribution to making this book even better than the earlier editions. Thanks, Eric! I can't imagine anyone better for the job. You deserve all the credit for raising the bar from good to great.

Like Eric and C#, there are fewer than a handful of people who know .NET multithreading as well as Stephen Toub. Accordingly, Stephen focused on the two rewritten (for a third time) multithreading chapters and their new focus on asynchronous support in C# 5.0. Thanks, Stephen!

Thanks to everyone at Addison-Wesley for their patience in working with me in spite of my frequent focus on everything else except the manuscript. Thanks to Elizabeth Ryan, Audrey Doyle, Vicki Rowland, Curt Johnson, and Joan Murray. Joan deserves a special medal of patience for the number of times I delayed not only with deliverables but even responding to emails.

About the Authors

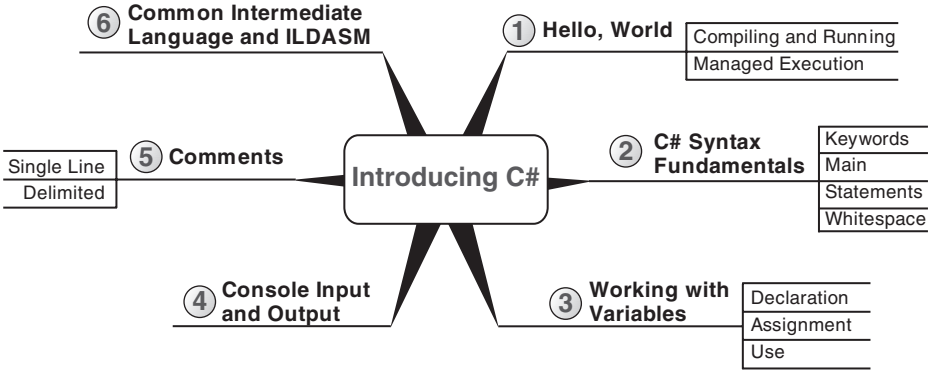
Mark Michaelis is the founder of IntelliTect and serves as the Chief Technical Architect and Trainer. Since 1996, he has been a Microsoft MVP for C#, Visual Studio Team System, and the Windows SDK, and in 2007 he was recognized as a Microsoft Regional Director. He also serves on several Microsoft software design review teams, including C#, the Connected Systems, Office/SharePoint, and Visual Studio. He speaks at developer conferences and has written numerous articles and other books. He holds a bachelor of arts degree in philosophy from the University of Illinois and a master's degree in computer science from the Illinois Institute of Technology. When not bonding with his computer, he is busy with his family or training for another triathlon (having completed his first Ironman in 2008). He lives in Spokane, Washington, with his wife Elisabeth and three children, Benjamin, Hanna, and Abigail.

Eric Lippert is a principal developer on the C# compiler team at Microsoft. He has worked on the design and implementation of the Visual Basic, VBScript, Jscript, and C# languages and on Visual Studio Tools For Office, and is a member of the C# language design team. When not writing or editing books about C#, he does his best to keep his tiny sailboat upright. He lives in Seattle with his wife, Leah.

This page intentionally left blank

1
Introducing C#

C# IS NOW A WELL-ESTABLISHED LANGUAGE that builds on features found in its predecessor C-style languages (C, C++, and Java), making it immediately familiar to many experienced programmers.[1] Part of a larger, more complex execution platform called the Common Language Infrastructure (CLI), C# is a programming language for building software components and applications.



This chapter introduces C# using the traditional `HelloWorld` program. The chapter focuses on C# syntax fundamentals, including defining an entry point into the C# program executable. This will familiarize you with the C#

1. The first C# design meeting took place in 1998.

syntax style and structure, and it will enable you to produce the simplest of C# programs. Prior to the discussion of C# syntax fundamentals is a summary of managed execution context, which explains how a C# program executes at runtime. This chapter ends with a discussion of variable declaration, writing and retrieving data from the console, and the basics of commenting code in C#.

Hello, World

The best way to learn a new programming language is to write code. The first example is the classic `HelloWorld` program. In this program, you will display some text to the screen.

Listing 1.1 shows the complete `HelloWorld` program; in the following sections, you will compile the code.

LISTING 1.1: `HelloWorld` **in C#**[2]

```
class HelloWorld
{
 static void Main()
 {
   System.Console.WriteLine("Hello. My name is Inigo Montoya.");
 }
}
```

> **NOTE**
>
> C# is a case-sensitive language: Incorrect case prevents the code from compiling successfully.

Those experienced in programming with Java, C, or C++ will immediately see similarities. Like Java, C# inherits its basic syntax from C and C++.[3] Syntactic punctuation (such as semicolons and curly braces), features (such as case sensitivity), and keywords (such as `class`, `public`, and `void`)

2. Refer to the movie *The Princess Bride* if you're confused about the Inigo Montoya references.
3. When creating C#, the language creators sat down with the specifications for C/C++, literally crossing out the features they didn't like and creating a list of the ones they did like. The group also included designers with strong backgrounds in other languages.

are familiar to programmers experienced in these languages. Beginners and programmers from other languages will quickly find these constructs intuitive.

Compiling and Running the Application

The C# compiler allows any file extension for files containing C# source code, but `.cs` is typically used. After saving the source code to a file, developers must compile it. (Appendix A provides instructions for installing the compiler.) Because the mechanics of the command are not part of the C# standard, the compilation command varies depending on the C# compiler implementation.

If you place Listing 1.1 into a file called `HelloWorld.cs`, the compilation command in Output 1.1 will work with the Microsoft .NET compiler (assuming appropriate paths to the compiler are set up).[4]

Output 1.1

```
>csc.exe HelloWorld.cs
Microsoft (R) Visual C# Compiler version 4.0.30319.17626
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.
```

The exact output will vary depending on what version of the compiler you use.

Running the resultant program, `HelloWorld.exe`, displays the message shown in Output 1.2.

Output 1.2

```
>HelloWorld.exe
Hello. My name is Inigo Montoya.
```

The program created by the C# compiler, `HelloWorld.exe`, is an **assembly.** Instead of creating an entire program that can be executed independently,

4. Compilation using the Mono compiler, an open source compiler now available from www.mono-project.com, is virtually identical, except that the compiler name is `mcs.exe` rather than `csc.exe`. Although I would very much have liked to place instructions for each platform here, doing so detracts from the topic of introducing C#. See Appendix A for details on Mono.

developers can create a library of code that can be referenced by another, larger program. Libraries (or class libraries) use the filename extension `.dll`, which stands for Dynamic Link Library (DLL). A library is also an assembly. In other words, the output from a successful C# compile is an assembly regardless of whether it is a program or a library.

Language Contrast: Java—Filename Must Match Class Name

In Java, the filename must follow the name of the class. In C#, this convention is frequently followed but is not required. In C#, it is possible to have two classes in one file, and starting with C# 2.0, it's possible to have a single class span multiple files.

C# Syntax Fundamentals

Once you successfully compile and run the `HelloWorld` program, you are ready to start dissecting the code to learn its individual parts. First, consider the C# keywords along with the identifiers that the developer chooses.

BEGINNER TOPIC

Keywords

In order for the compiler to interpret the code, certain words within C# have special status and meaning. Known as **keywords,** they provide the concrete syntax that the compiler uses to interpret the expressions the programmer writes. In the `HelloWorld` program, `class`, `static`, and `void` are examples of keywords.

The compiler uses the keywords to identify the structure and organization of the code. Because the compiler interprets these words with elevated significance, C# requires that developers place keywords only in certain locations. When programmers violate these rules, the compiler will issue errors.

C# Keywords

Table 1.1 shows the C# keywords.

TABLE 1.1: C# Keywords

abstract	else	let* (3)	sizeof
add*	enum	lock	stackalloc
alias* (2)	equals* (3)	long	static
as	event	namespace	string
ascending* (3)	explicit	new	struct
async* (5)	extern	null	switch
await* (5)	false	object	this
base	finally	on* (3)	throw
bool	fixed	operator	true
break	float	orderby* (3)	try
by* (3)	for	out	typeof
byte	foreach	override	uint
case	from* (3)	params	ulong
catch	get* (1)	partial* (2)	unchecked
char	global* (2)	private	unsafe
checked	goto	protected	ushort
class	group* (3)	public	using
const	if	readonly	value* (1)
continue	implicit	ref	var* (3)
decimal	in	remove*	virtual
default	int	return	void
delegate	interface	sbyte	volatile
descending* (3)	internal	sealed	where* (3)
do	into* (3)	select* (3)	while
double	is	set* (1)	yield* (2)
dynamic* (4)	join* (3)	short	

* Contextual keyword

Numbers in parentheses (*n*) identify in which version the contextual keyword was added.

After C# 1.0, no new **reserved keywords** were introduced to C#. However, some constructs in later versions use **contextual keywords,** which are significant only in specific locations. Outside these designated locations, contextual keywords have no special significance.[5] By this method, most C# 1.0 code is compatible with the later standards.[6]

Identifiers

Like other languages, C# includes **identifiers** to identify constructs that the programmer codes. In Listing 1.1, `HelloWorld` and `Main` are examples of identifiers. The identifiers assigned to a construct are used to refer back to the construct later, so it is important that the names the developer assigns are meaningful rather than arbitrary.

A keen ability to select succinct and indicative names is an important characteristic of a strong programmer because it means the resultant code will be easier to understand and reuse. Clarity coupled with consistency is important enough that the .NET Framework Guidelines advise against the use of abbreviations or contractions in identifier names and even recommend avoiding acronyms that are not widely accepted. If an acronym is sufficiently well established (HTML, for example) use it consistently: Avoid spelling out the accepted acronym at some times but not others. Generally, adding the constraint that all acronyms be included in a glossary of terms places enough overhead on the use of acronyms such that they are not used flippantly. Ultimately, select clear, possibly even verbose names—especially when working on a team or when developing a library against which others will program.

There are two basic casing formats for an identifier. **Pascal case** (henceforth PascalCase), as the CLI creators refer to it because of its popularity in

---

5. For example, early in the design of C# 2.0, the language designers designated `yield` as a keyword, and Microsoft released alpha versions of the C# 2.0 compiler, with `yield` as a designated keyword, to thousands of developers. However, the language designers eventually determined that by using `yield return` rather than `yield`, they could ultimately avoid adding `yield` as a keyword because it would have no special significance outside its proximity to `return`.
6. There are some rare and unfortunate incompatibilities, such as the following:
   - C# 2.0 requiring implementation of `IDisposable` with the `using` statement, rather than simply a `Dispose()` method
   - Some rare generic expressions such as `F(G<A,B>(7));` in C# 1.0, that means `F( (G<A) , (B>7) )` and in C# 2.0, that means to call generic method `G<A,B>` with argument `7` and pass the result to `F`

the Pascal programming language, capitalizes the first letter of each word in an identifier name; examples include `ComponentModel`, `Configuration`, and `HttpFileCollection`. As `HttpFileCollection` demonstrates with HTTP, when using acronyms that are more than two letters long only the first letter is capitalized. The second format, camel case (henceforth camelCase), follows the same convention, except that the first letter is lowercase; examples include `quotient`, `firstName`, `httpFileCollection`, `ioStream`, and `theDreadPirateRoberts`.

Guidelines

DO favor clarity over brevity when naming identifiers.

DO NOT use abbreviations or contractions within identifier names.

DO NOT use any acronyms unless they are widely accepted, and even then, only when necessary.

Notice that although underscores are legal, generally there are no underscores, hyphens, or other nonalphanumeric characters in identifier names. Furthermore, C# doesn't follow its predecessors in that Hungarian notation (prefixing a name with a data type abbreviation) is not used. This avoids the variable rename that is necessary when data types change, and the inconsistency in the data type prefix that is frequently encountered when using Hungarian notation.

In some rare cases, some identifiers, such as `Main`, can have a special meaning in the C# language.

Guidelines

DO capitalize both characters in two-character acronyms, except for the first word of a camelCased identifier.

DO capitalize only the first character in acronyms with three or more characters, except for the first word of a camelCased identifier.

DO NOT capitalize any of the characters in acronyms at the beginning of a camelCased identifier.

DO NOT use Hungarian notation (that is, do not encode the type of a variable in its name).

ADVANCED TOPIC

Keywords

Although it is rare, keywords may be used as identifiers if they include "@" as a prefix. For example, you could name a local variable `@return`. Similarly (although it doesn't conform to the casing standards of C# coding standards), it is possible to name a method `@throw()`.

There are also four undocumented reserved keywords in the Microsoft implementation: `__arglist`, `__makeref`, `__reftype`, and `__refvalue`. These are required only in rare interop scenarios and you can ignore them for all practical purposes. Note that these four special keywords begin with two underscores. The designers of C# reserve the right to make any identifier that begins with two underscores into a keyword in a future version; for safety, avoid ever creating such an identifier yourself.

Type Definition

All executable code in C# appears within a type definition, and the most common type definition begins with the keyword `class`. A **class definition** is the section of code that generally begins with `class identifier { ... }`, as shown in Listing 1.2.

Listing 1.2: Basic Class Declaration

```
class HelloWorld
{
  //...
}
```

The name used for the type (in this case, `HelloWorld`) can vary, but by convention, it must be PascalCased. For this particular example, therefore, other possible names are `Greetings`, `HelloInigoMontoya`, `Hello`, or simply `Program`. (`Program` is a good convention to follow when the class contains the `Main()` method, described next.)

Guidelines

DO name classes with nouns or noun phrases.

DO use PascalCasing for all class names.

Generally, programs contain multiple types, each containing multiple methods.

Main

BEGINNER TOPIC

What Is a Method?

Syntactically, a **method** in C# is a named block of code introduced by a method declaration (for example, `static void Main()`) and (usually) followed by zero or more statements within curly braces. Methods perform computations and/or actions. Similar to paragraphs in written languages, methods provide a means of structuring and organizing code so that it is more readable. More importantly, methods can be reused and called from multiple places, and so avoid the need to duplicate code. The method declaration introduces the method and defines the method name along with the data passed to and from the method. In Listing 1.3, `Main()` followed by `{ ... }` is an example of a C# method.

The location where C# programs begin execution is the **Main method,** which begins with `static void Main()`. When you execute the program by typing `HelloWorld.exe` at the command console, the program starts up, resolves the location of `Main`, and begins executing the first statement within Listing 1.3.

LISTING 1.3: Breaking Apart `HelloWorld`

```
class HelloWorld
{
  static void Main()
  {
    System.Console.WriteLine("Hello, My name is Inigo Montoya");
  }
}
```

Method Declaration
Main
Statement
Class Definition

Although the Main method declaration can vary to some degree, `static` and the method name, `Main`, are always required for a program.

ADVANCED TOPIC

Declaration of the Main Method

C# requires that the Main method return either `void` or `int`, and that it take either no parameters, or a single array of strings. Listing 1.4 shows the full declaration of the Main method.

Listing 1.4: The Main Method, with Parameters and a Return

```
static int Main(string[] args)
{
    //...
}
```

The `args` parameter is an array of strings corresponding to the command-line arguments. However, the first element of the array is not the program name but the first command-line parameter to appear after the executable name, unlike in C and C++. To retrieve the full command used to execute the program use `System.Environment.CommandLine`.

The `int` returned from `Main` is the status code and it indicates the success of the program's execution. A return of a nonzero value generally indicates an error.

Language Contrast: C++/Java—`main()` Is All Lowercase

Unlike its C-style predecessors, C# uses an uppercase *M* for the Main method in order to be consistent with the PascalCased naming conventions of C#.

The designation of the `Main` method as `static` indicates that other methods may call it directly off the class definition. Without the `static` designation, the command console that started the program would need to perform additional work (known as **instantiation**) before calling the method. (Chapter 5 contains an entire section devoted to the topic of static members.)

Placing `void` prior to `Main()` indicates that this method does not return any data. (This is explained further in Chapter 2.)

One distinctive C/C++ style characteristic followed by C# is the use of curly braces for the body of a construct, such as the class or the method. For example, the `Main` method contains curly braces that surround its implementation; in this case, only one statement appears in the method.

Statements and Statement Delimiters

The `Main` method includes a single statement, `System.Console.WriteLine()`, which is used to write a line of text to the console. C# generally uses a semicolon to indicate the end of a **statement,** where a statement comprises one or more actions that the code will perform. Declaring a variable, controlling the program flow, and calling a method are typical uses of statements.

Language Contrast: Visual Basic—Line-Based Statements

Some languages are line-based, meaning that without a special annotation, statements cannot span a line. Until Visual Basic 2010, Visual Basic was an example of a line-based language. It required an underscore at the end of a line to indicate that a statement spans multiple lines. Starting with Visual Basic 2010, many cases were introduced where the line continuation character was optional.

ADVANCED TOPIC

Statements without Semicolons

Many programming elements in C# end with a semicolon. One example that does not include the semicolon is a `switch` statement. Because curly braces are always included in a `switch` statement, C# does not require a semicolon following the statement. In fact, code blocks themselves are considered statements (they are also composed of statements) and they don't require closure using a semicolon. Similarly, there are cases, such as the `using` declarative, in which a semicolon occurs at the end but it is not a statement.

Since creation of a newline does not separate statements, you can place multiple statements on the same line and the C# compiler will interpret the line to have multiple instructions. For example, Listing 1.5 contains two statements on a single line that, in combination, display `Up` and `Down` on two separate lines.

Listing 1.5: Multiple Statements on One Line

```
System.Console.WriteLine("Up");System.Console.WriteLine("Down");
```

C# also allows the splitting of a statement across multiple lines. Again, the C# compiler looks for a semicolon to indicate the end of a statement (see Listing 1.6).

Listing 1.6: Splitting a Single Statement across Multiple Lines

```
System.Console.WriteLine(
  "Hello. My name is Inigo Montoya.");
```

In Listing 1.6, the original `WriteLine()` statement from the `HelloWorld` program is split across multiple lines.

BEGINNER TOPIC

What Is Whitespace?

Whitespace is the combination of one or more consecutive formatting characters such as tab, space, and newline characters. Eliminating all whitespace between words is obviously significant, as is whitespace within a quoted string.

Whitespace

The semicolon makes it possible for the C# compiler to ignore whitespace in code. Apart from a few exceptions, C# allows developers to insert whitespace throughout the code without altering its semantic meaning. In Listing 1.5 and Listing 1.6, it didn't matter whether a newline was inserted within a statement or between statements, and doing so had no effect on the resultant executable created by the compiler.

Frequently, programmers use whitespace to indent code for greater readability. Consider the two variations on `HelloWorld` shown in Listing 1.7 and Listing 1.8.

Listing 1.7: No Indentation Formatting

```
class HelloWorld
{
static void Main()
{
System.Console.WriteLine("Hello Inigo Montoya");
}
}
```

LISTING 1.8: Removing Whitespace

```
class HelloWorld{static void Main()
{System.Console.WriteLine("Hello Inigo Montoya");}}
```

Although these two examples look significantly different from the original program, the C# compiler sees them as identical.

BEGINNER TOPIC

Formatting Code with Whitespace

Indenting the code using whitespace is important for greater readability. As you begin writing code, you need to follow established coding standards and conventions in order to enhance code readability.

The convention used in this book is to place curly braces on their own line and to indent the code contained between the curly brace pair. If another curly brace pair appears within the first pair, all the code within the second set of braces is also indented.

This is not a uniform C# standard, but a stylistic preference.

Working with Variables

Now that you've been introduced to the most basic C# program, it's time to declare a local variable. Once a variable is declared, you can assign it a value, replace that value with a new value, and use it in calculations, output, and so on. However, you cannot change the data type of the variable. In Listing 1.9, `string max` is a variable declaration.

LISTING 1.9: Declaring and Assigning a Variable

```
class MiracleMax
{
  static void Main()
  {
        data type
        string max;
               variable
      max = "Have fun storming the castle!";

      System.Console.WriteLine(max);
  }
}
```

BEGINNER TOPIC

Local Variables

A **variable** refers to a storage location by a name that the program can later assign and modify. *Local* indicates that the programmer **declared** the variable within a method.

To declare a variable is to define it, which you do by

1. Specifying the type of data which the variable will contain
2. Assigning it an identifier (name)

Data Types

Listing 1.9 declares a variable with the data type `string`. Other common data types used in this chapter are `int` and `char`.

- `int` is the C# designation of an integer type that is 32 bits in size.
- `char` is used for a character type. It is 16 bits, large enough for (nonsurrogate) Unicode characters.

The next chapter looks at these and other common data types in more detail.

BEGINNER TOPIC

What Is a Data Type?

The type of data that a variable declaration specifies is called a **data type** (or object type). A data type, or simply **type,** is a classification of things that share similar characteristics and behavior. For example, *animal* is a type. It classifies all things (monkeys, warthogs, and platypuses) that have animal characteristics (multicellular, capacity for locomotion, and so on). Similarly, in programming languages, a type is a definition for several items endowed with similar qualities.

Declaring a Variable

In Listing 1.9, `string max` is a variable declaration of a string type whose name is `max`. It is possible to declare multiple variables within the same

statement by specifying the data type once and separating each identifier with a comma. Listing 1.10 demonstrates this.

LISTING 1.10: Declaring Two Variables within One Statement

```
string message1, message2;
```

Because a multivariable declaration statement allows developers to provide the data type only once within a declaration, all variables will be of the same type.

In C#, the name of the variable may begin with any letter or an underscore (_), followed by any number of letters, numbers, and/or underscores. By convention, however, local variable names are camelCased (the first letter in each word is capitalized, except for the first word) and do not include underscores.

> **Guidelines**
>
> **DO** use camelCasing for local variables.

Assigning a Variable

After declaring a local variable, you must assign it a value before reading from it. One way to do this is to use the = **operator,** also known as the **simple assignment operator.** Operators are symbols used to identify the function the code is to perform. Listing 1.11 demonstrates how to use the assignment operator to designate the string values to which the variables `max`[7] and `valerie` will point.

LISTING 1.11: Changing the Value of a Variable

```
class MiracleMax
{
  static void Main()
  {
      string valerie;
      string max = "Have fun storming the castle!";

      valerie = "Think it will work?";

      System.Console.WriteLine(max);
      System.Console.WriteLine(valerie);
```

7. `max` does not mean the math function here, but rather is used as a variable name.

```
        max = "It would take a miracle.";
        System.Console.WriteLine(max);
    }
}
```

From this listing, observe that it is possible to assign a variable as part of the variable declaration (as it was for `max`), or afterward in a separate statement (as with the variable `valerie`). The value assigned must always be on the right side.

Running the compiled `MiracleMax.exe` program produces the code shown in Output 1.3.

OUTPUT 1.3

```
>MiracleMax.exe
Have fun storming the castle!
Think it will work?
It would take a miracle.
```

C# requires that local variables be determined by the compiler to be "definitely assigned" before they are read. Additionally, an assignment returns a value. Therefore, C# allows two assignments within the same statement, as demonstrated in Listing 1.12.

LISTING 1.12: Assignment Returning a Value That Can Be Assigned Again

```
class MiracleMax
{
  static void Main()
  {
    // ...
    string requirements, max;
    requirements = max = "It would take a miracle.";
    // ...
  }
}
```

Using a Variable

The result of the assignment, of course, is that you can then refer to the value using the variable identifier. Therefore, when you use the variable `max` within the `System.Console.WriteLine(max)` statement, the program displays `Have fun storming the castle!`, the value of `max`, on the console. Changing the

value of max and executing the same System.Console.WriteLine(max) statement causes the new max value, It would take a miracle., to be displayed.

ADVANCED TOPIC

Strings Are Immutable

All data of type string, whether string literals or otherwise, is immutable (or unmodifiable). For example, it is not possible to change the string "Come As You Are" to "Come As You Age." A change such as this requires that you reassign the variable to point to a new location in memory, instead of modifying the data to which the variable originally referred.

Console Input and Output

This chapter already used System.Console.WriteLine repeatedly for writing out text to the command console. In addition to being able to write out data, a program needs to be able to accept data that a user may enter.

Getting Input from the Console

One way to retrieve text that is entered at the console is to use System.Console.ReadLine(). This method stops the program execution so that the user can enter characters. When the user presses the Enter key, creating a newline, the program continues. The output, also known as the **return,** from the System.Console.ReadLine() method is the string of text that was entered. Consider Listing 1.13 and the corresponding output shown in Output 1.4.

LISTING 1.13: Using System.Console.ReadLine()

```
class HeyYou
{
  static void Main()
  {
      string firstName;
      string lastName;

      System.Console.WriteLine("Hey you!");

      System.Console.Write("Enter your first name: ");
      firstName = System.Console.ReadLine();
```

```
        System.Console.Write("Enter your last name: ");
        lastName = System.Console.ReadLine();

        ...
    }
}
```

OUTPUT 1.4

```
>HeyYou.exe
Hey you!
Enter your first name: Inigo
Enter your last name: Montoya
```

After each prompt, this program uses the `System.Console.ReadLine()` method to retrieve the text the user entered and assign it to an appropriate variable. By the time the second `System.Console.ReadLine()` assignment completes, `firstName` contains the value `Inigo` and `lastName` refers to the value `Montoya`.

ADVANCED TOPIC

System.Console.Read()

In addition to the `System.Console.ReadLine()` method, there is also a `System.Console.Read()` method. However, the data type returned by the `System.Console.Read()` method is an integer corresponding to the character value read, or `-1` if no more characters are available. To retrieve the actual character, it is necessary to first cast the integer to a character, as shown in Listing 1.14.

LISTING 1.14: Using `System.Console.Read()`

```
int readValue;
char character;
readValue = System.Console.Read();
character = (char) readValue;
System.Console.Write(character);
```

The `System.Console.Read()` method does not return the input until the user presses the Enter key; no processing of characters will begin, even if the user types multiple characters before pressing the Enter key.

In C# 2.0, there appears a new method called `System.Console.ReadKey()` which, in contrast to `System.Console.Read()`, returns the input after a single keystroke. It allows the developer to intercept the keystroke and perform actions such as key validation, restricting the characters to numerics.

Writing Output to the Console

In Listing 1.13, you prompt the user for his first and last names using the method `System.Console.Write()` rather than `System.Console.WriteLine()`. Instead of placing a newline character after displaying the text, the `System.Console.Write()` method leaves the current position on the same line. In this way, any text the user enters will be on the same line as the prompt for input. The output from Listing 1.13 demonstrates the effect of `System.Console.Write()`.

The next step is to write the values retrieved using `System.Console.ReadLine()` back to the console. In the case of Listing 1.15, the program writes out the user's full name. However, instead of using `System.Console.WriteLine()` as before, this code will use a slight variation. Output 1.5 shows the corresponding output.

Listing 1.15: Formatting Using `System.Console.WriteLine()`

```
class HeyYou
{
  static void Main()
  {
      string firstName;
      string lastName;

      System.Console.WriteLine("Hey you!");

      System.Console.Write("Enter your first name: ");
      firstName = System.Console.ReadLine();

      System.Console.Write("Enter your last name: ");
      lastName = System.Console.ReadLine();

      System.Console.WriteLine(
        "Your full name is {0} {1}.", firstName, lastName);
  }
}
```

Output 1.5

```
Hey you!
Enter your first name: Inigo
Enter your last name: Montoya
Your full name is Inigo Montoya.
```

Instead of writing out `Your full name is` followed by another `Write` statement for `firstName`, a third `Write` statement for the space, and finally a `WriteLine` statement for `lastName`, Listing 1.15 writes out the entire output using **composite formatting.** With composite formatting, the code first supplies a **format string** to define the output format. In this example, the format string is `"Your full name is {0} {1}."`. It identifies two indexed placeholders for data insertion in the string.

Note that the index value begins at zero. Each inserted parameter (known as a **format item**) appears after the format string in the order corresponding to the index value. In this example, since `firstName` is the first parameter to follow immediately after the format string, it corresponds to index value `0`. Similarly, `lastName` corresponds to index value `1`.

Note that the placeholders within the format string need not appear in order. For example, Listing 1.16 switches the order of the indexed placeholders and adds a comma, which changes the way the name is displayed (see Output 1.6).

Listing 1.16: Swapping the Indexed Placeholders and Corresponding Variables

```
System.Console.WriteLine("Your full name is {1}, {0}",
  firstName, lastName);
```

Output 1.6

```
Hey you!
Enter your first name: Inigo
Enter your last name: Montoya
Your full name is Montoya, Inigo
```

In addition to not having the placeholders appear consecutively within the format string, it is possible to use the same placeholder multiple times within a format string. Furthermore, it is possible to omit a placeholder. It is not possible, however, to have placeholders that do not have a corresponding parameter.

Comments

In this section, we modify the program in Listing 1.15 by adding comments. In no way does this vary the execution of the program; rather, providing comments within the code makes the code more understandable. Listing 1.17 shows the new code, and Output 1.7 shows the corresponding output.

LISTING 1.17: Commenting Your Code

```
class CommentSamples
{
  static void Main()
  {
                                        single-line comment
      string firstName; // Variable for storing the first name
      string lastName;  // Variable for storing the last name

      System.Console.WriteLine("Hey you!");

                          delimited comment inside statement
      System.Console.Write /* No new line */ (
          "Enter your first name: ");
      firstName = System.Console.ReadLine();

      System.Console.Write /* No new line */ (
          "Enter your last name: ");
      lastName = System.Console.ReadLine();

      /* Display a greeting to the console
         using composite formatting. */      } delimited comment
      System.Console.WriteLine("Your full name is {0} {1}.",
          firstName, lastName);
      // This is the end
      // of the program listing
  }
}
```

OUTPUT 1.7

```
Hey you!
Enter your first name: Inigo
Enter your last name: Montoya
Your full name is Inigo Montoya.
```

In spite of the inserted comments, compiling and executing the new program produces the same output as before.

Programmers use comments to describe and explain the code they are writing, especially where the syntax itself is difficult to understand, or perhaps a particular algorithm implementation is surprising. Since comments are pertinent only to the programmer reviewing the code, the compiler ignores comments and generates an assembly that is devoid of any trace that comments were part of the original source code.

Table 1.2 shows four different C# comment types. The program in Listing 1.17 includes two of these.

TABLE 1.2: C# Comment Types

Comment Type	Description	Example
Delimited comments	A forward slash followed by an asterisk, `/*`, identifies the beginning of a delimited comment. To end the comment use an asterisk followed by a forward slash: `*/`. Comments of this form may span multiple lines in the code file or appear embedded within a line of code. The asterisks that appear at the beginning of the lines but within the delimiters are simply for formatting.	`/*comment*/`
Single-line comments	Comments may also be declared with a delimiter comprising two consecutive forward slash characters: `//`. The compiler treats all text from the delimiter to the end of the line as a comment. Comments of this form comprise a single line. It is possible, however, to place sequential single-line comments one after another, as is the case with the last comment in Listing 1.17.	`//comment`
XML delimited comments	Comments that begin with `/**` and end with `**/` are called XML delimited comments. They have the same characteristics as regular delimited comments, except that instead of ignoring XML comments entirely, the compiler can place them into a separate text file. XML delimited comments were only explicitly added in C# 2.0, but the syntax is compatible with C# 1.0.	`/**comment**/`
XML single-line comments	XML single-line comments begin with `///` and continue to the end of the line. In addition, the compiler can save single-line comments into a separate file with the XML delimited comments.	`///comment`

A more comprehensive discussion of the XML comments appears in Chapter 9, where we further discuss the various XML tags.

There was a period in programming history where a prolific set of comments implied a disciplined and experienced programmer. This is no longer the case. Instead, the code that is readable without comments is more valuable than that which requires comments in order to clarify what it does. If developers find it necessary to enter comments in order to clarify what a particular code block is doing, they should favor rewriting the code more clearly over commenting it. Writing comments that are simply a repeat of what the code clearly shows serves only to clutter, decrease readability, and increase the likelihood of the comments going out of date because the code changes without the comments getting updated.

Guidelines

DO NOT use comments unless they describe something that is not obvious to someone other than the developer who wrote the code.

DO favor writing clearer code over entering comments to clarify a complicated algorithm.

BEGINNER TOPIC

Extensible Markup Language (XML)

The Extensible Markup Language (XML) is a simple and flexible text format frequently used within Web applications and for exchanging data between applications. XML is extensible because included within an XML document is information that describes the data, known as **metadata.** Here is a sample XML file.

```
<?xml version="1.0" encoding="utf-8" ?>
<body>
  <book title="Essential C# 5.0">
      <chapters>
          <chapter title="Introducing C#"/>
          <chapter title="Operators and Control Flow"/>
          ...
      </chapters>
  </book>
</body>
```

The file starts with a header indicating the version and character encoding of the XML file. After that appears one main "book" element. Elements begin with a word in angle brackets, such as `<body>`. To end an element, place the same word in angle brackets and add a forward slash to prefix the word, as in `</body>`. In addition to elements, XML supports attributes. `title="Essential C# 5.0"` is an example of an XML attribute. Note that the metadata (book title, chapter, and so on) describing the data ("Essential C# 5.0", "Operators and Control Flow") is included in the XML file. This can result in rather bloated files, but it offers the advantage that the data includes a description to aid in interpreting the data.

Managed Execution and the Common Language Infrastructure

The processor cannot directly interpret an assembly. Assemblies consist mainly of a second language known as the **Common Intermediate Language** (**CIL**), or **IL** for short.[8] The C# compiler transforms the C# source file into this intermediate language. An additional step, usually performed at execution time, is required to change the CIL code into **machine code** that the processor can understand. This involves an important element in the execution of a C# program: the **Virtual Execution System** (**VES**). The VES, also casually referred to as the **runtime,** compiles CIL code as needed (this process is known as **just-in-time** compilation or **jitting**). The code that executes under the context of an agent such as the runtime is **managed code,** and the process of executing under control of the runtime is **managed execution.** The code is "managed" because the runtime controls significant portions of the program's behavior by managing aspects such as memory allocation, security, and just-in-time compilation. Code that does not require the runtime in order to execute is **native code** (or **unmanaged code**).

The specification for a VES is included in a broader specification known as the **Common Language Infrastructure** (**CLI**) specification.[9] An international standard, the CLI includes specifications for the following:

- The VES or runtime
- The CIL

8. A third term for CIL is Microsoft IL (MSIL). This book uses the term *CIL* because it is the term adopted by the CLI standard. IL is prevalent in conversation among people writing C# code because they assume that IL refers to CIL rather than other types of intermediate languages.
9. Miller, J., and S. Ragsdale. 2004. *The Common Language Infrastructure Annotated Standard*. Boston: Addison-Wesley.

- A type system that supports language interoperability, known as the **Common Type System** (**CTS**)
- Guidance on how to write libraries that are accessible from CLI-compatible languages (available in the **Common Language Specification** [**CLS**])
- Metadata that enables many of the services identified by the CLI (including specifications for the layout or file format of assemblies)
- A common programming framework, the Base Class Library (BCL), which developers in all languages can utilize

> **NOTE**
>
> The term *runtime* can refer to either execution time or the Virtual Execution System. To help clarify, this book uses the term *execution time* to indicate when the program is executing, and it uses the term *runtime* when discussing the agent responsible for managing the execution of a C# program while it executes.

Running within the context of a CLI implementation enables support for a number of services and features that programmers do not need to code for directly, including the following.

- *Language interoperability:* interoperability between different source languages. This is possible because the language compilers translate each source language to the same intermediate language (CIL).
- *Type safety:* checks for conversion between types, ensuring that only conversions between compatible types will occur. This helps prevent the occurrence of buffer overruns, a leading cause of security vulnerabilities.
- *Code access security:* certification that the assembly developer's code has permission to execute on the computer.
- *Garbage collection:* memory management that automatically de-allocates memory previously allocated by the runtime.
- *Platform portability:* support for potentially running the same assembly on a variety of operating systems. One obvious restriction is that no platform-dependent libraries are used; therefore, as with Java, there

are potentially some platform dependencies idiosyncrasies that need to be worked out.

- *BCL:* provides a large foundation of code that developers can depend on (in all CLI implementations) so that they do not have to develop the code themselves.

> **NOTE**
>
> This section gives a brief synopsis of the CLI to familiarize you with the context in which a C# program executes. It also provides a summary of some of the terms that appear throughout this book. Chapter 21 is devoted to the topic of the CLI and its relevance to C# developers. Although the chapter appears last in the book, it does not depend on any earlier chapters, so if you want to become more familiar with the CLI, you can jump to it at any time.

C# and .NET Versioning

Readers will notice that Output 1.1 refers to the ".NET Framework version 4.5." At the time of this writing, Microsoft had six major releases to the .NET Framework and only five C# compiler releases. .NET Framework version 3.0 was an additional set of API libraries released in between C# compiler releases (and Visual Studio 2005 and 2008 versions). As a result, the .NET Framework version that corresponded with C# 3.0 was 3.5. With the release of C# 4.0 and the .NET Framework 4.0, the version numbers were synchronized. However, the .NET version for C# 5.0 is .NET Framework 4.5, so version numbers in the current generation are no longer aligned. Table 1.3 is a brief overview of the C# and .NET releases.

The majority of all code within this text will work with platforms other than Microsoft's as long as the compiler version corresponds to the version of code required. Although providing full details on each C# platform would be helpful for some, it proved to detract from the focus of learning C#, so the main body of the text is restricted to information on Microsoft's platform, .NET. This is simply because Microsoft has the predominant (by far) implementation. Furthermore, translation to another platform is fairly trivial.

TABLE 1.3: C# and .NET Versions

Comment Type	Description
C# 1.0 with .NET Framework 1.0/1.1 (Visual Studio 2002 and 2003)	The initial release of C#. A language built from the ground up to support .NET programming.
C# 2.0 with .NET Framework 2.0 (Visual Studio 2005)	Added generics to the C# language and libraries that supported generics to the .NET Framework 2.0.
.NET Framework 3.0	An additional set of APIs for distributed communications (Windows Communication Foundation—WCF), rich client presentation (Windows Presentation Foundation—WPF), workflow (Windows Workflow—WF), and Web authentication (Cardspaces).
C# 3.0 with .NET Framework 3.5 (Visual Studio 2008)	Added support for LINQ, a significant improvement to the APIs used for programming collections. The .NET Framework 3.5 provided libraries that extended existing APIs to make LINQ possible.
C# 4.0 with .NET Framework 4 (Visual Studio 2010)	Added support for dynamic typing along with significant improvements in the API for writing multithreaded programs that capitalized on multiple processors and cores within those processors.
C# 5.0 with .NET Framework 4.5 (Visual Studio 11) and WinRT integration	Added support for asynchronous method invocation without the explicit registration of a delegate callback. An additional change in the framework was support for interoperability with the Windows Runtime (WinRT).

Perhaps the biggest feature added to .NET 4.5 was support for calling into components within the Windows Runtime (WinRT). Internally, WinRT is native code that provides an entirely new platform in which .NET can execute. However, in both the style of its API and the behavior of its runtime, it resembles .NET. For practical purposes, components in WinRT are a new set of APIs—with significant functionality overlap with the .NET Framework. The difference is that this new API is only available on Windows 8 and it has been designed from the ground up to provide a .NET-like programming experience for functionality that was previously only available in "Win32"

APIs with .NET wrappers. Although significant, the addition of WinRT as a platform is mostly orthogonal to the learning of .NET and C#. Therefore, this book does not provide in-depth coverage of WinRT.

Common Intermediate Language and ILDASM

As mentioned in the preceding section, the C# compiler converts C# code to CIL code and not to machine code. The processor can directly understand machine code, but CIL code needs to be converted before the processor can execute it. Given an assembly (either a DLL or an executable), it is possible to view the CIL code using a CIL disassembler utility to deconstruct the assembly into its CIL representation. (The CIL disassembler is commonly referred to by its Microsoft-specific filename, `ILDASM`, which stands for IL Disassembler.) This program will disassemble a program or its class libraries, displaying the CIL generated by the C# compiler.

The exact command used for the CIL disassembler depends on which implementation of the CLI is used. You can execute the .NET CIL disassembler from the command line as shown in Output 1.8.

Output 1.8

```
>ildasm /text HelloWorld.exe
```

The `/text` portion is used so that the output appears on the command console rather than in a new window. The stream of output that results by executing these commands is a dump of CIL code included in the `HelloWorld.exe` program. Note that CIL code is significantly easier to understand than machine code. For many developers, this may raise a concern because it is easier for programs to be decompiled and algorithms understood without explicitly redistributing the source code.

As with any program, CLI-based or not, the only foolproof way of preventing disassembly is to disallow access to the compiled program altogether (for example, only hosting a program on a web site instead of distributing it out to a user's machine). However, if decreased accessibility to the source code is all that is required, there are several obfuscators available. These obfuscators open up the IL code and munge the code so that it does the same thing but in a way that is much more difficult to understand.

This prevents the casual developer from accessing the code and instead creates assemblies that are much more difficult and tedious to decompile into comprehensible code. Unless a program requires a high degree of algorithm security, these obfuscators are generally sufficient.

ADVANCED TOPIC

CIL Output for `HelloWorld.exe`

Listing 1.18 shows the CIL code created by `ILDASM`.

LISTING 1.18: Sample CIL Output

```
//  Microsoft (R) .NET Framework IL Disassembler.  Version 4.0.30319.17369
//  Copyright (c) Microsoft Corporation.  All rights reserved.



// Metadata version: v4.0.30319
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         //
➥ .z\V.4..
  .ver 4:0:0:0
}
.assembly HelloWorld
{
  .custom instance void [mscorlib]System.Runtime.CompilerServices.Compilat
➥ ionRelaxationsAttribute::.ctor(int32) = ( 01 00 08 00 00 00 00 00 )
  .custom instance void [mscorlib]System.Runtime.CompilerServices.Runtime
➥ CompatibilityAttribute::.ctor() = ( 01 00 01 00 54 02 16 57 72 61 70 4E 6F
➥ 6E 45 78   // ....T..WrapNonEx

➥ 63 65 70 74 69 6F 6E 54 68 72 6F 77 73 01 )       // ceptionThrows.
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module HelloWorld.exe
// MVID: {D229AC10-1DEC-47A1-AA62-3BA19389E37E}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x00490000


// =============== CLASS MEMBERS DECLARATION ===================
```

```
.class private auto ansi beforefieldinit HelloWorld
       extends [mscorlib]System.Object
{
  .method private hidebysig static void  Main() cil managed
  {
    .entrypoint
    // Code size       13 (0xd)
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldstr      "Hello. My name is Inigo Montoya."
    IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_000b:  nop
    IL_000c:  ret
  } // end of method HelloWorld::Main

  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  } // end of method HelloWorld::.ctor

} // end of class HelloWorld


// =============================================================

// *********** DISASSEMBLY COMPLETE ***********************
```

The beginning of the listing is the manifest information. It includes not only the full name of the disassembled module (`HelloWorld.exe`), but also all the modules and assemblies it depends on, along with their version information.

Perhaps the most interesting thing that you can glean from such a listing is how relatively easy it is to follow what the program is doing compared to trying to read and understand machine code (assembler). In the listing, an explicit reference to `System.Console.WriteLine()` appears. There is a lot of peripheral information to the CIL code listing, but if a developer wanted to understand the inner workings of a C# module (or any CLI-based program) without having access to the original source code, it would be relatively easy unless an obfuscator is used. In fact, several free tools are available (such as Red Gate's Reflector, ILSpy, JustDecompile, dotPeek, and CodeReflect) that can decompile from CIL to C# automatically.

SUMMARY

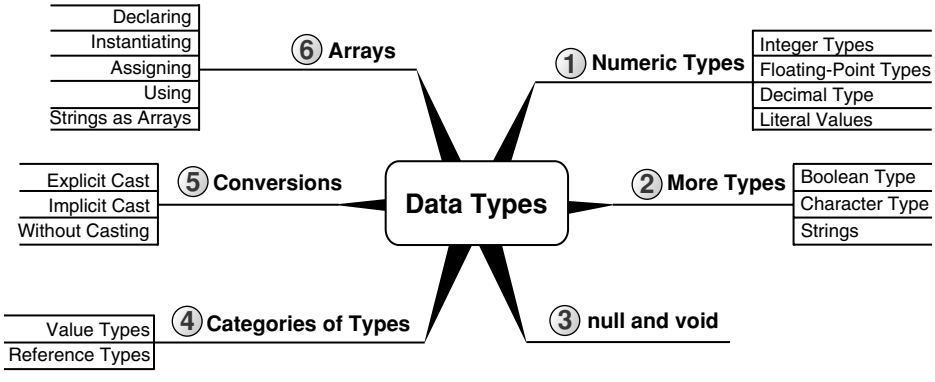
This chapter served as a rudimentary introduction to C#. It provided a means of familiarizing you with basic C# syntax. Because of C#'s similarity to C++ style languages, much of this might not have been new material to you. However, C# and managed code do have some distinct characteristics, such as compilation down to CIL. Although it is not unique, another key characteristic is that C# includes full support for object-oriented programming. Even tasks such as reading and writing data to the console are object-oriented. Object orientation is foundational to C#, and you will see this throughout this book.

The next chapter examines the fundamental data types that are part of the C# language, and discusses how you can use these data types with operands to form expressions.

This page intentionally left blank

2
Data Types

FROM CHAPTER 1'S `HelloWorld` program, you got a feel for the C# language, its structure, basic syntax characteristics, and how to write the simplest of programs. This chapter continues to discuss the C# basics by investigating the fundamental C# types.



Until now, you have worked with only a few primitive data types, with little explanation. In C# thousands of types exist, and you can combine types to create new types. A few types in C#, however, are relatively simple and are considered the building blocks of all other types. These types are **predefined types** or **primitives.** The C# language's primitive types include eight integer types, two binary floating-point types for scientific calculations and one decimal float for financial calculations, one Boolean type, and a

character type. This chapter investigates these primitives, looks more closely at the `string` type, and introduces arrays.

Fundamental Numeric Types

The basic numeric types in C# have keywords associated with them. These types include integer types, floating-point types, and a special floating-point type called `decimal` to store large numbers with no representation error.

Integer Types

There are eight C# integer types. This variety allows you to select a data type large enough to hold its intended range of values without wasting resources. Table 2.1 lists each integer type.

TABLE 2.1: Integer Types

Type	Size	Range (Inclusive)	BCL Name	Signed	Literal Suffix
`sbyte`	8 bits	–128 to 127	`System.SByte`	Yes	
`byte`	8 bits	0 to 255	`System.Byte`	No	
`short`	16 bits	–32,768 to 32,767	`System.Int16`	Yes	
`ushort`	16 bits	0 to 65,535	`System.UInt16`	No	
`int`	32 bits	–2,147,483,648 to 2,147,483,647	`System.Int32`	Yes	
`uint`	32 bits	0 to 4,294,967,295	`System.UInt32`	No	U or u
`long`	64 bits	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	`System.Int64`	Yes	L or l
`ulong`	64 bits	0 to 18,446,744,073,709,551,615	`System.UInt64`	No	UL or ul

Included in Table 2.1 (and in Tables 2.2 and 2.3) is a column for the full name of each type; we discuss the literal suffix later in the chapter. All the fundamental types in C# have a short name and a full name. The full name corresponds to the type as it is named in the Base Class Library (BCL). This name is the same across all languages and it uniquely identifies the type within an assembly. Because of the fundamental nature of primitive types C# also supplies keywords as short names or abbreviations to the full names of fundamental types. From the compiler's perspective, both names are exactly

the same, producing exactly the same code. In fact, an examination of the resultant CIL code would provide no indication of which name was used.

Although C# supports using both the full BCL name and the keyword, as developers we are left with the choice of which to use when. Rather than switching back and forth, it is better to use one or the other consistently. For this reason, C# developers generally go with using the C# keyword form—choosing, for example, `int` rather than `System.Int32` and `string` rather than `System.String` (or a possible shortcut of `String`).

Guidelines

DO use the C# keyword rather than the BCL name when specifying a data type (for example, `string` rather than `String`).
DO favor consistency rather than variety within your code.

The choice for consistency frequently may be at odds with other guidelines. For example, given the guideline to use the C# keyword in place of the BCL name, there may be occasions when you find yourself maintaining a file (or library of files) with the opposite style. In these cases it would better to stay consistent with the previous style than to inject a new style and inconsistencies in the conventions. Even so, if the "style" was in fact a bad coding practice that was likely to introduce bugs and obstruct successful maintenance, by all means correct the issue throughout.

Language Contrast: C++—`short` Data Type

In C/C++, the `short` data type is an abbreviation for `short int`. In C#, `short` on its own is the actual data type.

Floating-Point Types (`float`, `double`)

Floating-point numbers have varying degrees of precision, and binary floating-point types can only represent numbers exactly if they are a fraction with a power of two as the denominator. If you were to set the value of a floating-point variable to be 0.1, it could very easily be represented as

0.0999999999999999 or 0.10000000000000001 or some other number very close to 0.1. Similarly, setting a variable to a large number such as Avogadro's number, 6.02 x 10^{23}, could lead to a representation error of around 10^{8}, which after all is a tiny fraction of that number. The accuracy of a floating-point number is in proportion to the magnitude of the number it represents. A floating-point number is precise to a certain number of significant digits, not by a fixed value such as ±0.01.

C# supports the two binary floating-point number types listed in Table 2.2.

TABLE 2.2: Floating-Point Types

Type	Size	Range (Inclusive)	BCL Name	Significant Digits	Literal Suffix
`float`	32 bits	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	`System.Single`	7	F or f
`double`	64 bits	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	`System.Double`	15–16	无

Binary numbers appear as base 10 (denary) numbers for human readability. The number of bits (binary digits) converts to 15 decimal digits, with a remainder that contributes to a sixteenth decimal digit as expressed in Table 2.2. Specifically, numbers between 1.7 * 10^{307} and less than 1 * 10^{308} have only 15 significant digits. However, numbers ranging from 1 * 10^{308} to 1.7 * 10^{308} will have 16 significant digits. A similar range of significant digits occurs with the `decimal` type as well.

Decimal Type

C# also provides a decimal floating-point type with 128-bit precision (see Table 2.3). This type is suitable for financial calculations.

TABLE 2.3: `decimal` Type

Type	Size	Range (Inclusive)	BCL Name	Significant Digits	Literal Suffix
`decimal`	128 bits	1.0×10^{-28} to approximately 7.9×10^{28}	`System.Decimal`	28–29	M or m

Unlike binary floating-point numbers, the `decimal` type maintains exact accuracy for all denary numbers within its range. With the `decimal` type, therefore, a value of 0.1 is exactly 0.1. However, while the `decimal` type has greater precision than the floating-point types, it has a smaller range. Thus, conversions from floating-point types to the `decimal` type may result in overflow errors. Also, calculations with `decimal` are slightly (generally imperceptibly) slower.

ADVANCED TOPIC

Floating-Point Types Dissected

Denary numbers within the range and precision limits of the `decimal` type are represented exactly. In contrast, the binary floating-point representation of many denary numbers introduces a rounding error. Just as ⅓ cannot be represented exactly in any finite number of decimal digits, so too $^{11}/_{10}$ cannot be represented exactly in any finite number of binary digits. In both cases, we end up with a rounding error of some kind.

A `decimal` is represented by $\pm N * 10^k$ where the following is true.

- `N`, the mantissa, is a positive 96-bit integer.
- `k`, the exponent, is given by `-28 <= k <= 0`.

In contrast, a binary float is any number $\pm N * 2^k$ where the following is true.

- `N` is a positive 24-bit (for `float`) or 53-bit (for `double`) integer.
- `k` is an integer ranging from `-149` to `+104` for `float` and `-1075` to `+970` for `double`.

Literal Values

A **literal value** is a representation of a constant value within source code. For example, if you want to have `System.Console.WriteLine()` print out the integer value `42` and the `double` value `1.618034`, you could use the code shown in Listing 2.1.

LISTING 2.1: Specifying Literal Values

```
System.Console.WriteLine(42);
System.Console.WriteLine(1.618034);
```

Output 2.1 shows the results of Listing 2.1.

OUTPUT 2.1

```
42
1.618034
```

BEGINNER TOPIC

Use Caution When Hardcoding Values

The practice of placing a value directly into source code is called **hardcoding,** because changing the values means recompiling the code. Developers must carefully consider the choice between hardcoding values within their code and retrieving them from an external source, such as a configuration file, so that the values are modifiable without recompiling.

By default, when you specify a literal number with a decimal point, the compiler interprets it as a `double` type. Conversely, a literal value with no decimal point generally defaults to an `int`, assuming the value is not too large to be stored in an integer. If the value is too large, the compiler will interpret it as a `long`. Furthermore, the C# compiler allows assignment to a numeric type other than an `int`, assuming the literal value is appropriate for the target data type. `short s = 42` and `byte b = 77` are allowed, for example. However, this is appropriate only for constant values; `b = s` is not allowed without additional syntax, as discussed in the section Conversions between Data Types, later in this chapter.

As previously discussed in the section Fundamental Numeric Types, there are many different numeric types in C#. In Listing 2.2, a literal value is placed within C# code. Since numbers with a decimal point will default to the `double` data type, the output, shown in Output 2.2, is `1.61803398874989` (the last digit, 5, is missing), corresponding to the expected accuracy of a `double`.

LISTING 2.2: Specifying a Literal `double`

```
System.Console.WriteLine(1.618033988749895);
```

OUTPUT 2.2

```
1.6180339887498949
```

To view the intended number with its full accuracy, you must declare explicitly the literal value as a `decimal` type by appending an `M` (or `m`) (see Listing 2.3 and Output 2.3).

LISTING 2.3: Specifying a Literal `decimal`

```
System.Console.WriteLine(1.618033988749895M);
```

OUTPUT 2.3

```
1.618033988749895
```

Now the output of Listing 2.3 is as expected: `1.618033988749895`. Note that d is for `double`. To remember that `m` should be used to identify a `decimal`, remember that "m is for monetary calculations."

You can also add a suffix to a value to explicitly declare a literal as `float` or `double` by using the `F` and `D` suffixes, respectively. For integer data types, the suffixes are `U`, `L`, `LU`, and `UL`. The type of an integer literal can be determined as follows.

- Numeric literals with no suffix resolve to the first data type that can store the value in this order: `int`, `uint`, `long`, and `ulong`.
- Numeric literals with the suffix `U` resolve to the first data type that can store the value in the order `uint` and then `ulong`.
- Numeric literals with the suffix `L` resolve to the first data type that can store the value in the order `long` and then `ulong`.
- If the numeric literal has the suffix `UL` or `LU`, it is of type `ulong`.

Note that suffixes for literals are case-insensitive. However, uppercase is generally preferred to avoid any ambiguity between the lowercase letter *l* and the digit 1.

In some situations, you may wish to use exponential notation instead of writing out several zeroes before or after the decimal point. To use exponential notation, supply the `e` or `E` infix, follow the infix character with a positive or negative integer number, and complete the literal with the appropriate data type suffix. For example, you could print out Avogadro's number as a `float`, as shown in Listing 2.4 and Output 2.4.

Listing 2.4: Exponential Notation

```
System.Console.WriteLine(6.023E23F);
```

Output 2.4

```
6.023E+23
```

Guidelines

DO use uppercase literal suffixes (for example, 1.618033988749895M).

BEGINNER TOPIC

Hexadecimal Notation

Usually you work with numbers that are represented with a base of 10, meaning there are ten symbols (0–9) for each digit in the number. If a number is displayed with hexadecimal notation, it is displayed with a base of 16 numbers, meaning 16 symbols are used: 0–9, A–F (lowercase can also be used). Therefore, 0x000A corresponds to the decimal value 10 and 0x002A corresponds to the decimal value 42, being 2 x 16 + 10. The actual number is the same. Switching from hexadecimal to decimal or vice versa does not change the number itself, just the representation of the number.

Each hex digit is four bits, so a byte can represent two hex digits.

In all discussions of literal numeric values so far, we have covered only base 10 type values. C# also supports the ability to specify hexadecimal

values. To specify a hexadecimal value, prefix the value with `0x` and then use any hexadecimal digit, as shown in Listing 2.5.

Listing 2.5: Hexadecimal Literal Value

```
// Display the value 42 using a hexadecimal literal.
System.Console.WriteLine(0x002A);
```

Output 2.5 shows the results of Listing 2.5.

Output 2.5

```
42
```

Note that this code still displays `42`, not `0x002A`.

ADVANCED TOPIC

Formatting Numbers As Hexadecimal

To display a numeric value in its hexadecimal format, it is necessary to use the `x` or `X` numeric formatting specifier. The casing determines whether the hexadecimal letters appear in lowercase or uppercase. Listing 2.6 shows an example of how to do this.

Listing 2.6: Example of a Hexadecimal Format Specifier

```
// Displays "0x2A"
System.Console.WriteLine("0x{0:X}", 42);
```

Output 2.6 shows the results.

Output 2.6

```
0x2A
```

Note that the numeric literal (`42`) can be in decimal or hexadecimal form. The result will be the same.

ADVANCED TOPIC

Round-Trip Formatting

By default, `System.Console.WriteLine(1.618033988749895);` displays `1.61803398874989`, with the last digit missing. To more accurately identify the string representation of the double value it is possible to convert it using a format string and the round-trip format specifier, `R` (or `r`). `string.Format("{0:R}", 1.618033988749895)`, for example, will return the result `1.6180339887498949`.

The round-trip format specifier returns a string that, if converted back into a numeric value, will always result in the original value. Listing 2.7, therefore, will show the numbers are not equal without the round-trip format.

LISTING 2.7: Formatting Using the `R` Format Specifier

```
// ...
const double number = 1.618033988749895;
double result;
string text;

text = string.Format("{0}", number);
result = double.Parse(text);
System.Console.WriteLine("{0}: result != number",
  result != number);

text = string.Format("{0:R}", number);
result = double.Parse(text);
System.Console.WriteLine("{0}: result == number",
  result == number);
// ...
```

Output 2.7 shows the resultant output.

OUTPUT 2.7

```
True: result != number
True: result == number
```

When assigning `text` the first time, there is no round-trip format specifier and, as a result, the value returned by `double.Parse(text)` is not the same as the original `number` value. In contrast, when the round-trip format specifier is used, `double.Parse(text)` returns the original value.

For those unfamiliar with the == syntax from C-based languages, `result == number` returns `true` if `result` is equal to `number`, while `result != number` does the opposite. Both assignment and equality operators are discussed in the next chapter.

More Fundamental Types

The fundamental types discussed so far are numeric types. C# includes some additional types as well: `bool`, `char`, and `string`.

Boolean Type (`bool`)

Another C# primitive is a Boolean or conditional type, `bool`, which represents true or false in conditional statements and expressions. Allowable values are the keywords `true` and `false`. The BCL name for `bool` is `System.Boolean`. For example, in order to compare two strings in a case-insensitive manner, you call the `string.Compare()` method and pass a `bool` literal `true` (see Listing 2.8).

Listing 2.8: A Case-Insensitive Comparison of Two Strings

```
string option;
...
int comparison = string.Compare(option, "/Help", true);
```

In this case, you make a case-insensitive comparison of the contents of the variable option with the literal text /Help and assign the result to comparison.

Although theoretically a single bit could hold the value of a Boolean, the size of `bool` is 1 byte.

Character Type (`char`)

A `char` type represents 16-bit characters whose set of possible values are drawn from the Unicode character set's UTF-16 encoding. A `char` is the same size as a 16-bit unsigned integer (`ushort`), which represents values between 0 and 65,535. However, `char` is a unique type in C# and code should treat it as such.

The BCL name for `char` is `System.Char`.

BEGINNER TOPIC

The Unicode Standard

Unicode is an international standard for representing characters found in the majority of human languages. It provides computer systems with functionality for building **localized** applications, applications that display the appropriate language and culture characteristics for different cultures.

ADVANCED TOPIC

16 Bits Is Too Small for All Unicode Characters

Unfortunately, not all Unicode characters can be represented by just one 16-bit `char`. The original Unicode designers believed that 16 bits would be enough, but as more languages were supported, it was realized that this assumption was incorrect. As a result, some (rarely used) Unicode characters are composed of "surrogate pairs" of two `char` values.

To construct a literal `char`, place the character within single quotes, as in `'A'`. Allowable characters comprise the full range of keyboard characters, including letters, numbers, and special symbols.

Some characters cannot be placed directly into the source code and instead require special handling. These characters are prefixed with a backslash (\) followed by a special character code. In combination, the backslash and special character code are an **escape sequence.** For example, `\n` represents a newline, and `\t` represents a tab. Since a backslash indicates the beginning of an escape sequence, it can no longer identify a simple backslash; instead, you need to use `\\` to represent a single backslash character.

Listing 2.9 writes out one single quote because the character represented by `\'` corresponds to a single quote.

LISTING 2.9: Displaying a Single Quote Using an Escape Sequence

```
class SingleQuote
{
  static void Main()
  {
      System.Console.WriteLine('\'');
  }
}
```

In addition to showing the escape sequence, Table 2.4 includes the Unicode representation of characters.

TABLE 2.4: Escape Characters

Escape Sequence	Character Name	Unicode Encoding
`\'`	Single quote	`\u0027`
`\"`	Double quote	`\u0022`
`\\`	Backslash	`\u005C`
`\0`	Null	`\u0000`
`\a`	Alert (system beep)	`\u0007`
`\b`	Backspace	`\u0008`
`\f`	Form feed	`\u000C`
`\n`	Line feed (sometimes referred to as a newline)	`\u000A`
`\r`	Carriage return	`\u000D`
`\t`	Horizontal tab	`\u0009`
`\v`	Vertical tab	`\u000B`
`\uxxxx`	Unicode character in hex	`\u0029`
`\x[n][n][n]n`	Unicode character in hex (first three placeholders are options); variable length version of `\uxxxx`	`\u3A`
`\Uxxxxxxxx`	Unicode escape sequence for creating surrogate pairs	`\UD840DC01` (𠀁)

You can represent any character using Unicode encoding. To do so, prefix the Unicode value with `\u`. You represent Unicode characters in hexadecimal notation. The letter *A*, for example, is the hexadecimal value `0x41`. Listing 2.10 uses Unicode characters to display a smiley face (`:)`), and Output 2.8 shows the results.

LISTING 2.10: Using Unicode Encoding to Display a Smiley Face

```
System.Console.Write('\u003A');
System.Console.WriteLine('\u0029');
```

OUTPUT 2.8

```
:)
```

Strings

A finite sequence of zero or more characters is called a **string.** The string type in C# is `string`, whose BCL name is `System.String`. The string type includes some special characteristics that may be unexpected to developers familiar with other programming languages. The characteristics include a "verbatim string" prefix character, `@`, and the fact that strings are immutable.

Literals

You can enter a literal string into code by placing the text in double quotes ("), as you saw in the `HelloWorld` program. Strings are composed of characters, and because of this, character escape sequences can be embedded within a string.

In Listing 2.11, for example, two lines of text are displayed. However, instead of using `System.Console.WriteLine()`, the code listing shows `System.Console.Write()` with the newline character, `\n`. Output 2.9 shows the results.

LISTING 2.11: Using the `\n` Character to Insert a Newline

```
class DuelOfWits
{
  static void Main()
  {
      System.Console.Write(
          "\"Truly, you have a dizzying intellect.\"");
      System.Console.Write("\n\"Wait 'til I get going!\"\n");
  }
}
```

OUTPUT 2.9

```
"Truly, you have a dizzying intellect."
"Wait 'til I get going!"
```

The escape sequence for double quotes differentiates the printed double quotes from the double quotes that define the beginning and end of the string.

In C#, you can use the @ symbol in front of a string to signify that a backslash should not be interpreted as the beginning of an escape sequence. The resultant **verbatim string literal** does not reinterpret just the backslash character. Whitespace is also taken verbatim when using the @ string syntax. The triangle in Listing 2.12, for example, appears in the console exactly as typed, including the backslashes, newlines, and indentation. Output 2.10 shows the results.

LISTING 2.12: Displaying a Triangle Using a Verbatim String Literal

```
class Triangle
{
  static void Main()
  {
      System.Console.Write(@"begin
             /\
            /  \
           /    \
          /      \
         /________\
end");
  }
}
```

OUTPUT 2.10

```
begin
             /\
            /  \
           /    \
          /      \
         /________\
end
```

Without the @ character, this code would not even compile. In fact, even if you changed the shape to a square, eliminating the backslashes, the code still would not compile because a newline cannot be placed directly within a string that is not prefaced with the @ symbol.

The only escape sequence the verbatim string does support is "", which signifies double quotes and does not terminate the string.

Language Contrast: C++—String Concatenation at Compile Time

Unlike C++, C# does not automatically concatenate literal strings. You cannot, for example, specify a string literal as follows:

```
"Major Strasser has been shot. "
"Round up the usual suspects."
```

Rather, concatenation requires the use of the addition operator. (If the compiler can calculate the result at compile time, however, the resultant CIL code will be a single string.)

If the same literal string appears within an assembly multiple times, the compiler will define the string only once within the assembly and all variables will refer to the same string. That way, if the same string literal containing thousands of characters was placed multiple times into the code, the resultant assembly would reflect the size of only one of them.

String Methods

The string type, like the `System.Console` type, includes several methods. There are methods, for example, for formatting, concatenating, and comparing strings.

The `Format()` method in Table 2.5 behaves exactly like the `Console.Write()` and `Console.WriteLine()` methods, except that instead of displaying the result in the console window, `string.Format()` returns the result to the caller.

All of the methods in Table 2.5 are **static.** This means that, to call the method, it is necessary to prefix the method name (for example, `Concat`) with the type that contains the method (for example, `string`). As illustrated below, however, some of the methods in the string class are **instance** methods. Instead of prefixing the method with the type, instance methods use the variable name (or some other reference to an instance). Table 2.6 shows a few of these methods, along with an example.

TABLE 2.5: `string` Static Methods

Statement	Example

`static string string.Format(`
`  string format,`
`  ...)`
	`string text, firstName, lastName;`
	`//...`
	`text = string.Format("Your full name is {0} {1}.",`
	`  firstName, lastName);`
	`// Display`
	`// "Your full name is <firstName> <lastName>."`
	`System.Console.WriteLine(text);`

`static string string.Concat(`
`  string str0,`
`  string str1)`
	`string text, firstName, lastName;`
	`//...`
	`text = string.Concat(firstName, lastName);`
	`// Display "<firstName><lastName>", notice`
	`// that there is no space between names.`
	`System.Console.WriteLine(text);`

`static int string.Compare(`
`  string str0,`
`  string str1)`
	`string option;`
	`//...`
	`// String comparison in which case matters.`
	`int result = string.Compare(option, "/help");`
	
	`// Display:`
	`//  0 if equal`
	`//  negative if option < /help`
	`//  positive if option > /help`
	`System.Console.WriteLine(result);`

	`string option;`
	`//...`
	`// Case-insensitive string comparison`
	`int result = string.Compare(`
	`  option, "/Help", true);`
	
	`// Display:`
	`//  0 if equal`
	`//  < 0 if option < /help`
	`//  > 0 if option > /help`
	`System.Console.WriteLine(result);`

TABLE 2.6: `string` Methods

Statement	Example

`bool StartsWith(`
`  string value)`
`bool EndsWith(`
`  string value)`
	`string lastName`
	`//...`
	`bool isPhd = lastName.EndsWith("Ph.D.");`
	`bool isDr = lastName.StartsWith("Dr.");`

`string ToLower()`
`string ToUpper()`
	`string severity = "warning";`
	`// Display the severity in uppercase`
	`System.Console.WriteLine(severity.ToUpper());`

`string Trim()`
`string Trim(...)`
`string TrimEnd()`
`string TrimStart()`
	`// Remove any whitespace at the start or end.`
	`username = username.Trim();`

`string Replace(`
`  string oldValue,`
`  string newValue)`
	`string filename;`
	`//...`
	`// Remove ?'s from the string`
	`filename = filename.Replace("?", "");;`

New Line

When writing out a new line, the exact characters for the new line will depend on the operating system on which you are executing. On Microsoft Windows platforms, the new line is the combination of both the `\r` and `\n` characters, while a single `\n` is used on Unix. One way to overcome the discrepancy between platforms is simply to use `System.Console.WriteLine()` in order to output a blank line. Another approach, virtually essential when you are not outputting to the console yet still require execution on multiple platforms, is to use `System.Environment.NewLine`. In other words, `System.Console.WriteLine("Hello World")` and `System.Console.Write("Hello World" + System.Environment.NewLine)` are equivalent.

ADVANCED TOPIC

C# Properties

Technically, the `Length` member referred to in the following section is not actually a method, as indicated by the fact that there are no parentheses following its call. `Length` is a property of `string`, and C# syntax allows access to a property as though it were a member variable (known in C#

as a **field**). In other words, a property has the behavior of special methods called setters and getters, but the syntax for accessing that behavior is that of a field.

Examining the underlying CIL implementation of a property reveals that it compiles into two methods: `set_<PropertyName>` and `get_<PropertyName>`. Neither of these, however, is directly accessible from C# code, except through the C# property constructs. See Chapter 5 for more detail on properties.

String Length

To determine the length of a string you use a string member called `Length`. This particular member is called a **read-only property.** As such, it can't be set, nor does calling it require any parameters. Listing 2.13 demonstrates how to use the `Length` property, and Output 2.11 shows the results.

LISTING 2.13: Using `string`'s `Length` Member

```
class PalindromeLength
{
  static void Main()
  {
      string palindrome;

      System.Console.Write("Enter a palindrome: ");
      palindrome = System.Console.ReadLine();

      System.Console.WriteLine(
          "The palindrome, \"{0}\" is {1} characters.",
          palindrome, palindrome.Length);
  }
}
```

OUTPUT 2.11

```
Enter a palindrome: Never odd or even
The palindrome, "Never odd or even" is 17 characters.
```

The length for a string cannot be set directly; it is calculated from the number of characters in the string. Furthermore, the length of a string cannot change because a string is **immutable.**

Strings Are Immutable

A key characteristic of the `string` type is that it is immutable. A string variable can be assigned an entirely new value but there is no facility for modifying the contents of a `string`. It is not possible, therefore, to convert a `string` to all uppercase letters. It is trivial to create a new string that is composed of an uppercase version of the old string, but the old string is not modified in the process. Consider Listing 2.14 as an example.

LISTING 2.14: Error; `string` Is Immutable

```
class Uppercase
{
  static void Main()
  {
      string text;

      System.Console.Write("Enter text: ");
      text = System.Console.ReadLine();

      // UNEXPECTED:  Does not convert text to uppercase
      text.ToUpper();

      System.Console.WriteLine(text);
  }
}
```

Output 2.12 shows the results of Listing 2.14.

OUTPUT 2.12

```
Enter text: This is a test of the emergency broadcast system.
This is a test of the emergency broadcast system.
```

At a glance, it would appear that `text.ToUpper()` should convert the characters within `text` to uppercase. However, strings are immutable and, therefore, `text.ToUpper()` will make no such modification. Instead, `text.ToUpper()` returns a new string that needs to be saved into a variable or passed to `System.Console.WriteLine()` directly. The corrected code is shown in Listing 2.15, and its output is shown in Output 2.13.

LISTING 2.15: Working with Strings

```
class Uppercase
{
```

```
    static void Main()
    {
        string text, uppercase;

        System.Console.Write("Enter text: ");
        text = System.Console.ReadLine();

        // Return a new string in uppercase
        uppercase = text.ToUpper();

        System.Console.WriteLine(uppercase);
    }
}
```

OUTPUT 2.13

```
Enter text: This is a test of the emergency broadcast system.
THIS IS A TEST OF THE EMERGENCY BROADCAST SYSTEM.
```

If the immutability of a string is ignored, mistakes similar to those shown in Listing 2.14 can occur with other string methods as well.

To actually change the value in `text`, assign the value from `ToUpper()` back into `text`, as in the following:

```
text = text.ToUpper();
```

System.Text.StringBuilder

If considerable string modification is needed, such as when constructing a long string in multiple steps, you should use the data type `System.Text.StringBuilder` rather than `string`. The `StringBuilder` type includes methods such as `Append()`, `AppendFormat()`, `Insert()`, `Remove()`, and `Replace()`, some of which also appear on `string`. The key difference, however, is that on `StringBuilder` these methods will modify the data in the `StringBuilder` itself, and will not simply return a new string.

`null` and `void`

Two additional keywords relating to types are `null` and `void`. `null` is a value which indicates that the variable does not refer to any valid object. `void` is used to indicate the absence of a type or the absence of any value altogether.

null

null can also be used as a type of string "literal." null indicates that a variable is set to nothing. Reference types, pointer types, and nullable value types can be assigned the value null. The only reference type covered so far in this book is string; Chapter 5 covers the topic of creating classes (which are reference types) in detail. For now, suffice it to say that a variable of reference type contains a reference to a location in memory that is different from that of the variable. Code that sets a variable to null explicitly assigns the reference to refer to no valid value. In fact, it is even possible to check whether a reference refers to nothing. Listing 2.16 demonstrates assigning null to a string variable.

LISTING 2.16: Assigning null to a String

```
static void Main()
{
    string faxNumber;
    // ...

    // Clear the value of faxNumber.
    faxNumber = null;

    // ...
}
```

It is important to note that assigning the value null to a reference type is distinct from not assigning it at all. In other words, a variable that has been assigned null has still been set, and a variable with no assignment has not been set and, therefore, will often cause a compile error if used prior to assignment.

Assigning the value null to a string is distinctly different from assigning an empty string, "". null indicates that the variable has no value. "" indicates that there is a value: an empty string. This type of distinction can be quite useful. For example, the programming logic could interpret a faxNumber of null to mean that the fax number is unknown, while a faxNumber value of "" could indicate that there is no fax number.

The void "Type"

Sometimes the C# syntax requires a data type to be specified but no data is actually passed. For example, if no return from a method is needed, C# allows the use of void to be specified as the data type instead. The

declaration of `Main` within the `HelloWorld` program is an example. The use of `void` as the return type indicates that the method is not returning any data and tells the compiler not to expect a value. `void` is not a data type per se, but rather an indication that there is no data being returned.

Language Contrast: C++

In both C++ and C#, `void` has two meanings: as a marker that a method does not return any data, and to represent a pointer to a storage location of unknown type. In C++ programs it is quite common to see pointer types like `void**`. C# can also represent pointers to storage locations of unknown type using the same syntax, but this usage is comparatively rare in C# and typically only encountered when writing programs that interoperate with unmanaged code libraries.

Language Contrast: Visual Basic—Returning `void` Is Like Defining a Subroutine

The Visual Basic equivalent of returning a `void` in C# is to define a subroutine (`Sub`/`End Sub`) rather than a function that returns a value.

ADVANCED TOPIC

Implicitly Typed Local Variables

C# 3.0 added a contextual keyword, `var`, for declaring an **implicitly typed local variable.** As long as the code initializes a variable at declaration time with an expression of unambiguous type, C# 3.0 and later allow for the variable data type to be implied rather than stated, as shown in Listing 2.17.

Listing 2.17: Working with Strings

```
class Uppercase
{
  static void Main()
  {
```

```
        System.Console.Write("Enter text: ");
        var text = System.Console.ReadLine();

        // Return a new string in uppercase
        var uppercase = text.ToUpper();

        System.Console.WriteLine(uppercase);
    }
}
```

This listing is different from Listing 2.15 in two ways. First, rather than using the explicit data type `string` for the declaration, Listing 2.17 uses `var`. The resultant CIL code is identical to using `string` explicitly. However, `var` indicates to the compiler that it should determine the data type from the value (`System.Console.ReadLine()`) that is assigned within the declaration.

Second, the variables `text` and `uppercase` are not declared without assignment at declaration time. To do so would result in an error at compile time. As mentioned earlier, via assignment the compiler retrieves the data type of the right-hand side expression and declares the variable accordingly, just as it would if the programmer specified the type explicitly.

Although using `var` rather than the explicit data type is allowed, consider avoiding such use when the data type is known—for example, use `string` for the declaration of `text` and `uppercase`. Not only does this make the code more understandable, but it also verifies that the data type returned by the right-hand side expression is the type expected. When using a `var` declared variable, the right-hand side data type should be obvious; if it isn't, using the `var` declaration should be avoided.

`var` support was added to the language in C# 3.0 to support anonymous types. Anonymous types are data types that are declared "on the fly" within a method, rather than through explicit class definitions, as shown in Listing 2.18. (See Chapter 14 for more details on anonymous types.)

Listing 2.18: Implicit Local Variables with Anonymous Types

```
class Program
{
  static void Main()
  {
      var patent1 =
          new { Title = "Bifocals",
          YearOfPublication = "1784" };
      var patent2 =
          new { Title = "Phonograph",
          YearOfPublication = "1877" };
```

```
        System.Console.WriteLine("{0} ({1})",
            patent1.Title, patent1.YearOfPublication);
        System.Console.WriteLine("{0} ({1})",
            patent2.Title, patent1.YearOfPublication);
    }
}
```

The corresponding output is shown in Output 2.14.

OUTPUT 2.14

```
Bifocals (1784)
Phonograph (1784)
```

Listing 2.18 demonstrates the anonymous type assignment to an implicitly typed (`var`) local variable. This type of operation provides critical functionality with C# 3.0 support for joining (associating) data types or reducing the size of a particular type down to fewer data elements.

Categories of Types

All types fall into two categories: **value types** and **reference types.** The differences between the types in each category stem from how they are copied: Value type data is always copied by value, while reference type data is always copied by reference.

Value Types

With the exception of `string`, all the predefined types in the book so far have been value types. Variables of value types contain the value directly. In other words, the variable refers to the same location in memory where the value is stored. Because of this, when a different variable is assigned the same value, a copy of the original variable's value is made to the location of the new variable. A second variable of the same value type cannot refer to the same location in memory as the first variable. So changing the value of the first variable will not affect the value in the second. Figure 2.1 demonstrates this. `number1` refers to a particular location in memory that contains the value `42`. After assigning `number1` to `number2`, both variables will contain the value `42`. However, modifying either variable's value will not affect the other.

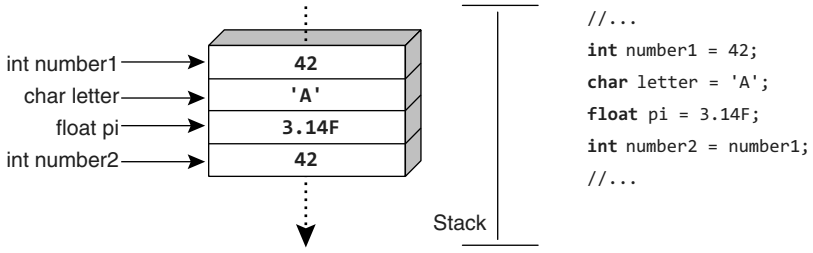


FIGURE 2.1: Value Types Contain the Data Directly

Similarly, passing a value type to a method such as `Console.WriteLine()` will also result in a memory copy, and any changes to the parameter inside the method will not affect the original value within the calling function. Since value types require a memory copy, they generally should be defined to consume a small amount of memory; value types should almost always be less than 16 bytes in size.

Reference Types

By contrast, the value of a variable of reference type is a reference to a storage location that contains data. Reference types store the reference where the data is located instead of storing the data directly, as value types do. Therefore, to access the data, the runtime will read the memory location out of the variable and then "jump" to the location in memory that contains the data. The memory area of the data a reference type points to is called the **heap** (see Figure 2.2).

A reference type does not require the same memory copy of the data that a value type does, which makes copying reference types far more efficient than copying large value types. When assigning the value of one reference type variable to another reference type variable, only the reference is copied, not the data referred to. In practice, a reference is always the same size as the "native size" of the processor: A 32-bit processor will copy a 32-bit reference and a 64-bit processor will copy a 64-bit reference, and so on. Obviously, copying the small reference to a large block of data is faster than copying the entire block, as a value type would.

Since reference types copy a reference to data, two different variables can refer to the same data. If two variables refer to the same object, changing

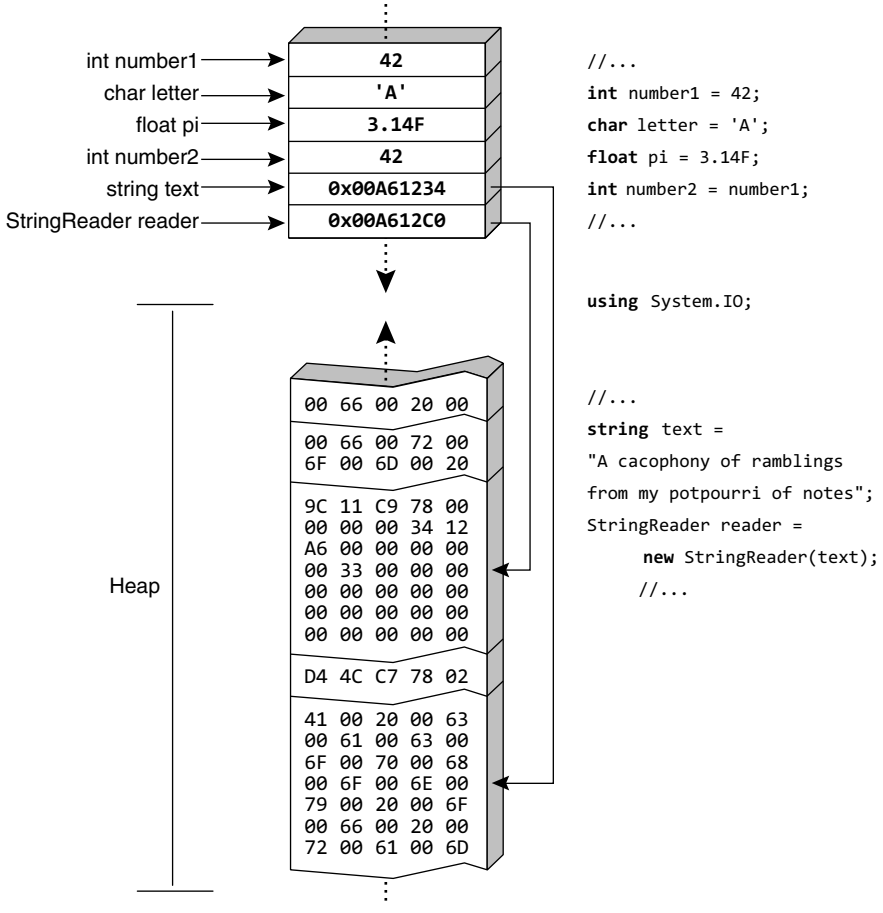


FIGURE 2.2: Reference Types Point to the Heap

a field of the object through one variable causes the effect to be seen when accessing the field via another variable. This happens both for assignment and for method calls. Therefore, a method can affect the data of a reference type, and that change can be observed when control returns to the caller. For this reason, a key factor when choosing between defining a reference type or a value type is whether the object is logically like an immutable value of fixed size (and therefore possibly a value type), or logically a mutable thing that can be referred to (and therefore likely to be a reference type).

Besides `string` and any custom classes such as `Program`, all types discussed so far are value types. However, most types are reference types.

Although it is possible to define custom value types, it is relatively rare to do so in comparison to the number of custom reference types.

Nullable Modifier

Value types cannot usually be assigned `null` because, by definition, they can't contain references, including references to nothing. However, this presents a problem because we frequently wish to represent values that are "missing." When specifying a count, for example, what do you enter if the count is unknown? One possible solution is to designate a "magic" value, such as `-1` or `int.MaxValue`, but these are valid integers. Rather, it is desirable to assign `null` to the value type because this is not a valid integer.

To declare variables of value type that can store `null` you use the nullable modifier, `?`. This feature, which was introduced with C# 2.0, appears in Listing 2.19.

Listing 2.19: Using the Nullable Modifier

```
static void Main()
{
  int? count = null;
  do
  {
      // ...
  }
  while(count == null);
}
```

Assigning `null` to value types is especially attractive in database programming. Frequently, value type columns in database tables allow `null`s. Retrieving such columns and assigning them to corresponding fields within C# code is problematic, unless the fields can contain `null` as well. Fortunately, the nullable modifier is designed to handle such a scenario specifically.

Conversions between Data Types

Given the thousands of types predefined in the various CLI implementations and the unlimited number of types that code can define, it is

important that types support conversion from one to another where it makes sense. The most common operation that results in a conversion is **casting.**

Consider the conversion between two numerical types: converting from a variable of type `long` to a variable of type `int`. A `long` type can contain values as large as 9,223,372,036,854,775,808; however, the maximum size of an `int` is 2,147,483,647. As such, that conversion could result in a loss of data—for example, if the variable of type `long` contains a value greater than the maximum size of an `int`. Any conversion that could result in a loss of magnitude or an exception because the conversion failed requires an **explicit cast.** Conversely, a conversion operation that will not lose magnitude and will not throw an exception regardless of the operand types is an **implicit conversion.**

Explicit Cast

In C#, you cast using the **cast operator.** By specifying the type you would like the variable converted to within parentheses, you acknowledge that if an explicit cast is occurring, there may be a loss of precision and data, or an exception may result. The code in Listing 2.20 converts a `long` to an `int` and explicitly tells the system to attempt the operation.

Listing 2.20: Explicit Cast Example

```
long longNumber = 50918309109;
int intNumber = (int) longNumber;
                cast operator
```

With the cast operator, the programmer essentially says to the compiler, "Trust me, I know what I am doing. I know that the value will fit into the target type." Making such a choice will cause the compiler to allow the conversion. However, with an explicit conversion, there is still a chance that an error, in the form of an exception, might occur while executing if the data does not convert successfully. It is, therefore, the programmer's responsibility to ensure the data will successfully convert, or else to provide the necessary error-handling code when it doesn't.

ADVANCED TOPIC

Checked and Unchecked Conversions

C# provides special keywords for marking a code block to indicate what should happen if the target data type is too small to contain the assigned data. By default, if the target data type cannot contain the assigned data, the data will truncate during assignment. For an example, see Listing 2.21.

LISTING 2.21: Overflowing an Integer Value

```
public class Program
{
  public static void Main()
  {
      // int.MaxValue equals 2147483647
      int n = int.MaxValue;
      n = n + 1 ;
      System.Console.WriteLine(n);
  }
}
```

Output 2.15 shows the results.

OUTPUT 2.15

```
-2147483648
```

Listing 2.21 writes the value `-2147483648` to the console. However, placing the code within a checked block, or using the `checked` option when running the compiler, will cause the runtime to throw an exception of type `System.OverflowException`. The syntax for a checked block uses the `checked` keyword, as shown in Listing 2.22.

LISTING 2.22: A Checked Block Example

```
public class Program
{
  public static void Main()
  {
      checked
      {
          // int.MaxValue equals 2147483647
          int n = int.MaxValue;
          n = n + 1 ;
          System.Console.WriteLine(n);
```

```
        }
    }
}
```

Output 2.16 shows the results.

OUTPUT 2.16

```
Unhandled Exception: System.OverflowException: Arithmetic operation
resulted in an overflow at Program.Main() in ...Program.cs:line 12
```

The result is that an exception is thrown if, within the checked block, an overflow assignment occurs at runtime.

The C# compiler provides a command-line option for changing the default checked behavior from unchecked to checked. C# also supports an unchecked block that overflows the data instead of throwing an exception for assignments within the block (see Listing 2.23).

LISTING 2.23: An Unchecked Block Example

```
using System;

public class Program
{
  public static void Main()
  {
      unchecked
      {
          // int.MaxValue equals 2147483647
          int n = int.MaxValue;
          n = n + 1 ;
          System.Console.WriteLine(n);
      }
  }
}
```

Output 2.17 shows the results.

OUTPUT 2.17

```
-2147483648
```

Even if the checked option is on during compilation, the unchecked keyword in the preceding code will prevent the runtime from throwing an exception during execution.

You cannot convert any type to any other type simply because you designate the conversion explicitly using the cast operator. The compiler will still check that the operation is valid. For example, you cannot convert a `long` to a `bool`. No such conversion is defined, and therefore, the compiler does not allow such a cast.

Language Contrast: Converting Numbers to Booleans

It may be surprising that there is no valid cast from a numeric type to a Boolean type, since this is common in many other languages. The reason no such conversion exists in C# is to avoid any ambiguity, such as whether –1 corresponds to true or false. More importantly, as you will see in the next chapter, this also reduces the chance of using the assignment operator in place of the equality operator (avoiding `if(x=42){...}` when `if(x==42){...}` was intended, for example).

Implicit Conversion

In other instances, such as going from an `int` type to a `long` type, there is no loss of precision and there will be no fundamental change in the value of the type. In these cases, code needs only to specify the assignment operator and the conversion is **implicit.** In other words, the compiler is able to determine that such a conversion will work correctly. The code in Listing 2.24 converts from an `int` to a `long` by simply using the assignment operator.

Listing 2.24: Not Using the Cast Operator for an Implicit Cast

```
int intNumber = 31416;
long longNumber = intNumber;
```

Even when no explicit cast operator is required (because an implicit conversion is allowed), it is still possible to include the cast operator (see Listing 2.25).

Listing 2.25: Using the Cast Operator for an Implicit Cast

```
int intNumber = 31416;
long longNumber = (long) intNumber;
```

Type Conversion without Casting

No conversion is defined from a string to a numeric type, so methods such as `Parse()` are required. Each numeric data type includes a `Parse()` function that enables conversion from a string to the corresponding numeric type. Listing 2.26 demonstrates this call.

Listing 2.26: Using `int.Parse()` to Convert a `string` to a Numeric Data Type

```
string text = "9.11E-31";
float kgElectronMass = float.Parse(text);
```

Another special type is available for converting one type to the next. The type is `System.Convert` and an example of its use appears in Listing 2.27.

Listing 2.27: Type Conversion Using `System.Convert`

```
string middleCText = "261.626";
double middleC = System.Convert.ToDouble(middleCText);
bool boolean = System.Convert.ToBoolean(middleC);
```

`System.Convert` supports only a predefined number of types and it is not extensible. It allows conversion from any primitive type (`bool`, `char`, `sbyte`, `short`, `int`, `long`, `ushort`, `uint`, `ulong`, `float`, `double`, `decimal`, `DateTime`, and `string`) to any other primitive type.

Furthermore, all types support a `ToString()` method that can be used to provide a string representation of a type. Listing 2.28 demonstrates how to use this method. The resultant output is shown in Output 2.18.

Listing 2.28: Using `ToString()` to Convert to a `string`

```
bool boolean = true;
string text = boolean.ToString();
// Display "True"
System.Console.WriteLine(text);
```

Output 2.18

```
True
```

For the majority of types, the `ToString()` method will return the name of the data type rather than a string representation of the data. The string representation is returned only if the type has an explicit implementation of