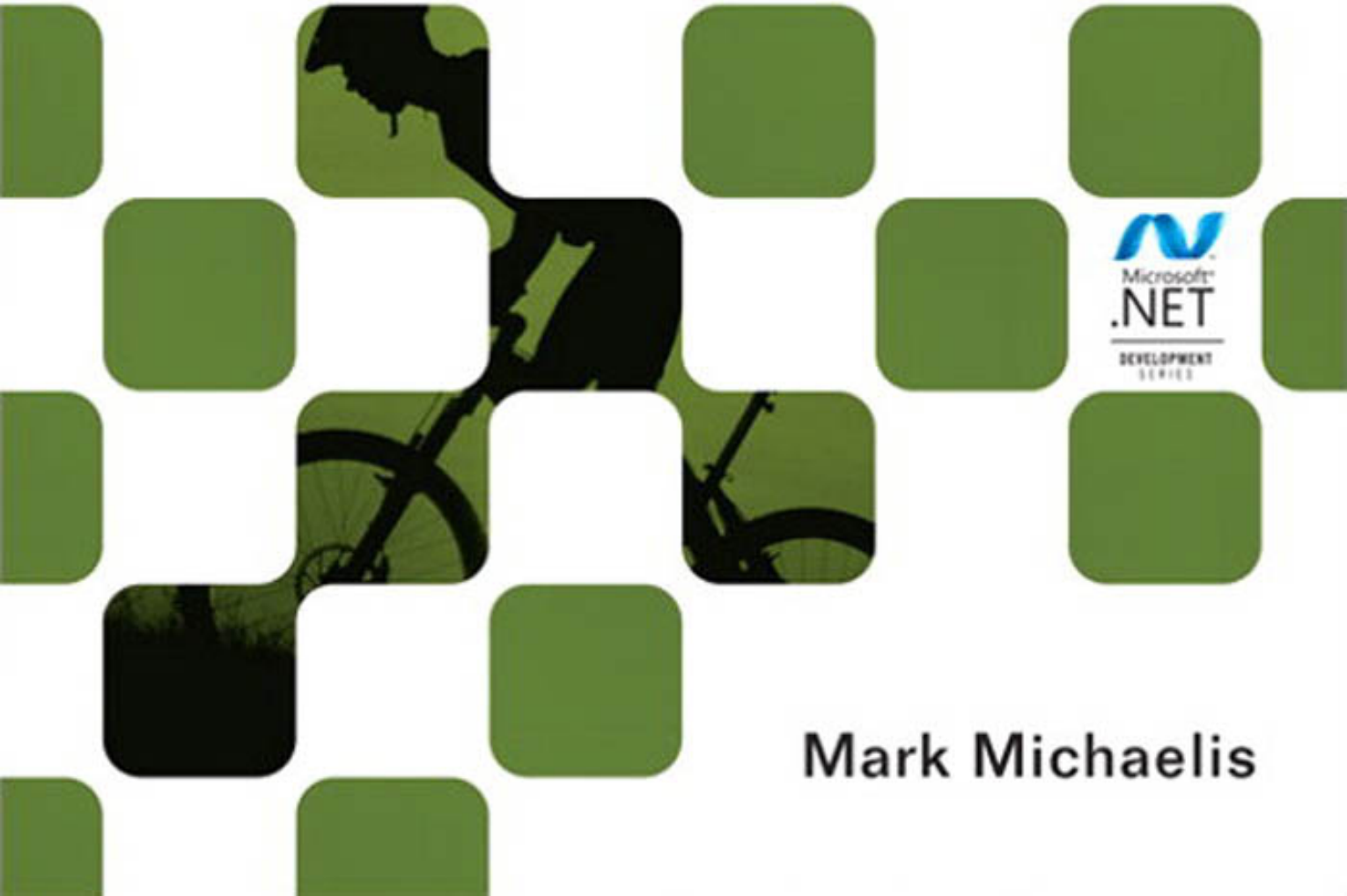
"If you want to be a C# developer, or if you want to enhance your C# programming skills, there is no more useful tool than a well-crafted book on the subject. You are holding such a book in your hands."

—From the Foreword by Charlie Calvert, Community Program Manager, Visual C#, Microsoft

Essential C# 4.0



Microsoft .NET DEVELOPMENT SERIES

Mark Michaelis

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The .NET logo is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries and is used under license from Microsoft.

Microsoft, Windows, Visual Basic, Visual C#, and Visual C++ are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries/regions.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Michaelis, Mark.
Essential C# 4.0 / Mark Michaelis.
p. cm.
Includes index.
ISBN 978-0-321-69469-0 (pbk. : alk. paper)
1. C# (Computer program language) I. Title.
QA76.73.C154M5237 2010
005.13'3—dc22
2009052592

Copyright © 2010 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax: (617) 671-3447

ISBN-13: 978-0-321-69469-0
ISBN-10: 0-321-69469-4
Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.
First printing, March 2010

Foreword

MARK MICHAELIS'S OVERVIEW OF THE C# language has become a standard reference for developers. In this, its third edition, programmers will find a thoughtful, well-written guide to the intricacies of one of the world's most popular computer languages. Having laid a strong foundation in the earlier editions of this book, Mark adds new chapters that explain the latest features in both C# and the .NET Framework.

Two of the most important additions to the book cover the latest tools for parallel programming and the new dynamic features found in C# 4.0. The addition of dynamic features to the C# language will give developers access to late-bound languages such as Python and Ruby. Improved support for COM Interop will allow developers to access Microsoft Office with an intuitive and easy-to-use syntax that makes these great tools easy to use. Mark's coverage of these important topics, along with his explanation of the latest developments in concurrent development, make this an essential read for C# developers who want to hone their skills and master the best and most vital parts of the C# language.

As the community PM for the C# team, I work to stay attuned to the needs of our community. Again and again I hear the same message: "There is so much information coming out of Microsoft that I can't keep up. I need access to materials that explain the technology, and I need them presented in a way that I can understand." Mark Michaelis is a one-man solution to a C# developer's search for knowledge about Microsoft's most recent technologies.

I first met Mark at a breakfast held in Redmond, Washington, on a clear, sunny morning in the summer of 2006. It was an early breakfast, and I like to sleep in late. But I was told Mark was an active community member, and so I woke up early to meet him. I'm glad I did. The distinct impression he made on me that morning has remained unchanged over the years.

Mark is a tall, athletic man originally from South Africa, who speaks in a clear, firm, steady voice with a slight accent that most Americans would probably find unidentifiable. He competes in Ironman triathlons and has the lean, active look that one associates with that sport. Cheerful and optimistic, he nevertheless has a businesslike air about him; one has the sense that he is always trying to find the best way to fit too many activities into a limited time frame.

Mark makes frequent trips to the Microsoft campus to participate in reviews of upcoming technology or to consult on a team's plans for the future. Flying in from his home in Spokane, Washington, Mark has clearly defined agendas. He knows why he is on the campus, gives his all to the work, and looks forward to heading back home to his family in Spokane. Sometimes he finds time to fit in a quick meeting with me, and I always enjoy them. He is cheerful and energetic, and nearly always has something provocative to say about some new technology or program being developed by Microsoft.

This brief portrait of Mark tells you a good deal about what you can expect from this book. It is a focused book with a clear agenda written in a cheerful, no-nonsense manner. Mark works hard to discover the core parts of the language that need to be explained and then he writes about them in the same way that he speaks: with a lucid, muscular prose that is easy to understand and totally devoid of condescension. Mark knows what his audience needs to hear and he enjoys teaching.

Mark knows not only the C# language, but also the English language. He knows how to craft a sentence, how to divide his thoughts into paragraphs and subsections, and how to introduce and summarize a topic. He consistently finds clear, easy-to-understand ways to explain complex subjects.

I read the first edition of Mark's book cover to cover in just a few evenings of concentrated reading. Like the current volume, it is a delight to

read. Mark selects his topics with care, and explains them in the simplest possible terms. He knows what needs to be included, and what can be left out. If he wants to explore an advanced topic, he clearly sets it apart from the rest of the text. He never shows off by first parading his intellect at the expense of our desire to understand.

A centrally important part of this new edition of the book continues to be its coverage of LINQ. For many developers the declarative style of programming used by LINQ is a new technology that requires developing new habits and new ways of thinking.

C# 3.0 contained several new features that enable LINQ. A main goal of the book is to lay out these features in detail. Explaining LINQ and the technologies that enable it is no easy task, and Mark has rallied all his formidable skills as a writer and teacher to lay this technology out for the reader in clear and easy-to-understand terms.

All the key technologies that you need to know if you want to understand LINQ are carefully explained in this text. These include

- Partial methods
- Automatic properties
- Object initializers
- Collection initializers
- Anonymous types
- Implicit local variables (`var`)
- Lambdas
- Extension methods
- Expression trees
- `IEnumerable<T>` and `IQueryable<T>`
- LINQ query operators
- Query expressions

The march to an understanding of LINQ begins with Mark's explanations of important C# 2.0 technologies such as generics and delegates. He then walks you step by step through the transition from delegates to lambdas. He explains why lambdas are part of C# 3.0 and the key role they play

in LINQ. He also explains extension methods, and the role they play in implementation of the LINQ query operators.

His coverage of C# 3.0 features culminates in his detailed explanation of query expressions. He covers the key features of query expressions such as projections, filtering, ordering, grouping, and other concepts that are central to an understanding of LINQ. He winds up his chapter on query expressions by explaining how they can be converted to the LINQ query method syntax, which is actually executed by the compiler. By the time you are done reading about query expressions you will have all the knowledge you need to understand LINQ and to begin using this important technology in your own programs.

If you want to be a C# developer, or if you want to enhance your C# programming skills, there is no more useful tool than a well-crafted book on the subject. You are holding such a book in your hands. A text such as this can first teach you how the language works, and then live on as a reference that you use when you need to quickly find answers. For developers who are looking for ways to stay current on Microsoft's technologies, this book can serve as a guide through a fascinating and rapidly changing landscape. It represents the very best and latest thought on what is fast becoming the most advanced and most important contemporary programming language.

—Charlie Calvert
Community Program Manager,
Visual C#, Microsoft
January 2010

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 written in the order in which they are executed. 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.

1. 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.

2. 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# 4.0 and .NET Framework 4.
3. 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 .NET Development Series provides a wealth of writing on these topics. *Essential C# 4.0* 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 and C# 4.0, 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 (Chapter 18)
- Parallel query processing with PLINQ
- 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# 4.0 is a language book that adheres to the core C# Language 4.0 Specification. To help you understand the various C# constructs, the book 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.

Code Samples

The code snippets in most of this text (see sample listing on the next page) 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 or 4.0 compliance is called out in the C# 3.0 and C# 4.0 indexes at the end of the book.

Here is a sample code listing.

LISTING 1.17: Commenting Your Code

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

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

      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. */
      System.Console.WriteLine("Your full name is {0} {1}.",
          firstName, lastName);
      // This is the end
      // of the program listing
  }
}
```

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
```

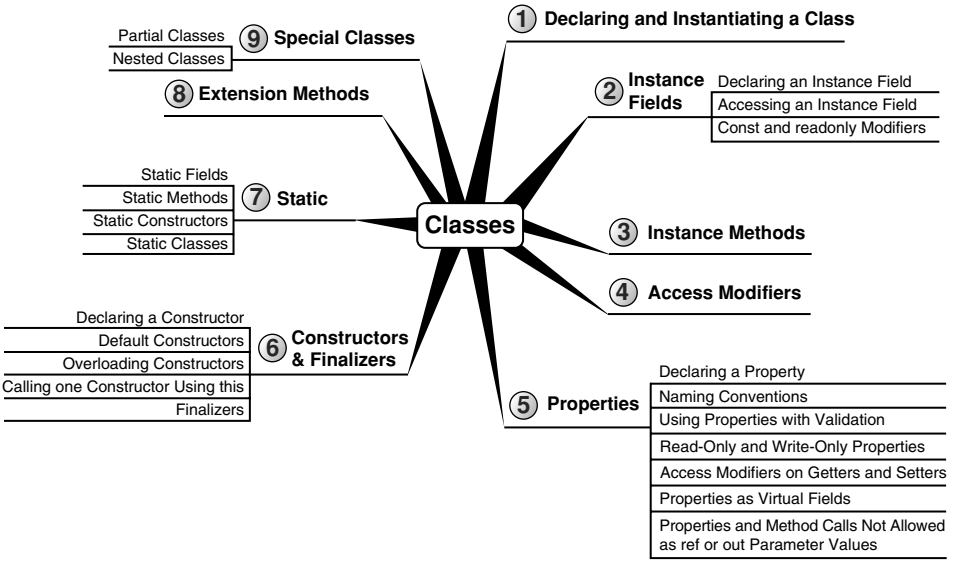
- User input for the program appears in italics.

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 and bonus mateial at intelliTechture.com/EssentialCSharp and at informit.com/msdotnetseries.

Mind Maps

Each chapter's introduction includes a **mind map**, which serves as an outline that provides an 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 and tabs 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# 4.0*. 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 or C# 4.0 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 a 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****—Synchronization and Other Multithreading Patterns:* 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 the 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—Full Source Code Listing:* In several cases, a full source code listing within a chapter would have made the chapter too long. To make

these listings still available to the reader, this appendix includes full listings from Chapters 3, 11, 12, 14, and 17.

- *Appendix C—Concurrent Classes from* `System.Collections.Concurrent`: This appendix provides overview diagrams of the concurrent collections that were added in the .NET Framework 4.
- *Appendixes D-F: C# 2.0, C# 3.0, C# 4.0 Topics:* These appendices provide a quick reference for any C# 2.0, C# 3.0, or C# 4.0 content. They are specifically designed to help programmers quickly get up to speed on C# features.

I hope you find this book to be a great resource in establishing your C# expertise and that you continue to reference it for the more obscure areas of C# and its inner workings.

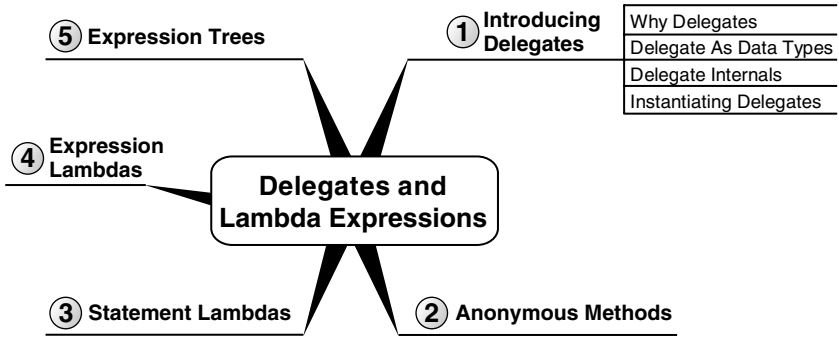
—*Mark Michaelis*
mark.michaelis.net

12

Delegates and Lambda Expressions

PREVIOUS CHAPTERS DISCUSSED extensively how to create classes using many of the built-in C# language facilities for object-oriented development. The objects instantiated from classes encapsulate data and operations on data. As you create more and more classes, you see common patterns in the relationships between these classes.

One such pattern is to pass an object that represents a method that the receiver can invoke. The use of methods as a data type and their support for publish-subscribe patterns is the focus of this chapter. Both C# 2.0 and C# 3.0 introduced additional syntax for programming in this area. Although C# 3.0 supports the previous syntax completely, in many cases C# 3.0 will deprecate the use of the older-style syntax. However, I have



placed the earlier syntax into Advanced Topic blocks, which you can largely ignore unless you require support for an earlier compiler.

Introducing Delegates

Veteran C and C++ programmers have long used method pointers as a means to pass executable steps as parameters to another method. C# achieves the same functionality using a **delegate,** which encapsulates methods as objects, enabling an indirect method call bound at runtime. Consider an example of where this is useful.

Defining the Scenario

Although not necessarily efficient, perhaps one of the simplest sort routines is a bubble sort. Listing 12.1 shows the `BubbleSort()` method.

LISTING 12.1: `BubbleSort()` Method

```
static class SimpleSort1
{
  public static void BubbleSort(int[] items)
  {
      int i;
      int j;
      int temp;

      if(items==null)
      {
        return;
      }

      for (i = items.Length - 1; i >= 0; i--)
      {
          for (j = 1; j <= i; j++)
          {
              if (items[j - 1] > items[j])
              {
                  temp = items[j - 1];
                  items[j - 1] = items[j];
                  items[j] = temp;
              }
          }
      }
  }
  // ...
}
```

This method will sort an array of integers in ascending order.

However, if you wanted to support the option to sort the integers in descending order, you would have essentially two options. You could duplicate the code and replace the greater-than operator with a less-than operator. Alternatively, you could pass in an additional parameter indicating how to perform the sort, as shown in Listing 12.2.

LISTING 12.2: BubbleSort() Method, Ascending or Descending

```
class SimpleSort2
{
  public enum SortType
  {
      Ascending,
      Descending
  }

  public static void BubbleSort(int[] items, SortType sortOrder)
  {
      int i;
      int j;
      int temp;

      if(items==null)
      {
        return;
      }

      for (i = items.Length - 1; i >= 0; i--)
      {
          for (j = 1; j <= i; j++)
          {
              switch (sortOrder)
              {
                  case SortType.Ascending :
                      if (items[j - 1] > items[j])
                        {
                          temp = items[j - 1];
                          items[j - 1] = items[j];
                          items[j] = temp;
                      }

                      break;

                  case SortType.Descending :
                      if (items[j - 1] < items[j])
                        {
                          temp = items[j - 1];
                          items[j - 1] = items[j];
```

```
                        items[j] = temp;
                    }

                    break;
                }
            }
        }
    }
    // ...
}
```

However, this handles only two of the possible sort orders. If you wanted to sort them alphabetically, randomize the collection, or order them via some other criterion, it would not take long before the number of `BubbleSort()` methods and corresponding `SortType` values would become cumbersome.

Delegate Data Types

To increase the flexibility (and reduce code duplication), you can pass in the comparison method as a parameter to the `BubbleSort()` method. Moreover, in order to pass a method as a parameter, there needs to be a data type that can represent that method—in other words, a delegate. Listing 12.3 includes a modification to the `BubbleSort()` method that takes a delegate parameter. In this case, the delegate data type is `ComparisonHandler`.

LISTING 12.3: `BubbleSort()` Method with Delegate Parameter

```
class DelegateSample
{
  // ...

  public static void BubbleSort(
      int[] items, ComparisonHandler comparisonMethod)
  {
      int i;
      int j;
      int temp;

      if(items==null)
      {
        return;
      }
      if(comparisonMethod == null)
      {
```

```
            throw new ArgumentNullException("comparisonMethod");
        }

        for (i = items.Length - 1; i >= 0; i--)
        {
            for (j = 1; j <= i; j++)
            {
                if (comparisonMethod(items[j - 1], items[j]))
                {
                  temp = items[j - 1];
                  items[j - 1] = items[j];
                  items[j] = temp;
                }
            }
        }
    }
    // ...
}
```

`ComparisonHandler` is a data type that represents a method for comparing two integers. Within the `BubbleSort()` method you then use the instance of the `ComparisonHandler`, called `comparisonMethod`, inside the conditional expression. Since `comparisonMethod` represents a method, the syntax to invoke the method is identical to calling the method directly. In this case, `comparisonMethod` takes two integer parameters and returns a Boolean value that indicates whether the first integer is greater than the second one.

Perhaps more noteworthy than the particular algorithm, the `ComparisonHandler` delegate is strongly typed to return a `bool` and to accept only two integer parameters. Just as with any other method, the call to a delegate is strongly typed, and if the data types do not match up, then the C# compiler reports an error. Let us consider how the delegate works internally.

Delegate Internals

C# defines all delegates, including `ComparisonHandler`, as derived indirectly from `System.Delegate`, as shown in Figure 12.1.[1]

The first property is of type `System.Reflection.MethodInfo`, which I cover in Chapter 17. `MethodInfo` describes the signature of a particular method, including its name, parameters, and return type. In addition to

1. The C# standard doesn't specify the delegate implementation's class hierarchy. .NET's implementation, however, does derive indirectly from `System.Delegate`.

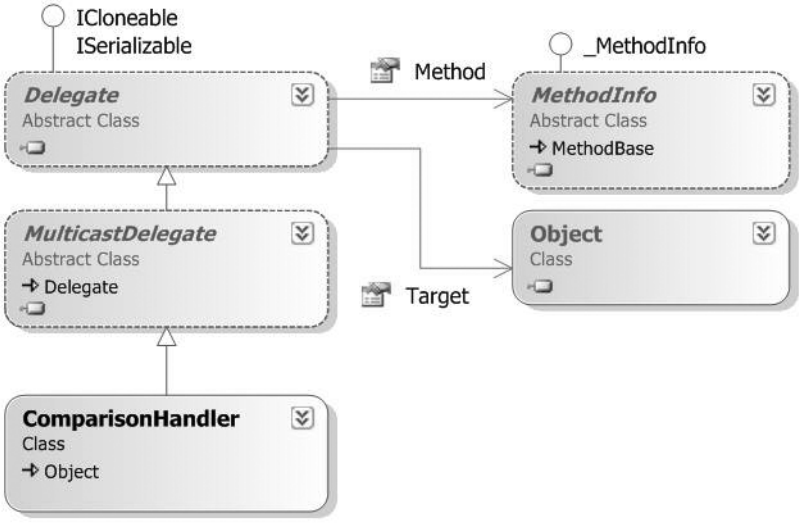


FIGURE 12.1: Delegate Types Object Model

`MethodInfo`, a delegate also needs the instance of the object containing the method to invoke. This is the purpose of the second property, `Target`. In the case of a static method, `Target` corresponds to the type itself. The purpose of the `MulticastDelegate` class is the topic of the next chapter.

It is interesting to note that all delegates are immutable. "Changing" a delegate involves instantiating a new delegate with the modification included.

Defining a Delegate Type

You saw how to define a method that uses a delegate, and you learned how to invoke a call to the delegate simply by treating the delegate variable as a method. However, you have yet to learn how to declare a delegate data type. For example, you have not learned how to define `ComparisonHandler` such that it requires two integer parameters and returns a `bool`.

Although all delegate data types derive indirectly from `System.Delegate`, the C# compiler does not allow you to define a class that derives directly or indirectly (via `System.MulticastDelegate`) from `System.Delegate`. Listing 12.4, therefore, is not valid.

Listing 12.4: System.Delegate Cannot Explicitly Be a Base Class

```
// ERROR: 'ComparisonHandler' cannot
// inherit from special class 'System.Delegate'
public class ComparisonHandler: System.Delegate
{
  // ...
}
```

In its place, C# uses the `delegate` keyword. This keyword causes the compiler to generate a class similar to the one shown in Listing 12.4. Listing 12.5 shows the syntax for declaring a delegate data type.

Listing 12.5: Declaring a Delegate Data Type

```
public delegate bool ComparisonHandler (
  int first, int second);
```

In other words, the `delegate` keyword is shorthand for declaring a reference type derived ultimately from `System.Delegate`. In fact, if the delegate declaration appeared within another class, then the delegate type, `ComparisonHandler`, would be a nested type (see Listing 12.6).

Listing 12.6: Declaring a Nested Delegate Data Type

```
class DelegateSample
{
  public delegate bool ComparisonHandler (
    int first, int second);
}
```

In this case, the data type would be `DelegateSample.ComparisonHandler` because it is defined as a nested type within `DelegateSample`.

Instantiating a Delegate

In this final step of implementing the `BubbleSort()` method with a delegate, you will learn how to call the method and pass a delegate instance—specifically, an instance of type `ComparisonHandler`. To instantiate a delegate, you need a method that corresponds to the signature of the delegate type itself. In the case of `ComparisonHandler`, that method takes two integers and returns a `bool`. The name of the method is not significant. Listing 12.7 shows the code for a greater-than method.

Listing 12.7: Declaring a `ComparisonHandler`-Compatible Method

```
public delegate bool ComparisonHandler (
  int first, int second);
```

```
class DelegateSample
{


  public static void BubbleSort(
    int[] items, ComparisonHandler comparisonMethod)
  {
      // ...
  }

  public static bool GreaterThan(int first, int second)
  {
    return first > second;
  }
  // ...
}
```

With this method defined, you can call `BubbleSort()` and pass the delegate instance that contains this method. Beginning with C# 2.0, you simply specify the name of the delegate method (see Listing 12.8).

Listing 12.8: Passing a Delegate Instance As a Parameter in C# 2.0

```
public delegate bool ComparisonHandler (
  int first, int second);
```

```
class DelegateSample
{
  public static void BubbleSort(
      int[] items, ComparisonHandler comparisonMethod)
  {
       // ...
  }

  public static bool GreaterThan(int first, int second)
  {
      return first > second;
  }

  static void Main()
  {
     int[] items = new int[100];

     Random random = new Random();
```

```
        for (int i = 0; i < items.Length; i++)
        {
            items[i] = random.Next(int.MinValue, int.MaxValue);
        }


        BubbleSort(items, GreaterThan);

        for (int i = 0; i < items.Length; i++)
        {
            Console.WriteLine(items[i]);
        }
    }

}
```

Note that the `ComparisonHandler` delegate is a reference type, but you do not necessarily use `new` to instantiate it. The facility to pass the name instead of using explicit instantiation is called **delegate inference,** a new syntax beginning with C# 2.0. With this syntax, the compiler uses the method name to look up the method signature and verify that it matches the method's parameter type.

ADVANCED TOPIC

Delegate Instantiation in C# 1.0

Earlier versions of the compiler require instantiation of the delegate demonstrated in Listing 12.9.

Listing 12.9: Passing a Delegate Instance As a Parameter Prior to C# 2.0

```
public delegate bool ComparisonHandler (
  int first, int second);
```

```
class DelegateSample
{
  public static void BubbleSort(
      int[] items, ComparisonHandler comparisonMethod)
  {
      // ...
  }

  public static bool GreaterThan(int first, int second)
  {
```

```
        return first > second;
    }

    static void Main(string[] args)
    {

        int i;
        int[] items = new int[5];

        for (i=0; i<items.Length; i++)
        {
            Console.Write("Enter an integer:");
            items[i] = int.Parse(Console.ReadLine());
        }

        BubbleSort(items,
            new ComparisonHandler(GreaterThan));

        for (i = 0; i < items.Length; i++)
        {
            Console.WriteLine(items[i]);
        }
    }

    // ...
}
```

Note that C# 2.0 and later support both syntaxes, but unless you are writing backward-compatible code, the 2.0 syntax is preferable. Therefore, throughout the remainder of the book, I will show only the C# 2.0 and later syntax. (This will cause some of the remaining code not to compile on version 1.0 compilers, unless you modify those compilers to use explicit delegate instantiation.)

The approach of passing the delegate to specify the sort order is significantly more flexible than the approach listed at the beginning of this chapter. With the delegate approach, you can change the sort order to be alphabetical simply by adding an alternative delegate to convert integers to strings as part of the comparison. Listing 12.10 shows a full listing that demonstrates alphabetical sorting, and Output 12.1 shows the results.

Listing 12.10: Using a Different `ComparisonHandler`-Compatible Method

```
using System;
class DelegateSample
{
```

```
public delegate bool ComparisonHandler(int first, int second);

public static void BubbleSort(
    int[] items, ComparisonHandler comparisonMethod)
{
    int i;
    int j;
    int temp;

    for (i = items.Length - 1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (comparisonMethod(items[j - 1], items[j]))
            {
                temp = items[j - 1];
                items[j - 1] = items[j];
                items[j] = temp;
            }
        }
    }
}

public static bool GreaterThan(int first, int second)
{
    return first > second;
}

public static bool AlphabeticalGreaterThan(
    int first, int second)
{
    int comparison;
    comparison = (first.ToString().CompareTo(
        second.ToString()));

    return comparison > 0;
}

static void Main(string[] args)
{

    int i;
    int[] items = new int[5];

    for (i=0; i<items.Length; i++)
    {
        Console.Write("Enter an integer: ");
        items[i] = int.Parse(Console.ReadLine());
    }
```

```
            BubbleSort(items, AlphabeticalGreaterThan);

        for (i = 0; i < items.Length; i++)
        {
            Console.WriteLine(items[i]);
        }
    }
}
```

OUTPUT 12.1:

```
Enter an integer: 1
Enter an integer: 12
Enter an integer: 13
Enter an integer: 5
Enter an integer: 4
1
12
13
4
5
```

The alphabetic order is different from the numeric order. Note how simple it was to add this additional sort mechanism, however, compared to the process used at the beginning of the chapter.

The only changes to create the alphabetical sort order were the addition of the `AlphabeticalGreaterThan` method and then passing that method into the call to `BubbleSort()`.

Anonymous Methods

C# 2.0 and later include a feature known as **anonymous methods**. These are delegate instances with no actual method declaration. Instead, they are defined inline in the code, as shown in Listing 12.11.

LISTING 12.11: Passing an Anonymous Method

```
class DelegateSample
{

  // ...

  static void Main(string[] args)
  {
```

```
        int i;
        int[] items = new int[5];
        ComparisonHandler comparisonMethod;

        for (i=0; i<items.Length; i++)
        {
            Console.Write("Enter an integer:");
            items[i] = int.Parse(Console.ReadLine());
        }

        comparisonMethod =
            delegate(int first, int second)
            {
                return first < second;
            };

        BubbleSort(items, comparisonMethod);

        for (i = 0; i < items.Length; i++)
        {
            Console.WriteLine(items[i]);
        }
    }
}
```

In Listing 12.11, you change the call to `BubbleSort()` to use an anonymous method that sorts `items` in descending order. Notice that no `LessThan()` method is specified. Instead, the `delegate` keyword is placed directly inline with the code. In this context, the `delegate` keyword serves as a means of specifying a type of "delegate literal," similar to how quotes specify a string literal.

You can even call the `BubbleSort()` method directly, without declaring the `comparisonMethod` variable (see Listing 12.12).

Listing 12.12: Using an Anonymous Method without Declaring a Variable

```
class DelegateSample
{

  // ...

  static void Main(string[] args)
  {

      int i;
      int[] items = new int[5];
```

```
        for (i=0; i<items.Length; i++)
        {
            Console.Write("Enter an integer:");
            items[i] = int.Parse(Console.ReadLine());
        }

        BubbleSort(items,
            delegate(int first, int second)
            {
                return first < second;
            }
        );

        for (i = 0; i < items.Length; i++)
        {
            Console.WriteLine(items[i]);
        }
    }
}
```

Note that in all cases, the parameter types and the return type must be compatible with the `ComparisonHandler` data type, the delegate type of the second parameter of `BubbleSort()`.

In summary, C# 2.0 included a new feature, anonymous methods, that provided a means to declare a method with no name and convert it into a delegate.

ADVANCED TOPIC

Parameterless Anonymous Methods

Compatibility of the method signature with the delegate data type does not exclude the possibility of no parameter list. Unlike with lambda expressions, statement lambdas, and expression lambdas (see the next section), anonymous methods are allowed to omit the parameter list (`delegate { return Console.ReadLine() != ""}`, for example). This is atypical, but it does allow the same anonymous method to appear in multiple scenarios even though the delegate type may vary. Note, however, that although the parameter list may be omitted, the return type will still need to be compatible with that of the delegate (unless an exception is thrown).

System-Defined Delegates: Func<>

.NET 3.5 (C# 3.0) included a series of generic delegates with the names "Action" and "Func." `System.Func` represents delegates that had return types while `System.Action` corresponds when no return type occurs. The signatures for these delegates are shown in Listing 12.13 (although the in/out type modifiers were not added until C# 4.0, as discussed shortly).

Listing 12.13: Func and Action Delegate Declarations

```
public delegate void Action ();
public delegate void Action<in T>(T arg)
public delegate void Action<in T1, in T2>(
    in T1 arg1, in T2 arg2)
public delegate void Action<in T1, in T2, in T3>(
    T1 arg1, T2 arg2, T3 arg3)
public delegate void Action<in T1, in T2, in T3, in T4(
    T1 arg1, T2 arg2, T3 arg3, T4 arg4)
...
public delegate void Action<
    in T1, in T2, in T3, in T4, in T5, in T6, in T7, in T8,
    in T9, in T10, in T11, in T12, in T13, in T14, in T16(
        T1 arg1, T2 arg2, T3 arg3, T4 arg4,
        T5 arg5, T6 arg6, T7 arg7, T8 arg8,
        T9 arg9, T10 arg10, T11 arg11, T12 arg12,
        T13 arg13, T14 arg14, T15 arg15, T16 arg16)

public delegate TResult Func<out TResult>();
public delegate TResult Func<in T, out TResult>(T arg)
public delegate TResult Func<in T1, in T2, out TResult>(
    in T1 arg1, in T2 arg2)
public delegate TResult Func<in T1, in T2, in T3, out TResult>(
    T1 arg1, T2 arg2, T3 arg3)
public delegate TResult Func<in T1, in T2, in T3, in T4,
    out TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)
...
public delegate TResult Func<
    in T1, in T2, in T3, in T4, in T5, in T6, in T7, in T8,
    in T9, in T10, in T11, in T12, in T13, in T14, in T16,
    out TResult>(
        T1 arg1, T2 arg2, T3 arg3, T4 arg4,
        T5 arg5, T6 arg6, T7 arg7, T8 arg8,
        T9 arg9, T10 arg10, T11 arg11, T12 arg12,
        T13 arg13, T14 arg14, T15 arg15, T16 arg16)
```

Since the delegate definitions in Listing 12.13 are generic, it is possible to use them instead of defining a custom delegate. For example, rather than declaring the `ComparisonHandler` delegate type, code could simply declare `ComparisonHandler` delegates using `Func<int, int, bool>`. The last type parameter of `Func` is always the return type of the delegate. The earlier type parameters correspond in sequence to the type of delegate parameters. In the case of `ComparisonHandler`, the return is `bool` (the last type parameter of the `Func` declaration) and the type arguments `int` and `int` correspond with the first and second parameters of `ComparisonHandler`. In many cases, the inclusion of `Func` delegates in the .NET 3.5 Framework eliminates the necessity to define delegates. You should use `System.Action`, or one of the generic versions, for delegates that have no return (`TResult`) and that take no parameters.

However, you should still declare delegate types when such a type would simplify coding with the delegate. For example, continuing to use the `ComparisonHandler` provides a more explicit indication of what the delegate is used for, whereas `Func<int, int, bool>` provides nothing more than an understanding of the method signature.

Evaluation about whether to declare a delegate is still meaningful and includes considerations such as whether the name of the delegate identifier is sufficient for indicating intent, whether the delegate type name would clarify its use, and whether the use of a .NET 3.5 type will limit the use of the assembly to .NET 3.5 clients unnecessarily.

Note that even though you can use a generic `Func` delegate in place of an explicitly defined delegate, the types are not compatible. You cannot assign one delegate type to a variable of another delegate type even if the type parameters match. For example, you cannot assign a `ComparisonHandler` variable to a `Func<int, int, bool>` variable or pass them interchangeably as parameters even though both represent signatures for a delegate that takes two `int` parameters and returns a `bool`.

However, notice the type parameter modifiers decorating the delegates in Listing 12.13. These do allow for some degree of casting between them, thanks to the variance support added in C# 4.0. Consider the following contravariant example: Because `void Action<in T>(T arg)` has the `in` type parameter decorator, it is possible to assign type `Action<string>` an object

of type `Action<object>`. In other words, any methods with a `void` return and an `object` parameter will implicitly cast to the more restrictive delegate type that only allows parameters of type `string`. Similarly with covariance and a `Func` delegate—since `TResult Func<out TResult>()` includes the `out` type parameter modifier on `TResult`, it is possible to implicitly assign a `Func<object>` variable the value of a `Func<string>`. (See Listing 12.14.)

Listing 12.14: Using Variance for Delegates

```
// Contravariance
Action<object> broadAction =
  delegate(object data)
  {
      Console.WriteLine(data);
  };
Action<string> narrowAction = broadAction;

// Contravariance
Func<string> narrowFunction =
  delegate()
  {
      return Console.ReadLine();
  };
Func<object> broadFunction = narrowFunction;

// Contravariance & Covariance Combined
Func<object, string> func1 =
  delegate(object data)
  {
      return data.ToString();
  };
Func<string, object> func2 = func1;
```

The last part of the listing combines both variance concepts into a single example, demonstrating how they can occur simultaneously if both in and out type parameters are involved.

The need for variance support within these generic delegates was a key contributing factor for why C# now includes the feature.[2]

2. The other was support for covariance to `IEnumerable<out T>`.

Lambda Expressions

Introduced in C# 3.0, **lambda expressions** are a more succinct syntax of **anonymous functions** than anonymous methods, where *anonymous functions* is a general term that includes both lambda expressions and anonymous methods. Lambda expressions are themselves broken into two types: statement lambdas and expression lambdas. Figure 12.2 shows the hierarchical relationship between the terms.

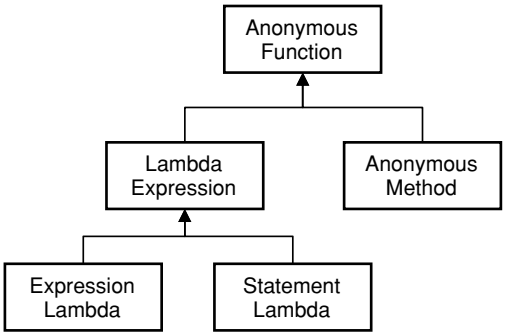


FIGURE 12.2: Anonymous Function Terminology

As mentioned earlier in the context of delegates, all anonymous functions are immutable.

Statement Lambdas

With statement lambdas, C# 3.0 provides a reduced syntax for anonymous methods, a syntax that does not include the `delegate` keyword and adds the **lambda operator,** `=>`. Listing 12.15 shows equivalent functionality to Listing 12.12, except that Listing 12.15 uses a statement lambda rather than an anonymous method.

LISTING 12.15: Passing a Delegate with a Statement Lambda

```
class DelegateSample
{

  // ...
```

```
    static void Main(string[] args)
    {

        int i;
        int[] items = new int[5];

        for (i=0; i<items.Length; i++)
        {
            Console.Write("Enter an integer:");
            items[i] = int.Parse(Console.ReadLine());
        }

        BubbleSort(items,
            (int first, int second) =>
            {
                return first < second;
            }
        );

        for (i = 0; i < items.Length; i++)
        {
            Console.WriteLine(items[i]);
        }
    }
}
```

When reading code that includes a lambda operator, you would replace the lambda operator with the words *go/goes to.* For example, you would read `n => { return n.ToString();}` as "n goes to return n dot ToString." In Listing 12.15, you would read the second `BubbleSort()` parameter as "integers `first` and `second` go to returning the result of `first` less than `second`."

As readers will observe, the syntax in Listing 12.15 is virtually identical to that in Listing 12.12, apart from the changes already outlined. However, statement lambdas allow for an additional shortcut via type parameter inference. Rather than explicitly declaring the data type of the parameters, statement lambdas can omit parameter types as long as the compiler can infer the types. In Listing 12.16, the delegate data type is `bool ComparisonHandler(int first, int second)`, so the compiler verifies that the return type is a `bool` and infers that the input parameters are both integers (`int`).

Listing 12.16: Omitting Parameter Types from Statement Lambdas

```
// ...

        BubbleSort(items,
            (first, second) =>
            {
                return first < second;
            }
        );

// ...
```

In general, statement lambdas do not need parameter types as long as the compiler can infer the types or can implicitly convert them to the requisite expected types. If the types are specified, however, there must be an exact match for the delegate type. In cases where inference is not possible, the data type is required, although even when it is not required, you can specify the data type explicitly to increase readability; once the statement lambda includes one type, all types are required.

In general, C# requires a lambda expression to have parentheses around the parameter list regardless of whether the data type is specified. Even parameterless statement lambdas, representing delegates that have no input parameters, are coded using empty parentheses (see Listing 12.17).

Listing 12.17: Parameterless Statement Lambdas

```
using System;
  // ...
  Func<string> getUserInput =
      () =>
      {
          string input;
          do
          {
              input = Console.ReadLine();
          }
          while(input.Trim().Length==0);
          return input;
      };
// ...
```

The exception to the parenthesis rule is that if the compiler can infer the data type and there is only a single input parameter, the statement lambda does not require parentheses (see Listing 12.18).

Listing 12.18: Statement Lambdas with a Single Input Parameter

```
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
  // ...
  IEnumerable<Process> processes = Process.GetProcesses().Where(
      process => { return process.WorkingSet64 > 2^30; });
  // ...
```

(In Listing 12.18, `Where()` returns a query for processes that have a physical memory utilization greater than 1GB.)

Note that back in Listing 12.17, the body of the statement lambda includes multiple statements inside the statement block (via curly braces). Although a statement lambda can have any number of statements, typically a statement lambda uses only two or three statements in its statement block.

Expression Lambdas

Unlike a statement lambda, which includes a statement block and, therefore, zero or more statements, an expression lambda has only an expression, with no statement block. Listing 12.19 is the same as Listing 12.15, except that it uses an expression lambda rather than a statement lambda.

Listing 12.19: Passing a Delegate with a Statement Lambda

```
class DelegateSample
{

  // ...

  static void Main(string[] args)
  {

      int i;
      int[] items = new int[5];

      for (i=0; i<items.Length; i++)
      {
          Console.Write("Enter an integer:");
          items[i] = int.Parse(Console.ReadLine());
      }

      BubbleSort(items, (first, second) => first < second; );
```

```
        for (i = 0; i < items.Length; i++)
        {
            Console.WriteLine(items[i]);
        }
    }
}
```

The difference between a statement and an expression lambda is that the statement lambda has a statement block on the right side of the lambda operator, whereas the expression lambda has only an expression (no return statement or curly braces, for example).

Generally, you would read a lambda operator in an expression lambda in the same way you would a statement lambda: "go/goes to." In addition, "becomes" is sometimes clearer. In cases such as the `BubbleSort()` call, where the expression lambda specified is a **predicate** (returns a Boolean), it is frequently clearer to replace the lambda operator with "such that." This changes the pronunciation of the statement lambda in Listing 12.19 to read "first and second such that first is less than second." One of the most common places for a predicate to appear is in the call to `System.Linq.Enumerable()`'s `Where()` function. In cases such as this, neither "such that" nor "goes to" is needed. We would read `names.Where(name => name.Contains(" "))` as "names where names dot Contains a space," for example. One pronunciation difference between the lambda operator in statement lambdas and in expression lambdas is that "such that" terminology applies more to expression lambdas than to statement lambdas since the latter tend to be more complex.

The anonymous function does not have any intrinsic type associated with it, although implicit conversion is possible for any delegate type as long as the parameters and return type are compatible. In other words, an anonymous method is no more a `ComparisonHandler` type than another delegate type such as `LessThanHandler`. As a result, you cannot use the `typeof()` operator (see Chapter 17) on an anonymous method, and calling `GetType()` is possible only after assigning or casting the anonymous method to a delegate variable.

Table 12.1 contains additional lambda expression characteristics.

TABLE 12.1: Lambda Expression Notes and Examples

Statement	Example

Lambda expressions themselves do not have type. In fact, there is no concept of a lambda expression in the CLR. Therefore, there are no members to call directly from a lambda expression. The `.` operator on a lambda expression will not compile, eliminating even the option of calls to `object` methods.	`// ERROR: Operator '.' cannot be applied to`
	`// operand of type 'lambda expression'`
	`Type type = ((int x) => x).ToString();`

Given that a lambda expression does not have an intrinsic type, it cannot appear to the right of an `is` operator.	`// ERROR: The first operand of an 'is' or 'as'`
	`// operator may not be a lambda expression or`
	`// anonymous method`
	`bool boolean = ((int x) => x) is Func<int, int>;`

Although there is no type on the lambda expression on its own, once assigned or cast, the lambda expression takes on a type. Therefore, it is common for developers to informally refer to the type of the lambda expression concerning type compatibility, for example.	`// ERROR: Lambda expression is not compatible with`
	`//        Func<int, bool> type.`
	`Func<int, bool> expression = ((int x) => x);`

A lambda expression cannot be assigned to an implicitly typed local variable since the compiler does not know what type to make the variable given that lambda expressions do not have type.	`// ERROR: Cannot assign lambda expression to an`
	`//        implicitly typed local variable`
	`var thing = (x => x);`

Continues

TABLE 12.1: Lambda Expression Notes and Examples *(Continued)*

Statement	Example

C# does not allow jump statements (`break`, `goto`, `continue`) inside anonymous functions if the target is outside the lambda expression. Similarly, you cannot target a jump statement from outside the lambda expression (or anonymous methods) into the lambda expression.	`// ERROR: Control cannot leave the body of an`
	`// anonymous method or lambda expression`
	`string[] args;`
	`Func<string> expression;`
	`switch(args[0])`
	`{`
	`  case "/File":`
	`  expression = () =>`
	`      {`
	`          if (!File.Exists(args[1]))`
	`              {`
	`                  break;`
	`              }`
	`              // ...`
	`          return args[1];`
	`      };`
	`      // ...`
	`}`

Variables introduced within a lambda expression are visible only within the scope of the lambda expression body.	`// ERROR:  The name 'first' does not`
	`//         exist in the current context`
	`Func<int, int, bool> expression =`
	`    (first, second) => first > second;`
	`first++;`

TABLE 12.1: Lambda Expression Notes and Examples *(Continued)*

Statement	Example

The compiler's flow analysis is unable to detect initialization of local variables in lambda expressions.	`int number;`
	`Func<string, bool> expression =`
	`  text => int.TryParse(text, out number);`
	`if (expression("1"))`
	`{`
	`  // ERROR: Use of unassigned local variable`
	`  System.Console.Write(number);`
	`}`

	`int number;`
	`Func<int, bool> isFortyTwo =`
	`  x => 42 == (number = x);`
	`if (isFortyTwo(42))`
	`{`
	`  // ERROR: Use of unassigned local variable`
	`  System.Console.Write(number);`
	`}`

ADVANCED TOPIC

Lambda Expression and Anonymous Method Internals

Lambda expressions (and anonymous methods) are not an intrinsic construct within the CLR. Rather, the C# compiler generates the implementation at compile time. Lambda expressions provide a language construct for an inline-declared delegate pattern. The C# compiler, therefore, generates the implementation code for this pattern so that the compiler automatically writes the code instead of the developer writing it manually. Given the earlier listings, therefore, the C# compiler generates CIL code that is similar to the C# code shown in Listing 12.20.

Listing 12.20: C# Equivalent of CIL Generated by the Compiler for Lambda Expressions

```
class DelegateSample
{

  // ...

  static void Main(string[] args)
  {

      int i;
      int[] items = new int[5];

      for (i=0; i<items.Length; i++)
      {
          Console.Write("Enter an integer:");
          items[i] = int.Parse(Console.ReadLine());
      }


      BubbleSort(items,
          DelegateSample.__AnonymousMethod_00000000);

      for (i = 0; i < items.Length; i++)
      {
          Console.WriteLine(items[i]);
      }

  }

  private static bool __AnonymousMethod_00000000(
      int first, int second)
  {
```

```
        return first < second;
    }

}
```

In this example, an anonymous method is converted into a separately declared static method that is then instantiated as a delegate and passed as a parameter.

Outer Variables

Local variables declared outside a lambda expression (including parameters), but **captured** (accessed) within the lambda expression, are **outer variables** of that lambda. `this` is also an outer variable. Outer variables captured by anonymous functions live on until after the anonymous function's delegate is destroyed. In Listing 12.21, it is relatively trivial to use an outer variable to count how many times `BubbleSort()` performs a comparison. Output 12.2 shows the results of this listing.

Listing 12.21: Using an Outer Variable in a Lambda Expression

```
class DelegateSample
{

  // ...

  static void Main(string[] args)
  {

      int i;
      int[] items = new int[5];
      int comparisonCount=0;

      for (i=0; i<items.Length; i++)
      {
          Console.Write("Enter an integer:");
          items[i] = int.Parse(Console.ReadLine());
      }

      BubbleSort(items,
          (int first, int second) =>
          {
              comparisonCount++;
              return first < second;
```

```
            }
        );

        for (i = 0; i < items.Length; i++)
        {
            Console.WriteLine(items[i]);
        }

        Console.WriteLine("Items were compared {0} times.",
                          comparisonCount);
    }
}
```

OUTPUT 12.2:

```
Enter an integer:5
Enter an integer:1
Enter an integer:4
Enter an integer:2
Enter an integer:3
5
4
3
2
1
Items were compared 10 times.
```

`comparisonCount` appears outside the lambda expression and is incremented inside it. After calling the `BubbleSort()` method, `comparisonCount` is printed out to the console.

As this code demonstrates, the C# compiler takes care of generating CIL code that shares `comparisonCount` between the anonymous method and the call site, even though there is no parameter to pass `comparisonCount` within the anonymous delegate, nor within the `BubbleSort()` method. Given the sharing of the variable, it will not be garbage-collected until after the delegate that references it is garbage-collected. In other words, the lifetime of the captured variable is at least as long as that of the longest-lived delegate object capturing it.

ADVANCED TOPIC

Outer Variable CIL Implementation

The CIL code generated by the C# compiler for outer variables is more complex than the code for a simple anonymous method. Listing 12.22 shows the C# equivalent of the CIL code used to implement outer variables.

Listing 12.22: C# Equivalent of CIL Code Generated by Compiler for Outer Variables

```
class DelegateSample
{

  // ...

  private sealed class __LocalsDisplayClass_00000001
  {
      public int comparisonCount;
      public bool __AnonymousMethod_00000000(
          int first, int second)
      {
              comparisonCount++;
              return first < second;
      }
  }

  ...

  static void Main(string[] args)
  {

      int i;
      LocalsDisplayClass_00000001 locals =
          new __LocalsDisplayClass_00000001();
      locals.comparisonCount=0;
      int[] items = new int[5];

      for (i=0; i<items.Length; i++)
      {
          Console.Write("Enter an integer:");
          items[i] = int.Parse(Console.ReadLine());
      }

      BubbleSort(items, locals.__AnonymousMethod_00000000);
      for (i = 0; i < items.Length; i++)
      {
          Console.WriteLine(items[i]);
      }

      Console.WriteLine("Items were compared {0} times.",
                        locals.comparisonCount);
  }
}
```

Notice that the captured local variable is never "passed" anywhere and is never "copied" anywhere. Rather, the captured local variable (`comparisonCount`) is a single variable whose lifetime we have extended by implementing it as an instance field rather than as a local variable. All references to the local variable are rewritten to be references to the field.

The generated class, `__LocalsDisplayClass`, is a **closure**—a data structure (class in C#) that contains an expression and the variables (public fields in C#) necessary to evaluate the expression. The variables (such as `comparisonCount`) enable the passing of data from one invocation of the expression to the next without changing the signature of the expression.

Expression Trees

Lambda expressions provide a succinct syntax for defining a method inline within your code. The compiler converts the code so that it is executable and callable later, potentially passing the delegate to another method. One feature for which it does not offer intrinsic support, however, is a representation of the expression as data—data that may be traversed and even serialized.

Using Lambda Expressions As Data

Consider the lambda expression in the following code:

```
persons.Where(
    person => person.Name.ToUpper() == "INIGO MONTOYA");
```

Assuming that `persons` is an array of `Persons`, the compiler compiles the lambda expression to a `Func<person, bool>` delegate type and then passes the delegate instance to the `Where()` method. Code and execution like this works very well. (The `Where()` method is an `IEnumerable` extension method from the class `System.Linq.Enumerable`, but this is irrelevant within this section.)

What if `persons` was not a `Person` array, but rather a collection of `Person` objects sitting on a remote computer, or perhaps in a database? Rather than returning all items in the `persons` collection, it would be preferable to send data describing the expression over the network and have the filtering occur remotely so that only the resultant selection returns over the network. In scenarios such as this, the data about the expression is needed, not the compiled CIL. The remote computer then compiles or interprets the expression data.

Interpreting is motivation for adding **expression trees** to the language. Lambda expressions that represent data about expressions rather than

compiled code are expression trees. Since the expression tree represents data rather than compiled code, it is possible to convert the data to an alternative format—to convert it from the expression data to SQL code (SQL is the language generally used to query data from databases) that executes on a database, for example. The expression tree received by `Where()` may be converted into a SQL query that is passed to a database, for example (see Listing 12.23).

Listing 12.23: Converting an Expression Tree to a SQL `where` Clause

```
persons.Where( person => person.Name.ToUpper() == "INIGO MONTOYA");

select * from Person where upper(Name) = 'INIGO MONTOYA';
```

Recognizing the original `Where()` call parameter as data, you can see that it is made up of the following:

- The call to the `Person` property, `Name`
- A call to a `string` method called `ToUpper()`
- A constant value, `"INIGO MONTOYA"`
- An equality operator, `==`

The `Where()` method takes this data and converts it to the SQL `where` clause by iterating over the data and building a SQL query string. However, SQL is just one example of what an expression tree may convert to.

Expression Trees Are Object Graphs

The data that an expression tree translates to is an object graph, an object graph that is represented by `System.Linq.Expressions.Expression`. Although an expression tree includes a method that will compile it into a delegate constructor call (executable CIL code), it is more likely that the expression tree (data) will be converted into a different format or set of instructions.

Any lambda expression, for example, is a type of expression that has a read-only collection of parameters, a return type, and a body—which is another expression (see Figure 12.3).

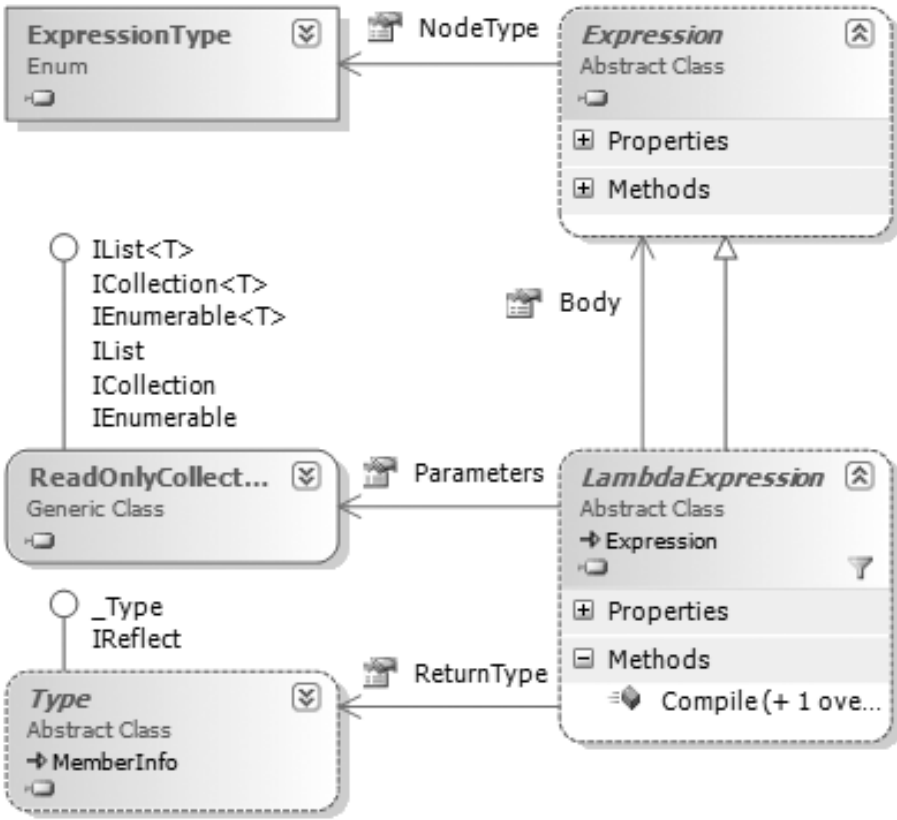


FIGURE 12.3: Object Graph of a Lambda Expression

This object graph is the data required to compile the `LambdaExpression` into CIL (or to convert some other representation). Similarly, we can create an object graph for a unary expression or binary expression (see Figure 12.4).

A unary expression (such as `count++`) is an expression composed of an `Operand` (of type `Expression`) and a `Method`—the operator. The `BinaryExpression`, which also derives from `Expression`, has two expression associations (`Left` and `Right`) in addition to the operator (`Method`). These object graphs sufficiently represent these types of expressions. However, there are another 30 or so expression types, such as `NewExpression`, `ParameterExpression`, `MethodCallExpression`, `LoopExpression`, and so forth.

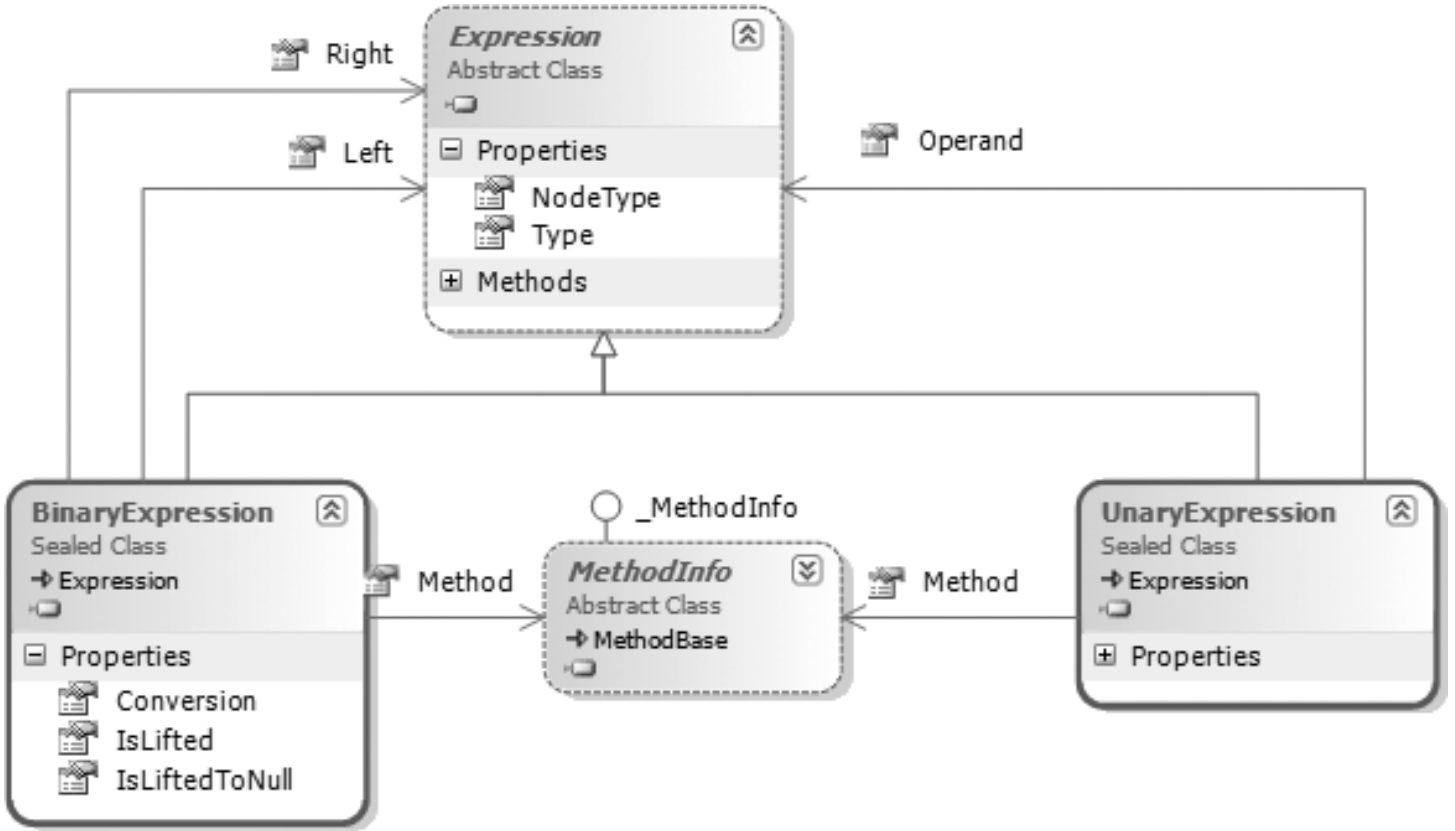


FIGURE 12.4: Object Graph of Unary and Binary Expressions

Lambda Expressions versus Expression Trees

Both a lambda expression for delegates and a lambda expression for an expression tree are compiled, and in both cases the syntax of the expression is verified at compile time with full semantic analysis. The difference, however, is that a lambda expression is compiled into a delegate in CIL. In contrast, an expression tree is compiled into a data structure of type `System.Linq.Expressions.Expression`.

Let us consider an example that highlights the difference between a delegate and an expression tree. `System.Linq.Enumerable` and `System.Linq.Queryable` are very similar. They each provide virtually identical extension methods to the collection interfaces they extend (`IEnumerable` and `IQueryable`, respectively). Consider, for example, the `Where()` method from Listing 12.23. Given a collection that supports `IEnumerable`, a call to `Where()` could be as follows:

```
persons.Where( person => person.Name.ToUpper() ==
    "INIGO MONTOYA");
```

Conceptually, the `Enumerable` extension method signature is defined on `IEnumerable<TSource>` as follows:

```
public IEnumerable<TSource> Where<TSource>(
    Func<TSource, bool> predicate);
```

However, the equivalent `Queryable` extension on the `IQueryable<TSource>` method call is identical, even though the conceptual `Where()` method signature (shown) is not:

```
public IQueryable<TSource> Where<TSource>(
    Expression<Func<TSource, bool>> predicate);
```

The calling code for the argument is identical because the lambda expression itself does not have type until it is assigned/cast.

`Enumerable`'s `Where()` implementation takes the lambda expression and converts it to a delegate that the `Where()` method's implementation calls. In contrast, when calling `Queryable`'s `Where()`, the lambda expression is converted to an expression tree so that the compiler converts the lambda expression into data. The object implementing `IQueryable` receives

the expression data and manipulates it. As suggested before, the expression tree received by `Where()` may be converted into a SQL query that is passed to a database.

Examining an Expression Tree

Capitalizing on the fact that lambda expressions don't have intrinsic type, assigning a lambda expression to a `System.Linq.Expressions.Expression<TDelegate>` creates an expression tree rather than a delegate.

In Listing 12.24, we create an expression tree for the `Func<int, int, bool>`. (Recall that `Func<int, int, bool>` is functionally equivalent to the `ComparisonHandler` delegate.) Notice that just the simple act of writing an expression to the console, `Console.WriteLine(expression)` (where `expression` is of type `Expression<TDelegate>`), will result in a call to `expression`'s `ToString()` method. However, this doesn't cause the expression to be evaluated or even to write out the fully qualified name of `Func<int, int, bool>` (as would happen if we used a delegate instance). Rather, displaying the expression writes out the data (in this case, the expression code) corresponding to the value of the expression tree.

Listing 12.24: Examining an Expression Tree

```
using System;
using System.Linq.Expressions;

class Program
{
    static void Main()
    {
        Expression<Func<int, int, bool>> expression;
        expression = (x, y) => x > y;
        Console.WriteLine("-------------{0}-------------",
            expression);
        PrintNode(expression.Body, 0);
        Console.WriteLine();
        Console.WriteLine();
        expression = (x, y) => x * y > x + y;
        Console.WriteLine("-------------{0}-------------",
            expression);
        PrintNode(expression.Body, 0);
        Console.WriteLine();
        Console.WriteLine();
    }
```

```
    public static void PrintNode(Expression expression,
        int indent)
    {
        if (expression is BinaryExpression)
            PrintNode(expression as BinaryExpression, indent);
        else
            PrintSingle(expression, indent);
    }
    private static void PrintNode(BinaryExpression expression,
      int indent)
    {
        PrintNode(expression.Left, indent + 1);
        PrintSingle(expression, indent);
        PrintNode(expression.Right, indent + 1);
    }
    private static void PrintSingle(
        Expression expression, int indent)
    {
        Console.WriteLine("{0," + indent * 5 + "}{1}",
          "", NodeToString(expression));
    }
    private static string NodeToString(Expression expression)
    {
        switch (expression.NodeType)
        {
            case ExpressionType.Multiply:
                return "*";
            case ExpressionType.Add:
                return "+";
            case ExpressionType.Divide:
                return "/";
            case ExpressionType.Subtract:
                return "-";
            case ExpressionType.GreaterThan:
                return ">";
            case ExpressionType.LessThan:
                return "<";
            default:
                return expression.ToString() +
                    " (" + expression.NodeType.ToString() + ")";
        }
    }
}
```

In Output 12.3, we see that the `Console.WriteLine()` statements within `Main()` print out the body of the expression trees as text.

OUTPUT 12.3:

```
------------- (x, y) => x > y -------------
     x (Parameter)
>
     y (Parameter)


------------- (x, y) => (x * y) > (x + y) -------------
          x (Parameter)
     *
          y (Parameter)
>
          x (Parameter)
     +
          y (Parameter)
```

The output of the expression as text is due to conversion from the underlying data of an expression tree—conversion similar to the `PrintNode()` and `NodeTypeToString()` functions, only more comprehensive. The important point to note is that an expression tree is a collection of data, and by iterating over the data, it is possible to convert the data to another format. In the `PrintNode()` method, Listing 12.24 converts the data to a horizontal text interpretation of the data. However, the interpretation could be virtually anything.

Using recursion, the `PrintNode()` function demonstrates that an expression tree is a tree of zero or more expression trees. The contained expression trees are stored in an `Expression`'s `Body` property. In addition, the expression tree includes an `ExpressionType` property called `NodeType` where `ExpressionType` is an enum for each different type of expression. There are numerous types of expressions: `BinaryExpression`, `ConditionalExpression`, `LambdaExpression` (the root of an expression tree), `MethodCallExpression`, `ParameterExpression`, and `ConstantExpression` are examples. Each type derives from `System.Linq.Expressions.Expression`.

Generally, you can use statement lambdas interchangeably with expression lambdas. However, you cannot convert statement lambdas into expression trees. You can express expression trees only by using expression lambda syntax.

SUMMARY

This chapter began with a discussion of delegates and their use as references to methods or callbacks. It introduced a powerful concept for passing a set of instructions to call in a different location, rather than immediately, when the instructions are coded.

Following on the heels of a brief look at the C# 2.0 concept of anonymous methods, the chapter introduced the C# 3.0 concept of lambda expressions, a syntax which supersedes (although doesn't eliminate) the C# 2.0 anonymous method syntax. Regardless of the syntax, these constructs allow programmers to assign a set of instructions to a variable directly, without defining an explicit method that contains the instructions. This provides significant flexibility for programming instructions dynamically within the method—a powerful concept that greatly simplifies the programming of collections through an API known as LINQ, which stands for Language Integrated Query.

Finally, the chapter ended with the concept of expression trees, and how they compile into data that represents a lambda expression, rather than the delegate implementation itself. This is a key feature that enables such libraries as LINQ to SQL and LINQ to XML, libraries that interpret the expression tree and use it within contexts other than CIL.

The term *lambda expression* encompasses both *statement lambda* and *expression lambda.* In other words, both statement lambdas and expression lambdas are types of lambda expressions.

One thing that the chapter mentioned but did not elaborate on was multicast delegates. The next chapter investigates multicast delegates in detail and explains how they enable the publish-subscribe pattern with events.

Index

16-bit characters, 41
42 as strings *versus* as integers, 187

; (semicolons)
  statements without, 10–11
  whitespace, 11–12
~ (bitwise complement) operator, 120
# (hash) symbol, 139
- (minus) operator, 84–92
+ (plus) operator, 84–92
= (simple assignment) operator, 14
_ (underscore), 15
!= (inequality) operator, 110, 370
! (logical notation) operator, 113
% (remainder) operator, 85
&& (AND) operator, 112, 373
' (single quote), 42
() (cast operator), 375–376
* (multiplication) operator, 85
+ (addition) operator, 85, 371–373
++ (increment) operator, 94–97
- - (decrement) operator, 94–97
/ (division) operator, 85
/// (three-forward-slash), 387
< (less than) operator, 110
<= (less than or equal to) operator, 110
== (equality) operator, 110, 370
> (greater than) operator, 110
>= (greater than or equal to) operator, 110
? (conditional) operator, 113–114
?? (null coalescing) operator, 114–115
@ symbol, 45
\ (backslash), 42
\n (newline) character, 42, 48
^ (exclusive OR) operator, 112
{} (code blocks), 105–107
|| (OR) operator, 111–112, 373
  constraints, 450

A

abstract classes, inheritance, 293–299
abstract members, 294
  declaring, 297
accessibility modifiers, 381
accessing
  arrays, 70
  CAS (Code Access Security), 659, 852
  class instances with `Me` keyword, 214
  instance fields, 210–211
  members, referent types, 839
  metadata, reflection, 652–662
  security code, 25
access modifiers, 220–222
  circumventing, 852
  classes, 380–381
  on getters and setters, 231–232
  `private`, 275
  `protected`, 276
acronyms, common C#, 862–863
actions, `System.Action`, 483–484
Active Template Library (ATL), 278
adding
  comments, 20–23
  items to `Dictionary<TKey, TValue>`, 623
  operators, 371–373
addition (+) operator, 85, 371–373
`Add()` method, 543
addresses, pointers and, 830–839

aggregation
  multiple inheritance, 280
  single inheritance, 279
aliasing, 164–165
  qualifiers, namespaces, 384–385
allocating data on call stacks, 836
AllowMultiple parameter, 674
ambiguity, avoiding, 213–217
AND operator (&&), 112, 373
anonymous functions, 486
anonymous methods, 480–482
  internals, 494–495
  parameterless, 482
anonymous types, 245–246
  arrays, initializing, 545–546
  collection interfaces, 536–538
  generating, 542–543
  implicit local variables, 54
  projection to, 558
  within query expressions, 593
APIs (application programming interfaces)
  encapsulation, 826–828
  VirtualAllocEx(), declaring, 818–819
  wrappers, simplifying calls with, 828–829
APMs (Asynchronous Programming Models), 783–797
  TPL (Task Parallel Library), calling, 791–796
AppDomain, unhandled exceptions on, 744–746
applicable calls, 185
applications
  domains, CLI (Common Language Infrastructure), 854–855
  HelloWorld program, 2–4, 28–30
  single instance, 766–767
  task-related finalization exceptions suppressed during shutdown, 717
applying
  arrays, 70–76
  bitwise operators, 118
  characters in arithmetic operations, 88–89
  factory inheritance, 451
  FlagsAttribute, 354–355
  generic classes, 427–429
  lambda expressions as data, 498–499
  post-increment operators, 95
  pre-increment operators, 96
  SafeHandle, 823–824
  strings, 50
  StructLayoutAttribute for sequential layout, 820–821
  System.Threading.Interlocked class, 761–763
  validation to properties, 228–229
  variables, 12–16
  variance in delegates, 485
  weak references, 391–393
ArgumentNullException, 407
arithmetic operators, 85
arrays, 64–80
  accessing, 70
  anonymous types, initializing, 545–546
  applying, 70–76
  assigning, 66–70
  common errors, 78–80
  declaring, 65–66, 70
  errors, 69
  foreach loops, 546–547
  instance methods, 75–76
  instantiation, 66–70
  length of, 72
  methods, 73–75
  parameters, 173–176
  redimensioning, 75
  strings as, 76–78
  support for covariance and contravariance in, 462–463
as operators, 302
assemblies, 3
  attributes, 665
  CLI (Common Language Infrastructure), 855–858
  metadata reflection, 652–662
  multimode, building, 856n5
  referencing, 377–381
  targets, modifying, 378–379
Assert() methods, 91
assigning
  arrays, 66–70
  indexer property names, 632–633
  null to strings, 51
  pointers, 834–837
  variables, 13, 14–16
assignment operators, 92–98
  binary operators, combining, 373
  bitwise, 120
associating
  data in classes, 250
  XML comments with programming constructs, 386–388
associativity, order of, 86
Asynchronous Delegate Invocation, 797–801
asynchronous operations with System.Threading.Thread, 738–740
Asynchronous Programming Models, Se APMs
AsyncState property, 710
ATL (Active Template Library), 278
atomicity, 704–705, 752
attributes, 663–688
  assemblies, 665

command-line, 881–888
constructors, initializing, 668–673
custom, 666–667
FlagsAttribute, 354–355, 675
limiting, 674
metadata reflection, 652–662
predefined, 676–677
return, specifying, 666
searching, 667–668
serialization, 680–682
System.ConditionalAttribute, 677–679
System.NonSerializable, 682–683
System.ObsoleteAttribute, 679–680
System.Runtime.Serialization.OptionalFieldAttribute, 686
ThreadStaticAttribute, 775–777
automatically implemented properties, 225–227
AutoResetEvent, semaphores over, 772
availability of types, 380
Average function, 585
avoiding
ambiguity, 213–217
copying, 345
deadlock, 759, 764–765
equality conditionals, 91
string types, 759–760
synchronization, 760
this type, 759–760
typeof types, 759–760
unboxing, 345
unnecessary locking, 765–766

B

BackgroundWorker class patterns, 804–809
backslash (\), 42
base classes, 204
constraints, 444–445
overriding, 281–293
refactoring, 271
Base Class Library. *See* BCL
base interfaces, using in class declarations, 320
base members, 291–292
base types, casting between derived types, 272–273
BCL (Base Class Library), 25
CLI (Common Language Infrastructure), 860
behaviors, dynamic data type, 690–693
benefits of generics, 430–431
best practices, synchronization design, 674
binary operators, 85, 371–373
BinarySearch() method, 75, 620
Binary Tree and Pair, full source code listings, 876–881
BinaryTree<T> class with no constraints, declaring, 439
binding, dynamic, 694
bits, 115
bitwise operators, 115–121
assignment, 120
complement (~) operator, 120
blocks
catch, general, 409–410
code blocks, 831
code blocks (), 105–107
System.Exception, 195–196
unchecked, 418
Boolean expressions, 109–115
Boolean types, 40–41
boxing, 339–346
break statements, 132–135
BubbleSort() method, 470–472
buffers
overflow bugs, 72
overruns, 72
building
custom collections, 611–612
multimode assemblies, 856n5
bytes, 115

C

C#
acronyms, 862–863
CLI (Common Language Infrastructure), compiling to machine code, 847–849
compilers
downloading, 865
installing, 865–867
custom collection interfaces, 612–613
delegate instantiation, 477–480
general catch blocks in, 409–410
LINQ, projection using query expressions, 592–593
overview of, 1–2
preprocessor directives, 138–145
properties, 48
syntax fundamentals, 4–12
VirtualAllocEx()APIs, declaring, 818–819
without generics, 422–439
C++
array declaration, 66
buffer overflow bugs, 72
delete operator, 208
deterministic destruction, 399, 850
dispatch method calls during construction, 286
global methods, 158
global variables and functions, 248
header files, 160
implicit overriding, 283
multiple inheritance, 278
operators, errors, 110
pointers, declaring, 833
preprocessing, 138
pure virtual functions, 297

C++ *(contd.)*
`struct` defines type with public members, 337
`switch` statements, 132
templates, 442
`var`, 540
`Variant`, 540
`void*`, 540
`void` as data types, 52
calculating
pi, 725
values, 115
callbacks, invoking, 787
caller variables, matching parameter names, 168
calling
APMs (Asynchronous Programming Models), 784–786, 791–796
applicable, 185
binary operators, 372–373
call sites, 168
constructors, 237, 243–244
external functions, 826–828
methods, 150–156
object initializers, 240
`SelectMany()` method, 580–582
stacks, 168
allocating data on, 836
exceptions, 412
wrappers, simplifying APIs with, 828–829
cancellation
cooperative, 719
parallel loops, 729–734
tasks, 718–722
`CancellationToken-Source.Token` property, 731
CAS (Code Access Security), 659, 852
case sensitivity, 2
casting
between base and derived types, 272–273
within inheritance chains, 274
inside generic methods, 456–457
cast operator (()), 58, 375–376
catch blocks
general, 409–410
`System.Exception`, 195–196
catching exceptions, 191–192, 196, 407–408, 411
categories of types, 55–57, 332–339
CD-ROM drives, 274
`Cell` type, 427
centralizing initialization, 244–245
chains, casting within inheritance, 274
characters
arithmetic operations, applying, 88–89
escape, 42, 43
newline (\n), 42, 48
Unicode, 41–43
`char` types, 41
checked conversions, 59–61, 417–419
checking
for `null`, 513–514
types, 851
child classes, 205
child collections, creating, 578
CIL (Common Intermediate Language), 23
boxing code in, 340
CLI (Common Language Infrastructure), 858
`dynamic data type`, 693
and `ILDASM`, 27–30
out variable implementation, 496–498
representation of generics, 463–464
`System.SerializableAttr ibute`, 687–688
circular wait condition, 765
`class`, iterators, 645
classes, 201–202
abstract, inheritance, 293–299
access modifiers, 220–222, 380–381
associated data, 250
`BackgroundWorker` patterns, 804–809
base, 204
constraints, 444–445
overriding, 281–293
refactoring, 271
`BinaryTree<T>`, declaring with no constraints, 439
concrete, 293
concurrent collection, 773–774
concurrent from `Systems.Collections .Concurrent`, 895–898
`ConsoleListControl`, 307
constructors, 236–247
declaring, 205–209
defining, 206
definitions, 7
deriving, 270
encapsulation, 258–260
exceptions, inheritance, 192
extension methods, 256–258
generics, 427–429, 661–662
hierarchies, 204, 473n1
inner, 262
instances
fields, 209–211
methods, 211–212
instantiating, 205–209
interfaces
compared with, 328–329
duplicating, 433–434
iterators, creating multiple in, 648–649
libraries, 377–378, 378
`LinkedList<T>`, 629
`List<T>`, 617–621
members, 209
`Monitor`, synchronization, 754–758

nested, 260–262, 265
object-oriented programming, 203–205
partial, 262–267
primary collections, 617–630
properties, 222–236
`Queue<T>`, 629
sealed, 281
`SortedDictionary<TKey, TValue>`, 626–628
`SortedList<T>`, 626–628
`Stack`, 422, 425
`Stack<T>`, 628
static, 255
static members, 247–256
`System.Threading.Interlocked`, 761–763
`System.Threading.WaitHandle`, 768–769
`this` keyword, 213–220
clauses
`into`, query continuation with, 605–606
`Let`, 600–602
query expressions, 590
`where`, converting expression trees to, 499
cleanup, resources, 790–791, 823–824
well-formed types, 393–400
`Clear()` method, 75
CLI (Common Language Infrastructure), 1, 24, 843–844
application domains, 854–855
assemblies, 855–858
BCL (Base Class Library), 860
C#, compiling to machine code, 847–849
CIL (Common Intermediate Language), 858
CLS (Common Language Specification), 859–860
CTS (Common Type System), 858–859
defining, 844–845
implementation, 845–846
manifests, 855–858
metadata, 860–861
modules, 855–858
P/Invoke, 816–830
runtime, 849–854
CLS (Common Language Specification), 24
CLI (Common Language Infrastructure), 859–860
CLU language, 635
clusters, 635
code
access security, 25
`Binary Tree` and `Pair`, 876–881
CAS (code access security), 659
CIL, boxing in, 340
command-line attributes, 881–888
comments, 20–23
conventions, events, 526–528
declaration space, 107–109
`HelloWorld` program, 2–4
invalid, indentation, 106
machine, 844, 847–849
management, 24
multithreading. *See* multithreading
paths, 159
P/Invoke, 816–830
`ProductSerialNumber`, 874–876
pseudocode, executing, 752
reusing, 378
scope, 107–109
styles, avoiding ambiguity, 213–217
`Tic-Tac-Toe`, 869–874
unsafe, 831–832
values, hardcoding, 35–37
virtual computer detection using P/Invoke, 888–894
whitespace, formatting, 11–12
Code Access Security (CAS), 659, 852
code blocks (), 105–107
collections
concurrent, 773–774
custom, building, 611–612
dictionary, 622–626
`IComparable<T>` interfaces, 614–617
`IDictionary<TKey, TValue>` interface, 614–617
`IList<T>` interface, 614–617
index operators, 630–634
initializers, 240–241, 543–546
interfaces, 612–613
anonymous types, 536–538
`IEnumerable<T>`, 546–552
implicitly typed local variables, 538–540
with standard query operators, 535–536
iterators, 634–650
linked lists, 629–630
`null`, returning, 634
primary collections classes, 617–630
queues, 629
sorting, 626–628
stacks, 628
`Collect()` method, 391
COM
controlling, 813
DLL registration, 858
combining binary operators and assignment operators, 373
command-line
arguments to `Main()` methods, passing, 166
attributes, full source code listings, 881–888
options, 76

CommandLineHandler.TryParse() method, 671
comments, 20–23
  delimited, 21
  single-line, 22
  XML, 385–389
common errors, arrays, 78–80
Common Intermediate Language. *See* CIL
Common Language Infrastructure. *See* CLI
Common Language Specification. *See* CLS
CompareTo() method, 442
ComparisonHandler-Compatible method, 478–479
compatibility, types between enums, 349–350
compilers
  C#
    downloading, 865
    installing, 865–867
  extracting XML data, 385n2
compiling
  case sensitivity, 2
  C# to machine code, 847–849
  HelloWorld program, 3–4
  JIT (just-in-time) compilers, 848
  LINQ query expressions, 607
  static compilation *versus* dynamic programming, 695–696
  string concatenation, 45
computers, virtual, 816
concatenation of strings compile time, 45
Concat() standard query operator, 584
concrete classes, 293
concurrent classes from Systems.Collections.Concurrent, 895–898
concurrent collection classes, 773–774
conditional (?) operator, 113–114
conditionals, 109. *See also* Boolean expressions
conditions, removing, 765
connecting
  publishers, 511–512
  subscribers, 511–512
console executable, 378
ConsoleListControl class, 307
consoles, input and output, 16–20
ConsoleSyncObject, 797
constants
  expressions, 98
  mathematics, 107
const fields, 258–259
constraints
  base classes, 444–445
  constructors, 446–447, 451
  generics, 439–457
  inheritance, 447–448, 450
  interfaces, 442–444
  limitations, 449–452
  multiple, 446
  struct/class, 445
constructors
  attributes, initializing, 668–673
  calling, 237, 243–244
  classes, 236–247
  constraints, 446–447, 451
  declaring, 237–238
  default, 239
  defining, 434–435
  inheritance, 292–293
  overloading, 241–242
  static, 253–254
constructs
  metadata reflection, 652–662
  programming, associating XML comments with, 386–388
contextual keywords, 6–7
Continuation Passing Style. *See* CPS
continue statements, 135–136
ContinueWith() method, 711–715, 717, 795–796
contravariance, generics, 457–463
control flow, 83–84
  statements, 121–132
controlling
  COM, 813
  threads, 706–738
conventions
  code, events, 526–528
  naming. *See* naming conventions
conversion
  as operators, 302
  checked, 59–61, 417–419
  C# to CIL, 847
  customizing, 274
  between data types, 58–64
  between enums and strings, 348, 350–351
  expression trees to SQL where clauses, 499
  generics to type parameters, 457
  implicit, 62, 273
  interfaces between implementing classes and, 318
  numbers to Booleans, 61
  numeric conversion with TryParse() method, 198–199
  operators, 375
    guidelines for, 377
    implementation, 376
  strings, 63
  unchecked, 59–61, 417–419
cooperative cancellation, 719
copying, avoiding, 345
Copy() method, 257
CopyTo() method, 617
CountdownEvent, 772
Count() function, 585
counting elements with Count() method, 561
Count property, 617
covariance, 438
  generics, 457–463

IEnumewrable<out T>, 485n2
C pointers, declaring, 833
CPS (Continuation Passing Style), 787–789
CTS (Common Type System), 858–859
Current Programming with Windows, 801n1
custom attributes, 666–667
custom collections
  building, 611–612
  IComparable<T> interfaces, 614–617
  IDictionary<TKey, TValue> interface, 614–617
  IList<T> interface, 614–617
  index operators, 630–634
  interfaces, 612–613
  iterators, 634–650
  linked lists, 629–630
  null, returning, 634
  primary collections classes, 617–630
  queues, 629
  sorting, 626–628
  stacks, 628
custom dynamic object implementation, 696–699
customizing
  conversions, defining, 274
  event implementation, 532–533
  exceptions, defining, 414–419
  LINQ, 585
  serialization, 683–684

D

data
  allocating on call stacks, 836
  to and from an alternate thread, passing, 799–801
  fixing, 835
  persistence, 217
  retrieval from files, 218
DataStore() method, 545
data types, 13–14, 31–32, 40–57
  arrays, 64–80
  categories of, 55–57
  conversions between, 58–64
  delegates, 472–473
  dynamic, principles and behaviors, 690–693
  fundamental numeric types, 32–40
  nullable modifiers, 57–58
  null keyword, 51–52
  parameters, 818–819
  short, 33
  strings, 43–51
  System.Text.StringBuilder, 51
  void keyword, 52–55
deadlock, 705–706, 760
  avoiding, 759, 764–765
decimal types, 34–35
declaration space, 107–109
declaring
  abstract members, 297
  arrays, 65–66, 70
  BinaryTree<T> class with no constraints, 439
  classes, 8, 205–209
  constant fields, 258
  constructors, 237–238
  delegates
    data types, 475
    with method returns, 522
  events, 525–526
  external functions, 817
  fields as volatile, 760–761
  finalizers, 393
  generics
    classes, 430
    delegate types, 529
    interfaces, 432
    multiple type parameters, 436
  instance fields, 209–210
  interfaces, constraints, 443–444
  jagged arrays, 71
  Main() method, 9–10
  methods, 157–161, 159–160
  parameters, 159
  pointers, 832–834
  properties, 223–225
  static constructors, 253–254
  static properties, 254
  two-dimensional arrays, 68
  Type alias, 164
  variables, 13, 14
    applying anonymous methods, 481
    of the Class Type, 206
  VirtualAllocEx() APIs, 818–819
  Win32 APIs, 818n1
decorating properties, 663, 664
decrement (- -) operator, 94–97
default constructors, 239
default() operators, 68, 338, 435
default values, specifying, 435–436
deferred execution
  with LINQ query expressions, 593–598
  standard query operators, 562–566
defining
  abstract classes, 294
  abstract members, 295
  cast operators, 275, 375
  classes, 7, 206
  CLI (Common Language Infrastructure), 844–845
  constructors, 434–435
  custom conversions, 274
  custom exceptions, 414–419
  delegates, types, 474–475
  enums, 347
  finalizers, 393–395, 434–435
  generic methods, 453
  index operators, 631–632
  inheritance, 269–270
  interfaces, 307
  iterators, 636

defining *(contd.)*
namespaces, well-formed types, 382–385
nested classes, 260, 265
objects, 206
preprocessor symbols, 141
properties, 224
publishers, events, 510–511
simple generic classes, 429–430
specialized `Stack` classes, 425
`struct`, 334
subroutines, 53
subscriber methods, 508–510
types, 7–8
delegates
class hierarchies, 473n1
data types, 472–473
events, 528–530
instantiating, 475–480
internals, 473–474
invoking, 512–513
multicast, 508
coding observer patterns with, 508–523
internals, 518–519
operators, 514–516
overview of, 470–480
passing, 829
types, defining, 474–475
variance, applying, 485
`delete` operator, 208
deleting whitespace, 12
delimited comments, 21
XML, 387
delimiters, statements, 10
dereferencing
pointers, 837–839
reference types, 334
deriving
base types, casting between, 272–273
inheritance, 270–281
one interface from another, 318
preventing, 281
design, synchronization best practices, 674
destruction, deterministic, 208, 399, 850
detecting virtual computers using P/Invoke, 888–894
deterministic destruction, 208, 399, 850
deterministic finalization, 395–398
diagrams
interfaces, 325
sequences, 520
Venn, 568
dialog boxes, Windows Error Reporting, 715
dictionary collections, 622–626
directives
`import`, wildcards in, 162
preprocessor, C#, 138–145
`using`, 161–168
disambiguation, multiple `Main()` methods, 167
dispatch method calls during construction, 286
`Dispose()` method, 397
disposing tasks, 723–724
distinct members, 606–607
`Distinct()` standard query operator, 584
dividing `float` by zero, 91
division (/) operator, 85
documentation, generating XML, 388–389
domains, applications, 854–855
`double` type, 36
`do/while` loops, 121–123
downloading C# compilers, 865
Duffy, Joe, 801n1
duplicating interfaces, 433–434
dynamic binding, 694
`dynamic` data type principles and behaviors, 690–693
dynamic objects
custom implementation, 696–699
programming with, 688–699
reflection, invoking, 689–690
dynamic programming, static compilation *versus*, 695–696

E

EAPs (Event-based Asynchronous Patterns), 801–804
editors, visual hints for, 144–145
`Eject()` method, 274
emitting errors, 141–142
empty catch block internals, 411
empty collections, returning, 634
enabling Intellisense, 592
encapsulation, 203
APIs, 826–828
circumventing, 852
classes, 258–260
information hiding, 220
objects group data with methods, 208–209
publication, 524–535
subscriptions, 523–524
of types, 379–380
enums
defining, 347
flags, 351–355
`FlagsAttribute`, 354–355
string conversion, 350–351
type compatibility between, 349–350
value types, 346–355
equality conditionals, avoiding, 91
equality (==) operators, 110–111, 370
`Equals()` method, overriding, 361–369
errors
arrays, 69, 78–80
emitting, 141–142
handling
C# 3.0, 519–520
P/Invoke, 821–823
infinite recursion, 178

methods, 186–199
operators, 110
reporting, 196
trapping, 187–192
escape sequences, 42
Event-based Asynchronous Patterns. *See* EAPs
events, 507–508
code conventions, 526–528
declaring, 525–526
delegates, 528–530
generics, 528–530
implementation, customizing, 532–533
internals, 530–523
multicast delegates, coding observer patterns with, 508–523
notifications
firing, 527–528
with multiple threads, 763–764
overview of, 523–533
publishers, defining, 510–511
reset, 768–771
exceptions
catching, 191–192, 196, 407–408, 411
class inheritance, 192
customizing, defining, 414–419
error handling, 186–199
general catch blocks, 409–410
handling, 405–419
background worker patterns, 808–809
subscribers, 520
unhandled exception handling on `Task`, 715–718
hiding, 411–412
inner, 415
multiple types, 405–407
reports, 412
rethrowing, 197, 413
serializable, 416
throwing, 406–407
types, 193–194
unhandled exceptions on `AppDomain`, 744–746
exclusive OR (^) operator, 112
executing
deferred
with LINQ query expressions, 593–598
standard query operators, 562–566
implicit execution, implementing, 607–608
iterations in Parallel, 724–734
management, 23–30
`ManualResetEvent` synchronization, 770
pseudocode, 752
threads, 704. *See also* multithreading
time, 24
VES (Virtual Execution System), 844
explicit cast, 58–59
explicit member implementation, 314–315
exponential notation, 37
exposing Async methods, 810
expressions. *See also* LINQ
Boolean, 109–115
constants, 98
lambda, 401, 486–505
queries
LINQ, 589–590
PLINQ (Parallel LINQ), 736
trees, 498–505
converting to SQL `where` clauses, 499
object graphs, 499–501
viewing, 503–505
`typeof`, 654–655
Extensible Markup Language. *See* XML
extensions
interfaces, 322–323
`IQueryable<T>`, 585
methods, 256–258, 278
external functions, calling, 826–828
extracting XML data, 385n2

F

factory inheritance, 451
`false` operator, 373–375
FCL (Framework Class Library), 860
fields
`const`, 258–259
instances, 209–211, 249
static, 248–250
virtual, properties as, 232–234
`volatile`, declaring as, 760–761
filenames, must match class names (Java), 4
files
data retrieval, 218
header, 160
loading, 216
XML, 22–23, 388–389. *See also* XML
filtering
LINQ query expressions, 598–599
with `System.Linq.Enumerable.Where()`, 562
with `Where()` methods, 556–557
finalization
deterministic, 395–398
garbage collection and, 398–399
guidelines, 400
task-related, 717
finalizers, 241, 393–395
defining, 434–435
`FindAll()` method, 621–622
firing event notifications, 527–528
fixing data, 835
flags, enums, 351–355
`FlagsAttribute`, 354–355, 675

floating-point types, 33–34
  inequality with, 89–92
  special characteristics of, 89
flow. *See* control flow
foreach loops
  with `IEnumerable<T>`, 547–551
  without `IEnumerable<T>`, 551–552
`foreach` loops, 127–130
  with arrays, 546–547
  collections, iterating over, 613
  modifying, 552
  parallel execution of, 727
`for` loops, 124–127
format items, 19
`Format()` method, 46
formatting
  code, avoiding ambiguity, 213–217
  indentation, 12
  Java
    lowercase, 9
    uppercase, 9
  numbers as hexadecimal, 38–39
  PLINQ (Parallel LINQ), 736–738
  round-trip, 39–40
  single instance applications, 766–767
  whitespace, 11–12
Forms, Windows, 809–811
Framework Class Library (FCL), 860
f-reachable objects, 390
from clause, 590
full outer joins, 569
full source code listings
  `Binary Tree` and `Pair`, 876–881
  command-line attributes, 881–888
  `ProductSerialNumber`, 874–876
  `Tic-Tac-Toe`, 869–874
  virtual computer detection using P/Invoke, 888–894
functions
  anonymous, 486
  `Average`, 585
  `Count()`, 585
  external
    calling, 826–828
    declaring, 817
  global variables and, 248
  `Max()`, 585
  `Min()`, 585
  pointers, passing delegates, 829
  pure virtual, 297
  `Sum()`, 585
fundamental numeric types, 32–40

G

garbage collection, 25, 849–851
  and finalization, 398–399
  well-formed types, 390–393
general catch blocks, 409–410
generating
  anonymous types, 542–543
  XML documentation files, 388–389
generics, 421
  benefits of, 430–431
  `catch`, 194
  classes, 427–429
  collection interface hierarchies, 613
  constraints, 439–457
  contravariance, 457–463
  covariance, 457–463
  C# without, 422–439
  events, 528–530
  interfaces, 432–433
  internals, 463–467
  lazy loading and, 401
  methods, 453–457
  structs, 432–433
  types, 427–439
    nested, 438–439
    reflection, 660–662
    `Tuple`, 437–438
`GetHashCode()` method, overriding, 358–361
`GetSummary()` member, 296
getters, access modifiers, 231–232
`GetType()` member, 653–654
GhostDoc, 389n3
global variables and functions, 248
`goto` statements, 137–138
graphs, objects, 499–501
greater than (>) operator, 110
greater than or equal to (>=) operator, 110
groupby clause, 590
`GroupBy()` method, grouping results with, 575–577
grouping
  LINQ query expressions, 602–605
  results with `GroupBy()` method, 575–577
  statements into methods, 150
`GroupJoin()` method, 577–580
guidelines
  for conversion operators, 377
  for exception handling, 411–413
  finalization, 400
  P/Invoke, 829–830

H

handling
  errors
    C# 3.0, 519–520
    methods, 186–199
    P/Invoke, 821–823
  exceptions, 405–419
    background worker patterns, 808–809
    subscribers, 520
    unhandled exception handling on `Task`, 715–718
hardcoding values, 35–37
hash symbol (#), 139
header files, 160

heaps, reference types, 333
HelloWorld program, 2–4
  CIL output for, 28–30
hexidecimal notation, 38
hiding
  exceptions, 411–412
  information, 220
hierarchies
  classes, 204, 473n1
  collections, 613
hints for visual editors, 144–145
hold and wait condition, 764
hooking up background worker patterns, 807–808

I

ICollection<T> interface, 616–617
IComparable<T> interface, 443, 614–617
IComparer<T> interface,sorting, 614–615
identifiers, 6–7
  keywords used as, 7
  type parameters, 429
IDictionary<TKey, TValue> interface, 614–617
IDisposable interface, using explicitly in place of SafeHandle, 825–826
Id property, 710
IEnumerable<T>
  collections interfaces, 546–552
  foreach loops with, 547–551
  foreach loops without, 551–552
IEnumewrable<out T>, covariance, 485n2
if statements, 102–103
  followed by code blocks (), 105
ILDASM, CIL and, 27–30
IList<T> interface, 614–617
immutable anonymous types, 541
immutable strings, 16, 49–51
immutable value types, 336
implementing
  CLI (Common Language Infrastructure), 845–846
  conversion operators, 376
  custom dynamic objects, 696–699
  Equals() method, 366
  events, customizing, 532–533
  explicit member, 314–315
  generic interfaces, 432
  GetHashCode() method, 359
  implicit execution, 607–608
  implicit member, 315–316
  interfaces, 308–312, 312–318, 433–434
  multiple interface inheritance, 324–326
  new operator, 238
  one-to-many relationships, 577–580
  outer joins, 579
  virtual methods, 283
implicit base type casting, 273
implicit conversion, 62, 273
  cast operators, 376
implicit execution, implementing, 607–608
implicitly typed local variables, 53–55, 538–540
implicit member implementation, 315–316
implicit overriding, 283
import directive, wildcards in, 162
incompatibilities, 6n6
increment (++) operator, 94–97
indentation
  formatting, 12
  invalid code, 106
indexer property names, assigning, 632–633
index operators, 630–634
  items to Dictionary<TKey, TValue>, adding, 623
indiscriminate synchronization, 758
inequality (!=) operator, 110, 370
inequality with floating-point types, 89–92
inferencing types, 454–455
infinite recursion errors, 178
infinity, negative, 92
information hiding, 220
infrastructure, languages, 23–30. *See also* CLI
inheritance, 203, 269–270
  abstract classes, 293–299
  as operators, 302
  base classes, overriding, 281–293
  chains, casting within, 274
  constraints, 447–448, 450
  definitions, 269–270
  derivation, 270–281
  exceptions, classes, 192
  factory, 451
  interfaces, 318–321
  is operators, 301
  methods, 271
  multiple, 278
  multiple interfaces, 321–322, 324–326
  polymorphism, 297–299
  single, 278–281
  System.Object, 299–301
  types, 205
  value types, 338–339
initializers
  collection, 240–241, 543–546
  objects, 239–241
initializing
  anonymous type arrays, 545–546
  attributes through constructors, 668–673
  centralizing, 244–245
  jagged arrays, 70
  lazy initialization, well-formed types, 400–402
  structs, 336–337

initializing *(contd.)*
three-dimensional arrays, 69
two-dimensional arrays, 69
inner classes, 262
inner exceptions, 415
inner joins, 568
with `Join()` method, performing, 572–575
input, consoles, 16–20
installing C# compilers, 865–867
instances
array methods, 75–76
custom attributes, retrieving, 670
fields, 209–211, 249
methods, 47, 211–212
single applications, 766–767
instantiating, 9
arrays, 66–70
classes, 205–209
delegates, 475–480
generics
based on reference types, 465–467
based on value types, 464–465
integers
types, 32–33
values, overflowing, 59
Intellisense, enabling, 592
interfaces, 305–307
collection, 535–536. *See also* collection interfaces
compared with classes, 328–329
constraints, 442–444
conversion between implementing classes and, 318
custom collections, 612–613
defining, 307
diagramming, 325
duplicating, 433–434
explicit member implementation, 314–315
extension methods on, 322–323
generics, 432–433
`ICollection<T>`, 616–617
`IComparable<T>`, 443, 614–617
`IComparer<T>`, 614–615
`IDictionary<TKey, TValue>`, 614–617
`IDisposable`, using explicitly in place of `SafeHandle`, 825–826
`IList<T>`, 614–617
implementation, 312–318
implicit member implementation, 315–316
inheritance, 318–321
multiple inheritance, 321–322, 324–326
`Parallel.For()` API, 726
polymorphism through, 307–312
support, 440
value types, 338–339
versioning, 327–328
`VirtualAllocEx()`, declaring, 818–819
Windows UI programming, 809–813
internals
anonymous methods, 494–495
delegates, 473–474
events, 530–523
generics, 463–467
lambda expressions, 494–495
multicast delegates, 518–519
properties, 235–236
interoperability of languages, 25
`Intersect()` standard query operator, 584
`into` clauses, query continuation with, 605–606
`in` type parameter, enabling contravariance with, 460–462
invalid code, indenting, 106
invalid reference types, 833
invoking
callbacks, 787
delegates, 512–513, 522
members, 655–660
P/Invoke (Platform Invoke), 816–830
reflection, dynamic objects, 689–690
sequential invocation, 516–517
`using` statements, 397
`IQueryable<T>`, 585
`IsCompleted` property, 710
`is` operators, 301
items, formatting, 19
iterations
`Dictionary<Tkey, TValue>`, 624
executing in Parallel, 724–734
`foreach` loops, modifying, 552
over `foreach` loops, 613
iterators
`class`, 645
classes, creating multiple in, 648–649
collections, 634–650
defining, 636
examples of, 641–643
overview of, 646–648
and state, 639–641
`struct`, 645
syntax, 636–637
values, yielding, 637–639
`yield break`, 645–646
`yield` statements, 649

J

jagged arrays. *See also* arrays
declaring, 71
initializing, 70
Java
array declaration, 66
exception specifiers, 408
filenames must match class names, 4
generics, 467
implicit overriding, 283
inner classes, 262

virtual methods by default, 282
wildcards in `import` directive, 162
JavaScript
`var`, 540
`Variant`, 540
`void*`, 540
JIT (just-in-time) compilers, 848
jitting, 24
`Join()` method, performing with inner joins, 572–575
joins, 568, 569
jump statements, 132–138
just-in-time (JIT) compilers, 848

K

keywords, 4–6
contextual, 6–7
`lock`, 757–758
`Me`, accessing class instances with, 214
`new`, 67
`null`, 51–52
`string`, 163n2
`this`, classes, 213–220
used as identifiers, 7
`var`, 53
`void`, 52–55
`yield`, 6n5
Knoppix, 867

L

lambdas
expressions, 401, 486–505
statements, 486–489
languages, 158
accessing class instances with `Me` keyword, 214
buffer overflow bugs, 72
CIL (Common Intermediate Language), 23
COM DLL registration, 858
`delete` operator, 208
deterministic destruction, 399, 850
dispatch method calls during construction, 286
exception specifiers, 408
generics, 467
global variables and functions, 248
header files, 160
implicit overriding, 283
infrastructure, 23–30
inner classes, 262
interoperability, 25
Java
filename must match class names, 4
`main()` is all lowercase, 9
multiple inheritance, 278
operator errors, 110
origin of iterators, 635
preprocessing, 138
project scope `Imports` directive, 162
pure virtual functions, 297
redimensioning arrays, 75
returning `void`, 53
`short` data types, 33
string concatenation at compile time, 45
`struct` defines type with public members, 337
templates, 442
UML (Unified Modeling Language), 325n1
virtual methods by default, 282
Visual Basic line-based statements, 10
`void*`, 540
`void` as data types, 52
wildcards in `import` directive, 162
last in, first out (LIFO), 422
lazy initialization, well-formed types, 400–402
left outer joins, 568
length
of arrays, 72
strings, 48–49
less than (<) operator, 110
less than or equal to (<=) operator, 110
`Let` clause, 600–602
libraries
class, 378
classes, 377–378
LIFO (last in, first out), 422
limiting
attributes, 674
constraints, 449–452
line-based statements, 10
lines, specifying numbers, 143–144
linked lists, collections, 629–630
`LinkedList<T>` class, 629
LINQ
customizing, 585
distinct members, 606–607
implicit execution, implementing, 607–608
`Let` clause, 600–602
queries
continuation with `into` clauses, 605–606
running in parallel, 734–738
query expressions, 589–590
compiling, 607
deferred execution with, 593–598
filtering, 598–599
grouping, 602–605
as method invocations, 608–609
overview of, 590–592
projection using, 592–593
sorting, 599–600
Linux, 867
Liskov, Barbara, 635
`List<T>` class, 617–621
literals
strings, 44–46
values, 35, 68
loading files, 216
local storage, threads, 774–777

local variables, 13
  implicitly typed, 53–55, 538–540
  multiple threads, 753–753
lock keyword, 757–758
  ConsoleSyncObject, 797
  objects, selecting, 758–759
locks, avoiding unnecessary, 765–766
lock statements, value types in, 343
logical Boolean operators, 111–113
logical notation (!) operator, 113
logical operators, 117–118
logs, exceptions, 412
long-running threads, 722–723
loops
  for, 124–127
  decrement (- -) operators, 94
  do/while, 121–123
  foreach, 127–130
    with arrays, 546–547
    with IEnumerable<T>, 547–551
    iterating over, 613
    modifying, 552
    parallel execution of, 727
    without IEnumerable<T>, 551–552
  parallel, canceling, 729–734
  while, 121–123
  yield returns, placing in, 643–645
lowercase, Java, 9

M

machine code, 844, 847–849
Main() method, 8
  declarations, 9–10
  parameters, 165–168
  returns, 165–168
managing
  code, 24
  execution, 23–30
  resources, 823–824
  threads, 740–742
manifests, CLI (Common Language Infrastructure), 855–858
ManualResetEvent, 768–771
ManualResetEventSLim, 768–771
many-to-many relationships, 569
matching caller variables with parameter names, 168
mathematics constants, 107
Max() function, 585
Me keyword, accessing class instances with, 214
members
  abstract, 294
  base, 291–292
  classes, 209
  distinct, 606–607
  explicit member implementation, 314–315
  GetSummary(), 296
  GetType(), 653–654
  implicit member implementation, 315–316
  invoking, 655–660
  object, overriding, 357–369
  private, 220
  referent types, accessing, 839
  static, 247–256
  System.Object, 299–301
  variables, 209
messages, turning off warning (#pragma), 142–143
metadata, 25
  CLI (Common Language Infrastructure), 860–861
  reflection, 652–662
methodImpAttribute, avoiding synchronization, 760
methods, 149–150
  Add(), 543
  anonymous, 480–482
    internals, 494–495
    parameterless, 482
  arrays, 73–75
  Assert(), 91
  BinarySearch(), 75, 620
  BubbleSort(), 470–472
  calling, 150–156
  Clear(), 75
  Collect(), 391
  CommandLineHandler.TryParse(), 671
  CompareTo(), 442
  ComparisonHandler-Compatible, 478–479
  ContinueWith(), 711–715, 717, 795–796
  Copy(), 257
  CopyTo(), 617
  Count(), counting elements with, 561
  DataStore(), 545
  declaring, 157–161
  Dispose(), 397
  Eject(), 274
  Equals(), overriding, 361–369
  error handling, 186–199
  extension, 256–258
  extensions, 278
  FindAll(), 621–622
  Format(), 46
  generics, 453–457
    casting inside, 456–457
    determining support for, 661–662
  GetHashCode(), overriding, 358–361
  GroupBy(), grouping results with, 575–577
  GroupJoin(), 577–580
  inheritance, 271
  instances, 47, 75–76, 211–212
  Join(), performing inner joins with, 572–575
  Main(), 8. *See also* Main() method
    declarations, 9–10

multiple Main(), disambiguation, 167
optional parameters, 182–185
OrderBy(), sorting with, 566–572
overloading, 179–182
overview of, 150–152
parameters, 168–176
partial, 264–267
Pop(), 422
Pulse(), 756
Push(), 422
query expressions as invocations, 608–609
recursion, 176–179
refactoring into, 158
resolution, 185
returns, 155–156, 522–523
Run(), 285
Select(), 557–560, 734
SelectMany(), 580–582
SetName(), 213
starting, 707
static, 251–253
Store(), 216
strings, 46–47
stringStatic, 46
subscriber, defining, 508–510
System.Console.ReadKey(), 18
System.Console.ReadLine(), 16
System.Console.Write(), 18–20
System.Threading.Interlocked, 762
ThenBy(), sorting with, 566–572
ToString(), overriding, 358
ToUpper(), 50
TryParse(), 63, 198–199
type names, 154
for unsafe code, 831
Where(), filtering with, 556–557
Min() function, 585
minus (-) operator, 84–92
models, APMs (Asynchronous Programming Models), 783–797
modifiers
access, 220–222, 852
accessibility, 381
new, 286–291
nullable, 57–58
readonly, 259
sealed, 291
virtual, 282–286
modifying
foreach loops, 552
targets, assemblies, 378–379
values, variables, 15
modules, 378
CLI (Common Language Infrastructure), 855–858
Monitor class synchronization, 754–758
Mono compilers, 3n4, 866–867
MTA (Multithreaded Apartment), 813
multicast delegates, 508
coding observer patterns with, 508–523
internals, 518–519
multidimensional array errors, 69
multimode assemblies, building, 856n5
multiple constraints, 446
multiple duplication of interfaces, 433–434
multiple exception types, 405–407
multiple inheritance, 278
multiple interface inheritance, 321–322
implementing, 324–326
multiple iterators, creating, 648–649
multiple Main() methods, disambiguation, 167
multiple threads
event notification with, 763–764
and local variables, 753–753
thread-safe, 752
multiple type parameters, 436
multiplication (*) operator, 85
Multithreaded Apartment (MTA), 813
multithreading, 701–706
before .NET Framework 4, 738–743
uncertainty, 706
unhandled exceptions on AppDomain, 744–746
mutual exclusion condition, 764

N

names
indexer property, assigning, 632–633
parameters, 184, 674–676
type methods, 154
namespaces, 152–154, 161
aliasing, 164–165
alias qualifiers, 384–385
nesting, 383
well-formed types, defining, 382–385
naming conventions
parameter types, 431
properties, 228–229
types, 7
NDoc, 389n4
negative infinity, 92
nesting
classes, 260–262, 265
delegate data types, declaring, 475
generic types, 438–439
if statements, 103–105
namespaces, 383
using declaratives, 163
.NET, 865–866
Framework, multithreading before version 4, 738–743
garbage collection, 849–850

.NET *(contd.)*
garbage collection in, 390–391
lazy initialization, 401
versioning, 26–27
`new` keyword, 67
newline (\n) characters, 42, 48
`new` modifiers, 286–291
`new` operator
implementation, 238
value types, 337
`NGEN` tool, 848
no preemption condition, 764
notation
exponential, 37
hexidecimal, 38
notifications, events
firing, 527–528
with multiple threads, 763–764
Novell, 3n4
`nowarn:<warn list>` option, 143
null
checking for, 513–514
returning, 634
nullable modifiers, 57–58
nullable value types, 425–427
null coalescing (??) operator, 114–115
`null` keyword, 51–52
numbers
to Booleans, conversion, 61
conversion with `TryParse()` method, 198–199
hexidecimal, formatting, 38–39
lines, specifying, 143–144
types, 32–40

O

`object` members, overriding, 357–369
object-oriented programming, classes, 203–205
objects
associated data, 250
CTS (Common Type System), 859
defining, 206
dynamic
implementing custom, 696–699
invoking reflection, 689–690
programming with, 688–699
f-reachable, 390
graphs, 499–501
group data with methods, 208–209
initializers, 239–241
`lock`, selecting, 758–759
resurrecting, 399–400
observers, 508
patterns, coding multicast delegates with, 508–523
`OfType<T> ()` standard query operator, 584
omitting parameter types from statement lambdas, 488
one-to-many relationships, 569
implementing, 577–580
operands, 84, 92
operators, 83–84, 84–98
AND (&&), 112, 373
adding, 371–373
addition (+), 85, 371–373
arithmetic, 85
`as`, 302
assignment, 92–98, 120
binary, 371–373
bitwise, 115–121
bitwise complement (~), 120
cast, 58, 275
cast (), 375–376
conditional (?), 113–114
constraints, 449
conversion, 375, 377
decrement (- -), 94–97
`default()`, 68, 338, 435
delegates, 514–516
`delete`, 208
division (/), 85
equality (==), 110–111, 370
errors, 110
exclusive OR (^), 112
`false`, 373–375
greater than (>), 110
greater than or equal to (>=), 110
increment (++), 94–97
index, 623, 630–634
inequality (!=), 110, 370
`is`, 301
less than (<), 110
less than or equal to (<=), 110
logical, 117–118
logical Boolean, 111–113
logical notation (!), 113
minus (-), 84–92
multiplication (*), 85
`new`
implementation, 238
value types, 337
null coalescing (??), 114–115
OR (||), 111–112, 373, 450
overloading, 369–377
parenthesis, 92–98
plus (+), 84–92
postfix increment, 96
post-increment, 95
precedence, 86
prefix increment, 96
pre-increment, 96
remainder (%), 85
shift, 116–117
simple assignment (=), 14
standard query, 535–536. *See also* collection interfaces; standard query operators
`true`, 373–375
unary, 373–375
optional parameters, 182–185
options
command-line, 76
`nowarn:<warn list>`, 143
parallel, 731–734

OrderBy() method, sorting with ThenBy() method, 566–572
order of associativity, 86
origin of iterators, 635
OR (||) operator, 111–112, 373
  constraints, 450
outer joins, 568
  implementing, 579
outer variables, 495–496
out parameter values, 234–235
output
  consoles, 16–20
  parameters, 171–173
out type parameter, enabling covariance with, 458–460
overflowing
  bounds of a float, 92
  integer values, 59
overloading
  constructors, 241–242
  methods, 179–182
  operators, 369–377
  System.Threading.Interlocked class methods, 762
overriding
  base classes, 281–293
  Equals() method, 361–369
  GetHashCode() method, 358–361
  implicit, 283
  object members, 357–369
  properties, 282
  ToString() method, 358
overruns, buffer, 72

P

parallel
  exception handling with System.AggregateException, 728–729
  iterations, executing in, 724–734
  loops, canceling, 729–734
  results and options, 731–734
Parallel.For() API, 726
Parallel LINQ (PLINQ), 559–560, 703, 736–738
parameterized types, 427
parameterless anonymous methods, 482
parameterless statement lambdas, 488
parameters, 149–150
  AllowMultiple, 674
  arrays, 173–176
  data types, 818–819
  declaring, 159
  Main() method, 165–168
  methods, 155, 168–176
  named, 184, 674–676
  optional, 182–185
  output, 171–173
  references, 170–171
  single input, statement lambdas with, 489
  types, 429, 660–661
    in, 460–462
    inferring, 454–455
    multiple, 436
    naming conventions, 431
    out, 458–460
  values, 168–169
  variables, defining index operators, 633–634
parent classes, 205
parenthesis operator, 92–98
partial classes, 262–267
partial methods, 264–267
pass-by references, 522
passing
  anonymous methods, 480–481
  command-line arguments to Main() methods, 166
  data to and from an alternate thread, 799–801
  delegates, 486–487, 489–490, 829
  states between APM (Synchronous Programming Model) methods, 789–790
paths, code, 159
patterns
  BackgroundWorker class, 804–809
  EAPs (Event-based Asynchronous Patterns), 801–804
  observers, coding multicast delegates with, 508–523
  publish-subscribe, 508
performance, 853–854
  synchronization, affect on, 758
performing inner joins with Join() method, 572–575
permanent values, 259
permissions, CAS (code access security), 659
persistence, data, 217
pi, calculating, 725
P/Invoke (Platform Invoke), 816–830
  errors, handling, 821–823
  virtual computer detection using, 888–894
placeholders, 19
  values, 115
Platform Invoke. *See* P/Invoke
platform portability, 852–853
platforms, 865–867
  portability, 25
PLINQ (Parallel LINQ), 559–560, 703, 736–738
plus (+) operator, 84–92
pointers
  and addresses, 830–839
  assigning, 834–837
  declaring, 832–834
  dereferencing, 837–839
  functions, passing delegates, 829
polymorphism, 205
  inheritance, 297–299
  through interfaces, 307–312
pools, threads, 706, 742–743
Pop() method, 422

portability
platform, 852–853
platforms, 25
postfix increment operators, 96
post-increment operators, applying, 95
precedence, operators, 86
predefined attributes, 676–677
predefined types, 31
prefix increment operators, 96
pre-increment operators, applying, 96
preprocessor directives, C#, 138–145
preventing
covariance maintains homogeneity, 457
derivation, 281
primary collections classes, 617–630
primitives, 31
principles, `dynamic` data type, 690–693
`private` access modifiers, 275
private members, 220
`ProductSerialNumber`, 874–876
programming
APMs (Asynchronous Programming Models), 783–797
`Binary Tree` and `Pair`, 876–881
command-line attributes, 881–888
comments, 20–23
constructs, associating XML comments with, 386–388
dynamic, static compilation *versus*, 695–696
with dynamic objects, 688–699
`HelloWorld` program, 2–4
object-oriented, classes, 203–205
`ProductSerialNumber`, 874–876
`Tic-Tac-Toe`, 869–874
values, hardcoding, 35–37
virtual computer detection using P/Invoke, 888–894
Windows UI, 809–813
programs
CIL output for, 28–30
`HelloWorld`, 2–4
projecting
LINQ query expressions, 592–593
with `Select()` method, 557–560
project scope `Imports` directive, 162
properties
attributes, 663, 664
automatically implemented, 225–227
C#, 48
classes, 222–236
`Count`, 617
declaring, 223–225
defining, 224
indexer property names, assigning, 632–633
internals, 235–236
lazy loading, 402
naming conventions, 228–229
overriding, 282
read-only, 230–231
static, 254–256
validation, applying, 228–229
as virtual fields, 232–234
write-only, 230–231
`protected` access modifiers, 276
pseudocode, executing, 752
publication, encapsulating, 524–535
public constants, 259
publishers
connecting, 511–512
events
defining, 510–511
publish-subscribe patterns, 508
`Pulse()` method, 756
pure virtual functions, 297
`Push()` method, 422

Q

qualifiers, aliasing namespaces, 384–385
quantum, 704
queries. *See also* LINQ
continuation with `into` clauses, 605–606
LINQ, 589–590, 734–738
PLINQ (Parallel LINQ), 559–560, 736–738
standard query operators. *See* standard query operators
queues, collections, 629
`Queue<T>` class, 629

R

RCW (runtime callable wrapper), 813
`readonly` modifiers, 259
read-only properties, 230–231
recursion
infinite recursion errors, 178
methods, 176–179
redimensioning arrays, 75
reentrant (locks), 765
refactoring
base classes, 271
into methods, 158
references
assemblies, 377–381
parameters, 170–171
pass-by, 522
root, 390
strong, 391
types, 56–57, 169–170, 333–336, 465–467
weak, 391–393

referent types, 832
  members, accessing, 839
reflection, 652–662
  dynamic objects, invoking, 689–690
  on generic types, 660–662
ref parameter values, 234–235, 819–820
registering
  COM DLL, 858
  for unhandled exceptions, 744–745
relational operators, 110–111
relationships
  many-to-many, 569
  one-to-many, 569, 577–580
remainder (%) operator, 85
removing
  conditions, 765
  whitespace, 12
reports
  errors, 196
  exceptions, 412
reserved words, 4. *See also* keywords
reset events, 768–771
resolution, methods, 185
resources
  cleanup, 393–400, 790–791
  managing, 823–824
  utilization, 400
results
  GroupBy() method, 575–577
  parallel, 731–734
  tasks, returning, 709
resurrecting objects, 399–400
rethrowing exceptions, 197, 413
retrieving
  attributes, 667–668
  specific attributes, 669
return attributes, specifying, 666
returning
  empty collections, 634
  null, 634
  task results, 709
  void, 53
returns
  Main() method, 165–168
  methods, 159–160, 522–523
  yield returns, placing in loops, 643–645
return statements, 160
return values, 15
reusing code, 378
Reverse() standard query operator, 584
reversing strings, 77
right outer joins, 569
root references, 390
round-trip formatting, 39–40
Run() method, 285
running
  HelloWorld program, 3–4
  LINQ queries in parallel, 734–738
  Parallel LINQ (PLINQ) queries, 559–560
  threads, 706–738
    canceling tasks, 718–722
    disposing tasks, 723–724
    long-running threads, 722–723
    unhandled exception handling on Task, 715–718
runtime, 24
  arrays, defining array size at, 68
  CLI (Common Language Infrastructure), 849–854
  metadata, reflection, 652–662
  virtual methods, 283
runtime callable wrapper (RCW), 813

S

SafeHandle, applying, 823–824
safety, types, 25, 541, 851
scope, 107–109, 155
sealed classes, 281
sealed modifiers, 291
searching
  attributes, 667–668
  List<T> class, 619
security
  access, 25
  CAS (Code Access Security), 659, 852
select clause, 590
selecting lock objects, 758–759
SelectMany() method, calling, 580–582
Select() method, 734
  projecting with, 557–560
SemaphoreSlim, 772
semaphores over AutoResetEvent, 772
semicolons (;)
  statements without, 10–11
  whitespace, 11–12
SequenceEquals() standard query operator, 584
sequences
  deferred execution, 565
  escape, 42
  invocation, 516–517
  layout, StructLayoutAttribute for, 820–821
  multithreading, 703. *See also* multithreading
serialization
  attributes, 680–682
  customizing, 683–684
  exceptions, 416
  versioning, 684–687
SetName() method, 213
setters, access modifiers, 231–232
shift operators, 116–117
short data types, 33
shutdown, applications, 717
signatures, APMs (Asynchronous Programming Models), 786–787
Silverlight, 536n1
simple assignment (=) operators, 14
simple generic classes, defining, 429–430
simplifying API calls with wrappers, 828–829
single inheritance, 278–281
single input parameters, statement lambdas with, 489

single instance applications, creating, 766–767
single-line comments, 22
single-line XML comments, 386–387
single quote ('), 42
sites, call, 168
sizing
  arrays at runtime, 68
  types, 752
SortedDictionary<TKey, TValue> class, 626–628
SortedList<T> class, 626–628
sorting
  collections, 626–628
  IComparer<T> interface, 614–615
  LINQ query expressions, 599–600
  with OrderBy() method and ThenBy() method, 566–572
space, declaring, 107–109
specialized Stack classes, defining, 425
specializing types, 205
specifiers, exceptions, 408
specifying
  constraints, 455
  default values, 435–436
  line numbers, 143–144
  literals, 36
  multiple constraints, 446
  parameters by name, 184
  return attributes, 666
SQL
  query expressions, 592
  where clauses, converting expression trees to, 499
Stack class, 422
  specialized, defining, 425
stacks
  calling, 168, 836
  collections, 628
  unwinding, 168
Stack<T> class, 628
standard query operators, 552–586, 582–586
  collection interfaces with, 535–536. *See also* collection interfaces
  Count() method, counting elements with, 561
  deferred execution, 562–566
  grouping results with GroupBy() method, 575–577
  implementing one-to-many relationships, 577–580
  performing inner joins with Join() method, 572–575
  Select() method, projecting with, 557–560
  sorting with OrderBy() method and ThenBy() method, 566–572
  Where() method, filtering with, 556–557
starting methods, 707
statements, 10
  Assert(), 92
  break, 132–135
  continue, 135–136
  control flow, 121–132
  delimiters, 10
  goto, 137–138
  groups into methods, 150
  if, 102–103, 105
  jump, 132–138
  lambdas, 486–489
  line-based, 10
  lock, 343
  versus method calls, 156
  nested if, 103–105
  return, 160
  switch, 130–132, 160
  System.Console.WriteLine(), 10
  Throw, 196
  using, 217n1, 395–398
  without semicolons (;), 10–11
  yield, 649
states
  APM (Synchronous Programming Model) methods, passing between, 789–790
  callbacks, invoking, 787
  iterators and, 639–641
  unsynchronized, 750
STAThreadAttribute, controlling COM threading models with, 813
static classes, 255
static compilation *versus* dynamic programming, 695–696
static constructors, 253–254
static fields, 248–250
static members, 247–256
static methods, 251–253
static properties, 254–256
Status property, 710
storage, local, 774–777
Store() method, 216
string keyword, 163n2
strings, 43–51
  applying, 50
  as arrays, 76–78
  concatenation at compile time, 45
  conversion, 63
  enums, 350–351
  immutable, 16, 49–51
  length, 48–49
  literals, 44–46
  methods, 46–47
  plus (+) operator, using with, 87–88
  reversing, 77
stringStatic methods, 46
string type, avoiding, 759–760
strong references, 391
struct
  class constraints, 445
  defining, 334
  generics, 432–433
  initializing, 336–337
  iterators, 645

StructLayoutAttribute for sequential layout, applying, 820–821
styles
  code, avoiding ambiguity, 213–217
  CPS (Continuation Passing Style), 787–789
subroutines, defining, 53
subscribers
  connecting, 511–512
  exceptions, handling, 520
  methods, defining, 508–510
subscriptions, encapsulating, 523–524
subtypes, 204
Sum() function, 585
super types, 204
support, interfaces, 440
switch statements, 130–132, 160
synchronization
  design best practices, 674
  lock, ConsoleSyncObject, 797
  methodImpAttribute, avoiding, 760
  Monitor class, 754–758
  threads, 750–777
  types, 766–774
  when to provide, 765
syntax, 1–2
  fundamentals, 4–12
  iterators, 636–637
System.Action, 483–484
System.ArgumentException, 405
System.AsyncCallback, 787–789
System.AttributeUsageAttribute, 673–674
System.Collections.Generic.ICollection<T>, 544
System.Collections.Generics namespace, 153
System.Collections namespace, 153
System.Collection.Stack, 423
System.ConditionalAttribute, 677–679
System.Console.ReadKey() method, 18
System.Console.ReadLine() method, 16
System.Console.WriteLine() statement, 10
System.Console.Write() method, 18–20
System.Data namespace, 153
System-defined delegates: Func, 483–485
System.Drawing namespace, 153
System.Exception
  catch blocks, 195–196
  use of, 412
System.IO namespace, 153
System.Linq.Enumerable.Where(), 562
System.Linq namespace, 153
System.Linq.Queryable, 585
System namespace, 153
System.NonSerializable attribute, 682–683
System.Object inheritance, 299–301
System.ObsoleteAttribute, 679–680
System.Runtime.CompilerServices.CompilerGeneratedAttribute, 236
System.Runtime.Serialization.OptionalFieldAttribute, 686
Systems.Collections.Concurrent, 895–898
System.SerializableAttribute, 687–688
Systems.Timer.Timer, 780
System.Text namespace, 153
System.Text.StringBuilder data type, 51
System.Threading.Interlocked class, 761–763
System.Threading.Mutex, 766–767
System.Threading namespace, 153
System.Threading.Tasks namespace, 153
System.Threading.Thread, 738–740
System.Threading.WaitHandle class, 768–769
System.Type, accessing metadata, 653–655
System.Web namespace, 154
System.Web.Services namespace, 154
System.Windows.Forms namespace, 154
System.Xml namespace, 154

T

targets, modifying assemblies, 378–379
Task.CurrentID property, 711
Task Parallel Library (TPL), 703
task-related finalization, 717
tasks
  canceling, 718–722
  disposing, 723–724
  results, returning, 709
templates, C++, 442
text, comments, 20–23
ThenBy() method, sorting with OrderBy() method, 566–572
thermostat, 508n1
this keyword, 213–220
this type, avoiding, 759–760
ThreadLocal<T>, 774–775
threads. *See also* multithreading
  controlling, 706–738
  data to and from an alternate, passing, 799–801
  local storage, 774–777
  long-running, 722–723
  managing, 740–742

threads *(contd.)*
multiple. *See* multiple threads
overview of, 703–706
pools, 706, 742–743
running, 706–738
canceling tasks, 718–722
disposing tasks, 723–724
long-running threads, 722–723
unhandled exception handling on `Task`, 715–718
synchronization, 750–777
thread-safe, 752
incrementing and decremeting, 96
`ThreadStaticAttribute`, 775–777
three-dimensional arrays, initializing, 69
three-forward-slash (`///`), 387
throwing exceptions, 406–407
`Throw` statement, 196
`Tic-Tac-Toe`, 869–874
timers, 778–783
time slices, 704
torn reads, 753
`ToString()` method, overriding, 358
`ToUpper()` method, 50
TPL (Task Parallel Library), 703
APMs (Asynchronous Programming Models), calling, 791–796
trapping errors, 187–192
trees, expressions, 498–505
object graphs, 499–501
viewing, 503–505
troubleshooting arrays, 69, 78–80
`true` operator, 373–375
`TryParse()` method, 63
numeric conversion with, 198–199
`Tuple` generic types, 437–438
turning off warning messages (`#pragma`), 142–143
two-dimensional arrays. *See also* arrays
declaring, 68
initializing, 69
`Type` alias, declaring, 164
`typeof` expressions, 654–655
`typeof` type, avoiding, 759–760
types
aliasing, 164–165
anonymous, 245–246
collection interfaces, 536–538
implicit local variables, 54
projection to, 558
base, casting between derived and, 272–273
Boolean, 40–41
categories of, 55–57, 332–339
`Cell`, 427
`char`, 41
checking, 851
comments, 21–22
compatibility between enums, 349–350
conversion without casting, 62
data, 13–14. *See also* data types
delegates, 472–473
parameters, 818–819
decimal, 34–35
definitions, 7–8
delegates, defining, 474–475
encapsulation of, 379–380
enums, defining, 348
exceptions, 193–194
floating-point, 33–34
inequality with, 89–92
special characteristics of, 89
generics, 427–439
nested, 438–439
reflection, 660–662
`Tuple`, 437–438
inferencing, 454–455
inheritance, 205
integers, 32–33
metadata, reflection, 652–662
multiple exception, 405–407
names, methods, 154
numeric, 32–40
parameterized, 427
parameters, 429
`in`, 460–462
determining type of, 660–661
multiple, 436
naming conventions, 431
`out`, 458–460
predefined, 31
references, 56–57, 169–170, 333–336, 465–467
referent, 832, 839
safety, 25, 541, 851
sizes, 752
specializing, 205
`string`, avoiding, 759–760
synchronization, 766–774
`this`, avoiding, 759–760
`typeof`, avoiding, 759–760
underlying
unboxing, 342
verifying, 301
unmanaged, 833
values, 55–56, 169–170, 331, 332
boxing, 339–346
enums, 346–355
inheritance, 338–339
instantiating generics based on, 464–465
interfaces, 338–339
nullable, 425–427
well-formed, 357. *See also* well-formed types

U

UML (Unified Modeling Language), 204, 325n1
unary operators, 373–375
  minus (-), 84–92
  plus (+), 84–92
unboxing, 339, 342
  avoiding, 345
uncertainty, multithreading, 706
unchecked conversions, 59–61, 417–419
underlying types
  unboxing, 342
  verifying, 301
underscore (_), 15
unhandled exceptions
  on `AppDomain`, 744–746
  handling on `Task`, 715–718
Unicode characters, 41–43
Unified Modeling Language. *See* UML
`Union()` standard query operator, 584
unmanaged types, 833
unnecessary locking, avoiding, 765–766
unsafe code, 831–832
unsynchronized states, 750
unwinding stacks, 168
updating `CommandLineHandler.TryParse()` method, 671
uppercase, Java, 9
`using` directive, 161–168
`using` statements, 217n1, 395–398
utilization of resources, 400

V

validation, applying properties, 228–229
values
  calculating, 115
  CTS (Common Type System), 859
  default, specifying, 435–436
  hardcoding, 35–37
  hexidecimal notation, 38
  integers, overflowing, 59
  iterators, yielding, 637–639
  literals, 35, 68
  parameters, 168–169
  permanent, 259
  placeholders, 115
  types, 55–56, 169–170, 331, 332
    boxing, 339–346
    enums, 346–355
    inheritance, 338–339
    instantiating generics based on, 464–465
    interfaces, 338–339
    nullable, 425–427
  variables, modifying, 15
variables
  applying, 12–16
  assigning, 13, 14–16
  declaring, 13, 14
    applying anonymous methods, 481
    of the Class Type, 206
  implicitly typed local, 53–55
  local, 13
    implicitly typed, 538–540
    multiple threads, 753–753
  members, 209
  outer, 495–496
  parameters, defining index operators, 633–634
  values, modifying, 15
variance, applying delegates, 485
`var` keyword, 53
Venn diagrams, 568
verbatim string literals, 44
verifying underlying types, 301
versioning
  interfaces, 327–328
  .NET, 26–27
  serialization, 684–687
VES (Virtual Execution System), 24, 844
viewing expression trees, 503–505
`VirtualAllocEx()` APIs, declaring, 818–819
virtual computers, 816
  detection using P/Invoke, 888–894
Virtual Execution System. *See* VES
virtual fields, properties as, 232–234
`virtual` modifiers, 282–286
Visual Basic
  accessing class instances with `Me` keyword, 214
  global methods, 158
  global variables and functions, 248
  line-based statements, 10
  redimensioning arrays, 75
  `var`, 540
  `Variant`, 540
  `void*`, 540
Visual Basic.NET, project scope `Imports` directive, 162
visual editors, hints for, 144–145
Visual Studio, XML comments in, 386
`void` keyword, 52–55
`volatile`, declaring fields as, 760–761

W

weak references, 391–393
well-formed types, 357
  accessibility modifiers, 381
  assemblies, referencing, 377–381
  garbage collection, 390–393
  lazy initialization, 400–402
  namespaces, defining, 382–385
  `object` members, overriding, 357–369
  operators, overloading, 369–377
  resource cleanup, 393–400
  XML comments, 385–389
`where` clauses, converting expression trees to, 499

`Where()` method, filtering with, 556–557
`while` loops, 121–123
whitespace, formatting, 11–12
wildcards in `import` directive, 162
Win32 APIs, declaring, 818n1
Windows
  Error Reporting dialog box, 715
  executable, 378
  Forms, 809–811
  Presentation Foundation (WPF), 811–813
  UI programming, 809–813
WPF (Windows Presentation Foundation), 811–813
wrappers
  RCW (runtime callable wrapper), 813
  simplifying API calls with, 828–829
write-only properties, 230–231
writing
  comments, 20–23
  output to consoles, 18–20
www.pinvoke.net, 818n1

X

XML (Extensible Markup Language), 22–23
  comments, 385–389
  delimited comments, 22
  documentation files, generating, 388–389
  single-line comments, 22
XOR (exclusive OR) operator, 112

Y

`yield break`, iterators, 645–646
yielding iterator values, 637–639
`yield` keyword, 6n5
`yield returns`, placing in loops, 643–645
`yield` statements, 649