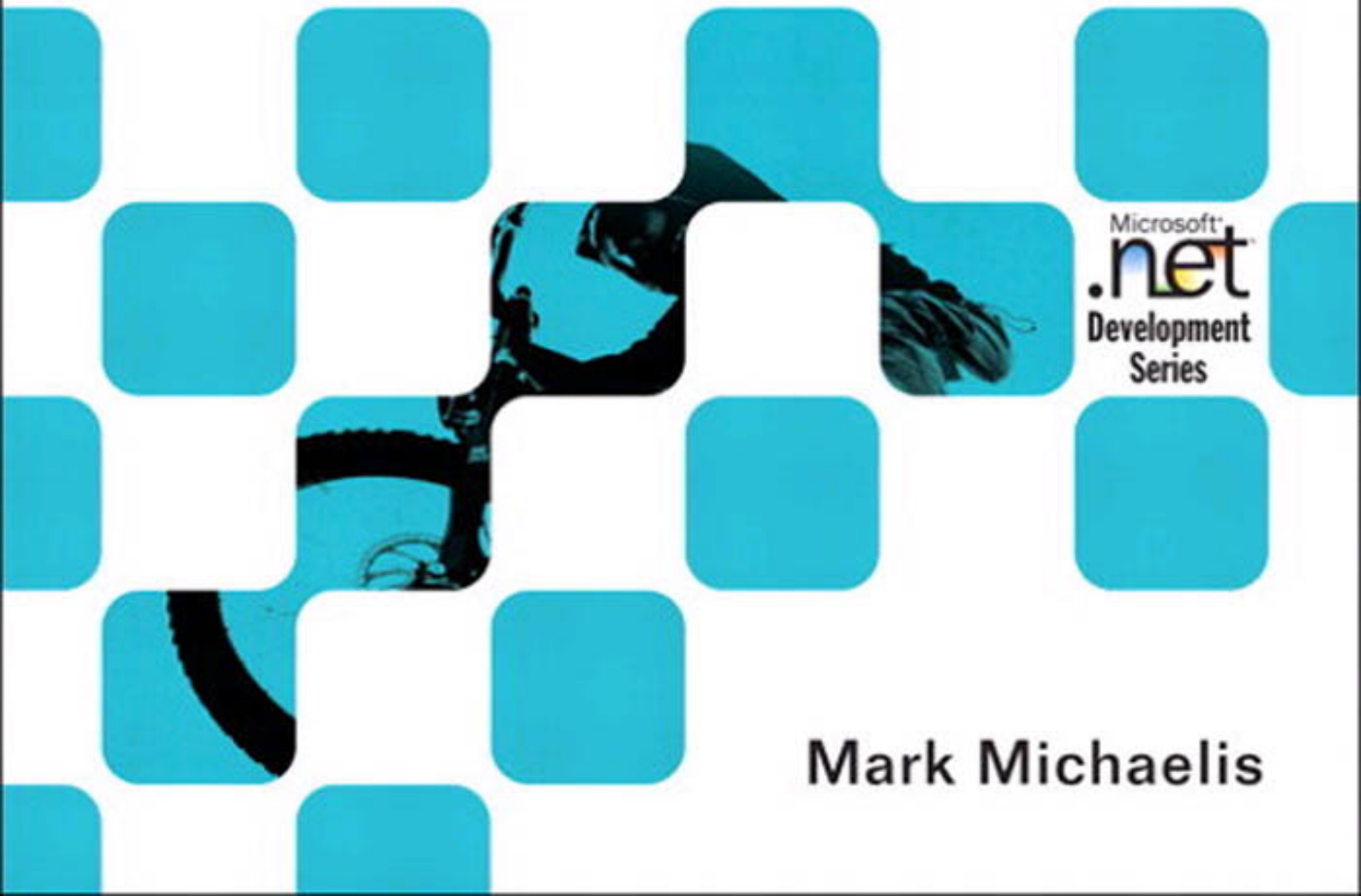
"If you want to be a C# developer, or if you want to enhance your C# programming skills, there is no more useful tool than a well-crafted book on the subject. You are holding such a book in your hands."

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

Essential C# 3.0

For .NET Framework 3.5



Microsoft .net Development Series

Mark Michaelis

Library of Congress Cataloging-in-Publication Data

Michaelis, Mark.
Essential C# 3.0 : for .NET Framework 3.5 / Mark Michaelis.
p. cm.
Includes index.
ISBN 978-0-321-53392-0 (pbk. : alk. paper) 1. C# (Computer program language)
2. Microsoft .NET Framework. I. Title.

QA76.73.C154M5235 2008
006.7'882—dc22

2008023595

ISBN-13: 978-0-321-53392-0
ISBN-10: 0-321-53392-5
Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
First printing, August 2008

Foreword

AS THE COMMUNITY PROGRAM MANAGER 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 every 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.

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, 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 is its coverage of language integrated query (LINQ). For many developers, the declarative style of programming used by LINQ will be a new technology that requires developing new habits and new ways of thinking.

C# 3.0 contains several new features that enable LINQ. One of the main goals of this new edition 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 out this technology for the reader in terms that are clear and easy to understand.

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 the 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 that 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 like 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 thoughts on what is fast becoming the most advanced and most important contemporary programming language.

—*Charlie Calvert,*
Community Program Manager, Visual C#, Microsoft
April 2008

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 collection APIs), and, eventually, rudimentary forms of declarative programming (in Chapter 17) via attributes.

This book has three main functions.

- It provides comprehensive coverage of the C# language, going beyond a tutorial and offering a foundation upon which you can begin effective software development projects.

- For readers already familiar with C#, this book provides insight into some of the more complex programming paradigms and provides in-depth coverage of the features introduced in the latest version of the language, C# 3.0 with .NET 3.5.
- It serves as a timeless reference, even after you gain proficiency with the language.

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

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

Target Audience for This Book

My challenge with this book was how 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 seldom addressed. Most important, 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, some of the most prominent enhancements are
  - Implicitly typed variables (see Chapter 2)
  - Extension methods (see Chapter 5)
  - Partial methods (see Chapter 5)
  - Lambda expressions (see Chapter 12)
  - Expression trees (see Chapter 12)

  - Anonymous types (see Chapter 14)
  - Standard query operators (see Chapter 14)
  - Query expressions (see Chapter 15)
- These topics are covered in detail for those who are 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# 3.0 is a language book that adheres to the core C# Language 3.0 Specification. To help you understand the various C# constructs, it provides numerous examples demonstrating each feature. Accompanying each concept are guidelines and best practices, ensuring that code compiles, avoids likely pitfalls, and achieves maximum maintainability.

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

Code Samples

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

Here is a sample code listing.

Listing 1.17: Commenting Your Code

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

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

```
                                Delimited Comment Inside Statement
        System.Console.Write /* No new line */ (
            "Enter your first name: ");
        firstName = System.Console.ReadLine();

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

        /* Display a greeting to the console                    } Delimited Comment
           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 distract you from learning a particular topic. Therefore, you need to modify the code samples before you can incorporate them into your programs. The core omission is error checking, such as exception handling. Also, code samples do not explicitly include `using System` statements. You need to assume the statement throughout all samples.

You can find sample code at http://mark.michaelis.net/EssentialCSharp.

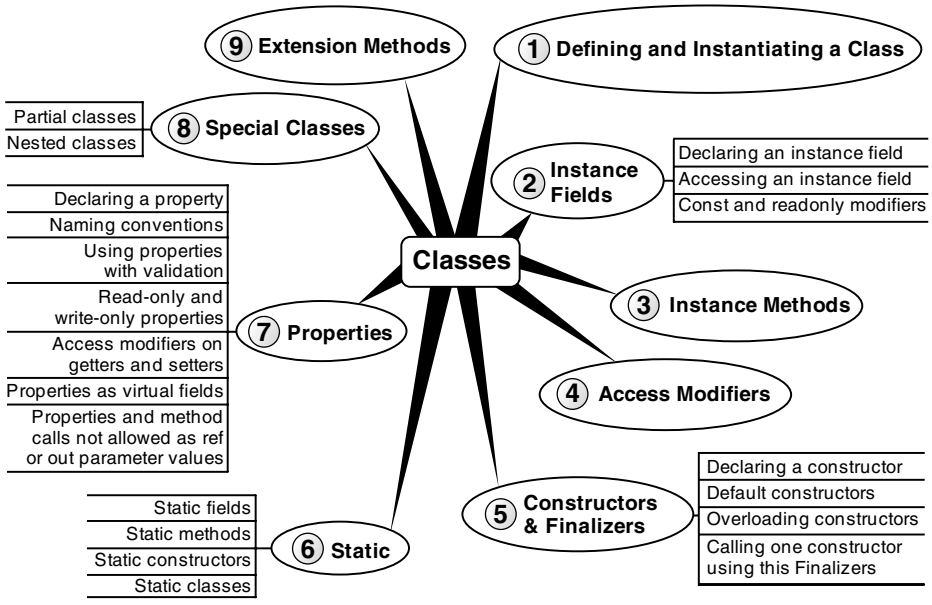
Helpful Notes

Depending on your level of experience, special code blocks and margin notations 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.

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.

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# 3.0.* Chapters 1–4 introduce structured programming, which enables 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, the chapters that follow next.

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. (Asterisks indicate entirely new chapters dedicated to C# 3.0 material. **Bold** chapter titles indicate chapters that contain C# 3.0 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#, default parameters are not supported, and this chapter explains why and how to provide the same functionality.

- ***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 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 new feature.
- ***Chapter 12—Delegates and Lambda Expressions:** Delegates begin to clearly distinguish 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 in Chapter 14.

- *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—Query Expressions:** Using standard query operators alone results in some long statements that are hard to decipher. However, query expressions provide an alternate 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 frequently 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 and Attributes:* 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.
- *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—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 been too long. To make these listings still available to the reader, this appendix includes full listing from Chapters 3, 11, 12, 14, and 17.
- *Appendix C—C# 3.0 Topics:* This appendix provides a quick reference for any C# 3.0 content. It is specifically designed to help C# 2.0 programmers to quickly get up to speed on the 3.0 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.

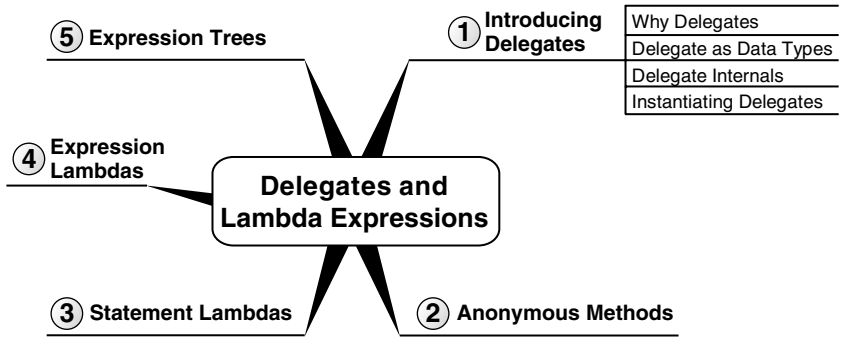
—Mark Michaelis
http://mark.michaelis.net

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 describes 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]

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

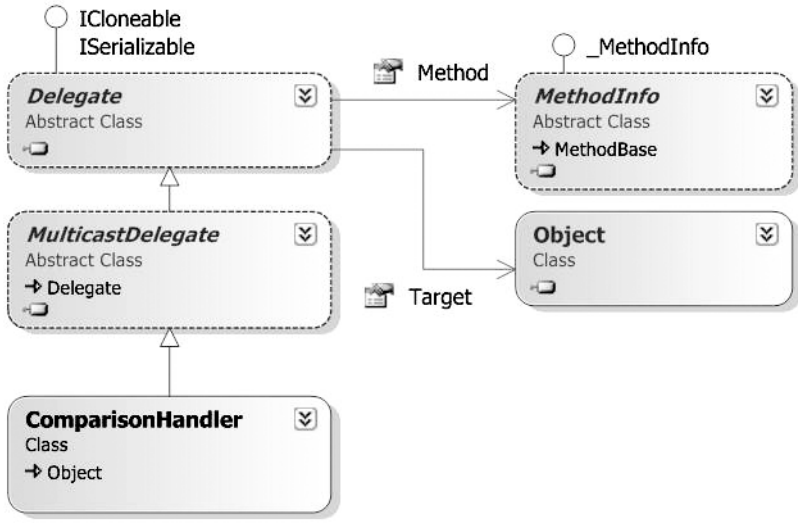


FIGURE 12.1: Delegate Types Object Model

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 `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.

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 rather than explicit instantiation is **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 above 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 above syntax. (This will cause some of the remaining code not to compile on version 1.0 compilers without modification 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 above 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<>

In .NET 3.5 (C# 3.0), there exists a series of generic delegates with the name "Func." The signatures for these delegates are shown in Listing 12.13.

Listing 12.13: Func Delegate Declarations

```
public delegate TResult Func<TResult>();
public delegate TResult Func<T, TResult>(T arg)
public delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2)
public delegate TResult Func<T1, T2, T3, TResult>(
    T1 arg1, T2 arg2, T3 arg3)
public delegate TResult Func<T1, T2, T3, T4, TResult>(
    T1 arg1, T2 arg2, T3 arg3, T4 arg4)
```

Since these delegate definitions 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 into the .NET 3.5 Framework eliminates the necessity to define delegates with four or fewer parameters that return a value. (You should use `System.Action` for delegates that have no return and that take no parameters.)

However, you should still declare delegate types when a specific delegate 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 `Func` generic delegate in place of an explicitly defined delegate, the types are not compatible. You cannot

assign any expression of one delegate type to a variable of another delegate type. For example, you cannot assign a `ComparisonHandler` variable to a `Func<int, int, bool>` variable or pass them interchangeably as parameters.

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.

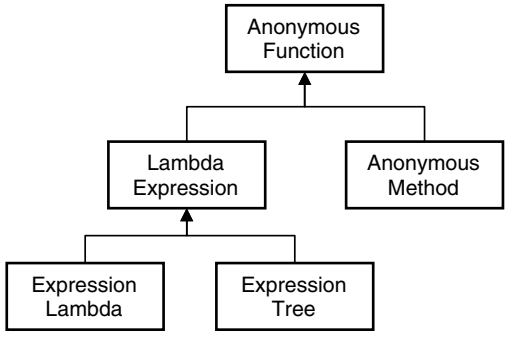


Figure 12.2: Anonymous Function Terminology

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 12.14 shows equivalent functionality to Listing 12.12, except that Listing 12.14 uses a statement lambda rather than an anonymous method.

Listing 12.14: 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 12.14 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 12.15, the delegate data type is `bool Comparison-Handler(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 12.15: 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. 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 12.16).

Listing 12.16: 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 12.17).

Listing 12.17: 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 > 1000000000; });
  // ...
```

(In Listing 12.17, `Where()` returns a query for processes that have a physical memory utilization greater than 1GB.)

Note that back on Listing 12.16, the body of the statement lambda includes multiple statements inside the statement block (via curly braces). Although there can be any number of statements in a statement lambda, typically a statement lambda uses only two or three statements in its statement block. In contrast, the body of an expression lambda does not even make up a full statement since there is no 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 12.18 is the same as Listing 12.14, except that it uses an expression lambda rather than a statement lambda.

Listing 12.18: 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 12.18 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 statements lambda 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 on 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 12.19.

Listing 12.19: 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;
  }

}
```

Outer Variables

Local variables (including parameters) declared outside an anonymous function (such as a lambda expression), but **captured** (accessed) within the lambda expression, are **outer variables** of that anonymous function. `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 12.20, it is relatively trivial to use an outer variable to count how many times `swap` is called by `BubbleSort()`. Output 12.2 shows the results of this listing.

Listing 12.20: Using an Outer Variable in a Lambda Expression

```
class DelegateSample
{

  // ...

  static void Main(string[] args)
  {

      int i;
      int[] items = new int[5];
      int swapCount=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) =>
          {
              bool swap = first < second;
              if(swap)
              {
                  swapCount++;
              }
              return swap;
          }
      );

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

      Console.WriteLine("Items were swapped {0} times.",
                        swapCount);
  }
}
```

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 swapped 4 times.
```

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

As this code demonstrates, the C# compiler takes care of generating CIL code that shares `swapCount` between the anonymous method and the call site, even though there is no parameter to pass `swapCount` 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 goes out of scope.

ADVANCED TOPIC

Outer Variable Internals

The CIL code generated by the C# compiler for outer variables is more complex than the code for a simple anonymous method, because the outer variable must be captured in a thread-safe manner. Listing 12.21 shows the C# equivalent of the CIL code used to implement outer variables.

Listing 12.21: C# Equivalent of CIL Code Generated by Compiler for Outer Variables

```
class DelegateSample
{

  // ...

  private sealed class __LocalsDisplayClass_00000001
  {
      public int swapCount;
      public bool __AnonymousMethod_00000000(
          int first, int second)
```

```
        {
            bool swap = first < second;

            if (swap)
            {
                swapCount++;
            }

            return swap;
        }
    }

    ...

    static void Main(string[] args)
    {

        int i;
        LocalsDisplayClass_00000001 locals =
            new __LocalsDisplayClass_00000001();
        locals.swapCount=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 swapped {0} times.",
                          locals.swapCount);
    }
}
```

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

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.

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<string, bool>` delegate type and then passes the delegate instance to the `Where()` method. Code and execution like this work 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 12.22).

Listing 12.22: 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.

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, whereas an expression tree is compiled into a data structure of type `System.Linq.Expressions.Expression`. As a result, when a lambda expression is an expression lambda, it may execute—it is CIL instructions for what the runtime should do. However, if the lambda expression is an expression tree, it is not a set of CIL instructions, but rather a data structure. Although an expression tree includes a method that will compile it into a delegate constructor call, it is more likely that the expression tree (data) will be converted into a different format or set of instructions.

`System.Linq.Enumerable` versus `System.Linq.Queryable`

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 12.22. 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 12.23, 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 12.23: 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 12.23 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 that 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, 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, statement lambdas and expression lambdas are both types of lambda expressions.

One thing 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

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z