O'REILLY®

6th Edition
Covers .NET 4.6 & the Roslyn Compiler

C# 6.0 in a Nutshell

THE DEFINITIVE REFERENCE

Joseph Albahari & Ben Albahari

C# 6.0

in a Nutshell

Joseph Albahari & Ben Albahari

Beijing • Boston • Farnham • Sebastopol • Tokyo

C# 6.0 in a Nutshell

by Joseph Albahari and Ben Albahari

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com.*

- Editor: Brian MacDonald
- Production Editor: Kristen Brown
- Proofreader: Amanda Kersey
- Indexer: Angela Howard
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Rebecca Demarest
- December 2015: Sixth Edition

Revision History for the Sixth Edition

- 2015-11-03: First Release
- 2015-12-18: Second Release
- 2016-04-01: Third Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491927069* for release details.

978-1-491-92706-9

[M]

Preface

C# 6.0 represents the fifth major update to Microsoft's flagship programming language, positioning C# as a language with unusual flexibility and breadth. At one end, it offers high-level abstractions such as query expressions and asynchronous continuations; while at the other end, it allows low-level efficiency through constructs such as custom value types and the optional use of pointers.

The price of this growth is that there's more than ever to learn. Although tools such as Microsoft's IntelliSense—and online references—are excellent in helping you on the job, they presume an existi map of conceptual knowledge. This book provides exactly that map of knowledge in a concise and unified style—free of clutter and long introductions.

Like the past three editions, *C# 6.0 in a Nutshell* is organized around concepts and use cases, making friendly both to sequential reading and to random browsing. It also plumbs significant depths while assuming only basic background knowledge—making it accessible to intermediate as well as advanced readers.

This book covers C#, the CLR, and the core Framework assemblies. We've chosen this focus to allov space for difficult topics such as concurrency, security, and application domains—without compromising depth or readability. Features new to C# 6.0 and the associated Framework are flagge so that you can also use this book as a C# 5.0 reference.

Intended Audience

This book targets intermediate to advanced audiences. No prior knowledge of C# is required, but som general programming experience is necessary. For the beginner, this book complements, rather than replaces, a tutorial-style introduction to programming.

If you're already familiar with C# 5.0, you'll find updated language sections, and a new chapter on "Roslyn," the compiler-as-a-service.

This book is an ideal companion to any of the vast array of books that focus on an applied technology such as WPF, ASP.NET, or WCF. The areas of the language and .NET Framework that such books omit, *C# 6.0 in a Nutshell* covers in detail—and vice versa.

If you're looking for a book that skims every .NET Framework technology, this is not for you. This book is also unsuitable if you want to learn about APIs specific to tablet or Windows Phone development.

How This Book Is Organized

The first three chapters after the introduction concentrate purely on C#, starting with the basics of syntax, types, and variables, and finishing with advanced topics such as unsafe code and preprocesso directives. If you're new to the language, you should read these chapters sequentially.

The remaining chapters cover the core .NET Framework, including such topics as LINQ, XML, collections, code contracts, concurrency, I/O and networking, memory management, reflection, dynamic programming, attributes, security, application domains, and native interoperability. You ca read most of these chapters randomly, except for Chapters 6 and 7, which lay a foundation for subsequent topics. The three chapters on LINQ are also best read in sequence, and some chapters assume some knowledge of concurrency, which we cover in Chapter 14.

What You Need to Use This Book

The examples in this book require a C# 6.0 compiler and Microsoft .NET Framework 4.6. You will also find Microsoft's .NET documentation useful to look up individual types and members (which is available online).

While it's possible to write source code in Notepad and invoke the compiler from the command line, you'll be much more productive with a *code scratchpad* for instantly testing code snippets, plus an *integrated development environment* (IDE) for producing executables and libraries.

For a code scratchpad, download LINQPad 5 or later from *http://www.linqpad.net* (free). LINQPad fully supports C# 6.0 and is maintained by one of the authors.

For an IDE, download Microsoft Visual Studio 2015: any edition, except the free express edition, is suitable for what's taught in this book.

> **NOTE**
>
> All code listings for Chapters 2 through 10, plus the chapters on concurrency, parallel programming, and dynamic programming are available as interactive (editable) LINQPad samples. You can download the whole lot in a single click: go to LINQPad's Sample Libraries page and choose "C# 6.0 in a Nutshell."

Conventions Used in This Book

The book uses basic UML notation to illustrate relationships between types, as shown in Figure P-1. slanted rectangle means an abstract class; a circle means an interface. A line with a hollow triangle denotes inheritance, with the triangle pointing to the base type. A line with an arrow denotes a one-way association; a line without an arrow denotes a two-way association.

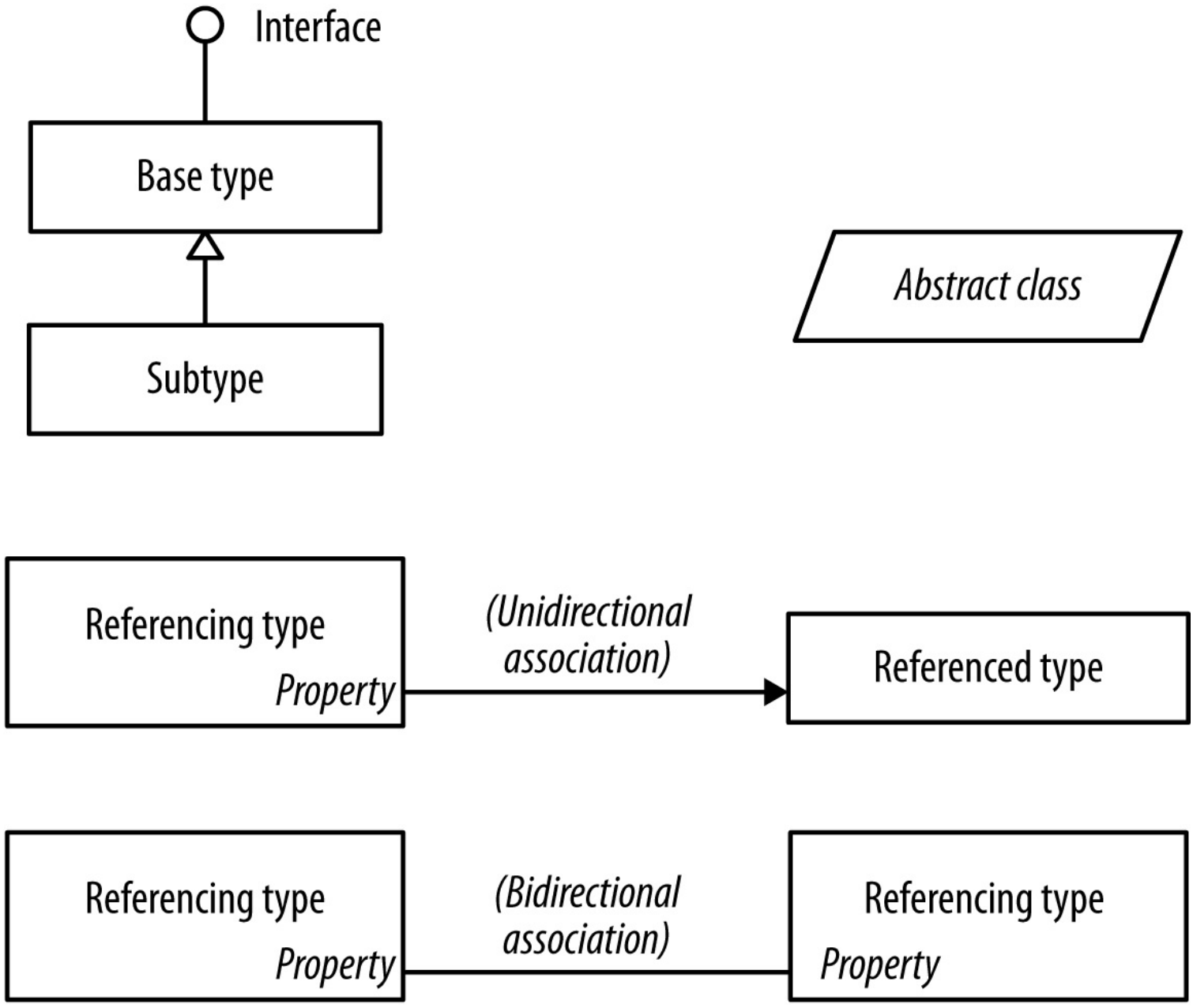

Figure P-1. Sample diagram

The following typographical conventions are used in this book:

Italic

Indicates new terms, URIs, filenames, and directories

`Constant width`

Indicates C# code, keywords and identifiers, and program output

`Constant width bold`

Shows a highlighted section of code

`Constant width italic`

Shows text that should be replaced with user-supplied values

TIP

This element signifies a tip or suggestion.

NOTE

This element signifies a general note.

WARNING

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at LINQPad's Sample Libraries page: choose "C# 6.0 in a Nutshell."

This book is here to help you get your job done. In general, if example code is offered with this book you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*C# 6.0 in a Nutshell* by Joseph Albahari and Ben Albahari (O'Reilly). Copyright 2016 Joseph Albahari and Ben Albahari, 978-1-491-92706-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and

individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

- O'Reilly Media, Inc.
- 1005 Gravenstein Highway North
- Sebastopol, CA 95472
- 800-998-9938 (in the United States or Canada)
- 707-829-0515 (international or local)
- 707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/c-sharp6_nutshell*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Joseph Albahari

First, I want to thank my brother, Ben Albahari, for persuading me to take on *C# 3.0 in a Nutshell*, whose success has spawned three subsequent editions. Ben shares my willingness to question conventional wisdom and tenacity to pull things apart until it becomes clear how they *really* work.

It's been an honor to have superb technical reviewers on the team. In this edition, we had invaluable and extensive feedback from Jared Parsons, Stephen Toub, Matthew Groves, Dixin Yan, Lee Coward

Bonnie DeWitt, Wonseok Chae, Lori Lalonde and James Montemagno.

The book was built on previous editions, whose technical reviewers I owe a similar honor: Eric Lippert, Jon Skeet, Stephen Toub, Nicholas Paldino, Chris Burrows, Shawn Farkas, Brian Grunkemeyer, Maoni Stephens, David DeWinter, Mike Barnett, Melitta Andersen, Mitch Wheat, Brian Peek, Krzysztof Cwalina, Matt Warren, Joel Pobar, Glyn Griffiths, Ion Vasilian, Brad Abrams, Sam Gentile, and Adam Nathan.

I appreciate that many of the technical reviewers are accomplished individuals at Microsoft, and I particularly thank you for taking out time to raise this book to the next quality bar.

Finally, I want to thank the O'Reilly team, including my best ever editor, Brian MacDonald, and extend personal thanks to Miri and Sonia.

Ben Albahari

Because my brother wrote his acknowledgments first, you can infer most of what I want to say. :) We've actually both been programming since we were kids (we shared an Apple IIe; he was writing his own operating system while I was writing Hangman), so it's cool that we're now writing books together. I hope the enriching experience we had writing the book will translate into an enriching experience for you reading the book.

I'd also like to thank my former colleagues at Microsoft. Many smart people work there, not just in terms of intellect but also in a broader emotional sense, and I miss working with them. In particular, learned a lot from Brian Beckman, to whom I am indebted.

Chapter 1. Introducing C# and the .NET Framework

C# is a general-purpose, type-safe, object-oriented programming language. The goal of the language programmer productivity. To this end, the language balances simplicity, expressiveness, and performance. The chief architect of the language since its first version is Anders Hejlsberg (creator o Turbo Pascal and architect of Delphi). The C# language is platform-neutral, but it was written to wor well with the Microsoft .NET Framework.

Object Orientation

C# is a rich implementation of the object-orientation paradigm, which includes *encapsulation*, *inheritance*, and *polymorphism*. Encapsulation means creating a boundary around an *object*, to separate its external (public) behavior from its internal (private) implementation details. The distinctive features of C# from an object-oriented perspective are:

Unified type system
: The fundamental building block in C# is an encapsulated unit of data and functions called a *type*. C# has a *unified type system*, where all types ultimately share a common base type. This means that all types, whether they represent business objects or are primitive types such as numbers, share the same basic set of functionality. For example, an instance of any type can be converted t a string by calling its `ToString` method.

Classes and interfaces
: In a traditional object-oriented paradigm, the only kind of type is a class. In C#, there are several other kinds of types, one of which is an *interface*. An interface is like a class, except that it only *describes* members. The implementation for those members comes from types that *implement* the interface. Interfaces are particularly useful in scenarios where multiple inheritance is required (unlike languages such as C++ and Eiffel, C# does not support multiple inheritance of classes).

Properties, methods, and events
: In the pure object-oriented paradigm, all functions are *methods* (this is the case in Smalltalk). In C#, methods are only one kind of *function member*, which also includes *properties* and *events* (there are others, too). Properties are function members that encapsulate a piece of an object's state, such as a button's color or a label's text. Events are function members that simplify acting on object state changes.

While C# is primarily an object-oriented language, it also borrows from the *functional programming* paradigm. Specifically:

Functions can be treated as values
: Through the use of *delegates*, C# allows functions to be passed as values to and from other

functions.

C# supports patterns for purity

Core to functional programming is avoiding the use of variables whose values change, in favor of declarative patterns. C# has key features to help with those patterns, including the ability to write unnamed functions on the fly that "capture" variables (*lambda expressions*) and the ability to perform list or reactive programming via *query expressions*. C# 6.0 also includes read-only auto-properties to help with writing *immutable* (read-only) types.

Type Safety

C# is primarily a *type-safe* language, meaning that instances of types can interact only through protocols they define, thereby ensuring each type's internal consistency. For instance, C# prevents yo from interacting with a *string* type as though it were an *integer* type.

More specifically, C# supports *static typing*, meaning that the language enforces type safety at *compile time*. This is in addition to type safety being enforced at *runtime*.

Static typing eliminates a large class of errors before a program is even run. It shifts the burden away from runtime unit tests onto the compiler to verify that all the types in a program fit together correctly. This makes large programs much easier to manage, more predictable, and more robust. Furthermore, static typing allows tools such as IntelliSense in Visual Studio to help you write a program, since it knows for a given variable what type it is, and hence what methods you can call on that variable.

> **NOTE**
>
> C# also allows parts of your code to be dynamically typed via the `dynamic` keyword (introduced in C# 4.0). However, C# remains a predominantly statically typed language.

C# is also called a *strongly typed language* because its type rules (whether enforced statically or at runtime) are very strict. For instance, you cannot call a function that's designed to accept an integer with a floating-point number, unless you first *explicitly* convert the floating-point number to an integer. This helps prevent mistakes.

Strong typing also plays a role in enabling C# code to run in a sandbox—an environment where every aspect of security is controlled by the host. In a sandbox, it is important that you cannot arbitrarily corrupt the state of an object by bypassing its type rules.

Memory Management

C# relies on the runtime to perform automatic memory management. The Common Language Runtime has a garbage collector that executes as part of your program, reclaiming memory for objec that are no longer referenced. This frees programmers from explicitly deallocating the memory for a

object, eliminating the problem of incorrect pointers encountered in languages such as C++.

C# does not eliminate pointers: it merely makes them unnecessary for most programming tasks. For performance-critical hotspots and interoperability, pointers may be used, but they are permitted only in blocks that are explicitly marked `unsafe`.

Platform Support

Historically, C# was used almost entirely for writing code to run on Windows platforms. Recently, however, Microsoft and other companies have invested in other platforms, including Mac OS X and iOS, and Android. Xamarin™ allows cross-platform C# development for mobile applications, and Portable Class Libraries are becoming increasingly widespread. Microsoft's ASP.NET 5 is a new wel hosting framework that can run either on the .NET Framework or on .NET Core, a new small, fast, open source, cross-platform runtime.

C#'s Relationship with the CLR

C# depends on a runtime equipped with a host of features such as automatic memory management an exception handling. The design of C# closely maps to the design of Microsoft's *Common Language Runtime* (CLR), which provides these runtime features (although C# is technically independent of the CLR). Furthermore, the C# type system maps closely to the CLR type system (e.g., both share the same definitions for predefined types).

The CLR and .NET Framework

The .NET Framework consists of the CLR plus a vast set of libraries. The libraries consist of core libraries (which this book is concerned with) and applied libraries, which depend on the core librarie Figure 1-1 is a visual overview of those libraries (and also serves as a navigational aid to the book).

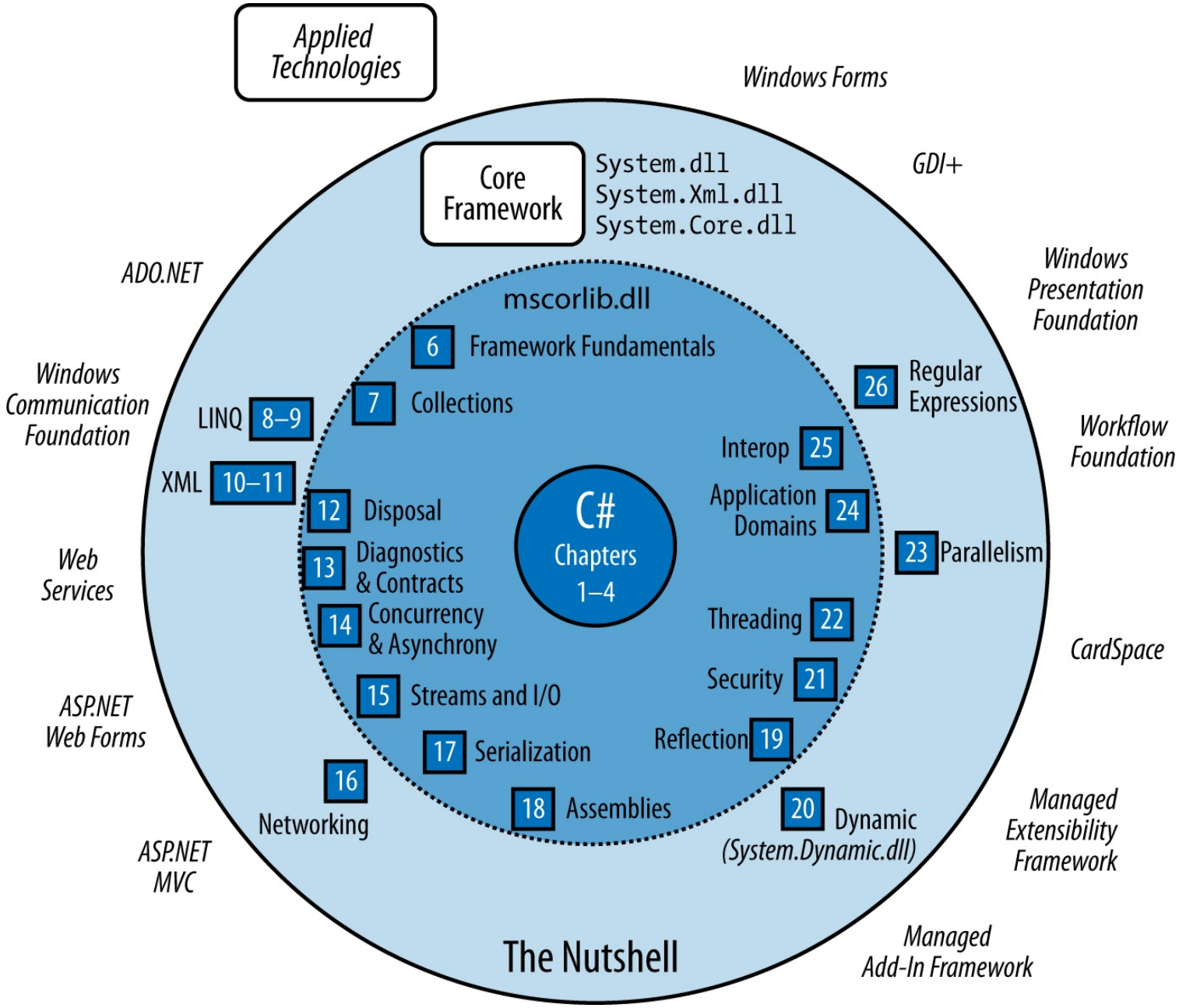

Figure 1-1. Topics covered in this book and the chapters in which they are found. Topics not covered are shown outside the large circle.

The CLR is the runtime for executing *managed code*. C# is one of several *managed languages* that ge compiled into managed code. Managed code is packaged into an *assembly*, in the form of either an executable file (an *.exe*) or a library (a *.dll*), along with type information, or *metadata*.

Managed code is represented in *intermediate language* or *IL*. When the CLR loads an assembly, it converts the IL into the native code of the machine, such as x86. This conversion is done by the CLR JIT (just-in-time) compiler. An assembly retains almost all of the original source language construct which makes it easy to inspect and even generate code dynamically.

NOTE

You can examine and decompile the contents of an IL assembly with tools such as ILSpy, dotPeek (JetBrains) or Reflector (Red Gate).

When writing Windows Store apps, you also now have the option of generating native code directly (".NET Native"). This improves startup performance and memory usage (which is particularly beneficial on mobile devices) and also runtime performance through static linking and other optimizations.

The CLR performs as a host for numerous runtime services. Examples of these services include memory management, the loading of libraries, and security services. The CLR is language-neutral, allowing developers to build applications in multiple languages (e.g., C#, F#, Visual Basic .NET and Managed C++).

The .NET Framework contains libraries for writing just about any Windows- or web-based application. Chapter 5 gives an overview of the .NET Framework libraries.

C# and Windows Runtime

C# also interoperates with *Windows Runtime* (WinRT) libraries. WinRT is an execution interface and runtime environment for accessing libraries in a language-neutral and object-oriented fashion. It ship with Windows 8 and newer and is (in part) an enhanced version of Microsoft's *Component Object Model* or COM (see Chapter 25).

Windows 8 and newer ship with a set of unmanaged WinRT libraries that serve as a framework for touch-enabled applications delivered through Microsoft's application store. (The term *WinRT* also refers to these libraries.) Being WinRT, the libraries can easily be consumed not only from C# and VB, but C++ and JavaScript.

> **WARNING**
>
> Some WinRT libraries can also be consumed in normal non-tablet applications. However, taking a dependency on WinRT gives your application a *minimum* OS requirement of Windows 8.

The WinRT libraries support the new "modern" user interface (for writing immersive touch-first applications), mobile device-specific features (sensors, text messaging and so on), and a range of cor functionality that overlaps with parts of the .NET Framework. Because of this overlap, Visual Studio includes a *reference profile* (a set of .NET *reference assemblies*) for Windows Store projects that hides the portions of the .NET Framework that overlap with WinRT. This profile also hides large portions of the .NET Framework considered unnecessary for tablet apps (such as accessing a database). Microsoft's application store, which controls the distribution of software to consumer devices, rejects any program that attempts to access a hidden type.

NOTE

A *reference assembly* exists purely to compile against and may have a restricted set of types and members. This allows developers to install the full .NET Framework on their machines while coding certain projects as though they had only a subset. The actual functionality comes at runtime from assemblies in the *global assembly cache* (see Chapter 18) that may superset the reference assemblies.

Hiding most of the .NET Framework eases the learning curve for developers new to the Microsoft platform, although there are two more important goals:

- It *sandboxes* applications (restricts functionality to reduce the impact of malware). For instance, arbitrary file access is forbidden, and there the ability to start or communicate with other program on the computer is extremely restricted.
- It allows low-powered Windows RT-only tablets to ship with a reduced .NET Framework, lowerin the OS footprint.

What distinguishes WinRT from ordinary COM is that WinRT *projects* its libraries into a multitude languages, namely C#, VB, C++ and JavaScript, so that each language sees WinRT types (almost) as though they were written especially for it. For example, WinRT will adapt capitalization rules to suit the standards of the target language, and will even remap some functions and interfaces. WinRT assemblies also ship with rich *metadata* in *.winmd* files, which have the same format as .NET assembly files, allowing transparent consumption without special ritual. In fact, you might even be unaware that you're using WinRT rather than .NET types, aside of namespace differences. Another clue is that WinRT types are subject to COM-style restrictions; for instance, they offer limited suppo for inheritance and generics.

NOTE

WinRT does not supersede the full .NET Framework. The latter is still recommended (and necessary) for standard desktop and server-side development, and has the following advantages:

- Programs are not restricted to running in a sandbox.
- Programs can use the entire .NET Framework and any third-party library.
- Application distribution does not rely on the Windows Store.
- Applications can target the latest Framework version without requiring users to have the latest OS version.

What's New in C# 6.0

C# 6.0's biggest new feature is that the compiler has been completely rewritten in C#. Known as project "Roslyn," the new compiler exposes the entire compilation pipeline via libraries, allowing yo to perform code analysis on arbitrary source code (see Chapter 27). The compiler itself is open sourc and the source code is available at *github.com/dotnet/roslyn*.

In addition, C# 6.0 features a number of minor but significant enhancements, aimed primarily at reducing code clutter.

The *null-conditional* ("Elvis") operator (see "Null Operators", Chapter 2) avoids having to explicitly check for null before calling a method or accessing a type member. In the following example, `result` evaluates to null instead of throwing a `NullReferenceException`:

```
System.Text.StringBuilder sb = null;
string result = sb?.ToString();      // result is null
```

Expression-bodied functions (see "Methods", Chapter 3) allow methods, properties, operators, and indexers that comprise a single expression to be written more tersely, in the style of a lambda expression:

```
public int TimesTwo (int x) => x * 2;
public string SomeProperty => "Property value";
```

Property initializers (Chapter 3) let you assign an initial value to an automatic property:

```
public DateTime Created { get; set; } = DateTime.Now;
```

Initialized properties can also be read-only:

```
public DateTime Created { get; } = DateTime.Now;
```

Read-only properties can also be set in the constructor, making it easier to create immutable (read-only) types.

Index initializers (Chapter 4) allow single-step initialization of any type that exposes an indexer:

```
new Dictionary<int,string>()
{
  [3] = "three",
  [10] = "ten"
}
```

String interpolation (see "String Type", Chapter 2) offers a succinct alternative to `string.Format`:

```
string s = $"It is {DateTime.Now.DayOfWeek} today";
```

Exception filters (see "try Statements and Exceptions", Chapter 4) let you apply a condition to a catch block:

```
try
{
  new WebClient().DownloadString("http://asef");
}
catch (WebException ex) when (ex.Status == WebExceptionStatus.Timeout)
```

```
{
  ...
}
```

The `using static` (see "Namespaces", Chapter 2) directive lets you import all the static members o a type, so that you can use those members unqualified:

```
using static System.Console;
...
WriteLine ("Hello, world");  // WriteLine instead of Console.WriteLine
```

The `nameof` (Chapter 3) operator returns the name of a variable, type or other symbol as a string. Thi avoids breaking code when you rename a symbol in Visual Studio:

```
int capacity = 123;
string x = nameof (capacity);   // x is "capacity"
string y = nameof (Uri.Host);   // y is "Host"
```

And finally, you're now allowed to `await` inside `catch` and `finally` blocks.

What Was New in C# 5.0

C# 5.0's big new feature was support for *asynchronous functions* via two new keywords, `async` and `await`. Asynchronous functions enable *asynchronous continuations*, which make it easier to write responsive and thread-safe, rich-client applications. They also make it easy to write highly concurren and efficient I/O-bound applications that don't tie up a thread resource per operation.

We cover asynchronous functions in detail in Chapter 14.

What Was New in C# 4.0

The features new to C# 4.0 were:

- Dynamic binding
- Optional parameters and named arguments
- Type variance with generic interfaces and delegates
- COM interoperability improvements

Dynamic binding (Chapters 4 and 20) defers *binding*—the process of resolving types and members—from compile time to runtime and is useful in scenarios that would otherwise require complicated reflection code. Dynamic binding is also useful when interoperating with dynamic languages and COM components.

Optional parameters (Chapter 2) allow functions to specify default parameter values so that callers

can omit arguments, and *named arguments* allow a function caller to identify an argument by name rather than position.

Type variance rules were relaxed in C# 4.0 (Chapters 3 and 4), such that type parameters in generic interfaces and generic delegates can be marked as *covariant* or *contravariant*, allowing more natural type conversions.

COM interoperability (Chapter 25) was enhanced in C# 4.0 in three ways. First, arguments can be passed by reference without the `ref` keyword (particularly useful in conjunction with optional parameters). Second, assemblies that contain COM interop types can be *linked* rather than *referenced*. Linked interop types support type equivalence, avoiding the need for *Primary Interop Assemblies* and putting an end to versioning and deployment headaches. Third, functions that return COM-Variant types from linked interop types are mapped to `dynamic` rather than `object`, eliminating the need for casting.

What Was New in C# 3.0

The features added to C# 3.0 were mostly centered on *Language Integrated Query* capabilities, or *LINQ* for short. LINQ enables queries to be written directly within a C# program and checked *statically* for correctness, and to query both local collections (such as lists or XML documents) or remote data sources (such as a database). The C# 3.0 features added to support LINQ comprised implicitly typed local variables, anonymous types, object initializers, lambda expressions, extension methods, query expressions, and expression trees.

Implicitly typed local variables (`var` keyword, Chapter 2) let you omit the variable type in a declaration statement, allowing the compiler to infer it. This reduces clutter as well as allowing *anonymous types* (Chapter 4), which are simple classes created on the fly that are commonly used in the final output of LINQ queries. Arrays can also be implicitly typed (Chapter 2).

Object initializers (Chapter 3) simplify object construction by allowing properties to be set inline after the constructor call. Object initializers work with both named and anonymous types.

Lambda expressions (Chapter 4) are miniature functions created by the compiler on the fly and are particularly useful in "fluent" LINQ queries (Chapter 8).

Extension methods (Chapter 4) extend an existing type with new methods (without altering the type's definition), making static methods feel like instance methods. LINQ's query operators are implemented as extension methods.

Query expressions (Chapter 8) provide a higher-level syntax for writing LINQ queries that can be substantially simpler when working with multiple sequences or range variables.

Expression trees (Chapter 8) are miniature code DOMs (Document Object Models) that describe lambda expressions assigned to the special type `Expression<TDelegate>`. Expression trees make it possible for LINQ queries to execute remotely (e.g., on a database server) because they can be introspected and translated at runtime (e.g., into a SQL statement).

C# 3.0 also added automatic properties and partial methods.

Automatic properties (Chapter 3) cut the work in writing properties that simply `get`/`set` a private backing field by having the compiler do that work automatically. *Partial methods* (Chapter 3) let an auto-generated partial class provide customizable hooks for manual authoring which "melt away" if unused.

Chapter 2. C# Language Basics

In this chapter, we introduce the basics of the C# language.

> **NOTE**
>
> All programs and code snippets in this and the following two chapters are available as interactive samples in LINQPad. Working through these samples in conjunction with the book accelerates learning in that you can edit the samples and instantly see the results without needing to set up projects and solutions in Visual Studio.
>
> To download the samples, go to LINQPad's Sample Libraries page and choose "C# 6.0 in a Nutshell." LINQPad is free—go to *http://www.linqpad.net*.

A First C# Program

Here is a program that multiplies 12 by 30 and prints the result, 360, to the screen. The double forwa slash indicates that the remainder of a line is a *comment*:

```
using System;                     // Importing namespace

class Test                        // Class declaration
{
  static void Main()              // Method declaration
  {
    int x = 12 * 30;              // Statement 1
    Console.WriteLine (x);        // Statement 2
  }                               // End of method
}                                 // End of class
```

At the heart of this program lie two *statements*:

```
int x = 12 * 30;
Console.WriteLine (x);
```

Statements in C# execute sequentially and are terminated by a semicolon (or a *code block*, as we'll s later). The first statement computes the *expression* `12 * 30` and stores the result in a *local variable*, named `x`, which is an integer type. The second statement calls the `Console` class's `WriteLine` *metho* to print the variable `x` to a text window on the screen.

A *method* performs an action in a series of statements, called a *statement block*—a pair of braces containing zero or more statements. We defined a single method named `Main`:

```
static void Main()
{
  ...
}
```

Writing higher-level functions that call upon lower-level functions simplifies a program. We can *refactor* our program with a reusable method that multiplies an integer by 12 as follows:

```
using System;

class Test
{
  static void Main()
  {
    Console.WriteLine (FeetToInches (30));      // 360
    Console.WriteLine (FeetToInches (100));     // 1200
  }

  static int FeetToInches (int feet)
  {
    int inches = feet * 12;
    return inches;
  }
}
```

A method can receive *input* data from the caller by specifying *parameters* and *output* data back to the caller by specifying a *return type*. We defined a method called `FeetToInches` that has a parameter for inputting feet, and a return type for outputting inches:

```
static int FeetToInches (int feet ) {...}
```

The *literals* `30` and `100` are the *arguments* passed to the `FeetToInches` method. The `Main` method in our example has empty parentheses because it has no parameters, and is `void` because it doesn't return any value to its caller:

```
static void Main()
```

C# recognizes a method called `Main` as signaling the default entry point of execution. The `Main` method may optionally return an integer (rather than `void`) in order to return a value to the execution environment (where a nonzero value typically indicates an error). The `Main` method can also optionally accept an array of strings as a parameter (that will be populated with any arguments passed to the executable). For example:

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

> **NOTE**
>
> An array (such as `string[]`) represents a fixed number of elements of a particular type. Arrays are specified by placing square brackets after the element type and are described in "Arrays".

Methods are one of several kinds of functions in C#. Another kind of function we used in our example

program was the * *operator*, which performs multiplication. There are also *constructors*, *properties*, *events*, *indexers*, and *finalizers*.

In our example, the two methods are grouped into a class. A *class* groups function members and data members to form an object-oriented building block. The `Console` class groups members that handle command-line input/output functionality, such as the `WriteLine` method. Our `Test` class groups two methods—the `Main` method and the `FeetToInches` method. A class is a kind of *type*, which we will examine in "Type Basics".

At the outermost level of a program, types are organized into *namespaces*. The `using` directive was used to make the `System` namespace available to our application, to use the `Console` class. We could define all our classes within the `TestPrograms` namespace, as follows:

```
using System;

namespace TestPrograms
{
  class Test  {...}
  class Test2 {...}
}
```

The .NET Framework is organized into nested namespaces. For example, this is the namespace that contains types for handling text:

```
using System.Text;
```

The `using` directive is there for convenience; you can also refer to a type by its fully qualified name, which is the type name prefixed with its namespace, such as `System.Text.StringBuilder`.

Compilation

The C# compiler compiles source code, specified as a set of files with the *.cs* extension, into an *assembly*. An assembly is the unit of packaging and deployment in .NET. An assembly can be either an *application* or a *library*. A normal console or Windows application has a `Main` method and is an *.exe* file. A library is a *.dll* and is equivalent to an *.exe* without an entry point. Its purpose is to be called upon (*referenced*) by an application or by other libraries. The .NET Framework is a set of libraries.

The name of the C# compiler is *csc.exe*. You can either use an IDE such as Visual Studio to compile, or call `csc` manually from the command line. (The compiler is also available as a library; see Chapter 27.) To compile manually, first save a program to a file such as *MyFirstProgram.cs*, and then go to the command line and invoke `csc` (located in *%ProgramFiles(X86)%\msbuild\14.0\bin*) as follows:

```
csc MyFirstProgram.cs
```

This produces an application named *MyFirstProgram.exe*.

WARNING

Peculiarly, .NET Framework 4.6 ships with the C# 5 compiler. To obtain the C# 6 command-line compiler, you must install Visual Studio or MSBuild 14.

To produce a library (*.dll*), do the following:

```
csc /target:library MyFirstProgram.cs
```

We explain assemblies in detail in Chapter 18.

Syntax

C# syntax is inspired by C and C++ syntax. In this section, we will describe C#'s elements of syntax, using the following program:

```
using System;

class Test
{
  static void Main()
  {
    int x = 12 * 30;
    Console.WriteLine (x);
  }
}
```

Identifiers and Keywords

Identifiers are names that programmers choose for their classes, methods, variables, and so on. These are the identifiers in our example program, in the order they appear:

```
System   Test   Main   x   Console   WriteLine
```

An identifier must be a whole word, essentially made up of Unicode characters starting with a letter or underscore. C# identifiers are case-sensitive. By convention, parameters, local variables, and private fields should be in camel case (e.g., `myVariable`), and all other identifiers should be in Pascal case (e.g., `MyMethod`).

Keywords are names that mean something special to the compiler. These are the keywords in our example program:

```
using   class   static   void   int
```

Most keywords are *reserved*, which means that you can't use them as identifiers. Here is the full list of C# reserved keywords:

```
abstract do       in        protected  true
as       double   int       public     try
base     else     interface readonly   typeof
bool     enum     internal  ref        uint
break    event    is        return     ulong
byte     explicit lock      sbyte      unchecked
case     extern   long      sealed     unsafe
catch    false    namespace short      ushort
char     finally  new       sizeof     using
checked  fixed    null      stackalloc virtual
class    float    object    static     void
const    for      operator  string     volatile
continue foreach  out       struct     while
decimal  goto     override  switch
default  if       params    this
delegate implicit private   throw
```

Avoiding conflicts

If you really want to use an identifier that clashes with a reserved keyword, you can do so by qualifying it with the `@` prefix. For instance:

```
class class  {...}      // Illegal
class @class {...}      // Legal
```

The `@` symbol doesn't form part of the identifier itself. So `@myVariable` is the same as `myVariable`.

> **NOTE**
>
> The `@` prefix can be useful when consuming libraries written in other .NET languages that have different keywords.

Contextual keywords

Some keywords are *contextual*, meaning they can also be used as identifiers—without an `@` symbol. These are:

```
add        dynamic into    partial when
ascending  equals  join    remove  where
async      from    let     select  yield
await      get     nameof  set
by         global  on      value
descending group   orderby var
```

With contextual keywords, ambiguity cannot arise within the context in which they are used.

Literals, Punctuators, and Operators

Literals are primitive pieces of data lexically embedded into the program. The literals we used in our example program are `12` and `30`.