4th Edition

Covers CLR 4.0

07 #3

C# 4.0

IN A NUTSHELL

The Definitive Reference

O'REILLY®

Joseph Albahari

& Ben Albahari

& Ben Albahari

C#/Microsoft .NET

C# 4.0 IN A NUTSHELL

When you have questions about how to use C# 4.0 or the .NET CLR, this highly acclaimed best-seller has precisely the answers you need. Uniquely organized around concepts and use cases, this fourth edition includes in-depth coverage of new C# topics such as parallel programming, code contracts, dynamic programming, security, and COM interoperability. You'll also find updated information on LINQ, including examples that work with both LINQ to SQL and Entity Framework. This handy book has all you need to stay on track with C# 4.0.

"C# 4.0 in a Nutshell *is one of the few books I keep on my desk as a quick reference. It is a book I recommend.*"

—Scott Guthrie
Corporate Vice President, .NET Developer Platform, Microsoft

work. This handy book has all you need to stay on track with C# 4.0.

.NET Developer Platform,
Microsoft

- Get up to speed on C# language basics, including syntax, types, and variables
- Explore advanced topics such as unsafe code and preprocessor directives
- Learn C# 4.0 features such as dynamic binding, type parameter variance, and optional and named parameters
- Work with .NET 4's rich set of features for parallel programming, code contracts, and the code security model
- Understand .NET topics, including XML, collections, I/O and networking, memory management, reflection, attributes, security, and native interoperability

"A must-read for a concise but thorough examination of the parallel programming features in .NET Framework 4."

—Stephen Toub
Parallel Computing Platform
Program Manager, Microsoft

"This wonderful book is a great reference for developers of all levels."

developers of all levels."

—Chris Burrows
C# Compiler Team, Microsoft

O'REILLY®
oreilly.com

US $49.99 **CAN $62.99**

ISBN: 978-0-596-80095-6

C# 4.0

IN A NUTSHELL

C#

4.0

4.0

IN A NUTSHELL

Fourth Edition

Joseph Albahari and Ben Albahari

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

C# 4.0 in a Nutshell, Fourth Edition

by Joseph Albahari and Ben Albahari

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://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Laurel R.T. Ruma
Production Editor: Loranah Dimant
Copyeditor: Audrey Doyle

Copyeditor: Audrey Doyle
Proofreader: Colleen Toporek

Indexer: John Bickelhaupt
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:

March 2002:
August 2003:
September 2007:

September 2007:
January 2010:

First Edition.
Second Edition.
Third Edition.
Fourth Edition.

ISBN: 978-0-596-80095-6
[M]
1263924338

Table of Contents

3.

V

Structs
Access Modifiers
Interfaces
Enums
Nested Types

6.

7.
8.
9.
10.

14.

Serialization

Emitting Generic Methods and Types
Awkward Emission Targets
Parsing IL

723
725
728

19.

Threading

832
840
849
851
852
855
857
860
862
865
869

22.
23.
24.
25.
26.

Preface

C# 4.0 further enhances Microsoft's flagship programming language with much-requested features—including support for dynamic programming, type parameter variance, and optional and named parameters. At the same time, the CLR and .NET Framework have grown to include a rich set of features for parallel programming, code contracts, and a new code security model.

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 existing 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 previous edition, *C# 4.0 in a Nutshell* is organized entirely around concepts and use cases, making it friendly both to sequential reading and to random browsing. It also plumbs significant depths while assuming only basic background

and use cases, making it 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 allow space for difficult topics such as concurrency, security, and application domains—without compromising depth or readability. Features new to C# 4.0 and the associated Framework are flagged so that you can also use this book as a C# 3.0 reference.

Intended Audience

This book targets intermediate to advanced audiences. No prior knowledge of C# is required, but some 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# 3.0, you'll find more than 100 pages dedicated to the new features of C# 4.0 and Framework 4.0. In addition, many chapters have

If you're already familiar with C# 3.0, you'll find more than 100 pages dedicated to the new features of C# 4.0 and Framework 4.0. In addition, many chapters have been enhanced from the previous edition, most notably the chapters on the C# language, .NET Framework fundamentals, memory management, threading, and COM interoperability. We've also updated the LINQ chapters to make the examples friendly to both LINQ to SQL and Entity Framework programmers.

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# 4.0 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 a replacement for IntelliSense (i.e., the alphabetical listings of types and type members that appeared in the C# 1.1 edition of this book).

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

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 preprocessor 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, I/O and networking, memory management, reflection, dynamic programming, attributes, security, concurrency, application domains, and native interoperability. You can 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.

What You Need to Use This Book

The examples in this book require a C# 4.0 compiler and Microsoft .NET Framework 4.0. You will also find Microsoft's .NET documentation useful to look up individual types and members. The easiest way to get all three—along with an integrated development environment—is to install Microsoft Visual Studio 2010. Any edition is suitable for what's taught in this book, including Visual Studio Express (a free download). Visual Studio also includes an express edition of SQL Server, required to run the LINQ to SQL and Entity Framework examples, and IntelliSense,

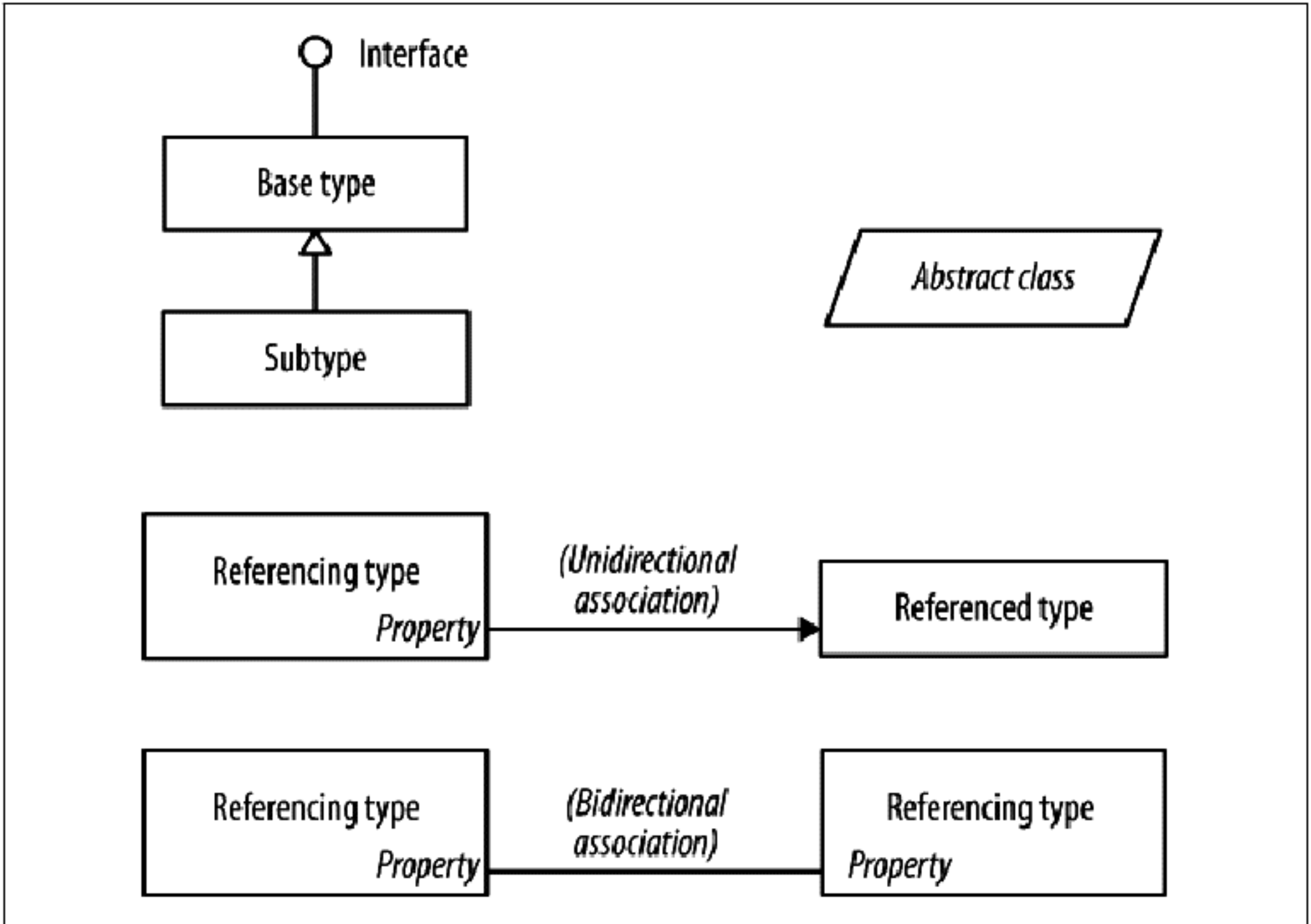
quired to run the LINQ to SQL and Entity Framework examples, and IntelliSense, which pops up type member listings as you type.

For Chapters 2 through 4, Chapter 6, Chapters 8 through 10, and Chapter 24, the code samples are available in the free code-snippet IDE, LINQPad. The samples include everything in those chapters from simple expressions to complete programs and are fully editable, allowing you to learn interactively. You can download LINQPad from *http://www.linqpad.net*; to obtain the additional samples, click "Download more samples" in the Samples tab at the bottom left. You can then advance through each sample with a single click.

Conventions Used in This Book

The book uses basic UML notation to illustrate relationships between types, as shown in Figure P-1. A 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.

an arrow denotes a two-way association.



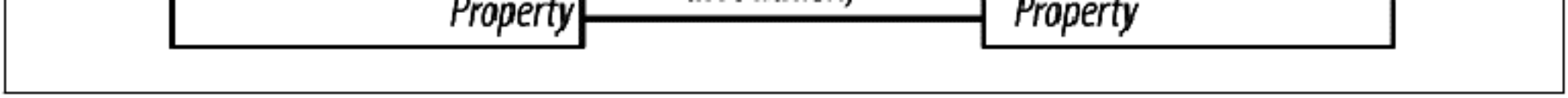


Figure P-1. Sample diagram

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URIs, filenames, and directories

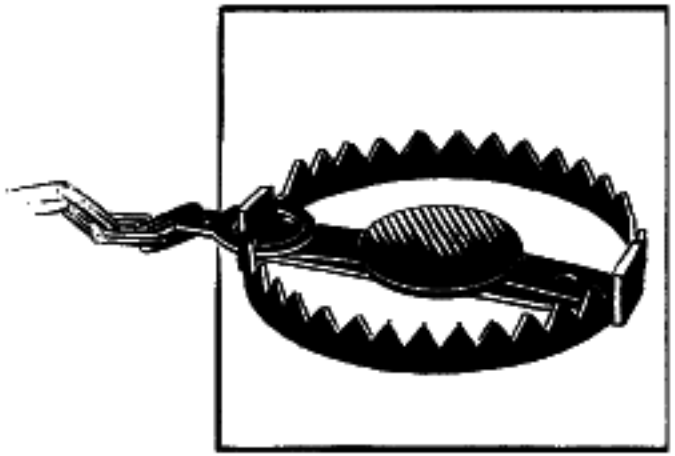
`Constant width`
: Indicates C# code, keywords and identifiers, and program output

`Constant width bold`
: Shows a highlighted section of code

`Constant width italic`
: Shows text that should be replaced with user-supplied values

This icon signifies a tip, suggestion, or general note

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code

This book is here to help you get your job done. In general, you may use the code in this book 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# 4.0 in a Nutshell* by Joseph Albahari and Ben Albahari. Copyright 2010 Joseph Albahari and Ben Albahari, 978-0-596-80095-6."

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

We'd Like to Hear from You

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

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://www.oreilly.com/catalog/9780596800956

Code listings and additional resources are provided at:

http://www.albahari.com/nutshell/

To comment or ask technical questions about this book, send email to the following

To comment or ask technical questions about this book, send email to the following, quoting the book's ISBN (9780596800956):

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our website at:

http://www.oreilly.com

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages,

and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

Acknowledgments

Joseph Albahari

First, I want to thank my brother and coauthor, Ben Albahari, for initially persuading me to take on what has become a highly successful project. I particularly enjoy working with Ben in probing difficult topics: he shares my willingness to question conventional wisdom, and the tenacity to pull things apart until it becomes clear how they *really* work.

I am most indebted to the superb technical reviewers. Starting with the reviewers at

I am most indebted to the superb technical reviewers. Starting with the reviewers at Microsoft, the extensive input from Stephen Toub (Parallel Programming team) and Chris Burrows (C# Compiler team) significantly enhanced the chapters on concurrency, dynamic programming, and the C# language. From the CLR team, I received invaluable input on security and memory management from Shawn Farkas, Brian Grunkemeyer, Maoni Stephens, and David DeWinter. And on Code Contracts, the feedback from Brian Grunkemeyer, Mike Barnett, and Melitta Andersen raised this chapter to the next quality bar. Thank you, people—both for your prompt feedback and for answering all my questions. I really appreciate it!

I have the highest praise for Jon Skeet (author of *C# in Depth* and Stack Overflow extraordinaire), whose perceptive suggestions enhanced numerous chapters (you work for Google, but we'll forgive you!). I'm similarly grateful for the keen eye of C# MVP Nicholas Paldino, who spotted errors and omissions that others missed. I'd also like to thank C# MVPs Mitch Wheat and Brian Peek, and reviewers of the 3.0 edition upon which this book was based. This includes the aforementioned Nicholas Paldino, who applied his thoroughness and breadth of knowledge to most chapters of the book, and Krzysztof Cwalina, Matt Warren, Joel Pobar, Glyn Griffiths, Ion Vasilian, Brad Abrams, Sam Gentile, and Adam Nathan.

Griffiths, Ion Vasilian, Brad Abrams, Sam Gentile, and Adam Nathan.

Finally, I want to thank the O'Reilly team, including my prompt and efficient editor, Laurel Ruma, my publicist, Kathryn Barrett, my copyeditor, Audrey Doyle, and members of my family, 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, I learned a lot from Brian Beckman, to whom I am indebted.

1

Introducing C# and the .NET Framework

C# is a general-purpose, type-safe, object-oriented programming language. The goal

C# is a general-purpose, type-safe, object-oriented programming language. The goal of the language is 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 of Turbo Pascal and architect of Delphi). The C# language is platform-neutral, but it was written to work 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

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, any type can be converted to a string by calling its `ToString` method.

Classes and interfaces

In the pure 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* (similar to Java interfaces). An interface is like a class except it is only a definition for a type, not an implementation. It's 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.

state changes.

Type Safety

C# is primarily a *type-safe* language, meaning that types can interact only through protocols they define, thereby ensuring each type's internal consistency. For instance, C# prevents you 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 *dynamic* type safety, which the .NET CLR enforces 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.

given variable what type it is, and hence what methods you can call on that variable.

C# 4.0 allows parts of your code to be dynamically typed via the new `dynamic` keyword. However, C# remains a predominately statically typed language.

C# is called a *strongly typed language* because its type rules (whether enforced statically or dynamically) 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 CLR has

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

Introducing C# and .NET

C# is typically used for writing code that runs on Windows platforms. Although Microsoft standardized the C# language and the CLR through ECMA, the total amount of resources (both inside and outside of Microsoft) dedicated to supporting C# on non-Windows platforms is relatively small. This means that languages such as Java are sensible choices when multiplatform support is of primary concern. Having said this, C# can be used to write cross-platform code in the following scenarios:

-
-
-

C# code may run on the server and dish up DHTML that can run on any platform. This is precisely the case for ASP.NET.

C# code may run on a runtime other than the Microsoft Common Language Runtime. The most notable example is the Mono project, which has its own C# compiler and runtime, running on Linux, Solaris, Mac OS X, and Windows.

C# code may run on a host that supports Microsoft Silverlight (supported for Windows and Mac OS X). This is a new technology that is analogous to Adobe's Flash Player.

C#'s Relationship with the CLR

C# depends on a runtime equipped with a host of features such as automatic memory management and exception handling. The design of C# closely maps to the design of the 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 primitive types).

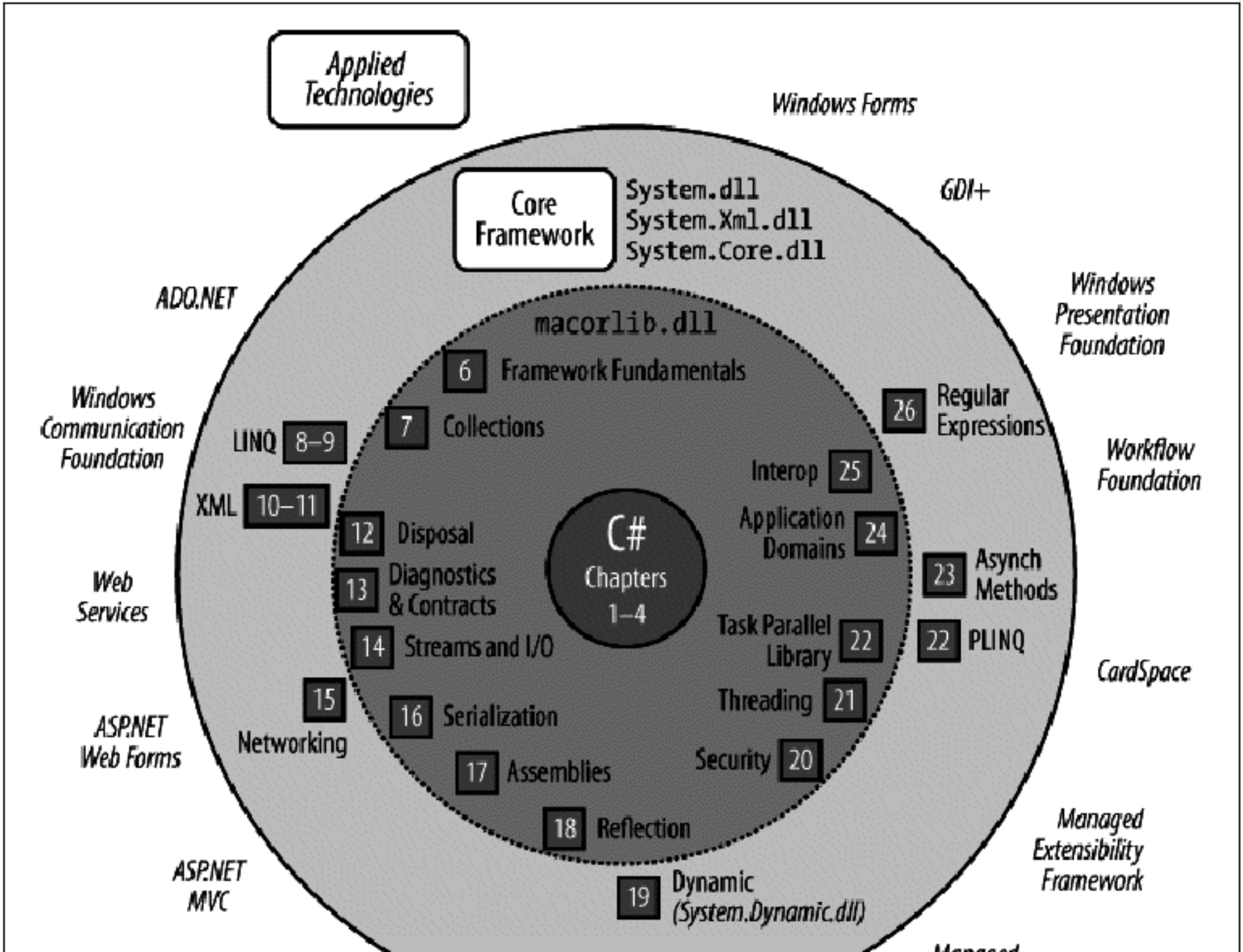
The CLR and .NET Framework

The CLR and .NET Framework

The .NET Framework consists of a runtime called the *Common Language Runtime* (CLR) and 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 libraries. Figure 1-1 is a visual overview of those libraries (and also serves as a navigational aid to the book).

The CLR is the runtime for executing *managed code*. C# is one of several *managed languages* that get 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's JIT (Just-In-Time) compiler. An assembly retains


Applied Technologies
Windows Forms
GDI+
Windows Presentation Foundation
Workflow Foundation
CardSpace
Managed Extensibility Framework
ASP.NET MVC
ASP.NET Web Forms
Web Services
Windows Communication Foundation
ADO.NET
Core Framework
System.dll
System.Xml.dll
System.Core.dll
mscorlib.dll
6 Framework Fundamentals
7 Collections
LINQ 8–9
XML 10–11
12 Disposal
13 Diagnostics & Contracts
14 Streams and I/O
15 Networking
16 Serialization
17 Assemblies
18 Reflection
19 Dynamic (System.Dynamic.dll)
C# Chapters 1–4
26 Regular Expressions
Interop 25
Application Domains 24
23 Asynch Methods
Task Parallel Library 22
22 PLINQ
Threading 21
Security 20

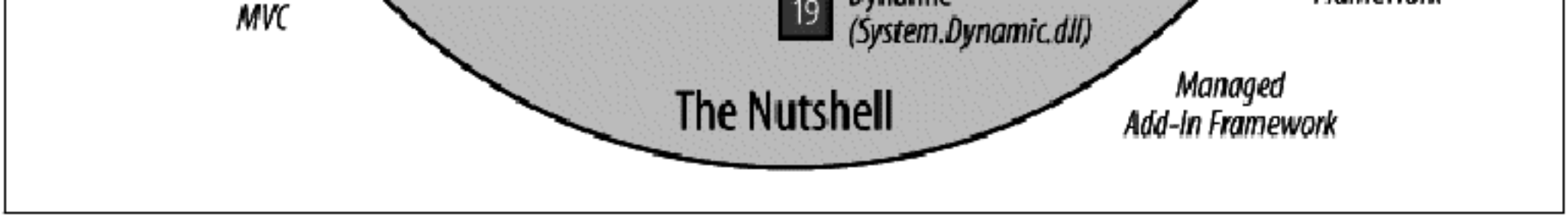


Figure 1-1. This depicts the topics covered in this book and the chapters in which they are found. The names of specialized frameworks and class libraries beyond the scope of this book are grayed out and displayed outside the boundaries of The Nutshell.

almost all of the original source language constructs, which makes it easy to inspect and even generate code dynamically.

Red Gate's .NET Reflector application is an invaluable tool for examining the contents of an assembly (you can also use it as a decompiler).

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#, Visual Basic .NET, Managed C++, Delphi.NET, Chrome .NET, and J#).

languages (e.g., C#, Visual Basic .NET, Managed C++, Delphi.NET, Chrome .NET, and J#).

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

What's New in C# 4.0

The new features in C# 4.0 are:

-
-
-
-
-

Dynamic binding

Dynamic binding

Type variance with generic interfaces and delegates

Optional parameters

Named arguments

COM interoperability improvements

Dynamic binding (Chapters 4 and 19) is C# 4.0's biggest innovation. This feature was inspired by dynamic languages such as Python, Ruby, JavaScript, and Smalltalk. Dynamic binding defers *binding*—the process of resolving types and members—from compile time to runtime. Although C# remains a predominantly statically typed language, a variable of type `dynamic` is resolved in a late-bound manner. For example:

```
dynamic d = "hello";
Console.WriteLine (d.ToUpper());
Console.WriteLine (d.Foo());

// HELLO
// Compiles OK but gives runtime error
```

Calling an object dynamically is useful in scenarios that would otherwise require

Calling an object dynamically 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. An optional parameter declaration such as:

```
void Foo (int x = 23) { Console.WriteLine (x); }
```

can be called as follows:

```
Foo(); // 23
```

Named arguments (Chapter 2) allow a function caller to identify an argument by name rather than position. For example, the preceding method can now be called as follows:

```
Foo (x:5);
```

Type variance (Chapters 3 and 4) allows generic interfaces and generic delegates to mark their type parameters as covariant or contravariant. This enables code such as

Type variance (Chapters 3 and 4) allows generic interfaces and generic delegates to mark their type parameters as covariant or contravariant. This enables code such as the following to work:

```
IEnumerable<string> x = ...;
IEnumerable<object> y = x;
```

COM interoperability (Chapter 25) has been enhanced in C# 4.0 in three ways. First, arguments can be passed by reference without the `ref` keyword. This feature is particularly useful in conjunction with optional parameters. It means that the following C# 3.0 code to open a Word document:


```
object o1 = "foo.doc";
object o2 = Missing.Value;
object o3 = Missing.Value;
...
```

```
...
word.Open (ref o1, ref o2, ref o3...);
```

can now be simplified to:

```
word.Open ("Foo.doc");
```

Second, assemblies that contain COM interop types can now 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 *variant* types from linked interop types are mapped to `dynamic` rather than `object`, eliminating the need for casting.

2

C# Language Basics

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

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

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, click the Samples tab in LINQPad and then click "Download more samples." 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 forward slash indicates that the remainder of a line is a *comment*:

```
using System;
class Test
{
```

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

// Importing namespace
// Class declaration
//   Method declaration
//     Statement 1
```

```
//     Statement 1
//     Statement 2
//   End of method
// End of class
```

At the heart of this program lie two *statements*. Statements in C# execute sequentially. Each statement is terminated by a semicolon:

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


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` *method*, 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

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));
    Console.WriteLine (FeetToInches (100));
  }
  static int FeetToInches (int feet)
  {
    int inches = feet * 12;
    return inches;
  }
}
```

```
// 360
// 1200
```

```
// 1200
```

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. 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) {...}
```

passed to the executable). For example:

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


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" on page 32.

Methods are one of several kinds of functions in C#. Another kind of function we used was the * *operator*, used to perform 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" on page 12.

At the outermost level of a program, types are organized into *namespaces*. The

amine in "Type Basics" on page 12.

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;
```

```
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`.

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. 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 under *%SystemRoot%\Microsoft.NET\Framework\<framework-version>* where *%SystemRoot%* is your Windows directory) as follows:

tory) as follows:

```
csc MyFirstProgram.cs
```

This produces an application named *MyFirstProgram.exe*.

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

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

We explain assemblies in detail in Chapter 16.

Syntax

Syntax

C# syntax is based on 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
```

```
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 reserved by the compiler that you can't use as identifiers. These are the keywords in our example program:

```
using
```

```
class
```

```
static
```

```
void
```

```
void
int
```


Here is the full list of C# keywords:

```
abstract
byte
class
delegate
event
fixed
if
```

if
internal
new
override
readonly
short
struct
try
unsafe
void
as
case

case
const
do
explicit
float
implicit
is
null
params
ref
sizeof
switch
typeof

switch
typeof
ushort
while

base
catch
continue
double
extern
for
in
lock
object

```
object
private
return
stackalloc
this
uint
using

bool
char
decimal
else
false
```

else
false
foreach
int
long
operator
protected
sbyte
static
throw
ulong
virtual

virtual

break
checked
default
enum
finally
goto
interface
namespace
out
public
sealed

C# Basics

```
sealed
string
true
unchecked
volatile
```

Avoiding conflicts

If you really want to use an identifier that clashes with a 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`.

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

Contextual keywords

Contextual Keywords

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

```
add
from
join
select

ascending
get
let
set

by
```

```
by
global
on
value
descending
group
orderby
var
dynamic
in
partial
```