4th Edition
Covers CLR 4.0

Free Sampler

C# 4.0

IN A NUTSHELL

The Definitive Reference

Joseph Albahari
& Ben Albahari

O'REILLY®

O'Reilly Ebooks—Your bookshelf on your devices!

When you buy an ebook through oreilly.com, you get lifetime access to the book, and whenever possible we provide it to you in four, DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, and Android .apk ebook—that you can use on the devices of your choice. Our ebook files are fully searchable and you can cut-and-paste and print them. We also alert you when we've updated the files with corrections and additions.

Learn more at http://oreilly.com/ebooks/

You can also purchase O'Reilly ebooks through iTunes, the Android Marketplace, and Amazon.com.

C# 4.0 in a Nutshell, Fourth Edition
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://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
Proofreader: Colleen Toporek
Indexer: John Bickelhaupt
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:

March 2002:	First Edition.
August 2003:	Second Edition.
September 2007:	Third Edition.
January 2010:	Fourth Edition.

ISBN: 978-0-596-80095-6

[M] [3/10]

1268156389

Table of Contents

1

Introducing C# and the .NET Framework

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 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.

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.

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 a garbage collector that executes as part of your program, reclaiming memory for objects that are no longer referenced. This frees programmers from explicitly deal-

locating 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

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 .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

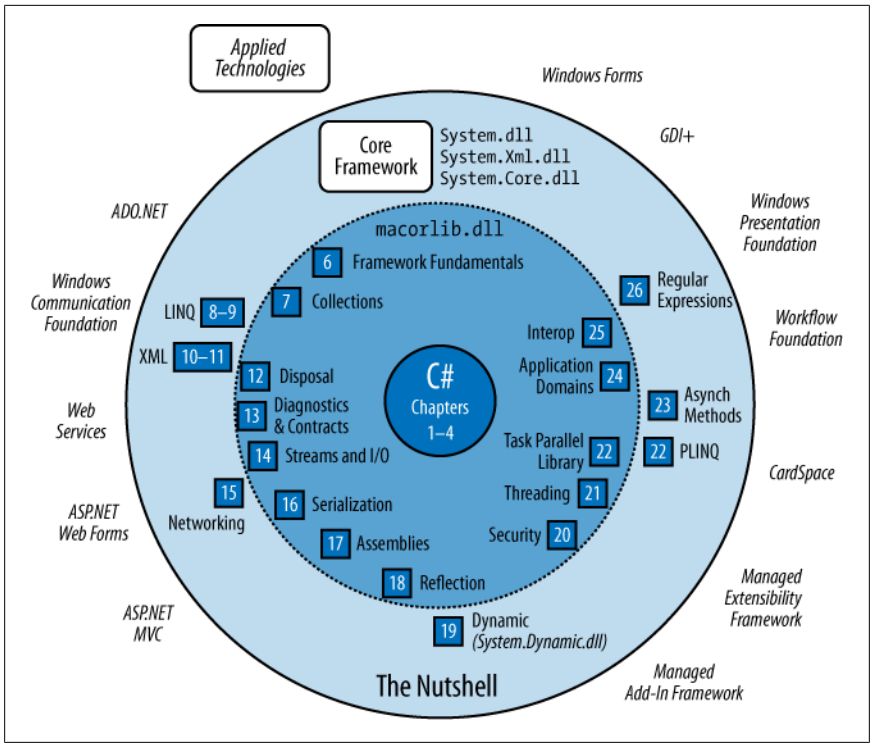

Figure 1-1. This depicts the topics covered in this book and the chapters in which they are found. The names of specialized frameworks and class libraries beyond the scope of this book are grayed out and displayed outside the boundaries of The Nutshell.

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 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#).

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
- 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());  // HELLO
Console.WriteLine (d.Foo());      // Compiles OK but gives runtime error
```

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 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.